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 1

US01 - Caio
US02 - Walber
US03 - Valdemar
US04 - João Pedro
US05 - Rafael

Finalização Etapa 1: 27/07/2025

Descrição das Atividades

US01 (Caio)
Participação Total: 20%

A US01, como uma das primeiras *User Stories* a serem desenvolvidas no projeto, demanda as operações básicas de criar, editar e remover ativos de investimento no sistema – um CRUD simples (*Create, Read, Update* e *Delete*). Contudo, somente o administrador do sistema é permitido a realizar essas ações, levando em conta, através da especificação do projeto, que o administrador é um usuário único com os seus devidos privilégios. Para isso, foi criado no diretório */model/asset*, a entidade *AssetModel* referente ao ativo do sistema, contendo como atributos: nome, tipo de ativo (*AssetType*) – Crypto (Criptomoeda), Treasury Bounds (Tesouro direto) e Stock (Ação), descrição, status (disponível ou indisponível) e o seu valor de cotação, atrelada a sua quantidade de cota.

Seguindo o padrão de camadas, foi criada a camada controladora (*AssetController*), de serviço (*AssetService*) e de repositório (*AssetRepository*), respeitando a comunicação entre elas. Para os DTOs (*Data Transfer Object*), foram criados alguns para a US01 e demais que utilizam do ativo em suas funcionalidades. Com relação a US01, os DTOs foram: *AssetPostRequestDTO* - responsável por receber os dados do lado cliente para a criação de um ativo, *AssetDeleteRequestDTO* - com as informações do admin no corpo da requisição (como relatado na US02), *AssetResponseDTO* - dados para a resposta do servidor ao cliente, e por fim, *AssetTypeResponseDTO* - com os campos correspondentes ao tipo de ativo, conforme design escolhido e discutido no tópico “Principais Decisões de Design”.

Na camada de serviço do ativo (*AssetServiceImpl*) é realizado todas as operações acerca de um ativo, como o seu CRUD, além de outros métodos descritos nas próximas USs. É importante ressaltar, as verificações das entradas para a manipulação e/ou criação de um ativo, que se mal tratadas, podem gerar exceções e erros em tempo de execução. Para isso, foram criadas algumas exceções customizadas para o uso na camada de serviço do ativo, e são detalhadas na US02 e US03.

Acerca dos testes, seguindo o padrão de camadas, foram realizados testes de integração com o controlador de ativos (*AssetControllerTest*), simulando com o uso de Mocks o ambiente web, e abordando casos de testes para requisições com entrada válidas e inválidas, seja de valores incorretos (nulos ou em branco), como também, de admin autorizado ou não autorizado. Por fim, os testes de unidade, tratando dos métodos responsáveis pela camada de serviço (*AssetServiceUnitTests*), garantindo uma maior cobertura dos testes, sendo boa parte, os mesmos realizados na camada de integração, mas agora, no lado do *service* e *repository*.

No tópico de “Principais Decisões de Design”, será explicado a tomada de decisão acerca da modelagem do tipo dos ativos, e a sua persistência no banco de dados através da JPA.

US02 (Walber)

Participação Total: 20%

Para a US02, foi implementada uma funcionalidade que permite ao administrador do sistema ativar ou desativar ativos previamente cadastrados, por meio do *endpoint PATCH assets/{idAsset}/activation*. Com intuito de garantir segurança, o administrador deve enviar, além do novo valor para *isActive*, o seu *email* e *access code* no corpo da requisição (*AssetActivationPatchRequestDTO*), que são autenticados por um serviço dedicado (*AdminService).* Esse serviço instancia um único objeto *AdminModel* com credenciais padrão e delega a validação de acesso por meio do método polimórfico *validateAccess*, definido na superclasse *UserModel*. A lógica de validação utiliza os modelos *EmailModel* e *AccessCodeModel* como *value objects* auxiliares.

A hierarquia de usuários é modelada com uma superclasse abstrata *UserModel* que declara o método polimórfico *validateAccess*, como mencionado anteriormente. No caso do administrador, a validação compara *email* e *access code*; para o cliente, a validação é feita apenas pelo código de acesso e ele é recuperado do *repository* com base no *ID*.

Ao ativar ou desativar um ativo, sua propriedade *isActive* é alterada. Ativos desativados não aparecem no *endpoint* de ativos disponíveis (*GET assets/available*), evitando que sejam comprados. Porém, a fim de garantir rastreabilidade, esses ativos permanecem visíveis no histórico de compras dos clientes.

