PROJETO 02 – TESTES AUTOMATIZADOS - PROF. MATIELO JOSE GERONIMO
ENTREGÁVEL DA SEMANA 02 (DUPLA OU INDIVIDUAL) - PRAZO: 16/09/2024 até 23h59 (SEGUNDA-FEIRA)

URNA VIRTUAL

1 CONTEXTO:

1.1 Desenvolver uma API RESTful para gerenciar VOTAÇÕES ON-LINE e implementar testes unitários e testes de integração variando os cenários para validar todas as regras de negócio descritas nos itens 2 e 3.

1.2 A complexidade nas regras de negócio é proposital, pois possibilitará a criação de cenários de teste mais completos e que fazem mais sentido em projetos reais do mercado de trabalho.

1.3 LEIAM com muita ATENÇÃO para não haver erro de interpretação.

1.4 O projeto mensal é de testes automatizados aplicados ao back-end. Então, a implementação incorreta das regras de negócio refletirá na avaliação. Estarei observando sempre o back-end + testes.

1.5 Esta atividade terá um peso maior na avaliação e será utilizada também no projeto mensal de front-end.

2 REGRAS DE NEGÓCIO SIMPLES: DADOS, RELACIONAMENTOS E VALIDATIONS (NAS ENTITIES):

2.1 O cadastro de um **ELEITOR** deverá conter nome completo (obrigatório), CPF válido (mas não obrigatório – pode ser nulo inicialmente), profissão (obrigatório), telefone celular válido (obrigatório), telefone fixo válido (mas não obrigatório), endereço de e-mail válido (mas não obrigatório – pode ser nulo inicialmente) e status (processado pelo sistema, conforme regras descritas no Item 3).

2.2 O cadastro de um **CANDIDATO** deverá conter nome completo (obrigatório), CPF válido (obrigatório), número do candidato (obrigatório e ÚNICO), função (campo obrigatório), status (processado pelo sistema, conforme regras descritas no Item 3) e votos apurados (transiente e calculado pelo sistema).

A função deverá ser identificada de forma numérica, sendo 1 para prefeito e 2 para vereador.

O campo de votos apurados é transiente, ou seja, não deverá ser persistido no banco, pois será apenas calculado. Portanto, deverá ter a annotation @Transient em cima. Isso evitará a criação da coluna no banco.

2.3 O registro de um **VOTO** deverá conter data e hora da votação (campo obrigatório e obtido pelo sistema), o candidato a prefeito escolhido (objeto obrigatório), o candidato a vereador escolhido (objeto obrigatório) e um hash que será o comprovante (campo texto gerado pelo sistema, conforme regras descritas no Item 3).

Um voto possui um único candidato a prefeito e um candidato a prefeito pode estar associado a vários votos.

Um voto possui um único candidato a vereador e um candidato a vereador pode estar associado a vários votos.

Atenção! Vereador e Prefeito não são entidades, são funções dos candidatos.

Em hipótese alguma o eleitor deverá ser identificado. Então, não deverá haver relacionamento entre eleitor e voto.

Não trataremos votos brancos e nulos. Então, os objetos dos candidatos são obrigatórios dentro do voto.

2.4 O sistema deverá ter uma classe de **APURAÇÃO** que não deverá ser persistida. Portanto, é uma classe que não deverá ter annotation @Entity e não terá tabela no banco de dados. Ela servirá apenas para representar um objeto com o resultado das eleições que será totalmente calculado.

Esta classe deverá conter o total de votos (inteiro), uma lista de objetos dos candidatos a prefeito e uma lista de objetos dos candidatos a vereador.

Como esta é uma classe que não persistirá, não coloquem annotation de relacionamento em cima das listas.

---

3 REGRAS DE NEGÓCIO MAIS COMPLEXAS: MÉTODOS (NOS SERVICES):

STATUS DO ELEITOR

3.1 O status não deverá ser enviado na requisição, mas sim processado pelo sistema.

3.2 Os status possíveis são: APTO, INATIVO, BLOQUEADO, PENDENTE e VOTOU.

3.3 INATIVO: Este status deverá ser atribuído quando houver requisição para deletar o eleitor. O sistema jamais deverá deletar de fato o eleitor, apenas mudar o status para INATIVO. Para isso, o método *delete*() do CRUD deverá ser adaptado e o método *findAll*() do CRUD deverá listar somente os eleitores ativos.

Se o usuário já votou, o eleitor não poderá ser inativado. Então, o status não deverá ser alterado e uma exceção deverá ser lançada = "*Usuário já votou. Não foi possível inativá-lo*".

3.4 PENDENTE: Este status deverá ser atribuído quando o eleitor a ser salvo ou atualizado estiver sem CPF ou sem endereço de e-mail cadastro. Para isso, os métodos *save*() e *update*() do CRUD deverão ser adaptados.

