Repositórios (Parte 2)

Site: Moodle institucional da UTFPR
Curso: CETEJ34 - Sistemas de Banco de Dados - JAVA_XXIX (2023_03)
Livro: Repositórios (Parte 2)

Impresso por: BEATRIZ NASCIMENTO GOMES
Data: domingo, 18 fev. 2024, 10:53

Índice

1. Repositórios (Parte 2)

Até o tópico anterior estudamos como trabalhar com os métodos de consultas provenientes das interfaces do Spring Data JPA. Contudo, em algum momento, será necessário desenvolver suas próprias consultas e o Spring Data JPA fornece algumas formas distintas de fazer isso. E uma delas é com o uso de palavras-chaves (*keywords*).

1.1. Consultas por Palavras-chaves

Este recurso é totalmente baseado no nome que é dado ao método, ou seja, na sua assinatura, a qual vai conter palavras-chave e a partir delas o Spring Data vai ser capaz de montar a consulta JPQL para então executá-la.

Veja *o* exemplo de uma consulta por palavras-chaves adicionada na **Listagem 4.6.**

LISTAGEM 4 .6: Consulta por Palavras-Chaves

```
public interface ContatoRpeository extends JpaRepository<Contato, Long> {

   Contato findByNomeAndIdade(String nome, Integer idade);

}
```

Analisando a assinatura do método **findByNomeAndIdade()** é possível selecionar quatro partes distintas:

- **findBy -** aqui a palavra chave é **By,** porém, anterior **a By** é necessário ter alguma instrução. Normalmente em consultas essa instrução é find, mas poderia ser, por exemplo, get . Além dessas, outras situações como count para quantidade ou remove para exclusão. Caso não haja uma instrução anterior a By, uma exceção será lançada;
- **Nome** - faz referência ao atributo nome da classe de entidade Contato;
- **And** - palavra-chave reservada do Spring Data JPA que tem a mesma função do AND na JPQL;
- **Idade** - faz referência ao atributo idade da classe de entidade Contato.

A assinatura do método apresentado vai gerar uma consulta JPQL conforme a seguinte instrução:

```
"from Contato e where e.nome = ?1 and e.idade = ?2"
```

Outro ponto importante a ressaltar é que cada palavra-chave da consulta deve ser iniciada por letras maiúsculas seguidas de letras minúsculas.

Esta prática vai evitar exceções, já que o Spring entende que uma nova palavra-chave foi inserida quando ele encontra uma letra maiúscula.

O Spring Data JPA fornece uma lista de palavras-chaves para serem usadas nas assinaturas dos métodos.

Algumas dessas palavras-chaves podem acabar gerando algum tipo de dúvida. Como exemplo, a **StartingWith, EndingWith** e **Containing.**

Estas três instruções usam o coringa (%) para analisar, respectivamente, o início, o final ou qualquer trecho de uma *string* armazenada na coluna da tabela.

O parâmetro que será enviado pelo método de consulta não deve conter o caractere coringa, a própria instrução no nome do método vai adicioná-la no momento apropriado.

As consultas com **In** ou **NotIn,** devem receber como parâmetro um objeto do tipo **Collection ,** ou então , um *array* ou *varargs.* Os elementos contidos neste objeto serão os valores analisados pela consulta com o critério.

Até agora, todas as consultas abordadas foram realizadas especificamente sobre a entidade **Contato,** usando como critério os atributos desta classe. Mas, como **Contato** tem um relacionamento do tipo **1 - 1** com **Endereco** é perfeitamente possível adicionar a partir do repositório de contato uma consulta que alcance algum atributo de **Endereco** com o critério.

Suponha que você queira localizar todos os contatos que estão registrados no banco de dados com o endereço na cidade do Rio de Janeiro. Sendo que, cidade é um atributo da classe de entidade **Endereco.**

Para realizar esta consulta, leve em consideração o link entre cada objeto das classes, como mostra o exemplo na **Listagem 4.7.**

LISTAGEM 4.7: Consulta por Palavras-Chaves

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {
   // código anterior omitido nesta listagem
   List<Contato> findByEnderecoCidade(String cidade);
}
```

Analisando a assinatura do método, após a palavra-chave **By,** temos a instrução **EnderecoCidade .** Esta instrução remete a algo como:

