Desafio Sicredi

Paulo Victor Sobrinho

[paulo.sobrinho@outlook.com.br](mailto:paulo.sobrinho@outlook.com.br)

**GitHub**: <https://github.com/pvsobrinho/desafio_sicredi_2024>

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**Objetivo da aplicação**

Para criar a API de acordo com o objetivo. O projeto terá as seguintes funcionalidades:

1. Cadastrar uma nova pauta;
2. Abrir uma sessão de votação;
3. Receber votos dos associados;
4. Contabilizar os votos e dar o resultado da votação;
5. Persistência dos dados para garantir que não sejam perdidos em caso de restart.

**Estratégias Comuns de Versionamento de API**

Estratégia escolhida: **Versionamento no Caminho da URL (Path Versioning)**

Essa é a estratégia mais comum, onde a versão da API é indicada diretamente na URL.

**Justificativa:**

A estratégia mais comum e recomendada é o versionamento no caminho da URL (Path Versioning). É a mais transparente e simples de implementar e permite uma separação clara entre versões da API, sem confundir os consumidores.

No entanto, se você precisar de maior flexibilidade ou quiser manter a URL estável, o versionamento no cabeçalho HTTP é uma ótima opção, mas exige que os clientes configurem os cabeçalhos adequadamente.

**Exemplo:**

bash

/api/v1/pautas

/api/v2/pautas

**Vantagens:**

* + Simples de entender e implementar.
  + Permite evoluir a API sem quebrar a compatibilidade com versões anteriores.
  + É fácil para os consumidores verem claramente qual versão estão usando.

**Desvantagens:**

* + Se a API crescer muito com várias versões, pode haver replicação de código entre as versões.
  + URLs podem ficar longas e repetitivas.

**Quando usar:**

* + Quando você precisa manter múltiplas versões da API ativas simultaneamente.
  + Quando os consumidores da API são diversos e precisam de visibilidade clara das versões.

Outras estratégias de versionamento de APIs:

1. Versionamento no Header HTTP (Header Versioning)
2. Versionamento no Parâmetro de Consulta (Query Parameter Versioning)
3. Versionamento por Subdomínio (Subdomain Versioning)

**Iniciando os testes locais:**

Texto

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo azul

Descrição gerada automaticamente

**Rodando a aplicação localmente.**

Importante definir o profile local ou azure antes de executar:

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

**Estrutura do Projeto:**

Texto

Descrição gerada automaticamente

**Por qual motivo optei por não usar injeção de dependência via @Autowired?**

**Vantagens de Usar Injeção por Construtor (Forma Atual)**

1. **Imutabilidade e Segurança**
   * Quando você usa **injeção por construtor** e declara as variáveis com final, as dependências se tornam **imutáveis**. Isso significa que uma vez que os valores são atribuídos pelo construtor, eles não podem ser alterados. Isso proporciona uma maior segurança e consistência no código, prevenindo alterações acidentais nas dependências durante a execução.
2. **Testabilidade Facilitada**
   * A **injeção por construtor** facilita a **injeção manual de mocks** ou objetos falsos ao escrever testes unitários. Isso torna o código mais **testável**, já que você pode passar as dependências manualmente no construtor durante os testes.
   * Exemplo de teste com injeção por construtor:
3. **Depreciação do @Autowired em Construtores**
   * A partir do Spring 4.3, o **Spring** recomenda o uso da **injeção por construtor** em vez de @Autowired nos campos ou até mesmo nos próprios construtores, pois a **injeção por construtor** é automática quando há apenas um construtor na classe. Não é necessário usar @Autowired no construtor para que o Spring saiba qual construtor usar.

Este é uma justificativa interessante pois estou usando Java 23 e Spring 5

1. **Evita Injeção de Campo Nula**
   * Usando @Autowired nos campos, há um risco maior de inicialização tardia, o que pode causar o problema do "null pointer exception" se a injeção falhar ou ocorrer fora de ordem. Na injeção por construtor, como as dependências são resolvidas na criação da instância, esse problema é evitado.
2. **Facilidade de Manutenção e Refatoração**
   * Quando as dependências são passadas pelo construtor, elas ficam visíveis imediatamente, facilitando o entendimento de quais dependências a classe precisa. Isso torna o código mais fácil de manter, especialmente em projetos maiores, onde classes complexas podem ter muitas dependências.

**Injeção com @Autowired nos Campos: Desvantagens**

* **Baixa Visibilidade das Dependências**
  + Quando usamos @Autowired nos campos, não fica claro de imediato quais dependências a classe realmente precisa, o que pode dificultar a manutenção do código.
* **Impossibilidade de Imutabilidade**
  + Com a injeção de dependência diretamente nos campos, você não pode marcar os campos como final, o que significa que eles podem ser alterados acidentalmente durante a execução, potencialmente causando problemas inesperados.
* **Dificuldade nos Testes**
  + Testar classes que usam @Autowired diretamente nos campos pode ser mais complicado, porque você precisaria usar um framework como o Spring ou ferramentas adicionais de mock para injetar dependências nos testes. Já com injeção por construtor, você pode passar as dependências diretamente no construtor durante os testes.

