# Repositórios

Site: Moodle institucional da UTFPR
Curso: CETEJ34 - Sistemas De Banco De Dados - JAVA_XIX (2020_01)
Livro: Repositórios

Impresso por: Alexandre Tiago Ximenes
Data: sábado, 22 ago 2020, 06:51

# Sumário

# 1. Repositórios

O Spring Data como já abordado, fornece três interface sprincipais que são: **Repository**, **CrudRepository** e **PagingAndSortingRepository**.

Estas interfaces são a base para o projeto Spring Data JPA, sendo assim, é importante conhecê-las um poucomais a fundo.

Nesta relação de interfaces, Repository é tida como o ponto inicial, ou a interface base que tem o objetivo de formar a infraestrutura dos repositórios no Spring Data. Já CrudRepository, que estende Repository, surge com uma lista de assinaturas de métodos.

Nesta lista, existem operações básicas para CRUD, aquelas mais comuns existentes na camada de persistência de um projeto com acesso a banco de dados. Vejamos quais são essas ope rações:

- **save(S)** - método para salvar uma entidadena base de dados;
- **save(Iterable<S>)** - este método insere uma lista de entidades no banco de dados, onde cada entidade desta lista é inserida por uma diferente operação de inserção;
- **findOne(ID)** - método de consulta que recebe um parâmetro ID (identificador) e retorna uma única entidade conforme a igualdade com o parâmetro;
- **exists(ID)** - pesquisa no banco de dados, via parâmetro ID, se a entidade existe ou não e retorna um booleano;
- **findAll( )** - retorn a toda s as entidades. Consulta idêntica a um: **select * from;**
- **findAll(Iterable<ID>)** - recebe um a lista de parâmetros do tipo ID e retorna todas as entidades referentes aos parâmetros;
- **count( )** - retorna a quantidadede entidades em uma tabela;
- **delete(ID)** - exclui uma entidadea partir do identificador;
- **delete(T)** - exclui uma entidade na tabela. O critério é um objeto referente à classe de entidade da tabela que sofrerá a ação;
- **delete(Iterable<? extends T>)** - recebe como critério uma lista de entidades e exclui as entidades referentes no banco de dados;
- **deleteAll( )** - realiza uma exclusão em massa sem qualquer critério. Este método vai remover todas as entidades de uma tabela.

A última interface da relação é a **PagingAndSortingRepository,** a qual tem como função oferecer consultas para ordenação e paginação de dados:

- **findAll(Sort)** - este método retorna todas as entidades de forma ordenada por ascendência ou descendência. O critério **Sort** é o objeto que vai conter a ordenação desejada;
- **findAll(Pageable)** - a consulta retorna todas as entidades conforme o critério **Pageable,** um objeto que tem como finalidade paginar os dados de retorno.

Até aqui foram abordadas as interfaces do SpringData, as quais são utilizadas como base para o projeto Spring Data JPA.

# 1.1. A Interface JpaRepository

O Spring Data JPA é um subprojeto do Spring Data. Desta forma, as três interfaces básicas, já analisadas, são também acessíveis via Spring Data JPA.

Além disso, existe outra interface, chamada **QueryByExampleExecutor**, que também é estendida pela JpaRepository.

Assim, ao estender **JpaRepository** nos repositórios de um projeto, se tem acesso a uma larga quantidade de métodos já predefini dos para uso.

Pode-se notar também, que muitos desses métodos possuem o mesmo nome. Porém, a grande diferença entre eles está no tipo de retorno e na lista de argumentos.

Estas diferenças possibilitam que o desenvolvedor possa escolher entre formas distintas de lidar com suas operações de CRUD.

A interface **QueryByExampleExecutor** fornece métodos que são voltados apenas para consultas.

Os argumentos desses métodos são objetos do tipo **Example,** proveniente do pacote **org.springframework.data.domain.Example.** Mas a **Example** pode também trabalhar em parceria com a classe **ExampleMatcher.**

Esta última fornece um pequeno grupo de métodos para definir alguns tipos de critérios em uma consulta. Esses critérios podem, por exemplo, suprimir a diferença entre letras maiúsculas e minúsculas, selecionar entidades a partir de uma sequência inicial ou final de caracteres, selecionar entidades simulando o **between** do SQL, entre outros.