```
contato.getEndereco().getCidade()
```

Porém , o objeto **contato,** não precisa estar presente no nome do método porque ele já é o ponto de partida dessa consulta. Isto porque, por padrão, ela parte do repositório referente a **Contato.**

Basta, então, incluir na assinatura do método os demais atributos, neste caso, endereco e cidade. Sendo assim, sempre que for necessário usar um atributo do objeto de relacionamento com a classe de entidade do repositório, faça esse link.

1.2. Limitando Resultados

Duas palavras-chaves são utilizadas para limitar os resultados: **First** e **Top**. Elas funcionam como uma instrução que vai retornar o primeiro elemento encontrado a partir da consulta. Ou seja, essas palavras-chaves trabalham como um limitador de resultados.

Então, vamos ver alguns exemplos que abordam o uso de **First e Top**. Para isso, suponha que a tabela **Contatos** tenha as seguintes entidades (Tabela 4.2).

Contatos

Id	Nome	Idade	Data-Cadastro	Endereço_ID
1	Ana da Silva	30	2013-10-05	1
2	Eduardo Coelho	25	2013-11-14	2
3	Juliana Boemo	39	2014-04-21	3
4	Aniele Vicentini da Silva	23	2014-08-18	4

Tabela 4 .2 : TABELA CONTATOS DA BASE DE DADOS

A partir desses registros, a seguinte consulta retornaria a entidade de **ID= 1**, quando executada na tabela de contatos.

```
Contato findFirstBy();
```

Analisando a assinatura do método, observe que após a palavra **find**, foi adicionada a palavra **First** seguida de **By**. Esta instrução vai fazer com que o retorno seja contido por uma única entidade, sem qualquer critério adicionado na consulta.

Por isso, o retorno foi o primeiro elemento da tabela. O mesmo resultado se aplicaria caso a palavra-chave **First** fosse substituída por **Top**, isto porque, elas têm exatamente a mesma função.

No entanto, se essa consulta fosse ordenada por idade, de forma ascendente (O - 9), o resultado seria diferente do anterior. Isto porque, primeiro é realizada a ordenação e depois dos resultados estarem ordenados é retornada a primeira entidade ou a entidade que ficou no topo desta ordenação.Sendo assim, a linha de **ID = 4** seria o retorno da seguinte consulta:

```
Contato findTopByOdrerByIdadeAsc();
```

Mas ainda existe outra característica que pode ser explorada no uso de **First** e **Top**, que é limitar o resultado a uma quantidade desejada de elementos. Algo como a função **limit** do MySQL, **top** do SqlServer ou **rownum** do Oracle.

Confira a seguinte consulta, que espera como retorno uma lista de contatos:

```
List<Contato> findFirst3ByOrderByIdadeAsc();
```

Note que após a palavra-chave **First**, foi incluído o número **3** (**First3**). Este número representa a quantidade limite de linhas que a consulta deve retornar.

Outro aspecto importante é que a consulta está usando a ordenação ascendente por idade, então o retorno seria as seguintes entidades: **ID = 4, ID = 2 e ID = 1**.

Isto porque, a ordenação por idade dispôs as linhas nesta sequência de identificadores: **4, 2, 1 e 3**. Mas, como a consulta foi limitada a apenas três entidades, a linha de **ID = 3** ficou de fora do retorno.

Então estas são as características necessárias que se deve conhecer sobre as palavras-chaves **First e Top**.

1.3. Anotação @Query

As consultas baseadas em palavras-chaves são uma forma bem prática e rápida de se trabalhar nos repositórios. Porém, em algumas situações, usar esse tipo de técnica pode não ser o mais apropriado.

Uma delas é quando *você* tem muitas regras ou critérios para um a determinada consulta, o que vai fazer com que a assinatura do método fique grande demais, o que pode causar uma dificuldade para outro membro da equipe de desenvolvimento compreendê-la se não tiver ainda experiência com esse tipo de recurso.

Outra situação pode ser o gosto pessoal do desenvolvedor, que prefere usar instruções JPQL ao invés do recurso por palavras-chaves.

Então, o Spring Data JPA fornece a anotação @Query, a qual tem como finalidade receber uma *string* contendo uma instrução JPQL. Deste modo, é perfeitamente possível substituir o recurso de palavras-chaves por JPQL.

Veja um exemplo na **Listagem 4.8** em que a consulta é declarada via anotação @Query.

LISTAGEM 4 .8 : JPQL VIA @Query

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {

  // código anterior omitido nesta listagem

  @Query(" select c from Contato c  where idade >= 18 order by nome asc")

  List<Contato> findByContatoMaiolrdade();

}
```

Observe o método **findByContatoMaiorIdade() ,** e veja que a anotação **@Query** foi incluída no topo de sua assinatura. E como parâmetro, a anotação recebeu uma *string* contendo a JPQL. Esta consulta vai localizar no banco de dados todas as linhas em que a idade seja maior ou igual a 18 e ainda ordenar os resultados por nome de forma ascendente (A - Z).

