Grupo 3

Caio Cesar Vieira Cavalcanti - 123110825
Walber Wesley Félix de Araújo Filho - 123110522
João Pedro Azevedo do Nascimento - 123110768
Rafael Barreto da Silva - 123110738
Valdemar Victor Leite Carvalho - 123110796

Planejamento

O planejamento e a alocação de tarefas entre membros da equipe também está disponível no GitHub Projects do nosso repositório.

Etapa 3

US09 - Rafael
US10 - Valdemar
US11 - Walber
US12 - João Pedro
US13 - Caio

Finalização Etapa 3: 24/08/2025

Descrição das Atividades

US09 (Rafael)

Participação Total: 20%

Para a US09, implementei o fluxo inicial de solicitação de compra de ativos por parte do cliente. A operação é realizada através de um endpoint no ClientController, que recebe como parâmetros o clientId, o assetId, a quantidade desejada e o código de acesso do cliente (via *DTO ClientPurchaseAssetRequestDTO*). Esse endpoint delega a lógica de negócio ao *ClientService*, que por sua vez valida o cliente (via *accessCode*), a quantidade de ativos requisitados e o encaminha para o *PurchaseServiceImpl*.

O *PurchaseServiceImpl* é responsável por criar uma nova instância de *PurchaseModel*, associando-a à carteira (*WalletModel*) e ao ativo (*AssetModel*). Essa instância é inicializada no estado REQUESTED, utilizando o padrão State implementado em conjunto com o *PurchaseModel*. O uso do *enum PurchaseStateEnum* em conjunto com o padrão State foi necessário para evitar problemas de dependência cíclica no banco de dados; a solução adotada foi o uso de *@PostLoad* para restaurar corretamente o estado transiente após carregar uma entidade persistida.

Além disso, antes de registrar a solicitação, o serviço garante que:

- O ativo está ativo e disponível;
- O cliente possui saldo suficiente na carteira para a operação.

Caso as condições não sejam atendidas, exceções personalizadas são lançadas, assegurando que a compra só possa ser registrada se respeitar as regras de negócio da corretora.

Os testes criados para essa US foram implementados na classe *ClientControllerTests*, cobrindo o fluxo end-to-end da solicitação de compra. Os cenários contemplados incluem:

- Solicitação bem-sucedida, retornando um *PurchaseResponseDTO* com os atributos esperados *(id, walletId, assetId, quantity, state e date*);
- Falhas devido a saldo insuficiente na carteira,o ativo não está disponível, entradas inválidas tipo *accessCode* errado, id de cliente inexistente/nulo, id de ativo inexistente/nulo e etc.

Esse fluxo implementado estabelece a base para as USs seguintes (US10–US13), pois é a partir da compra inicial no estado *REQUESTED* que os demais estados e operações são construídos.

US10 (Valdemar)

Participação Total: 20%

Foi realizado um conjunto abrangente de testes de unidade para as camadas de Model e Service da aplicação, focando principalmente no fluxo de compras e na gestão de carteiras (wallets).

Na camada de model, os testes abordaram os diferentes estados de uma compra (*PurchaseRequestedState*, *PurchaseAvailableState*, *PurchasePurchasedState* e *PurchaseInWalletState*). Para o estado *PurchaseRequestedState*, foram criados cenários que verificam:

- A transição correta para *PurchaseAvailableState* quando um administrador confirma a disponibilidade do ativo;
- O lançamento de exceções apropriadas quando um usuário não administrador tenta modificar a compra (*UnauthorizedUserAccessException*);
- A validação do ativo quanto à sua disponibilidade (*AssetIsInactiveException*) e quantidade disponível (*AssetQuantityAvailableIsInsufficientException*).

Nos estados *PurchasePurchasedState* e *PurchaseInWalletState*, os testes confirmaram que a modificação da compra altera corretamente o estado ou permanece inalterada, garantindo que a lógica de transição entre estados seja consistente com as regras de negócio. Além disso, testes foram criados para verificar o comportamento do método modify da própria *PurchaseModel*, assegurando que o estado transiente é carregado corretamente via *loadState* quando necessário. Na camada de Service, os testes cobriram também o método *addedInWallet* do *PurchaseServiceImpl* e a função *addPurchase* do *WalletService*. Foram contemplados principalmente dois cenários:

- Quando a compra é adicionada a uma holding já existente, garantindo que a quantidade e o preço acumulado são atualizados corretamente, que o repositório persiste a compra e que o DTO de resposta correto (*PurchaseResponseAfterAddedInWalletDTO*) é retornado;
- Quando a compra é adicionada e não existe uma holding correspondente, verifica-se a criação de uma nova holding, a persistência da compra e a consistência do DTO de resposta retornado.

