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 4

US14 - Valdemar
US15 - Caio
US16 - João Pedro
US17 - Walber
US18 - Rafael
US19 - Caio
US20 - Valdemar, João Pedro, Walber e Rafael

Finalização Etapa 4: 07/09/2025

Descrição das Atividades

US14 (Valdemar)

Participação Total: 15%

Foi realizada a implementação e um conjunto abrangente de testes de unidade e integração para as funcionalidades de retirada de ativos (withdraw) da aplicação, abrangendo as camadas de Controller, Service, além de DTOs, com foco principal no fluxo de gestão de holdings e carteira do cliente.

Na camada de Controller, os testes contemplaram o endpoint POST /clients/{clientId}/wallet/withdraw/{assetId} do ClientController, verificando cenários de sucesso e de validação. Foram criados testes que confirmam:

- A retirada parcial de um ativo existente em uma holding, garantindo que a quantidade retirada é corretamente refletida no WithdrawResponseDTO;

- A utilização do DTO ClientWithdrawAssetRequestDTO, incluindo a validação do accessCode e da quantidade a ser retirada, assegurando que requisições com dados válidos resultam em

sucesso (HTTP 200 OK).

Na camada de Service, os testes focaram no método withdrawClientAsset do ClientServiceImpl, garantindo a lógica completa de retirada de ativos. Foram contemplados os seguintes cenários:

- Retirada de quantidade menor que a holding existente, atualizando corretamente a quantidade e o preço acumulado na holding, e retornando o DTO de resposta esperado;
- Validação da existência da holding do ativo no wallet do cliente, lançando exceções apropriadas (HoldingNotFoundException) quando não encontrada;
- Confirmação do accessCode do cliente, garantindo que apenas usuários autorizados podem efetuar a retirada;
- Persistência adequada das alterações no WalletRepository e atualização das holdings envolvidas.

Na camada de Model, foram considerados os comportamentos relacionados à atualização das holdings e à consistência do WalletModel. Testes verificaram:

- Que a alteração de quantidade na holding reflete corretamente no estado do WalletModel;
- A integração com o AssetModel, garantindo que o ativo retirado mantém integridade de dados mesmo após a operação;
- Cenários com múltiplas holdings e compras pré-existentes, assegurando que a retirada de um ativo não afeta holdings de outros ativos.

Além disso, foram criados testes de integração utilizando MockMvc, validando a comunicação entre Controller e Service, bem como a serialização/deserialização dos DTOs (ClientWithdrawAssetRequestDTO e WithdrawResponseDTO). Todos os testes simularam interações reais com o banco de dados em memória (H2), incluindo a criação de clientes, wallets, holdings e assets, garantindo que o fluxo completo da operação de withdraw ocorra conforme esperado.

Esse conjunto de testes fortalece a confiabilidade da funcionalidade de retirada de ativos, assegurando que as validações, atualizações de holdings, persistência de dados e respostas ao cliente estão consistentes com os requisitos de negócio e respeitam as regras de autorização e integridade de dados.

US15 (Caio)

Participação Total: 10%

Na US15, é descrito que o sistema deve ser apto a calcular o imposto sobre cada operação de resgate feita pelo cliente, a um dado tipo de ativo e conforme o valor de lucro obtido. Para isso, delegamos a lógica do cálculo da taxa para o método abstrato *taxCalculate()* presente na classe abstrata *AssetType* – desenvolvida logo nas primeiras USs, também de minha responsabilidade. As classes concretas que se estendem dessa superclasse, são exatamente dos 3 tipos de ativos

existentes no sistema: *Stock, Crypto* e *TreasuryBounds*. Tal método, aplica às seguintes alíquotas, a depender do tipo de ativo:

1. Treasury Bounds: 10% sobre o lucro;
2. Stock: 15% sobre o lucro;
3. Crypto:
    a. 15% se o lucro for até R$ 5.000;
    b. 22,5% se o lucro for acima de R$ 5.000.

Para os valores presentes nas estruturas condicionais e nos cálculos, foi utilizado constantes definidas no início da classe, conforme dita a boa prática de evitar valores “mágicos” (Magic Numbers) espalhados no código, facilitando assim, legibilidade e principalmente manutenibilidade, em alterar, somente, o conteúdo dessas constantes, ao invés de modificar em todo o código em que o valor é recorrente.