Como mostra o exemplo, é bem simples incluir uma JPQL como instrução de consulta para substituir o uso de palavras-chaves. Algo importante a ressaltar é em relação ao nome do método, conforme exemplifica a **Listagem 4.9.**

LISTAGEM 4 .9: JPQL SOBRESCREVENDO PALAVRAS CHAVES @QUERY

```
public interface ContatoRepository extends JpaRepository <Contato, Long> {
```

```
  // código anterior omitido nesta listagem
```

```
  @Query("select c from Contato c where c.dtCadastro > ?1")
  List<Contato> findByDtCadastroAfter(Date dataCadastro);
}
```

Analisando com cuidado este código se pode notar que a assinatura do método usa exatamente as palavras-chaves que executam a mesma consulta JPQL incluída na **@Query.**

Quando uma situação dessas acontecer a anotação vai sobrescreveras palavras-chaves, ou seja, o nome do método não será levado em consideração.

A **@Query** ainda tem outra função que é trabalhar com instruções *em* SQL, algo que não seria possível com palavras-chaves. Na **Listagem 4.10** é apresentado um exemplo sobre esse caso.

LISTAGEM 4 .10: @QUERY COM NATIVE QUERY

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {
```

```
  // código anterior omitido nesta listagem
```

```
  @Query(value = "select * from contatos where data_cadastro = ?1", nativeQuery = true)
```

```
  List<Contato> findByDataCadastro(Date dataCadastro);
}
```

Observe que a consulta desta vez é em SQL e não em JPQL. É possível diferenciá-las por alguns aspectos principais como o uso do nome da tabela e de sua coluna *e* não o nome da classe de entidade *e* de seu atributo.

Para que um SQL seja aplicado à anotação é necessário incluir a propriedade **nativeQuery** inicializada com true. Assim, a anotação ativa o uso de instruções nativas ao invés de processar uma instrução JPQL e neste caso, a propriedade **value** deve ser atribuída com a *string* contendo a consulta.

Outro aspecto sobre a anotação @Query é que ela não precisa receber diretamente uma instrução JPQL ou SQL. A JPA fornece dois recursos chamados **NamedQuery** (JPQL) e **NamedNativeQuery** (SQL), os quais podem ser usados via anotações na classe de entidade ou arquivo XML.

O importante, é que a **@Query** pode trabalhar em conjunto *com* estes recursos. Sendo assim, uma pequena alteração será realizada na classe de entidade Contato, para a inclusão de duas anotações, conforme mostra a **Listagem 4.11.**

LISTAGEM 4 .11: ALTERAÇÃO NA CLASSE DE ENTIDADE CONTATO

```
@Entity
@Table(name = "CONTATOS")
@NamedQuery(
   name = "Contato.byIdade",
   query = "from Contato c where c.idade = ?1")
@NamedNativeQuery(
   name = "Contato.byNome",
   query = "select * from Contatos where nome like ?1",
   resultClass = Contato.class)
public class Contato extends AbstractPersistable<Long> {
   // código anterior omitido nesta listagem
}
```

Veja que as anotações **@NamedQuery** e **@NamedNativeQuery** foram incluídas sobre a assinatura da classe. Cada uma delas possui os atributos name, para a inclusão de um nome e query para a adição da instrução de consulta.

Já @NamedNativeQuery ainda tem o atributo **resultClass** para transformar o resultado da consulta nativa em um objeto Contato.

O importante, a saber, é como vamos vincular essas duas consultas a anotação @Query lá em ContatoRepository. Isto é feito de forma bem simples, apenas utilizando os nomes incluídos nos atributos name de cada anotação.

Então, confira na **Listagem 4.12** o processo que deve ser realizado para criar este vinculo.

LISTAGEM 4 .12: NAMED QUERY E NAMED NATIVE QUERY VIA REPOSITÓRIO

```
public interface Contato Repository extends JpaRepository <Contato, Long>  {
```

