Agendamento de consultas

Course	Spring Boot 3
Confidence	Not Confident
Next Review	@2 de agosto de 2023
Last Edited	@1 de agosto de 2023 10:45
Curso	Spring Boot 3: Documente teste e prepare uma API para o deploy

Topicos

- Apresentação
- Projeto inicial
- Nova funcionalidade
- Para saber mais: anotação @JsonAlias
- Para saber mais: formatação de datas
- Classe Service
- Validações de integridade
- Para saber mais: Service Pattern
- Escolha de médico aleatório
- Consultas com @Query
- Faça como eu fiz: cancelamento de consultas
- O que aprendemos?

Nova Funcionalidade

- [ ] Controller
- [ ] DTOs
- [ ] Entidade JPA
- [ ] Repository
- [ ] Migration
- [ ] Requisição para nova consulta

Para saber mais: anotação @JsonAlias

Aprendemos que os nomes dos campos enviados no JSON para a API devem ser idênticos aos nomes dos atributos das classes DTO, pois assim o Spring consegue preencher corretamente as informações recebidas.

Entretanto, pode acontecer de um campo ser enviado no JSON com um nome diferente do atributo definido na classe DTO. Por exemplo, imagine que o seguinte JSON seja enviado para a API:

```
{
    “produto_id” : 12,
    “data_da_compra” : “01/01/2022”
```

E a classe DTO criada para receber tais informações seja definida da seguinte maneira:

```
public record DadosCompra(
    Long idProduto,
    LocalDate dataCompra
){}
```

Se isso ocorrer, teremos problemas, pois o Spring vai instanciar um objeto do tipo `DadosCompra`, mas seus atributos não serão preenchidos e ficarão como `null` em razão de seus nomes serem diferentes dos nomes dos campos recebidos no JSON.

Temos duas possíveis soluções para essa situação:

1) Renomear os atributos no DTO para terem o mesmo nome dos campos no JSON;

2) Solicitar que a aplicação cliente, que está disparando requisições para a API, altere os nomes dos campos no JSON enviado.

A primeira alternativa citada anteriormente não é recomendada, pois os nomes dos campos no JSON não estão de acordo com o padrão de nomenclatura de atributos utilizado na linguagem Java.

A segunda alternativa seria a mais indicada, porém, nem sempre será possível "obrigar" os clientes da API a alterarem o padrão de nomenclatura utilizado nos nomes dos campos no JSON.

Para essa situação existe ainda uma terceira alternativa, na qual nenhum dos lados (cliente e API) precisam alterar os nomes dos campos/atributos. Basta, para isso, utilizar a anotação `@JsonAlias`:

```
public record DadosCompra(
    @JsonAlias("produto_id") Long idProduto,
    @JsonAlias("data_da_compra") LocalDate dataCompra
){}
```

A anotação `@JsonAlias` serve para mapear "apelidos" alternativos para os campos que serão recebidos do JSON, sendo possível atribuir múltiplos *alias*:

```
public record DadosCompra(
    @JsonAlias({"produto_id", "id_produto"}) Long idProduto,
    @JsonAlias({"data_da_compra", "data_compra"}) LocalDate dataCompra
){}
```

Dessa forma resolvemos o problema, pois o Spring, ao receber o JSON na requisição, vai procurar os campos considerando todos os *alias* declarados na anotação `@JsonAlias`.

Para saber mais: formatação de datas

Como foi demonstrado no vídeo anterior, o Spring tem um padrão de formatação para campos do tipo data quando esses são mapeados em atributos do

tipo `LocalDateTime`. Entretanto, é possível personalizar tal padrão para utilizar outras formatações de nossa preferência.

Por exemplo, imagine que precisamos receber a data/hora da consulta no seguinte formato: **dd/mm/yyyy hh:mm**. Para que isso seja possível, precisamos indicar ao Spring que esse será o formato ao qual a data/hora será recebida na API, sendo que isso pode ser feito diretamente no DTO, com a utilização da anotação `@JsonFormat`:

```
@NotNull
@Future
@JsonFormat(pattern = "dd/MM/yyyy HH:mm")
LocalDateTime data
```

No atributo ***pattern*** indicamos o padrão de formatação esperado, seguindo as regras definidas pelo padrão de datas do Java. Você pode encontrar mais detalhes nesta página do JavaDoc.