Embora a funcionalidade de compra ainda não esteja implementada, foi criado um modelo de domínio para suportar o requisito de histórico informado na descrição da *US* (“Ativos desativados não devem estar disponíveis para compra, mas ainda podem ser exibidos no histórico dos clientes"). Os principais componentes desse modelo feito para suportar o histórico, no momento, são:

- *WalletModel*: representa a carteira do cliente e mantém um conjunto de transações(*TransactionModel*);
- *TransactionModel* é uma entidade abstrata que referencia um ativo, quantidade, data e carteira;
- *PurchaseModel* herda de *TransactionModel* e adiciona o estado da compra (*PurchaseStateEnum*).

Para fins de demonstração e testes, são inseridos valores padrão no banco via *import.sql*, permitindo o retorno do histórico de compras pelo *endpoint GET* clients/{id}/purchases. Isso simula clientes que possuem compras associadas, inclusive de ativos que podem estar desativados.

Foram criados testes do *services* de *controllers* referentes a esta *US*. Os testes incluem ativação e desativação de ativos com sucesso, validação de campos obrigatórios no *DTO*, autenticação do administrador com falhas para *email* ou código de acesso inválidos, resposta adequada para ativos não encontrados, garantia de que o endpoint de ativos disponíveis retorna apenas ativos disponíveis, cliente inexistente, cliente sem compras (histórico vazio), clientes com compras.

Por fim, exceções específicas foram definidas e utilizadas para garantir uma resposta de erro mais legível. Seguem as exceções utilizadas:

- *UnauthorizedUserAccessException*: usada para sinalizar que o acesso a recursos do servidor foi negado a um usuário (cliente ou admin);
- *AssetReferencedInPurchaseException*: impede a exclusão de ativos referenciados em compras de modo a preservar a integridade do histórico;
- *ClientIdNotFoundException*: lançada quando um cliente buscado por *ID* não é encontrado;
- *AssetNotFoundException*: sinaliza que um *asset* buscado por *ID* não foi encontrado.

Todas essas exceções possuem uma forma default com mensagens mais genéricas quanto uma forma que podem receber mensagens específicas para facilitar a identificação do erro. Essas mensagens específicas são introduzidas como parâmetro no ponto do código em que são geradas. Por exemplo, quando um *Asset* não é encontrado com a partir do seu ID, criamos a seguinte exceção em *AssetServiceImpl: new AssetNotFoundException*(“*Asset not found with ID ” + idAsset*).

US03 (Valdemar)
Participação Total: 20%

A US03 demandou a criação de um endpoint updateQuotation (/asset/{idAsset}/quotation), o qual recebe o idAsset e um DTO exclusivo, AssetQuotationUpdateDTO, o qual continha o valor novo de quotation, o e-mail e o access code do admin, dado que é ele quem tem a permissão para fazer tal alteração.

Houve a necessidade de criar duas exceções personalizadas para dar mais robustez e clareza às exceções própria do endpoint, como a InvalidAssetTypeException que é lançada ao tentar ser feito um update em um Asset que não seja Stock ou Crypto, além da InvalidQuotationVariationException que ocorre quando é tentado atualizar a quotation com uma variação menor que 1%. Representando bem as restrições da US e garantindo confiabilidade para o endpoint. Ademais, dois handlers também foram criados para gerenciar as exceções. A exceção proveniente da tentativa falha de login ao colocar as credenciais incorretas do admin já está embutida na US02.

O uso de código limpo também foi observado, por exemplo, ao atribuir uma constante da variação mínima de 1% para MIN_QUOTATION_VARIATION.

Os testes de unidade realizados na classe AssetServiceUnitTests verificam o comportamento do updateQuotation da AssetServiceImpl, assegurando que a atualização da cotação de um ativo ocorra corretamente quando as condições esperadas são atendidas. Inicialmente, é testado o cenário

de sucesso em que a cotação é atualizada com sucesso para um novo valor válido, respeitando a variação mínima exigida, e garantindo que o repositório de ativos persiste a alteração. Também são testadas as variações mínimas aceitáveis de cotação, tanto positiva (exatamente 1% acima do valor atual) quanto negativa (1% abaixo), confirmando que o sistema permite a atualização nesses limites.

Além disso, os testes validam o lançamento certo de exceções quando o ativo a ser atualizado não é encontrado, quando a variação de cotação não atinge o mínimo necessário, e quando o tipo do ativo é inválido. Ademais, foram testados os casos em que o administrador que realiza a operação não está autorizado, garantindo que nesses casos a exceção apropriada é lançada e nenhuma persistência da alteração é realizada.