```
   // código anterior omitido nesta listagem
```

```

```

```
   @Query(name = "Contato.byIdade")
```

```
   List<Contato> findByIdade(Integer idade);
```

```
   @Query(name = "Contato.byNome")
```

```
   Contato findByNome(String nome);
```

```
}
```

```

```

Analisando o código-fonte se pode notar que as anotações @Query têm um atributo **name,** o qual deve ser atribuído com o nome referente a consulta NamedQuery ou NamedNativeQuery que se deseja executar.

1.4. Remodelando Resultados

Quando uma consulta é executada via repositório do Spring Data JPA, os resultados são baseados na classe de entidade que está vinculada ao repositório. Sendo assim, o retorno das consultas sempre vai trazer todos os atributos existentes na classe de entidade.

Porém, algumas vezes não é necessário que todos os atributos da classe façam parte do retorno da consulta. Por exemplo, a classe Contato, apresentada na Listagem 3.1, possui os segunites atributos: id, nome, idade, dtNascimento e enderecos.

Desta forma, cada vez que uma consulta é executada via ContatoRepository, todos esses atributos são retornados e aqueles que por ventura, não tenham valor algum lá na tabela, resultariam em objetos nulos. Por isso, vamos dar uma atenção especial ao relacionamento de Contato com a entidade Endereco. Este relacionamento exige uma consulta adicional por parte do Hibernate. Ou seja, o Hibernate busca a entidade Contato e depois ele roda uma nova consulta por conta própria para trazer o endereço vinculado ao contato retornado.

Então, para cada vez que uma consulta é executada o Hibernate vai rodar pelo menos duas consultas devido o relacionamento. E se a classe de entidade tiver mais de um relacionamento mapeado, mais consultas adicionais serão necessárias.

Deste modo, os relacionamentos acabam criando um objeto de retorno muito grande e às vezes com dados que não são necessários ou que o usuário não vai precisar utilizar. Por exemplo, se uma ação na aplicação precisar localizar um contato para exibir apenas o nome, a idade e a data de cadastro, por que trazer junto os dados referente ao endereço? Nesta situação, o ideal seria remodelar o retorno da consulta e projetar um retorno sem o endereço contido nele.

Para isso, o SpringData sugere que seja criada uma interface de projeção, onde são especificamente definidos os dados que a consulta deve retornar.

Confira então, a **Listagem 4.13**, para ver um exemplo deste processo.

LISTAGEM 4.13: Interface de Projeção SemEndereco

```
package com.curso.entity.projection;
import java.util.Date;
public interface SemEndereco {
   String getNome();
   Integer getIdade();
   Date getDtCadastro();
}
```

Observe no código-fonte que a interface SemEndereco possui três métodos. Esses métodos são os dados que se quer ter como retorno a partir de uma consulta.

Eles também fazem referência aos métodos get() da classe da entidade Contato. É dessa forma que o Spring Data vai conseguir vincular os dados de retorno com a interface SemEndereco. E assim, ao invés de retornar um objeto Contato, vai retornar um objeto SemEndereco.

Mas como o Spring Data vai saber que precisa retornar um objeto SemEndereco, ao invés de um Contato a partir da interface ContatoRepository?

Este processo é extremamente simples, basta incluir na assinatura do método de consulta o retorno conforme um objeto do tipo SemEndereco, como mostra a **Listagem 4.14**.

LISTAGEM 4.14: Remodelando o Retorno para SemEndereco

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {
```

```
   // código anterior omitido nesta listagem
```

```
   SemEndereco findContatoByNome(String nome);
}
```

Agora, toda vez que a consulta findContatoByNome() for executada, o retorno será um objeto SemEndereco contendo apenas os valores para as variáveis nome, idade e dtCadastro. Deste modo, é possível criar várias interfaces de projeção para remodelar seus objetos com apenas os dados que são realmente necessários em cada consulta. Então, vamos explorar um pouco mais esse recurso.

Suponha agora, que o retorno desejado de uma consulta contenha o nome do contato e a cidade cadastrada. Lembrando que, a cidade está armazenada na tabela de endereços, assim, é necessário usar o relacionamento existente entre as classes de entidades Contato e Endereco para obter o resultado.

Uma nova interface de projeção será criada, conforme a **Listagem 4.15**.

LISTAGEM 4.15: INTERFACE DE PROJEÇÃO NOMECIDADE

```
package com.curso.entity.projeciton;
import org.springframework.beans.factory.annotation.Value;
public interface NomeCidade {
```

```
   String getNome();
```

```
   @Value("#{(target.endereco.cidade)}")
   String getCidade();
}
```

Ao analisar a interface NomeCidade, é possível verificar uma diferença marcante em relação à interface SemEndereco já vista anteriormente.