**Isso pode se tornar uma dor de cabeça a medida que o projeto fica mais complexo. Realizar teste automatizado pode se tornar cada vez mais difícil, ou não a depender das escolhes arquiteturais e de quais recursos serão utiliozados.**

**Documentação da API com Swagger**

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Existem validações nos parâmetros e nos campos do requestBody. Fazer validação no dto é uma forma eficiente e de otimização para reutilização de código.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Uso de conversores garante maior segurança ao evitar o uso de uma entidade no controlador e melhor eficiência no uso de dados. Trafegando assim apenas os dados necessários e não todo o corpo de dados da entidade. Além de garantir um código limpo para quem usa a api. Apenas dados necessários são trafegados para determinada ação.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**Otimização no tratamento de exceções com o GlobalExceptionHandler**

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

**Arquitetura**

O código que construímos segue **alguns** princípios da arquitetura **SOLID**, mas ainda existem pontos que podem ser aprimorados para atender completamente a esses princípios. **Não foram aplicados todos os princípios pois não existe complexidade suficiente para justificar tal uso**. Vamos analisar cada princípio do SOLID e identificar como o código atual se alinha a eles:

**1. S - Single Responsibility Principle (SRP)**

**O que o código já faz bem:**

* Cada classe tem uma responsabilidade clara:
  + **Controladores** lidam com as requisições HTTP.
  + **Serviços** contêm a lógica de negócios.
  + **Repositórios** interagem com o banco de dados.
  + **Conversores** fazem a conversão entre DTOs e entidades.

**O que pode melhorar:**

* O código está bem próximo de cumprir o SRP, mas uma análise constante deve ser feita para garantir que classes de serviço não fiquem sobrecarregadas com responsabilidades adicionais. Se a lógica de negócios crescer, poderá ser necessário dividir algumas responsabilidades entre serviços menores ou criar classes auxiliares.

**2. O - Open/Closed Principle (OCP)**

**O que o código já faz bem:**

O design do sistema facilita a extensão sem a necessidade de modificar código existente. Por exemplo, novos endpoints ou funcionalidades podem ser adicionados sem alterar os controladores e serviços existentes.

**O que pode melhorar:**

* Em alguns casos, podemos precisar de estratégias mais avançadas, como a criação de interfaces e abstrações para comportar diferentes implementações sem modificar as classes existentes. No momento, os serviços e controladores estão relativamente simples e cumprem o papel. À medida que a aplicação crescer, interfaces e estratégias de injeção de dependência serão ainda mais importantes para garantir a adesão a esse princípio.

**3. L - Liskov Substitution Principle (LSP)**

**O que o código já faz bem:**

* O LSP se aplica mais a cenários onde há herança ou polimorfismo. Como o código atual usa principalmente injeção de dependência com interfaces para repositórios, não temos problemas diretos com LSP. Se decidirmos implementar classes abstratas ou subclasses, será importante garantir que essas substituições não alterem o comportamento esperado das classes de forma inesperada.

**O que pode melhorar:**

* Quando começarmos a usar mais interfaces ou subclasses, precisamos garantir que todas as substituições mantenham o comportamento esperado. Isso é mais relevante para o futuro, caso o projeto se expanda com mais interfaces e heranças.

**4. I - Interface Segregation Principle (ISP)**

**O que o código já faz bem:**

* O ISP sugere que as interfaces devem ser pequenas e especializadas. No momento, usamos repositórios que implementam interfaces do Spring Data JPA (JpaRepository), que são específicas e focadas nas operações de banco de dados.

**O que pode melhorar:**

* À medida que a aplicação crescer, devemos considerar evitar interfaces grandes e focar em criar interfaces especializadas para funções específicas de serviços ou repositórios, evitando que classes implementem métodos que não precisam.

**5. D - Dependency Inversion Principle (DIP)**

**O que o código já faz bem:**

* Estamos usando **injeção de dependências** com **repositórios** e **serviços**. Isso facilita a testabilidade e mantém o código desacoplado de implementações específicas.

**O que pode melhorar:**

* Podemos ir além e introduzir mais abstrações por meio de interfaces para serviços, o que nos permitirá criar implementações alternativas dos serviços sem alterar o código que depende dessas classes.

**Teste de Operações da API**

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

TESTE INFORMANDO UMA PAUTA INVALIDA OU NÃO EXISTENTE:

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Teste informando uma pauta existente com cpf invalido:

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Teste com CPF correto:

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Site

Descrição gerada automaticamente

**Testes Unitários Automatizados com jUnit**

Apesar de não ter sido solicitado no enunciado. Realizei um teste no serviço o principal. É de fundamental importância os testes unitários. Assim evitamos gerar novos bugs em ambiente de produção pelo fato de a automatização de testes capturar bugs antes de submeter novas alterações para o ambiente produtivo.

Apenas para não perder o costume criei uma classe de teste:

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**Configurando a Nuvem Azure (WebApp + Banco de Dados)**

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, Email, Site

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Email

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Tela de computador com fundo preto

Descrição gerada automaticamente

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Trying to deploy artifact to api-sicredi...

Deploying (C:\Users\victo\IdeaProjects\desafio\_sicredi\_2024\target\desafio\_sicredi-0.0.1-SNAPSHOT.jar)[jar] ...

URL: <https://api-sicredi-gmdpfnebhmdabxd3.canadacentral-01.azurewebsites.net>

**Link Swagger na Nuvem Azure:**

[api-sicredi-qmdpfnebhndabxd3.canadacentral-01.azurewebsites.net](https://api-sicredi-qmdpfnebhndabxd3.canadacentral-01.azurewebsites.net/swagger-ui/index.html)

[api-sicredi-gmdpfnebhmdabxd3.canadacentral-01.azurewebsites.net](https://api-sicredi-gmdpfnebhmdabxd3.canadacentral-01.azurewebsites.net/)

**Swagger**:

<https://api-sicredi-gmdpfnebhmdabxd3.canadacentral-01.azurewebsites.net/swagger-ui/index.html>

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

O serviço pode apresentar intermitência ou estar indisponível por conta das características do plano e limitação de créditos. De toda forma é possível testar localmente caso o serviço na nuvem não esteja disponível.

**Solução alternativa para teste caso de indisponibilidade do webapp Azure:**

Rodando local não é necessário instalar banco de dados já que a aplicação aponta para base de dados do azure com firewall desativado.