A integração desse cálculo no sistema, foi através do método *withdrawAsset* e *confirmWithdraw* (US16 e US17), presentes na classe *WithdrawServiceImpl*, com auxílio de dois métodos privados responsáveis, respectivamente, por calcular a taxa (*calculateWithdrawTax*) – contendo como parâmetros o holding, referente a compra do ativo, o ativo, e a quantidade de ativos a se resgatar – e por fim, calcular o valor total subtraído (ou não, em casos que o lucro do cliente seja negativo/prejuízo) da taxa calculada anteriormente, com o valor bruto (*gross*) e resultando no valor real de resgate com o imposto incluído (*calculateWithdrawValue*). Acerca dos detalhes sobre como é realizado o processo de resgate (pelo método auxiliar *proccessWithdraw*), fica a cargo das USs comentadas no início, mas que já incluem a lógica do imposto sobre o valor resgatado.

Como detalhe, a taxa e o eventual valor de resgate (incluído o imposto) são calculados no momento que o cliente solicita o resgate, e até a confirmação do administrador, esses valores não alteram. Mantendo a robustez e a confiabilidade do sistema, frente a uma operação sensível financeiramente.

Foram desenvolvidos testes de unidade para validar o comportamento da funcionalidade de retirada em diferentes cenários de ativos. Os testes verificam se o cálculo da taxa de retirada está sendo aplicado corretamente de acordo com o tipo de ativo: Stock (15%), Treasury Bounds (10%) e Crypto (15% para lucros até R$ 5.000 e 22,5% acima desse valor). Além disso, foi testado o caso de holding inexistente, garantindo que uma exceção seja lançada quando não há ativos suficientes para o saque. Com isso, os testes asseguram que a regra de tributação sobre o lucro está consistente com a lógica de negócio definida na US e que o sistema responde corretamente tanto em cenários de sucesso quanto de falha.

US16 (João Pedro)

Participação Total: 15%

Na US16, foi requerida a operação de acompanhamento do status de cada resgate iniciado, incluindo estados, data e impostos pagos. Para implementar essa funcionalidade, utilizei o padrão de projeto State, garantindo que as transições de estados fossem controladas e seguissem a lógica de negócio definida. Assim, criei uma hierarquia de estados de resgate, começando pelo estado *REQUESTED*, atribuído automaticamente no momento em que o cliente dispara o resgate de um ativo. Esse estado pode ser alterado para *CONFIRMED*, após a validação do administrador da corretora. Imediatamente após o estado ser alterado para *CONFIRMED*, ele pode ser evoluído para *IN_ACCOUNT*, quando o valor é efetivamente creditado na conta do cliente.

Cada estado foi implementado como uma classe concreta que implementa uma interface comum, responsável por definir as operações possíveis. Dessa forma, centralizei a lógica de mudança de estados nas próprias classes, evitando condicionais espalhadas no código e facilitando a manutenção e extensão futura. Além disso, cada resgate armazena informações de data e impostos, que são preservadas em todas as transições.

Por fim, realizei testes de integração para verificar as transições válidas entre os estados, garantindo que não fosse possível, por exemplo, pular diretamente de *REQUESTED* para *IN_ACCOUNT*.

US17 (Walber)

Participação Total: 15%

Foi implementada a funcionalidade que permite que um **administrador do sistema** confirme a solicitação de **resgate feita por um cliente**.

Se essa condição for atendida, o estado do resgate é atualizado de **“Solicitado” para “Confirmado”** e, em seguida, automaticamente para **“Em Conta”**, permitindo que o valor seja transferido para a conta do cliente.

Fluxo da Funcionalidade

1. O administrador envia uma requisição **POST /withdraws/{withdrawId}/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 do resgate é modificado através do padrão **State**, utilizando a classe WithdrawRequestedState.

    - Se o cliente não possuir quantidade suficiente do ativo, é lançada a exceção ClientHoldingIsInsufficientException.

    - Se o usuário não for administrador, é lançada a exceção UnauthorizedUserAccessException.

4. O sistema atualiza o estado do resgate para **CONFIRMED** e, em seguida, para **IN_ACCOUNT**.

5. O sistema processa o resgate na carteira do cliente, removendo ou atualizando o holding e ajustando o orçamento disponível.

6. Caso a confirmação seja bem-sucedida, o sistema envia uma **notificação ao cliente**, indicando que o resgate foi confirmado.

- A notificação é representada no terminal como uma mensagem detalhada (print/logger), incluindo:
    - Nome do ativo;
    - Status da solicitação;
    - Motivo da notificação.

Testes

Foram implementados **testes de unidade** e **testes de integração** que cobrem os fluxos principais da lógica da funcionalidade como a autenticação do administrador, mudança de estado do resgate, validação de quantidade suficiente do ativo.

A suíte de testes apresenta alta cobertura, garantindo confiabilidade e robustez para a funcionalidade entregue.