Além da interface **JpaRepository** ter acesso a diversos métodos de outras interfaces, veja que ela tem suas próprias assinaturas de métodos.

Esta interface é considerada mais sofisticada que, por exemplo, a **CrudRepository.** Isto porque o retorno dos métodos que resultam em listas de entidades são do tipo **java.util.List** e não **java.lang.Iterable.**

A seguir veja a função de alguns dos métodos da **JpaRepository:**

- **findAll** - os métodos nomeados como findAll são responsáveis por retornar uma lista de entidades. Existe uma variação deles nesta interface, onde um se difere do outro pelo tipo de parâmetro que é adicionado com o parte dos critérios da consulta;
- **save(Iterable<S>)** - este método recebe como parâmetro uma lista de entidades e as insere no banco de dados. Após estas operações, ele retorna um objeto **java.util.List** contendo as entidades já em estado persistente;
- **flush()** - este método tem como finali dade forçar uma atualização ou sincronizar o cache de primeiro nível do Hibernate com o que está realmente no banco de dados;
- **saveAndFlush(S)** - use este método para garantir que, ao inserir uma nova entidade no banco de dados, neste mesmo momento o cache de primeiro nível do Hibernate seja sincronizado com o banco de dados;
- **deleteInBacth(Iterable<T>)** - o método recebe como parâmetro uma lista de entidades para a exclusão na tabela. Porém, a operação de exclusão é executada em lote, ou seja, em uma única instrução de **delete**;
- **deleteAllinBatch()** - todas as entidades de uma tabela serão excluídas em uma única instrução de **delete;**
- **getOne(ID)** - retorna a referência da entidade conforme o identificador passado como critério na consulta. Este método tem a mesma função do **getReference( )** da especificação JPA.

O que era necessário conhecer sobre o grupo as interfaces do tipo Repository foi abordado neste capitulo. E a partir dos próximos capítulos será demonstrado como trabalhar com os recursos provenientes dessas interfaces.

# 1.2. Lidando com Repositórios

Como já visto, as interfaces do tipo Repository fornecem uma grande variedade de métodos, mas para usá-los, é necessário criar repositórios específicos.

Onde cada repositório, ou interface do tipo Repository, que se adiciona em um projeto com Spring Data JPA deve ter uma classe de entidade vinculada.

No capítulo anterior, foram abordadas duas classesde entidades, a Contato e a Endereco. Então, inicialmente, será criada a interface ContatoRepository, conforme a **Listagem 4.1.**

**LISTAGEM 4.1:** Repositório ContatoRepository

```
package com.curso.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.curso.entity.Contato;

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

Já se sabe que cada repositório deve estar ligado a uma classe de entidade. Esta ligação é feita de forma simples. Ao estender a interface **JpaRepository** ela deve ser configurada com dois *Generics*. O primeiro é a classe de entidade e é ela quem vai criar o vínculo com o repositório. O segundo é o tipo de dado que representa o **id** da classe de entidade. Ao criar essa ligação, o Spring Data JPA vai automaticamente conhecer os atributos da entidade, assim, ele será capaz de executar as operações de CRUD apenas a partir da assinatura dos métodos.Em **ContatoRepository** não há ainda nenhuma assinatura de método, mas já é perfeitamente possível executar operações de CRUD a partir dos métodos de JpaRepository. Mas, para isso, é preciso ter uma classe que faça, via injeção de dependências, o acesso ao *bean* ContatoRepository. Então, na **Listagem 4.2** temos a classe ContatoService, a qual vai fazer as chamadas aos métodos do repositório.

**LISTAGEM 4.2:** Classe ContatoService

```
package com.curso.service;

import com.curso.entity.Contato;
import com.curso.repository.ContatoRepository;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.Optional;
import java.util.List;