Os testes implementados em AssetControllerTests verificam a integração. Os testes de sucesso asseguram que a cotação é atualizada corretamente para ativos dos tipos permitidos, Stock e Crypto, e que os demais atributos do ativo permanecem inalterados após a operação. Também é validado que a atualização pode ocorrer mesmo com uma variação negativa, desde que respeite o limite mínimo de 1%.

Nos testes de falha, são contempladas as principais regras de negócio e validações esperadas pela aplicação. Isso inclui rejeitar variações de cotação abaixo do mínimo exigido (1%), garantir que ativos do tipo Treasury_Bounds não possam ser atualizados, tratar corretamente ativos inexistentes, impedir alterações com JSON malformado, cotação nula, corpo vazio e credenciais inválidas de administrador. Ademais, também foi verificado que o endpoint responde adequadamente quando as credenciais do administrador são válidas, cobrindo o fluxo completo de autenticação.

US04 (João Pedro)

Participação Total: 20%

A US04 demandava as operações de criação, leitura, edição e remoção de clientes com certas especificidades, as quais irei abordar a seguir. Primeiramente, no sistema há 2 tipos de usuários, admin e clientes, sendo assim, foi criado uma classe abstrata *UserModel* onde foram declarados os atributos que Admin e Clientes compartilham *(id, fullName, email e accessCode)*. Logo, *AdminModel* e *ClientModel* estendem *UserModel*, o que promove reuso de código e maior coesão. Em *ClientModel* foram declarados os atributos que pertencem apenas a ele, sendo eles *address, planType, budget e wallet*. Declaramos o atributo *planType* como tipo de um enum também chamado *PlanType*, o que evita valores inválidos e garante maior legibilidade. Para os atributos *address, accessCode e email* foram criadas classes para fazer validação dos valores de forma isolada, promovendo encapsulamento.

Com *ClientModel* declarado, agora partimos para abordar a *ClientController* com endpoint base *'clients'* e *ClientServiceImpl*. Inicialmente, foi criado o método POST a qual recebe um parâmetro por meio do body da requisição do tipo ClientPostRequestDTO, o qual deve conter todos valores personalizados do cliente e ainda faz a validação desses valores com anotações, como por exemplo *@NotBlank.* Para a operação de leitura de clientes, foram criados 2 métodos GET, sendo um que não recebe nenhum parâmetro que retorna todos os clientes e outro método que recebe um parâmetro na url, que corresponde ao ID de um cliente específico, que retorna um determinado cliente. A atualização de cliente só deve ser possível para o atributo *fullName,* sendo assim, foi optado pelo método PATCH para evitar sobrescrita de dados não enviados. Esse método recebe um parâmetro do body da requisição do tipo *ClientPatchFullNameDTO*, o qual possui apenas os atributos *fullName* e *accessCode,* sendo o primeiro o valor o que será atualizado no banco e o segundo para cumprir a especificação da US que exige a validação do *accessCode* do cliente pra operação de atualização, logo, o método na service, chamado pelo PATCH, usa o método *validateAccess* de *ClientModel* para averiguar se o *accessCode* passado na requisição é igual ao do cliente que

pretende ser atualizado. Se igual, a operação de atualização é realizada com sucesso, caso contrário, uma exceção é lançada. Por último, foi implementado o método DELETE que recebe um parâmetro na url correspondente ao id do cliente que deseja ser deletado e um outro parâmetro no body do tipo *ClientDeleteRequestDTO,* o qual tem apenas o atributo *accessCode* para realizar a mesma validação feita no método PATCH. Os métodos PATCH, POST e GET retornam um objeto do tipo ClientResponseDTO, o qual retorna todos os atributos de *ClientModel*, com exceção do *accessCode,* o que é exigido pela especificação da US. Foram criadas as classes ClientControllerTests e ClientServiceUnitTests onde foram realizados diversos testes sobre as operações abordadas acima.

US05 (Rafael)
Participação Total: 20%

A US05 demandou a criação de um endpoint *getActiveAssets* (/client/{id}/assets) que recebe como parâmetro o ID de um cliente e um DTO exclusivo contendo o access code do próprio cliente para autenticação, visto que é necessário acessar dados do cliente referente ao tipo de plano que ele possui para devolver uma lista de ativos disponíveis para compra.

Durante o processo de desenvolvimento houve dúvidas em relação a implementação dessa funcionalidade sem violar padrões de responsabilidade e encapsulamento e após discutir com colegas do grupo e fazer pesquisas chegou-se à conclusão de que a melhor maneira seria fazendo os *services* comunicarem entre si em demanda. Dessa forma, o endpoint em ClientController chama o metodo *redirectGetActiveAssets* em *ClientService*, que por sua vez faz verificações acerca do id e access code do cliente (responsabilidade do *ClientService*) e em seguida redireciona para AssetService com base no tipo de plano que o cliente possui e devolve uma lista de ativos filtrados, assim como é requisitado na US05: Clientes do plano Normal visualizam apenas Tesouro Direto; clientes do plano Premium visualizar todos os tipos.