US18 (Rafael)

Participação Total: 15%

Foi desenvolvida a funcionalidade de consulta ao histórico de resgates do cliente e modificada o histórico de compras já existente de uma US anterior. A implementação agora permite que o cliente filtre seu histórico de compras/resgates por tipo de ativo, período e status da operação, conforme especificado na *US18*.

O método *getWithdrawHistory* foi implementado em um novo controller chamado *WithdrawController* enquanto o *getPurchaseHistory* passou para o *PurchaseController*, operando da seguinte forma: primeiro, faz as validações para verificar se é um cliente de fato, em seguida recupera todos os resgates associados à carteira do cliente a partir do repositório *withdrawRepository*. Por fim, aplica filtros sequenciais com base nos parâmetros opcionais fornecidos pelo DTO *ClientWithdrawHistoryRequestDTO*. Os filtros incluem tipo de ativo (verificando se o tipo do ativo corresponde ao valor informado), status da operação (comparando com o estado do resgate) e data (filtrando por data específica). Caso nenhum filtro seja informado, todos os resgates são retornados.

Após a filtragem, os resultados são mapeados para o DTO de resposta *WithdrawHistoryResponseDTO* utilizando um serviço de mapeamento (*dtoMapperService*). A abordagem seguiu o padrão já estabelecido para consultas de histórico no sistema, garantindo consistência arquitetural e alinhamento com as convenções do projeto. A validação dos parâmetros de entrada é tratada automaticamente pelo *framework*, assegurando que requisições com dados incompletos ou inválidos sejam rejeitadas antes de atingir a lógica de filtragem.

A funcionalidade foi integrada com sucesso ao sistema, permitindo que clientes acessem seu histórico de resgates de forma flexível e intuitiva, com respostas padronizadas e aderentes às regras de negócio e foram implementados testes de unidade e testes de integração que cobrem os fluxos principais da lógica da funcionalidade.

US19 (Caio)

Participação Total: 10%

Para a US19, espera-se que o administrador do sistema, consiga consultar todas as operações realizadas por todos os clientes cadastrados no sistema, com filtros por tipo de ativo, data, cliente (pela identificação) e tipo de operação (compra/*purchase* ou resgate/*withdraw* - permitindo ambos). Consoante a isso, a principal mudança no sistema foi a criação de um novo controlador, *ReportController*, responsável por conter essa responsabilidade de emitir uma espécie de relatório de consulta, por todas as operações realizadas por todos os clientes. Ademais, seguindo a arquitetura de camadas, outras classes foram criadas, como *ReportService* e *ReportServiceImpl*, além de algumas importantes para todo o arcabouço desse novo endpoint, explicadas e detalhadas nos próximos tópicos.

No corpo da requisição (*OperationReportRequestDTO*), será passado as credenciais do administrador e, opcionalmente, os 4 campos de filtragem (tipo do ativo, data - inicial e/ou final, cliente e tipo de operação). Em caso de omissão de um desses campos (valor *null*), o padrão será considerar todos ou qualquer valor presente no sistema (incluindo no tipo de operação, os dois ao mesmo tempo). O retorno, como resposta da requisição, é uma lista de *OperationReportResponseDTO*, no qual possui como campos, todas as informações referentes a operação realizada pelo cliente, incluindo valores de taxa e resgate (com o imposto incluído) – no caso da operação filtrada ser de resgate, ou tais valores omitidos se a operação for de compra, e em ambas, a data que ocorreu tal transação. Para viabilizar essa funcionalidade, criamos o *ReportController*, que expõe o endpoint POST */reports/operations*. Esse controlador delega a responsabilidade de negócio para o serviço *ReportService* (interface) e sua implementação *ReportServiceImpl*.

O *ReportServiceImpl* valida as credenciais do administrador por meio do *AdminService* e, em seguida, utiliza uma coleção de fetchers/buscadores (*PurchaseFetcher* e *WithdrawFetcher*). Cada fetcher encapsula a lógica de consulta e mapeamento de um tipo de operação específico, aplicando os filtros recebidos no *OperationReportRequestDTO*. Ademais, o núcleo da lógica de consulta está nos repositórios, que utilizam queries JPQL para aplicar dinamicamente os filtros recebidos no *OperationReportRequestDTO*.

Os fetchers herdam de *OperationFetcherTemplate<T>*, que padroniza comportamentos comuns (como resolução de client, wallet, e date), enquanto deixam a busca (*findItems()*) e o mapeamento (*map()*) de cada operação a cargo das classes concretas (*PurchaseFetcher* e *WithdrawFetcher*). Essa combinação de herança e composição permite que novas operações futuras sejam adicionadas com baixo impacto no código existente, não violando um dos princípios do SOLID, o Princípio do Aberto-Fechado (OCP). Além disso, mapeamento para o DTO de saída (*OperationReportResponseDTO*) é centralizado em *DTOMapperService*, garantindo consistência e clareza no formato retornado, além de separar responsabilidades de transformação de dados.

