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 2

US06 - João Pedro e Rafael Barreto
US07 - Walber Wesley e Caio Cesar
US08 - Valdemar

Finalização Etapa 2: 10/08/2025

Descrição das Atividades

US06 (João e Rafael)

Participação Total: 33,3%

Implementação da lógica - João

Antes de iniciar a implementação da US06, verifiquei que na US07 mesmo quando um asset já estava disponível ou um access code inválido era passado pelo cliente, o marco de interesse ainda era criado. Para corrigir isso, criei métodos privados para fazer essa checagem e lançar exceções personalizadas quando esses casos ocorrerem.

Tratando agora da implementação da US06, Walber implementou o padrão Observer, o qual possibilitou bastante reuso de código implementado para US07, haja visto que ambas User Stories se configuram como inscrições em determinado evento e disparo de uma notificação quando o evento ocorrer. Sendo assim, ficou de minha responsabilidade a implementação da lógica de restrições para se marcar interesse na variação de preço de um ativo e a verificação de mudança significativa no valor para notificar os clientes inscritos no evento. Primeiramente, a US06 diz que essa operação só está disponível para clientes Premium, ativos disponíveis e do tipo ação ou criptomoeda. Então, criei métodos privados para cada uma dessas validações, tornando o código mais legível, que lançam exceções personalizadas *(ClientIsNotPremium, AssetIsInactive e AssetIsNotStockNeitherCrypto*), caso a restrição associada não seja cumprida. Além da validação típica de access code para operações do cliente. No método responsável por alterar o valor de um ativo, *updateQuotation*, fiz uma verificação para saber se a mudança de valor é igual ou maior que

10%, caso sim, há a chamada para o método *notifySubscribersByType* de *EventManager* que realiza a notificação aos clientes que marcaram interesse na mudança de preço desse ativo.

Implementação dos testes - Rafael

Acerca dos testes de unidade da US06 - fiquei responsável por testar os métodos da nova feature implementada responsável por marcar interesse na variação de preço, porém após discutir com Caio decidimos dividir uma parte dos testes da US06 e US07 um para o outro já que o sistema de inscrição é intrinsecamente conectado em ambas as user stories. Dessa forma, adicionei teste no *ClientControllerTests* referentes a ambos os endpoints (*markInterestInPriceVariationOfAsset & markInterestInAvailabilityOfAsset*) e criei os testes iniciais do *EventManagerUnitTests* referentes ao *subscribeToAssetEvent.* Além disso, foram ajustados às chamadas dos métodos mencionados acima no *ClientController* e melhorados/adicionados testes de User Stories passadas para serem menos redundantes, incluir casos extremos e aumentar significativamente o coverage do projeto.

US07 (Walber e Caio)

Participação Total: 33,3%

Implementação da lógica - Walber

Na aplicação, foi implementada a funcionalidade que permite a um cliente marcar interesse em um ativo indisponível específico — seja ele Tesouro Direto, ação ou criptomoeda — para receber uma notificação assim que o ativo voltar a ficar disponível para compra.

O fluxo inicia no endpoint *PATCH clients/{clientId}/interest/availability*, que recebe o clientId e um *ClientMarkInterestInAssetRequestDTO* contendo o identificador do ativo e o *accessCode* do cliente para validação. O *controller* delega a operação para o *ClientService*, que direciona a requisição ao *AssetService*.

No *AssetService*, o método *subscribeToAsset* obtém a instância do ativo e invoca o método subscribe no próprio *AssetModel*. Esse método garante que o EventManager esteja configurado — caso contrário, lança a exceção personalizada *EventManagerNotSetException* — e valida o ativo antes de registrar a assinatura por meio de *eventManager.subscribeToAssetEvent*(...).