Todos esses e outros testes de Service utilizaram mocks para simular dependências externas, como repositórios e serviços auxiliares, permitindo validar exclusivamente a lógica de negócio. Situações de sucesso e possíveis falhas foram contempladas, assegurando que o sistema lança as exceções corretas e mantém a consistência dos dados em diferentes cenários.

Esse conjunto de testes fortalece a confiabilidade do fluxo de compras e da gestão de carteiras, garantindo que transições de estado, validações e atualizações de holdings ocorram conforme o esperado e que as respostas retornadas aos clientes estejam consistentes com os requisitos de negócio.

US11 (Walber)

Participação Total: 20%

Foi implementada a funcionalidade que permite que um administrador do sistema confirme a disponibilidade de uma compra feita por um cliente. Essa confirmação verifica duas condições principais:

1. O ativo está ativo e disponível para negociação;
2. Há liquidez suficiente para atender à quantidade solicitada pelo cliente.

Se ambas as condições forem atendidas, o estado da compra é atualizado de **"Solicitado"** para **"Disponível"**.

Fluxo da Funcionalidade:

1. O administrador envia uma requisição POST /purchases/{purchaseId}/availability-confirmation com seu e-mail e código de acesso.
2. O sistema valida se o usuário é realmente um administrador e se os dados de acesso estão corretos.
3. O estado da compra é modificado através do padrão **State**, utilizando a classe PurchaseRequestedState.
4. Se o ativo não estiver ativo ou a quantidade disponível for insuficiente, exceções específicas são lançadas (AssetIsInactiveException, AssetQuantityAvailableIsInsufficientException).
5. Caso a confirmação seja bem-sucedida, o sistema envia uma **notificação ao cliente**, indicando que a compra está disponível. A notificação é representada no terminal como uma mensagem detalhada, incluindo:

    - Nome do ativo;
    - Quantidade solicitada;
    - Preço unitário e valor total;
    - Motivo da notificação

Considerações Técnicas:

- O padrão **State** foi utilizado para separar comportamentos de acordo com o estado da compra.
- O envio da notificação é realizado via log (logger.info) e também como mensagem no terminal.
- O código segue boas práticas de validação e tratamento de exceções, garantindo que apenas administradores possam confirmar a disponibilidade da compra.

Testes:

Foram implementados **testes de unidade** e **testes de integração** que cobrem todos os fluxos principais de lógica da funcionalidade. Esses testes validam os pontos importantes da US como a autenticação do administrador, mudança de estado, validação de liquidez e verificação se o Asset está ativo. A suíte de testes apresenta **alta cobertura**, garantindo confiabilidade e robustez para a funcionalidade entregue.

US12 (João Pedro)

Participação Total: 20%

Para a US12, criei um endpoint em *ClientController* para que o cliente execute a operação de confirmação da compra, que em seguida, de forma automática, irá disparar a adição dessa compra na carteira. Primeiramente, o método *confirmationByClient* em *ClientController* espera um *clientId*, *purchaseId* e um DTO, o qual contém apenas o *accessCode* do cliente, o qual é necessário para validar a operação. Esse método delega para *purchaseConfirmationByClient* em *ClientService*, que irá realizar a validação do cliente com o *accessCode* e, posteriormente, executa *confirmationByClient* do *purchaseService*, que realiza a modificação do estado de compra (Availability -> Purchased) e retorna a compra (*PurchaseModel*). Após isso, é realizado o decremento do saldo na carteira do cliente conforme o valor da compra, caracterizado pela quantidade de ativos multiplicado pelo preço dele. Por fim, o *purchaseConfirmationByClient* irá executar e retornar o método de *WalletService* chamado *addPurchase*, que, junto a método com mesmo nome em *PurchaseService*, irá atualizar o estado da compra (Purchased -> In_Wallet) e cria ou atualiza um *HoldingModel* – objeto que centraliza todas as compras de um mesmo ativo para diminuir os custos das operações sobre eles. Caso tenha criado um *HoldingModel*, irá também adicioná-lo em um *Map* nomeado como *holdings* em *Wallet*. Por fim, será retornado para *ClientController* um *PurchaseResponseDTO.* Toda implementação foi feita evitando bad smells, como métodos longos e operações realizadas por entidades não experts na informação, e mantendo o padrão State.