@Service
public class ContatoService {

   @Autowired
   private ContatoRepository repository;

   public void salvar(Contato contato) {
      repository.save(contato);
   }

   public <Optional>Contato buscaPorId(Long id) {
      return repository.findById(id);
   }

   public List<Contato> buscarTodos() {
      return repository.findAll();
   }
}
```

Analisando o código apresentado é necessário ressaltar dois aspectos. O primeiro é o uso da anotação **@Service,** a qual transforma a classe **ContatoService** em um *bean* gerenciado pelo SpringFramework.O segundo é a anotação **@Autowired**, que realiza o processo de injeção de de pendências entre **ContatoService** e **ContatoRepository**. Ou seja, sempre que em **ContatoService** for necessário ter um objeto inicializado de **ContatoRepository**, o Spring vai fornecê-lo.Por este motivo, o acesso aos métodos **save(), findById() e findAll()** são realizados diretamente pela variável **repository** sem o uso de qualquer instância.

Agora, preste atenção em cada um desses métodos e veja a função de cada um deles.O método **salvar()** recebe como parâmetro um objeto do tipo **Contato**. Suponha que este objeto esteja vindo de um formulário qualquer presente em uma página JSP. Assim, o código interno de salvar() executa o método save() do repositório e uma entidade contato é inserida no banco de dados.Veja que não existiu nenhuma complexidade de código em toda essa operação, por exemplo, abrir sessão, abrir transação, comitar ou mesmo fechar a sessão ou a transação. Essas responsabilidades ficam por conta do Spring Data, sem que você precise se preocupar com elas.Uma particularidade do método **save()** é que ele executa tanto uma operação de inserção como de atualização. Automaticamente o Spring Data JPA vai identificar se você está alterando um registro já existente na tabela ou inserindo um novo registro. Ele identifica se a operação deve ser um **save** ou um **update** pelo **id** do objeto. Se o objeto **contato** não tem um **id**, significa que ele é uma instrução de inserção. Mas caso haja no objeto um **id** presente, a operação será de atualização.Já o método **buscarPorId()**, recebe como parâmetro um valor do tipo **Long** e a partir do método **findById()**, presente em **CrudRepository**, a consulta é executada para localizar um contato coma chave primária igual ao valor do parâmetro **id** informado.Novamente, não houve complexidade alguma, apenas uma chamada simples ao método **findById()**. Por fim, no método **buscarTodos()** a consulta utilizada é a **findAll()** , de **JpaRepository**, a qual vai retornar uma lista com todas as linhas da tabela **Contatos**.Até aqui foi possível ver como é simples trabalhar com operações de CRUD já presentes nas interfaces que o Spring Data e o Spring Data JPA fornecem.

Mas em alguns casos, vai ser necessário ter algum tipo a mais de código para executar algum as operações. Um exemplo disso é se você pretende usar os métodos que recebem como parâmetro um objeto do tipo **Example**, conforme demonstrado na **Listagem 4.3.**

**LISTAGEM 4.3:** Consultas com Example

```
@Service
public class contatoService{

   // código anterior omitido nesta listagem

   public List<Contato> buscarPoridadeQBE(Integer idade){

      Contato contato = new Contato();
      contato.setIdade(idade);
      Example<Contato> example = Example.of(contato);

      return repository.findAll(example);
   }

   public List<Contato> buscarPorNomeQBE(String nome) {

      Contato contato = new Contato();
      contato.setNome(nome);

      ExampleMatcher matcher = ExampleMatcher
         .matching()
         .withIgnoreCase()
         .withStringMatcher(ExampleMatcher.StringMatcher.ENDING);

      Example<Contato> example = Example.of(contato, matcher);