O *EventManager* verifica se o cliente já está inscrito para aquele ativo e tipo de notificação. Caso já esteja inscrito, ele lança uma exceção personalizada *AlreadySubscribedException* a qual indica que o cliente já possui essa assinatura. Se não tiver inscrito, o *EventManagerI* persiste uma nova assinatura no *repository*. Dessa forma, garante-se que cada cliente só receba a notificação uma única vez quando o ativo voltar a estar disponível.

Quando um ativo tem seu *status* alterado via *changeActiveStatus*, o *AssetModel* detecta a transição de inativo para ativo e, caso haja um *EventManager* configurado, dispara o método *notifySubscribersByType* para todos os assinantes do tipo *AVAILABILITY* neste caso.

O *EventManager* então recupera todas as assinaturas relacionadas, monta uma mensagem de contexto indicando o motivo da notificação e chama o método *notify* de cada assinante (clientes

que implementam *ISubscriber*). Após o envio, a assinatura é removida do *repository* de modo a evitar notificações duplicadas para o mesmo evento de disponibilidade.

A notificação é representada de forma simples e direta: uma mensagem impressa no terminal, contendo o motivo e o destinatário. Além de receber uma cor diferente para garantir destaque das outras mensagens enviadas pelo *Spring* + *JPA*.

Com isso, o sistema garante que clientes interessados em ativos indisponíveis sejam informados imediatamente quando eles se tornarem disponíveis.

Implementação dos testes - Caio

Acerca dos testes de unidade da US07 – os testes de integração transcorreram por outra parte do grupo – foram adicionados nas classes *AssetServiceUnitTests*, testes para o método *updateQuotation()* e *setIsActive()* da camada de serviços de ativos, que juntos, invocam métodos presentes em *AssetModel*, como *updateQuotation()* e *changeActiveStatus()*. Tais testes, focam nas notificações que deveriam, ou não, serem enviadas para os clientes, partindo da validação das regras de negócios, assim como, verificações de dados inválidos ou incorretos, enviados como argumentos para esses métodos. Para a classe *EventManagerUnitTests*, o foco estava na garantia dos inscritos serem notificados, corretamente, conforme o seu tipo de assinatura (*PRICE_VARIATION* e *AVAILABILITY*), verificando também dados incorretos ou inválidos como entradas.

US08 (Valdemar)

Participação Total: 33,3%

Foi realizada uma refatoração importante na forma como o acesso aos ativos é tratado na aplicação. Anteriormente, existiam dois endpoints públicos voltados para o CRUD de Assets: um que listava todos os ativos (getAllAssets) e outro que retornava os detalhes de um ativo específico (getAssetById). No entanto, ambos foram removidos, visto que não há nenhum User Story (US) que preveja ou justifique o acesso direto a esses dados. Além disso, esse tipo de acesso expõe informações que, segundo o novo requisito apresentado nessa entrega, devem estar restritas ao cliente que detém o ativo.

Diante disso, foi implementado um novo fluxo de acesso que segue as restrições corretas e garante segurança e privacidade ao usuário. Um novo endpoint GET /clients/{clientId}/assets/{assetId} foi criado, exigindo no corpo da requisição um DTO chamado ClientAssetAccessRequestDTO. Esse DTO possui um único campo: o accessCode, que representa a autenticação do cliente. Tal estrutura reforça o princípio de que somente o cliente dono do ativo pode consultar seus detalhes e mantém a interdependência de outros DTOs.

Na camada de serviço (ClientServiceImpl), foi introduzido o método getAssetDetails, responsável por validar a existência do cliente, verificar se o accessCode fornecido é válido através do método validateAccess, e, somente após essa verificação, buscar os detalhes do ativo via assetService.getAssetById. Esse fluxo garante que qualquer tentativa de acesso com credenciais inválidas, ou mesmo a tentativa de acesso a um cliente inexistente, resulte nas exceções apropriadas, como ClientIdNotFoundException ou UnauthorizedUserAccessException, já tratadas na aplicação. O getAllAssets também foi excluído no AssetServiceImpl.

