Introdução

No cooperativismo, cada associado possui um voto e as decisões são tomadas em assembleias, por votação. Imagine que você deve criar uma solução para dispositivos móveis para gerenciar e participar dessas sessões de votação.

Essa solução deve ser executada na nuvem e promover as seguintes funcionalidades através de uma API REST:

- Cadastrar uma nova pauta
- Abrir uma sessão de votação em uma pauta (a sessão de votação deve ficar aberta por um tempo determinado na chamada de abertura ou 1 minuto por default)
- Receber votos dos associados em pautas (os votos são apenas 'Sim'/'Não'. Cada associado é identificado por um id único e pode votar apenas uma vez por pauta)
- Contabilizar os votos e dar o resultado da votação na pauta

Para fins de exercício, a segurança das interfaces pode ser abstraída e qualquer chamada para as interfaces pode ser considerada como autorizada. A solução deve ser construída em java, usando Spring-boot, mas os frameworks e bibliotecas são de livre escolha (desde que não infrinja direitos de uso).

É importante que as pautas e os votos sejam persistidos e que não sejam perdidos com o restart da aplicação.

O foco dessa avaliação é a comunicação entre o backend e o aplicativo mobile. Essa comunicação é feita através de mensagens no formato JSON, onde essas mensagens serão interpretadas pelo cliente para montar as telas onde o usuário vai interagir com o sistema. A aplicação cliente não faz parte da avaliação, apenas os componentes do servidor.

1. Arquitetura

Para a implementação da solução, usaremos uma arquitetura de Controller – Service – Repository. Por se tratar de uma aplicação mais simples, trata-se de uma arquitetura monolítica com segurança abstraída para fins de avaliação. A figura 1 ilustra a arquitetura proposta.

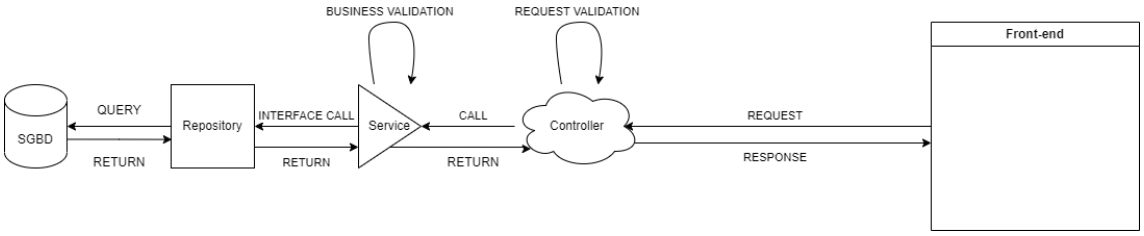

Figura 1 - Arquitetura do Sistema

2. Mapeamento de entidades

Considerando o problema dado, foi modelado um conjunto de entidades com os seguintes nomes:

- Agenda: Entidade que representa uma pauta
- VotingSession: Sessão de votos da pauta
- Associate: Associado que deve estar cadastrado na agenda para votar
- Vote: O voto em si.

Em cima dessas entidades, foi criado um relacionamento representado pelo diagrama UML (figura 2). Algumas observações:

- Inicialmente, a Agenda é criada apenas com nome e descrição. Isso se dá para que se possa cadastrar os associados posteriormente.
- Uma VotingSession possui uma data de início (startsAt) e uma data de fim (endsAt). A data de início é obrigatória. Se a data de fim não for submetida, será considerada como 1 minuto após a data de inicío.
- Ao criar uma VotingSession, ela deverá ser atribuída a alguma Agenda. A aplicação retornará exceções caso a Agenda não tenha associados ou se aquela Agenda já possua alguma VotingSession.
- A atribuição de Agenda à VotingSession só pode ser feita pelo lado da VotingSession.
- É possível editar a data de início ou a data de fim da VotingSession, desde que já não tenha começado ou finalizado.
- Se as datas enviadas para criação ou atualização da VotingSession estiverem no passado, uma exceção será retornada.