Em caso de atualização, se o eleitor já estiver INATIVO, o status de inativo deverá ser mantido, mesmo que haja pendência de cadastro e a persistência deverá seguir normalmente.

3.5 BLOQUEADO: Este status deverá ser atribuído quando um eleitor com status PENDENTE tentar votar.

O sistema deverá barrar o voto com a exceção "*Usuário com cadastro pendente tentou votar. O usuário será bloqueado!*" e o status do eleitor deverá ficar bloqueado.

3.6 VOTOU: Este status deverá ser atribuído quando um eleitor com status APTO concluir um voto válido.

3.7 APTO: Este status deverá ser atribuído quando o eleitor a ser salvo ou atualizado não tiver pendência de cadastro, não estiver inativo, não estiver bloqueado e não tiver votado. Então, ele estará apto a votar.

STATUS DO CANDIDATO

3.8 O status não deverá ser enviado na requisição, mas sim processado pelo sistema.

3.9 Os status possíveis são: ATIVO e INATIVO.

3.10 INATIVO: Este status deverá ser atribuído quando houver requisição para deletar o candidato.

O sistema jamais deverá deletar de fato o candidato, apenas mudar o status para INATIVO. Para isso, o método *delete*() do CRUD deverá ser adaptado e o método *findAll*() do CRUD deverá listar somente os candidatos ativos.

3.11 ATIVO: Este status deverá ser atribuído sempre para um novo candidato cadastrado.

O VOTO E APURAÇÃO

3.12 A Controller da entidade Voto não deverá ter métodos comuns de CRUD. Deverá ter somente os métodos *votar*() e *realizarApuracao*().

3.13 O método *votar*() deverá receber como argumento um objeto da classe Voto e o id do eleitor. Se tudo der certo, o atributo data e hora de votação deverá ser setado no objeto, o hash de votação deverá ser gerado e setado no objeto, o objeto deverá ser persistido e o hash deverá ser retornado.

3.14 Somente eleitores com status APTO poderão votar. Se algum eleitor com outro status tentar votar, lançar uma exceção= “Eleitor inapto para votação”.

3.15 Para gerar o hash de comprovante, utilize a seguinte lógica: String hash = UUID.*randomUUID*().toString();

3.16 O eleitor não deverá ser identificado no voto que será salvo no banco de dados, mas seu status deverá ser atualizado quando ele votar conforme o regramento ali em cima.

3.17 Antes de persistir um voto, o sistema deve verificar se o candidato enviado no objeto prefeito da requisição é de fato um candidato a prefeito.

Caso tenha sido enviado um candidato a vereador no lugar do objeto de um candidato a prefeito, o sistema deverá lançar uma exceção = “*O candidato escolhido para prefeito é um candidato a vereador. Refaça a requisição!*”.

A mesma verificação deverá ser feita para o objeto vereador.

3.18 O método *realizarApuracao*() não terá argumentos e deverá retornar um objeto da classe Apuração com o resultado das eleições. No service de voto, o método realizarApuracao() deverá realizar todos os cálculos para gerar o objeto de Apuração e devolver para controller, conforme o regramento a seguir:

3.19 No service de candidato, crie um método que recupere a lista de candidatos a prefeito ativos. Crie também outro método que recupere a lista de candidatos a vereador ativos.

3.20 Invoque estas listagens no método realizarApuracao() do service de voto. Depois, percorra cada lista e faça o set inserindo o total de votos de cada candidato dos loops. Para isso, crie um método customizado no repository para retornar o total (count(*)) de votos pelo ID do candidato.

3.21 Após setar todos os votos totais nos atributos transientes de cada candidato, ordene as duas listagens do mais votado para o menos votado. Dica de ordenação de objetos em lista por atributo específico:

Collections.sort(listaCandidatos, (list1, list2) -> Integer.compare(list2.getVotosTotais(), list1. getVotosTotais()));

OUTRAS REGRAS:

3.22 Não há necessidade de criar métodos de CRUD comuns para votação. Somente os métodos votar() e realizarApuracao().

3.23 Não há necessidade de criar controller, service ou repository para a classe Apuração. A apuração será utilizada somente no service e na controller de votação.

3.24 O campo de votos apurados do candidato somente será calculado na apuração. Não deverá ser incrementado na votação.

4 TESTES AUTOMATIZADOS:

4.1 Realizar testes de integração e testes unitários para validar as regras de negócio contidas nos itens 2 e 3 d este documento até atingir **80% de cobertura**.

4.2 Lembrem-se de que exceções geradas nos validations da entidade devem ser testadas com assertThrows e exceções lançadas por vocês nos services devem ser testadas com assertEquals para verificar se deu BAD REQUEST.