Além disso, foram criados testes de unidade e integração, que cobrem todos os aspectos críticos desse novo fluxo. Nos cenários de sucesso, os testes asseguram que um cliente válido, ao fornecer o accessCode correto, recebe com sucesso os detalhes do ativo, incluindo nome, tipo, valor e outros campos relevantes. Já nos cenários de falha, foram contempladas situações como o fornecimento de um código de acesso incorreto, um clientId inexistente ou um assetId inválido. Esses testes confirmam que o método responde corretamente com as exceções esperadas, e que em nenhuma dessas situações a requisição prossegue até o ponto de retornar os dados sensíveis do ativo.

Esse conjunto de mudanças reforça a segurança da aplicação, limita corretamente o acesso de acordo com os requisitos de negócio e melhora a coesão da responsabilidade entre administrador e cliente no sistema. Com isso, o sistema se aproxima ainda mais da fidelidade exigida pelas User Stories propostas.

Principais Decisões de Design

Dentro das decisões de design adotadas para a evolução da aplicação, uma escolha relevante foi a remoção dos endpoints públicos getAllAssets e getAssetById, que permitiam ao administrador consultar a lista completa de ativos ou os detalhes de um ativo específico. Essa decisão está diretamente relacionada aos princípios centrais da arquitetura da aplicação, como responsabilidade única, segurança de acesso e adesão estrita às regras de negócio descritas nas User Stories.

Não havia, até o momento, qualquer US que justificasse ou exigisse que o administrador pudesse visualizar diretamente os ativos registrados na aplicação. A presença desses endpoints, portanto, representava uma funcionalidade órfã do ponto de vista de negócio, além de abrir brechas para o acesso irrestrito a informações sensíveis, o que vai contra os princípios de encapsulamento de dados e limitação de exposição da API.

Como resposta a essa análise, foi implementada uma nova abordagem: um endpoint exclusivo para o cliente acessar os detalhes de seus próprios ativos, mediante autenticação. O novo GET /clients/{clientId}/assets/{assetId} exige um corpo com um DTO específico (ClientAssetAccessRequestDTO), contendo o accessCode do cliente. Isso fortalece a autenticidade da requisição e garante que apenas o proprietário do ativo, e mais ninguém, tenha acesso aos seus dados. Essa decisão reforça o princípio da responsabilidade: o cliente é o único responsável por visualizar suas informações, e o backend garante que essa lógica seja intransponível.

No contexto das principais decisões de design, essa escolha complementa a estratégia já adotada de separação clara de responsabilidades entre administradores e clientes, além de estar alinhada com o uso de interfaces e abstrações no serviço, facilitando testes e evoluções. O cliente agora interage diretamente com o serviço de ativos por meio do ClientService, que atua como intermediário para validar o acesso antes de delegar a busca ao AssetService, mantendo assim uma camada de controle específica para o domínio do cliente.

Essa mudança também valoriza a aplicação de exceções personalizadas como forma de tratar tentativas de acesso indevido. A ausência de endpoints de listagem aberta e a exigência de um accessCode tornam o sistema mais seguro, além de mais fiel ao comportamento esperado pelo negócio.

Por fim, a decisão reforça a coesão da arquitetura ao garantir que cada tipo de usuário (admin ou cliente) só possa executar operações estritamente autorizadas por sua função. Isso contribui para uma aplicação mais robusta, auditável e alinhada aos princípios de mínima exposição, segurança orientada ao domínio e coerência funcional com os requisitos especificados.

Implementação do Padrão Observer

Para a entrega do projeto, optamos por implementar o padrão *Observer* de forma a permitir que clientes interessados sejam notificados automaticamente sobre mudanças específicas em ativos (*AssetModel*), sem que seja necessário acoplar a lógica de notificação diretamente às regras de negócio de atualização de dados.

A classe *AssetModel* atua como *Publisher*, sendo responsável por gerenciar os eventos que podem ocorrer sobre um ativo, como variação de preço ou mudança de disponibilidade. No entanto, para manter a responsabilidade bem delimitada, ela não notifica diretamente os clientes. Em vez disso, delega essa tarefa a um componente especializado, o *EventManager*, que centraliza a lógica de inscrição e notificação.