Além disso, realizei os testes de integração que checam se o fluxo end-to-end deu certo, ou seja, se retornou status 200 tanto no cenário em que se cria um *HoldingModel* quanto no que atualiza um *HoldingModel* já existente. Ainda fiz checagens específicas – as quais também foram testadas isoladamente –, como verificação do saldo da carteira e estado da estrutura de dados holdings após a compra.

US13 (Caio)

Participação Total: 20%

Para a US13, é necessário um endpoint na classe *ClientController* (*/wallet-holding*) responsável pelo envio da requisição do cliente com o objetivo de visualizar sua carteira de investimentos, no atual momento da requisição, contendo todos os ativos comprados, incluindo propriedades como: tipo (tipo do ativo), quantidade, valor de aquisição (assumindo ser o valor unitário do ativo no momento da compra, obtendo pela razão do *accumulatedPrice/quantity*), valor atual (a cotação do ativo no momento) e o desempenho (relatando se o cliente teve lucro ou prejuízo nesse conjunto de ativos comprados).

Como resposta da requisição (*WalletHoldingResponseDTO*), o cliente recebe uma lista com todos os ativos comprados no sistema (*HoldingResponseDTO*), cada um contendo as informações desejadas acima e, ao final, um sumário do valor atual de todos os ativos, valor total investido, e a margem de lucro ou prejuízo (desempenho) com a subtração desses dois atributos, respectivamente.

Assim como as demais USs feitas, foi seguido rigorosamente uma implementação evitando *bad smells*, tal como más práticas, estudadas ao longo da disciplina, e as devidas correções, caso fossem encontradas.

Para os testes dessa US, seguiu-se o padrão de testes feitos até então. Testar o endpoint na classe exclusiva para testes de integração (*ClientControllerTests*), checando o fluxo end-to-end (ponta-a-ponta), em caso de sucesso (HTTP Code 200), ou casos de falhas, como *bad request* (HTTP Code 400), dado o código de acesso enviado pelo usuário no *ClientWalletRequestDTO* inválido ou nulo. Por fim, testes de unidade (*ClientServiceUnitTests*), com a finalidade de testar a camada de serviços e outras classes acopladas a ela e, utilizadas na lógica de negócio da US em

questão, como o cálculo envolvendo os valores de cada holding referente aos ativos comprados pelo cliente, no retorno dos DTOs como *HoldingResponseDTO* e *WalletHoldingResponseDTO*, contendo o tipo do ativo, quantidade adquirida, valor de aquisição, valor atual e o desempenho (lucro - margem positiva -, ou prejuízo - margem negativa), além de um sumário total desses últimos 3 atributos contendo a informação para toda a carteira.

Principais Decisões de Design

Durante a implementação inicial das User Stories foram tomadas algumas decisões importantes de design:

1. **Modificação da Wallet já existente**
    - A *WalletModel* foi removido os maps de compras e resgates para evitar ter que salvar múltiplas vezes todas as mudanças em carteira, dessa forma foi criado um repositório próprio para compras e resgates que tem conexão intrínseca com a carteira. Além disso, foi adicionado um Map de holdings, onde cada holding representa um ativo comprado pelo cliente.

2. **Separação da lógica de transações**
    - Foi criado um repositório próprio para o *PurchaseModel e WithdrawModel*, separando-o da Wallet. Essa decisão melhora a coesão, facilita o fluxo de transações e permite agrupar funcionalidades em seus serviços específicos.

3. **Criação do HoldingModel**
    - Como apresentado anteriormente, o *HoldingModel* foi introduzido para representar ativos dentro da carteira de forma consolidada. Ele permite agrupar múltiplas compras de um mesmo ativo, facilitando cálculos de quantidade total, preço médio e desempenho, além disso vai ser responsável por evitar refatoramento de código em User Stories futuras relacionadas às operações de venda.

4. **Padrão State + Enum**

    - O padrão **State** foi adotado para gerenciar as transições de estados da compra (Requested → Available → Purchased → In_Wallet). Porém, para resolver o problema de dependência cíclica no banco de dados (já que o padrão exige referência bidirecional entre estado e modelo), foi decidido utilizar também o *PurchaseStateEnum.*

    - O enum funciona como "estado persistido" no banco, enquanto o padrão **State** é utilizado em memória para orquestrar a lógica de transição. O @PostLoad garante que, ao carregar a entidade, o estado correto seja restaurado.

Diagrama: Link