- Votos não poderão ser registrados caso o associado já tenha votado anteriormente, se não estiver registrado na pauta, se a sessão não tiver iniciado ou já tiver terminado.
- VotingSessions possui quatro getters para descrever o resultado.
    - getYesVotes, que contabiliza a quantidade de votos positivos.
    - getNoVotes, que contabiliza a quantidade de votos negativos.
    - getTotalVotes, que contabiliza a quantidade total de votos.
    - getResult, que indica o resultado da VotingSession.
- Ainda, o resultado da VotingSession é enumerado da seguinte forma:
    - NOT_STARTED: a sessão ainda não começou.
    - COUNTING: a sessão começou e a votação está acontecendo.
    - YES_VOTES: a sessão terminou e os votos positivos venceram.
    - NO_VOTES: a sessão terminou e os votos negativos venceram.
    - DRAW: a sessão terminou e houve empate.
- Só será informada a quantidade de votos quando o resultado for diferente de NOT_STARTED.

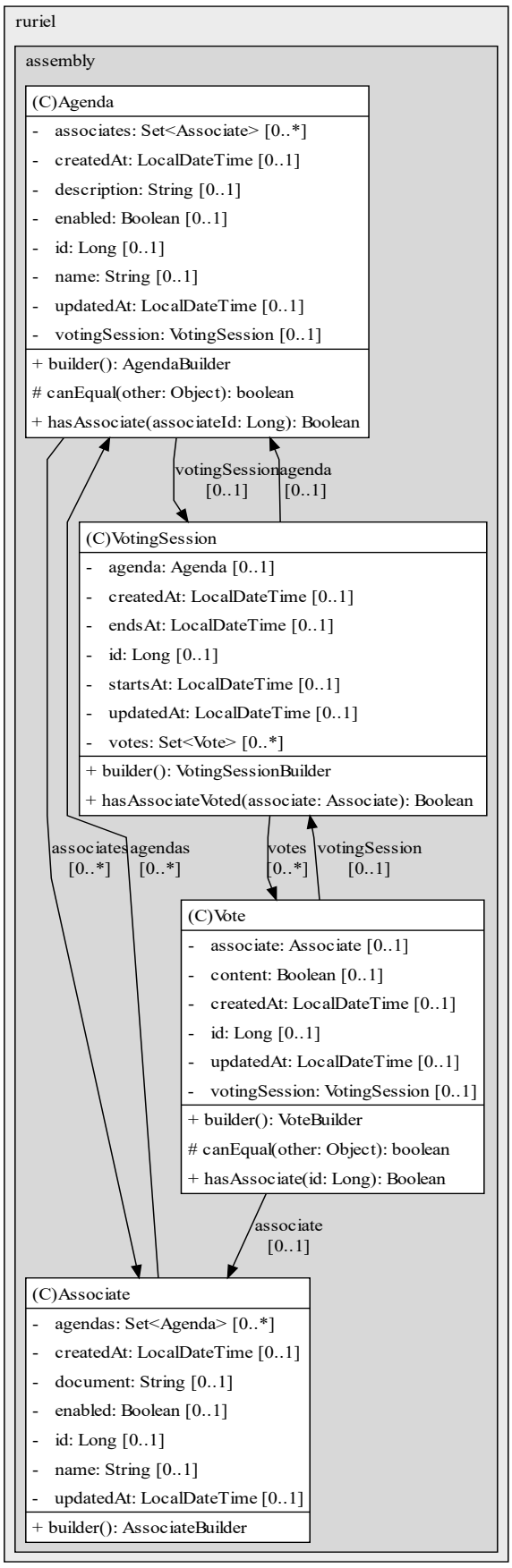

Figura 2 - Diagrama UML das entidades do projeto

3. Estrutura do diretório

O projeto está dividido em cinco pacotes. São eles:

- api: Pacote onde contém tudo relacionado a nossa API REST. Isso inclui controllers, recursos retornados e exceções. A parte de controllers e recursos ainda são organizadas de acordo com a versão da API.
- configuration: Pacote onde contém nossas classes de configuração.
    - Handlers de cada exceção jogada
    - Configuração do Jackson para formatar as datas corretamente,
    - Configuração do ModelMapper para mapeamento entre entidade e recurso.
    - Configuração do OpenApi para geração do Swagger.
- entities: As classes de entidades, detalhadas no capítulo anterior.
- repositories: Lista de interfaces que herdam de JpaRepository, utilizado para acessar o banco de dados.
- services: Classes que realizam a comunicação entre os controllers e os repositories.