A comunicação entre *AssetModel* e *EventManager* é feita através de uma interface (*EventManager*). Isso garante baixo acoplamento e facilita a substituição ou evolução da lógica de notificação no futuro, além de permitir a aplicação de injeção de dependência no contexto do *Spring*. A implementação concreta (*EventManagerImpl*) é responsável por:

- Registrar assinaturas no banco de dados via *SubscriptionRepository;*
- Notificar os inscritos sempre que o *Asset* dispara um evento;
- Remover inscrições de uso único.

Os observadores (clientes) implementam a interface *ISubscriber*, que define um único método *notify(String context).* Essa abstração garante que qualquer entidade capaz de receber notificações possa ser integrada ao facilmente ao sistema, sem depender de uma implementação específica.

O fluxo funciona assim:

1. O cliente se inscreve em um evento (via *controller -> service -> AssetModel.subscribe*);
2. O AssetModel chama o EventManager para registrar a inscrição;
3. Quando o ativo sofre uma alteração relevante (*upadateQuotation* ou *changeActiveStatus*), o *AssetModel* avalia se as condições para notificação foram atendidas (variação da quotation >= 10% ou mudança de disponibilidade – inativo para ativo);
4. O *AssetModel* delega ao *EventManager* a notificação dos observadores correspondentes;
5. O *EventManager* recupera os inscritos do banco, instancia cada *ISubscriber* válido e chama o método *notify.*

Essa arquitetura apresenta três vantagens principais:

- Baixo acoplamento: *AssetModel* não precisa conhecer os detalhes de como as notificações são enviadas;
- Extensibilidade: é simples adicionar novos tipos de eventos ou mudar a forma de envio de notificações sem alterar o código principal do ativo;
- Persistência de inscrições: assinaturas são mantidas no banco, permitindo que o sistema continue notificando mesmo após reinicializações.

Modelagem UML

Acerca da modelagem UML, houve modificações comparadas a da primeira entrega, como a classe *Client* não possuir agregação com a classe *Asset*, isto se dá ao fato que, a implementação do padrão Observer – mencionado acima – implica em que o cliente não armazene os ativos que ele está interessado. Se fosse preciso isso, teríamos que navegar por todos os clientes e por todos os *interestedAssets* (e *waitingAssetAvailable*) de cada cliente e verificar se tem algum ativo que acabou de receber essa alteração, por exemplo. A lógica, com o Observer, é mais simples e direta, isto é, já recuperamos do banco os Ids dos inscritos para determinado tipo de inscrição para cada *assetId* e notificamos essa lista recuperada diretamente, isso é controlado pelo *EventManager*.

Com isso, pensando nas futuras USs, correspondentes às classes *Transaction*, *Wallet* e demais que estão no código como *placeholders*, é interessante – e lógico – a agregação de *Asset* com a classe *Transaction*, que possui um relacionamento de composição com *Wallet*. Assim, removendo a lógica de ativo agregado ao cliente, separando-os das suas devidas responsabilidades.

Por fim, a principal mudança foi na inserção da lógica envolvendo o padrão comportamental Observer – ou Listener. O ativo é agregado a uma interface *EventManager*, que por sua vez, é implementada pela *EventManagerImpl*, presentes na camada de serviços da nossa aplicação. O cliente implementa a interface *ISubscriber* e sobrescreve o método *notify()*, utilizado nas notificações e mensagens exibidas no terminal. Com esse contrato e herança de tipo, o *EventManagerImpl* conversa com os inscritos unicamente por meio dessa interface, sem o conhecimento do cliente. Para finalizar, o *SubscriptionModel* é associado com *EventManagerImpl* – como mostra no diagrama, pela seta ->, referente a “quem ‘conhece’ quem”.

Diagrama: [Link]