Essa anotação também pode ser utilizada nas classes DTO que representam as informações que a API devolve, para que assim o JSON devolvido seja formatado de acordo com o pattern configurado. Além disso, ela não se restringe apenas à classe `LocalDateTime`, podendo também ser utilizada em atributos do tipo `LocalDate` e `LocalTime`.

Classe Service

A classe *Service* executa as regras de negócio e validações da aplicação

---

Validações de integridade

Nessa aula ele demonstra como podemos verificar se a um registro o banco de dados com o determinado id, tanto para um atributo ***obrigatório*** como para um atributo ***opcional***, usando o método `.existsById(id)`

Para saber mais: Service Pattern

O **Padrão Service** é muito utilizado na programação e seu nome é muito comentado. Mas apesar de ser um nome único, *Service* pode ser interpretado de várias maneiras: pode ser um ***Use Case*** (*Application Service*); um ***Domain Service***, que possui regras do seu domínio; um ***Infrastructure Service***, que usa algum pacote externo para realizar tarefas; etc.

Apesar da interpretação ocorrer de várias formas, a ideia por trás do padrão é separar as regras de negócio, as regras da aplicação e as regras de apresentação para que elas possam ser facilmente testadas e reutilizadas em outras partes do sistema.

Existem duas formas mais utilizadas para criar Services. Você pode criar Services mais genéricos, responsáveis por todas as atribuições de um Controller; ou ser ainda mais específico, aplicando assim o **S** do **SOLID**: *Single Responsibility Principle* (Princípio da Responsabilidade Única). Esse princípio nos diz que uma classe/função/arquivo deve ter apenas uma única responsabilidade.

Pense em um sistema de vendas, no qual provavelmente teríamos algumas funções como: *Cadastrar usuário*, *Efetuar login*, *Buscar produtos*, *Buscar produto por nome*, etc. Logo, poderíamos criar os seguintes Services: `CadastroDeUsuarioService`, `EfetuaLoginService`, `BuscaDeProdutosService`, etc.

Mas é importante ficarmos atentos, pois muitas vezes não é necessário criar um Service e, consequentemente, adicionar mais uma camada e complexidade desnecessária à nossa aplicação. Uma regra que podemos utilizar é a seguinte: se não houverem regras de negócio, podemos simplesmente realizar a comunicação direta entre os controllers e os repositories da aplicação.

Consultas com `@Query`

Ao analisar o código de uma *interface repository*, em um projeto com Spring Boot, você encontra nele o seguinte método:

```
@Query(“””
    select p from Produto p
    where
    p.preco >= :preco
    and
    p.dataCadastro >= data
“””)
List<Produto> buscarPorPrecoEData(BigDecimal preco, LocalDate data);
```

Qual o problema existente no método anterior? *Selecione uma alternativa*

- Não se pode utilizar o operador `>=` para atributos do tipo data.
  - É possível sim utilizar os operadores lógicos em atributos do tipo data.
- O retorno do método deveria ser `Produto` e não `List<Produto>`.
  - A consulta anterior pode devolver mais de um registro.
- O nome do método não seguiu o padrão de nomenclatura do Spring Data.
  - Ao utilizar a anotação `@Query`, não é obrigatório seguir o padrão de nomenclatura do Spring Data, com o nome do método em inglês.
- Faltou adicionar o caractere dois-pontos (:) antes do parâmetro data.
  - O caractere dois-pontos (:) indica um parâmetro dinâmico na query, sendo que nesse caso o parâmetro `data` deveria ter sido declarado da seguinte forma: `p.dataCadastro >= :data`.

Faça como eu fiz: cancelamento de consultas

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, porém, para a funcionalidade de **cancelamento de consultas**, cuja descrição encontra-se neste card do trello.

O que aprendemos?

Nesta aula, você aprendeu como:

- Implementar uma nova funcionalidade no projeto;
- Avaliar quando é necessário criar uma classe Service na aplicação;
- Criar uma **classe Service**, com o objetivo de isolar códigos de regras de negócio, utilizando para isso a anotação `@Service`;
- Implementar um algoritmo para a funcionalidade de agendamento de consultas;
- Realizar **validações de integridade** das informações que chegam na API;

- Implementar uma consulta JPQL (*Java Persistence Query Language*) complexa em uma *interface repository*, utilizando para isso a anotação `@Query`.