Na figura 3 podemos observar a estrutura de diretórios do projeto. Outros diretórios não listados incluem a pasta resources, que armazena a configuração do banco para desenvolvimento e produção, e o pacote de testes, que realiza os testes unitários de cada service e controller.

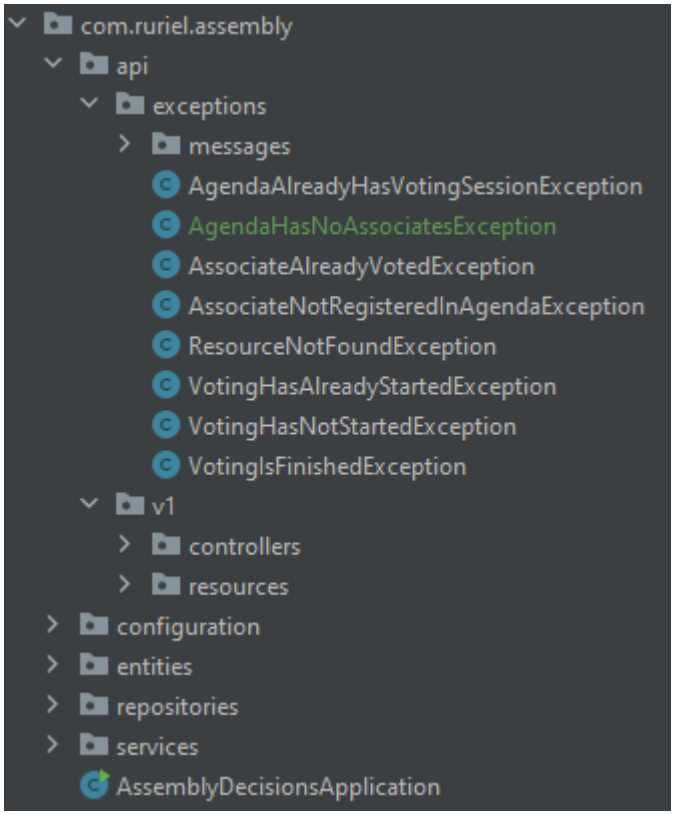

Figura 3 - Estrutura de diretórios do projeto

4. Controller

Entidades, serviços e repositórios são bem direto ao ponto. No entanto, é importante observar alguns pontos sobre a camada de controle da aplicação.

O versionamento da API é dado por Media Types customizados. Assim, caso houvesse a necessidade de criar uma segunda versão da API, bastaria produzir um Media Type diferente. Com isso, o front precisaria adicionar um cabeçalho Accept com Media Type desejado para garantir que vai receber a versão correta. O Media Type utilizado consiste no formato *application/vnd.assembly.api.vx+json*, onde x é o número da versão da API.

Para os corpos das requisições e respostas, foi utilizado o padrão DTO, localizado no pacote de resource. Com essa divisão, podemos eliminar o uso de campos da entidade que possam não ser interessantes para o endpoint em questão, evitar problemas de referências cíclicas e ainda validar os campos enviados na própria requisição. Foi utilizada a biblioteca ModelMapper para fazer a conversão de DTO para entidade e vice-versa. A API está documentada através do arquivo swagger.json. Também está disponível em /swagger-ui.html ao rodar a aplicação.

5. Sugestão de melhorias

Devido ao *deadline*, algumas partes do teste tiveram que ficar de fora. Serão listados nessa seção sugestões de melhorias que poderiam ser feitas:

- Comunicação com API de validação de CPF usando OpenFeign.
- Substituição do sfl4j pelo log4j2.
- Melhor tratamento da flag enabled na entidade de Associate, bem como tratamento dessa flag ao relacionar com a entidade de Agenda.
- Uso de alguma API como o CubeJS para trazer o resultado da tabela de votos em tempo real.
- Testes de integração.
- Testes unitários mais refinados nos controllers.