O código feito não viola nenhum padrão de design e é bem simples e direto. Dito isso, os testes implementados em *ClientControllerTests* testam apenas se para entradas inválidas o método devolve exceções e retorna os status do HTTPS corretos, foi uma decisão de design não incluir os testes da saída com assets pois exigiria ser instanciado o *AssetRepository*, *AssetServiceImpl* e realizar o *mock* utilizando *mockito*. Portanto, os testes foram incluídos em *AssetServiceUnitTests*, já que essa unidade de teste é apropriada para testes que necessitam desse repositório e serviço.

Nos testes realizados por mim da US05 é contemplado a requisição dos ativos disponíveis com id de client invalido, id de client nulo, access code invalido, access code nulo, retorno de ativos disponíveis (para cliente Premium) e o retorno de ativos disponíveis por tipo de ativo (para cliente Normal, embora vá ser sempre restringido ao *Treasury_Bounds*).

Principais Decisões de Design

Para promover flexibilidade e separação de responsabilidades optamos por utilizar interfaces para acessar os serviços da aplicação. Essa abordagem permite uma maior organização do código, facilita a aplicação de testes unitários e possibilita a substituição ou evolução das implementações concretas com menor impacto nas demais camadas da aplicação.

Além disso, implementamos exceções personalizadas para os principais cenários de erro, visando tornar as mensagens de resposta mais claras e legíveis. Essa decisão melhora a experiência tanto para os desenvolvedores quanto para os consumidores da API, já que facilita o diagnóstico de falhas e mantém um padrão mais coeso nas respostas de erro.

Agora, iremos discutir acerca da escolha de design para os tipos de ativos (*Asset Types*) que, por problemas de como a JPA (*Java Persistence API*) persiste os valores dos atributos para cada entidade do modelo, no banco de dados, obrigando a um tipo ser de classe abstrata ou concreta pelo uso da notação *@Entity*, forçou a mudança na própria modelagem, que antes seria uma interface – seguindo o princípio de OO de programar para uma interface e não para uma implementação – para uma classe abstrata. Tal classe, contém um método abstrato para as subclasses (antes que implementaria a interface) sobrescreverem do modo particular das mesmas – método esse será discutido na US15, sobre o cálculo do imposto, para os diferentes tipos de ativos. A JPA, então, marca essa classe como uma entidade, e utiliza a estratégia de herança *SINGLE_TABLE*, com uma coluna discriminadora, indicando o tipo daquele ativo, no formato de *string*. Na classe *AssetModel*, será armazenado um atributo do tipo *AssetType*, como junção de coluna, e com relacionamento *ManyToOne*, uma vez que, vários ativos diferentes podem estar associados a um mesmo tipo de ativo. Por fim, cada classe concreta do tipo de ativo, é uma entidade, e por estender da superclasse (*AssetType*), será mapeada para tabela de tipos de ativos, devido a estratégia de herança em tabela única – todas as subclasses vão ser armazenadas na mesma tabela.

Até o momento, discutimos o design escolhido no lado do modelo, ou do servidor propriamente dito, precisamos nos atentar ao lado do cliente, em como o mesmo envia, no corpo da requisição, o tipo do ativo a ser criado. Para isso, depois de algumas discussões, foi acordado a criação de um componente para o tipo de ativos, com um controlador, serviço e repositório. Com isso, o admin, na criação de um ativo, consegue listar todos os tipos de ativos registrados no sistema – inseridos no banco de dados, após a inicialização da aplicação, através do arquivo *import.sql* – e na requisição, é passado em *string* o nome referente ao tipo de ativo, e que será mapeado para o valor relativo do enum *AssetTypeEnum*. Por conta da utilização de um Enum, todo o trabalho de buscar no repositório dos tipos de ativo, e armazenar no atributo *assetType* de tipo *AssetType* da entidade *AssetModel*, fica mais fácil e simples de testar. Deste modo, as classes referentes aos DTOs não sabem da existência do tipo *AssetType* e, somente trabalham com o enum referente aos tipos de ativos existentes na aplicação, por exemplo, o DTO *AssetTypeResponseDTO* só retorna o id e nome do tipo de ativo, e que pode ser usado em outro DTO, como no *AssetResponseDTO*, referente ao campo *assetType*.

Diagrama: [Link]