      return repository.findAll(exmaple);
   }
}
```

O código apresentado possui dois métodos, os quais usam objetos Example para criar algumas regras que devem estar presentes nas consultas. Por esse motivo, é necessário trabalhar com um pouco mais de código que nas consultas anteriores.

Vamos começar analisando a consulta **buscarPorIdadeQBE().** Veja que ela recebe como parâmetro um objeto **Integer** que faz referência ao valor de uma idade.

Internamente, foi adicionada uma sequência de instruções para criar a regra da consulta. Primeiro, foi adicionada uma instância da classe de entidade **Contato** e via método **setIdade()** foi inserida nesta instância a idade recebida via parâmetro.Em seguida, um objeto **Example** é inicializado via método estático **of()**, que recebe como parâmetro a variável **contato**. Como na variável **contato** foi inserida a idade, a consulta vai saber que ela está sendo executada pelo critério de idade. Neste caso, vai ser retornada uma lista de entidades que correspondem ao valor do parâmetro **idade**.Agora, conferindo o método **buscarPorNomeQBE()**, note que foi usado como parâmetro um **String** que representa um nome. Novamente, uma instância de **Contato** é criada e a ela é adicionado o parâmetro **nome** via método **setNome()**. E desta vez, foi utilizada a classe **ExampleMatcher** para criar as regras da consulta.Como dito anteriormente, esta classe possui um pequeno grupo de métodos para adicionar as regras desejadas em uma consulta. O método inicial é **matching()** e, é ele que vai fornecer o acesso aos métodos que realmente formam as regras. O primeiro critério fica por conta do método **withIgnoreCase()**, que tem a função de desconsiderar se os nomes, na coluna **nome** da tabela, estão com letras maiúsculas ou minúsculas, já que em alguns bancos de dados essa diferença seria fundamental para a consulta.Já o método **withStringMatcher()** recebe como parâmetro uma informação para a consulta analisar apenas o final da *string* na coluna **nome** da tabela. Por exemplo, se o valor davariável nome for "Silva", a consulta retornará todas as linhas que na coluna nome tenha "Silva" como valor final.

Ou seja. "Andriele da Silva","Anelise Mattos da Silva" ou "Anete Silva". Mas não retornaria contatos com o nome "Maria Silva de Souza", "Paula da Silva Ferreira" ou "Aniele Silva Mattos". Isto porque, os nomes não são finalizados coma palavra "Silva".

Outro caso em que é preciso um pouco de código adicional é quando se trabalha com ordenação. Para ordenar os resultados da consulta é necessário incluir um objeto do tipo **Sort** contendo a ordenação desejada e por qual atributo ela será realizada. Na **Listagem 4.4,** há um exemplo de como ordenar os resultados pelo atributo **idade** de forma ascendente (0 - 9).

**LISTAGEM 4.4:** Consulta com Ordenação

```
@Service
public class ContatoSerivce {

   // côdigo anterior omitido nesta listagem

   public List<Contato>buscarTodosPorIdadeAsc(){

      Sort sort = new Sort(Direction.ASC, "idade");
      return repository.findAll(sort);
   }
}
```

Observe no código-fonte que o método **buscarTodosPoridadeAsc()** possui uma instância da classe **Sort**. Esta instância tem dois parâmetros adicionados no método construtor. O primeiro é o **Direction.ASC,** que de fine a ordenação dos resultados de forma ascendente. Essa ordenação é realizada via atributo **idade,** informado como um **String** na segunda posição do construtor. Esse segundo parâmetro deve sempre referenciar o nome do atributo da classe de entidade e não o nome da coluna no banco de dados. É muito importante memorizar esta informação.

Coma regra de ordenação pronta, basta adicionar ao método **findAll()** a variável **sort**. Assim, a consulta vai retornar uma lista de contatos ordenada pela idade de forma ascendente, ou seja, da menor para a maior idade.

Caso queira executar de forma que o retorno seja descendente (9 - 0), basta substituir **Direction.ASC** por **Direction.DESC**. Tanto a classe **Sort** como o *enum* **Direction** são encontradas *no* pacote **org.springframework.data.domain.**

# 1.3. Paginando Resultados

A paginação é um conceito ou técnica que pode ser usado tanto no lado cliente da aplicação como nolado servidor. Com o Spring Data JPA, é facilmente possível criar um processo de paginação no lado servidor.

A vantagem de utilizar este conceito está diretamente ligada ao desempenho de uma aplicação. Por exemplo,
em uma tabela com 1000 registros, uma consulta que retorna todas as linhas de uma única vez, vai ter um desempenho inferior a uma consulta que retorne apenas15 linhas.

Então, o que a paginação faz**,** é dividir a totalidade dos registros em pequenos blocos ou páginas. Assim, ao invés de realizar uma única consulta que retorne 1000 registros de uma só vez, a consulta é executada diversas vezes, trazendo em cada uma delas uma página diferente de registros.

Trabalhar com paginação no lado servidor muitas vezes é algo complexo, porque depende de uma lógica que deve ser bem elaborada e ainda é necessário lidar com uma grande quantidade de informações que devem ser levadas até o lado cliente da aplicação.

Já no Spring Data JPA. o processo de paginação de resultados é muito mais simples. Isto porque, existe um grupo de interfaces e classes que juntas, formam um objeto bem completo, com uma quantidade interessante de métodos para serem usados em vários tipos de paginações como: de tabelas, de páginas Web, de slides de imagens, entre outros.

No Spring Data JPA já há um método pronto para o uso de paginação, que é o **findAll()**. Se você lembrar, existe uma variedade de assinaturas de métodos com este nome entre as interfaces do tipo Repository. Mas, o que é usado para paginação é o seguinte:

```
Page<T> findAll(Paegable pageable);
```

O método pertence à interface **PagingAndSortingRepository** *e merece* uma atenção especial ao tipo de retorno **(Page)** *e* ao argumento **(Pageable).**

São esses objetos que vão proporcionar o uso de paginação, mas ambos são interfaces e não podem ser instanciados. Assim, a classe **PageRequest** fica com a responsabilidade da instância e de receber os dados necessários para o uso de paginação, os quais são: o número da página atual, o número *de* elemento s da página e caso necessário uma regra para ordenação.

E estes dados são passados via método construtor de **PageRequest** , a qual fornece três assinaturas, que são:

```
PageRequest(int page, int size)
PageRequest(int page, int size, Direction direction, String... properties)
PageReq uest(int page, int size, Sort sort)
```

Agora , confira na **Listagem 4.5** o método paginarResultados() que será usado para exemplificar o recurso de paginação.

**LISTAGEM 4 .5:** Método para Paginação

```
@Service
public class ContatoService {

public Page<Contato> paginaRresultados() {

   Sort sort = new Sort(Direction.ASC, "nome");
   Pageable pageable = new PageRequest(0,3,sort);
   return repository.findAll(pageable);
   }
}
```

Observe que neste método há uma instância da classe **PageRequest** atribuída auma variável do tipo **Pageable,** que é o tipo de parâmetro esperado pelo método **findAl().**

No construtor de **PageRequest** foram adicionadostrês valores:

- O primeiro é o número da página que vai ser localizada, neste caso, a página zero. A página inicial sempre será zero, assim como, qualquer lista em Java;
- O segundo é a quantidade máxima de elementos que será retornada nesta página, neste caso serão três elementos (entidades);
- O terceiro é a forma como os resultados serão ordenados. Sobre isso, é importante ressaltar que, se a tabela tem 10 elementos, primeiro a ordenação é realizada nestes 10 elementos, e então, os três primeiros elementos já ordenados farão parte da página 0 (zero), os próximos três da página 1 (um) e assim até a página com o décimo elemento.

Agora a variável **pageable** já contém as regras da paginação e pode ser adicionada como parâmetro do método **findAll()**.

Entretanto, ainda falta analisar o retorno desta consulta, ou seja, o que o objeto **Page** vai fornecer a você. Na verdade ele não retorna diretamente a lista de entidades, mas sim, uma lista de métodos com informações importantes para você lidar com a paginação entre o lado servidor e cliente da sua aplicação.

Veja quais são estes métodos e a função de cada um descrito a seguir:

- **previousPageable()** - vai fornecer dados específicos sobre a página anterior daquela que está sendo retornada. Antes de usar este método é importante ter certeza que existe uma página anterior, para isso, use o método **hasPrevious().** Isto porque, caso a página atual seja a página zero, não haverá uma página anterior e uma exceção será lançada em caso de acesso;
- **getNumber()** - retorna o número da página atual;
- **nextPageable()** - fornece acesso aosdados da próxima página. Use o método **hasNext()** para confirmar que existe uma próxima página
- e assim, evitar uma exceção, caso a página atual seja a última;
- **get NumberOf Elements()** - retorna o número de elementos existentes na página atual;
- **getSize()** - retorna o número máximo de elementos configurados por página no processo de paginação;
- **getTotalElements()** - este método retorna a quantidade total de elementos existentes na tabela ou em uma consulta sem a paginação;
- **getTotalPages()**- retorna a quantidade total de páginas envolvidas no processo de paginação;
- **getContent()** - este método retorna um objeto do tipo **java.util.List** contendo a lista de entidades retornadas para a página atual;
- **hasContent()** - retorna verdadeiro se existem elementos na lista de entidades e falso em caso negativo;
- **getSort()** - o retorno deste método contém as informações sobre as regras de ordenação da consulta;
- **isFirst() e isLast()** • estes métodos retornam verdadeiro ou falso no teste para verificar se a página atual é a primeira ou a última.

Com estas informações sobre paginação, você será capaz de criar sua própria tabela de dados ou mesmo usar os recursos de tabelas fornecidas pelo Bootstrap, jQuery Datatable, Dandelion, ou qualquer outro, de forma muito mais fácil.