O design dessa US foi estruturado sobre os padrões **Template Method** e **Strategy**, que organizaram e tornaram extensível o processo de busca e mapeamento das operações – serão detalhados na seção exclusiva sobre os padrões utilizados na etapa vigente.

Acerca dos testes de integração, foram desenvolvidos na classe *ReportControllerTests*, cobrindo cenários de sucesso e falha, no fluxo ponta-a-ponta (e2e). Os testes validam a aplicação dos filtros, a resposta correta para os tipos de operações de compras e/ou resgates, além de casos de borda como clientes inexistentes, intervalos de datas e resultados vazios.

Por fim, sobre os testes de unidade, na classe *ReportServiceUnitTests*, as dependências foram mockadas para isolar a lógica de negócio. Os testes verificaram a correta propagação dos filtros para os repositórios, a captura e envio dos valores ao mapper, bem como os métodos

auxiliares de datas, clientes e carteiras. Cobrindo em sua totalidade, todas as linhas inseridas durante a implementação da US.

US20 (Valdemar, João Pedro, Walber e Rafael)

Participação Combinada: 5%

Na US20, foi requerida a operação de exportação de um extrato completo das transações, compras e saques, de um cliente em formato CSV. Assim, foi necessário criar uma nova *Controller e Service,* chamadas *TransactionController e TransactionService,* pois *ClienteService* não poderia implementar as regras de negócio, a fim de evitar dependência circular, e nem seria coeso definir o endpoint para a operação descrita acima em *PurchaseService* ou *WithdrawService,* haja visto que a operação envolve tanto compras quanto saques. Em *TransactionController,* defini um endpoint que recebe um *Path Param,* o qual indica o id do cliente, e um *Body Param,* que contém o *accessCode* do cliente exigido para validar a operação.

O método *exportClientTransactionsCSV* em *TransactionService* é o responsável pela implementação da lógica de negócio, a qual acontece da seguinte forma: utilizo os métodos *getPurchaseHistory* e *getWithdrawHistory,* implementados para cumprimento da US18, com o objetivo de armazenar os históricos de compras e saques, respectivamente. Com a finalidade de manter ambos sobre o mesmo 'contrato', mapeei eles para um dto, chamado *TransactionExportDTO*, que possui os atributos necessários para atender as colunas desejadas no extrato. Dessa forma, com compra e saque sob o mesmo tipo, foi possível unificar os dois históricos em uma única lista de transações. A partir dessa lista, implementei o método *buildCSV*, responsável por montar uma string no formato CSV. Para isso, utilizei um *StringBuilder*, adicionando primeiro o cabeçalho com os nomes das colunas (type, asset, quantity, total-value, tax, date, state) e, em seguida, as transações. O conteúdo final foi convertido para byte[], que é retornado pelo endpoint.

Por fim, realizei os testes de integração para garantir o bom funcionamento do endpoint. Foram testados os casos inválidos básicos, como *accessCode* inválido ou cliente inexistente, casos limites, como quando o cliente ainda não fez nenhuma transação, e os casos tradicionais, em que há compra(s) e saque(s) ou somente saque(s).

Principais Decisões de Design

Padrão Template Method e Strategy (US19)

O **Strategy** foi aplicado por meio da lista de fetchers (*OperationFetcher*). Cada implementação (*PurchaseFetcher*, *WithdrawFetcher*) representa uma estratégia diferente para buscar e mapear operações. O *ReportServiceImpl* não precisa conhecer os detalhes internos de cada operação: ele apenas itera sobre os fetchers disponíveis, delegando a responsabilidade para a estratégia correta. Isso promove extensibilidade — isto é, novos tipos de operações podem ser adicionadas apenas criando outro fetcher que implemente a mesma interface, sem alterar o código já existente.

O **Template Method** foi aplicado na classe abstrata *OperationFetcherTemplate<T>*. Ela define a estrutura de como uma operação deve ser buscada e processada (ordem das etapas), mas delega a implementação de pontos específicos às subclasses (*PurchaseFetcher*, *WithdrawFetcher*), seguindo como o padrão comportamental sugere.

Métodos como *dateOf()*, *walletIdOf()*, *resolveWalletId()* e *walletIndex* fornecem um esqueleto reutilizável, enquanto métodos abstratos como *findItems()* e *map()* permitem customização conforme o tipo de operação. Isso garante reuso de código e consistência, evitando duplicação de lógica entre fetchers e facilitando a manutenção.

Diagrama: Link