Tal diferença fica pelo uso da anotação **@Value**, que tem como função receber uma instrução do tipo SpEL Expressions. Essa expressão nada mais faz do que indicar um atributo alvo que vai ser o valor do método anotado.

Ou seja, o método getCidade(), que não existe na classe Contato, vai receber o valor do atributo cidade da classe Endereco. O nome deste método poderia ser qualquer um, como get Cidadela() , getContatoCidade() , entre outros . Isto porque, quem está definindo o local de onde o valor da cidade será recuperado é a instrução presente na anotação.

Observe que expressão possui a palavra **target** , a qual é obrigatória no inicio da instrução. Em seguida, se navega entre as propriedades dos objetos para chegar ao destino desejado, neste caso, o atributo cidade do objeto endereco. Este tipo de instrução é denominada atributo virtual, já que o atributo alvo na verdade não faz parte da entidade da qual vai partir a consulta.

Por fim, basta criar o método de consulta em ContatoRepository, conforme demonstra a **Listagem 4.16**.

LISTAGEM 4.16 : REMODELANDO O RETORNO PARA NOMECIDADE

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {
```

```
   // código anterior omitido nesta listagem
```

```
   NomeCidade findContatoCidadeByNome(String nome);
}
```

Veja então, que foi possível remodelar o resultado de uma consulta de forma que o retorno passa a ser apenas o nome e a cidade do contato. A anotação @Value pode ser usada em qualquer método get() adicionado na interface de projeção e assim, permitir que o nome do método get() seja diferente daquele que faz referência ao atributo de origem.

É possível também, unir os valores de diferentes campos em um mesmo método get() com o uso dessa anotação. Por exemplo, suponha que uma classe qualquer tenha os atributos firstName e lastName e você queira criar uma projeção para gerar o nome completo. Para isso, basta fazer algo como o código a seguir:

```
@Value("#{target.firstName} #{target.lastName}")
```

```
String getFullName();
```

Perceba que foram adicionados, na anotação, dois alvos, o firstName e o lastName. Assim, o método getFullName() vai ter como resultado final a união desses dois valores formando o nome completo.

Para finalizar este tópico será apresentada outra abordagem sobre o recurso de projeção. Suponha que uma consulta na tabela de endereços deva retornar o nome da cidade e o total de vezes que ela foi cadastrada.

Para isso, será preciso utilizar um Group By na consulta e então contabilizar o número de vezes que cada cidade aparece na tabela.

Então, vamos começar pensando na interface de projeção que vai conter dois métodos para representar estes valores. Esses métodos são o getCidade() e o getTotal(), conforme mostra a **Listagem 4.17**.

LISTAGEM 4 .17: INTERFACE DE PROJEÇÃO PARA ENDEREÇOS

```
package com.curso.enitty.projection;

public interface CidadeTotal{
   String getCidade();
   Long getTotal();
}
```

Como a consulta será por endereços, o ideal é usar um repositório para a entidadeEndereco, conforme a **Listagem 4.18**.

LISTAGEM 4 .18 : INTERFACE ENDERECOREPOSITORY

```
package com.curso.repository;
```

```
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
```

```
import com.curso.entity.Endereco;
import com.curso.entity.projection.CidadeTotal;
```

```
public interface EnderecoRepository extends JpaRepository<Endereco, Long> {
```

```
  @Query("select e.cidade as cidade, count(e.cidade) as total"
  + " from Endereco e"
  + " group by e.cidade")

  List<CidadeTotal> findByCidadeTotal();
}
```

A interface EnderecoRepository não tem nada de novo em relação ao que já foi abordado em outros capítulos. O que tem de mais importante nela é a consulta JPQL desenvolvida para retornar um objeto remodelado como CidadeTotal.

Ou seja, essa consulta vai ter dois valores como resultado: o nome da cidade e o totalizador por cidade. Observe que a função count usa o alias total , o qual vai retornar o valor totalizado para o método getTotal(). E o alias cidade, retorna o nome da cidade para o método getCidade().

Desta forma, o retorno da consulta, por ser uma lista, teria em cada posição da lista um objeto CidadeTotal contendo um atributo cidade, com o nome da cidade e um atributo total , com a quantidade total de vezes que esta cidade aparece cadastrada no banco de dados.

Dai em diante, conhecendo o recurso de remodelagem de entidade, é possível que você seja capaz de criar consultas que retornem projeções do tipo soma (sum), média (avg), minimo (min), máximo (max) ou qualquer outra.