Desenvolvimento para Servidores-II

Queries no Spring Data

Neste tópico abordaremos a criação de queries no Spring Data e também a paginação.

Prof. Ciro Cirne Trindade

Queries

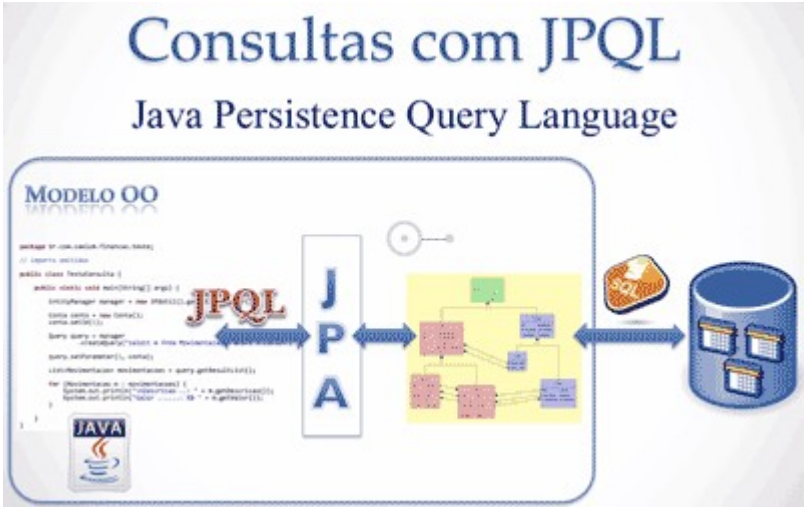
- O Spring Data permite criar *queries* de três formas diferentes
  - Através de *named queries* JPQL
  - Através da anotação `@Query`
  - Através de *query methods*

JPQL (Java Persistence Query Language) (1/3)

- Por padrão, todo banco de dados relacional aceita a SQL para realizar consultas e manipulações em modelos relacionais
- Quando adotamos o JPA, uma das coisas que buscamos é nos distanciar o máximo possível do modelo relacional e focar exclusivamente no modelo orientado a objetos e seus estados

JPQL (Java Persistence Query Language) (2/3)

- O JPQL é uma linguagem de consulta, assim como o SQL, porém orientada a objetos
- Isso significa que quando estivermos pesquisando dados, não consideramos nomes de tabelas ou colunas, e sim entidades e seus atributos



JPQL (Java Persistence Query Language) (3/3)

- Vantagens
  - Facilidade de escrita: por estar mais próxima ao modelo orientado a objetos
  - Portabilidade: podemos trocar a implementação do JPA (de Hibernate para EclipseLink, por exemplo)
  - Padronização: a forma de escrita de JPQL não muda, não importada o SGBD que você utilize

Sintaxe das *queries* JPQL (1/8)

- Após a cláusula **`select`** aparece um identificador que representa o objeto ou objetos que a *query* retorna
- Identificadores quaisquer são associados a entidades de tal forma que eles possam ser referenciados em qualquer lugar na *query*
- Forma geral:
  - `SELECT identicador FROM Entidade identificador WHERE identificador.atributo=valor`

Sintaxe das *queries* JPQL (2/8)

- A cláusula opcional **`where`** define critérios para os resultados da *query*
- Por exemplo:
  - `select c from Conta c` **`where`** `c.numero="4321-0"`

Sintaxe das *queries* JPQL (3/8)

- Palavras-chave em expressões JPQL são *case-insensitive*, mas entidades, identificadores e atributos não são
- Por exemplo, a expressão anterior também poderia ser escrita assim:
  - `SELECT c FROM Conta c WHERE c.numero="4321-0"`
- Mas não assim:
  - `SELECT c FROM Conta c WHERE c.NUMERO="4321-0"`

Sintaxe das *queries* JPQL (4/8)

- JPQL usa uma sintaxe *SQL-like* para os critérios da *query*
  - Os operadores lógicos **`and`** e **`or`** combinam múltiplos critérios
  - O operador = testa a igualdade, <> testa a diferença e os seguintes operadores são suportados para comparações **numéricas**: >, >=, <, <=
  - Por exemplo:
    - `select c from Conta c where c.id` **`>=`** `1` **`and`** `c.id` **`<=`** `5`

Sintaxe das *queries* JPQL (5/8)

- JPQL também inclui as seguintes cláusulas:
  - **[NOT] BETWEEN**: abreviação para expressar que um valor está entre 2 outros valores
    - As 2 *queries* a seguir são sinônimas:
      - `SELECT c FROM Conta c WHERE c.id >= 1 AND c.id <= 5`
      - `SELECT c FROM Conta c WHERE c.id BETWEEN 1 AND 5`

Sintaxe das *queries* JPQL (6/8)

- **[NOT] LIKE**: faz uma comparação de strings com suporte a *wildcard*
  - O caractere especial '_' no parâmetro significa qualquer caractere e o caractere especial '%' significa qualquer sequência de caracteres
  - Por exemplo:
    - `SELECT pf FROM PessoaFisica pf WHERE pf.nome LIKE '%José%'`
    - `SELECT pf FROM PessoaFisica pf WHERE pf.nome LIKE 'I__'`

Retorna objetos com atributo **nome** contendo "José"

Retorna objetos com atributo **nome** começando com 'I' seguido de 2 caracteres quaisquer

Sintaxe das *queries* JPQL (7/8)

- **[NOT] IN**: especifica que o atributo deve ser igual a um dos elementos de uma lista
  - As 2 queries a seguir são sinônimas:
    - `SELECT pf FROM PessoaFisica pf WHERE pf.endereco IN ('Santos', 'São Paulo', 'Natal')`
    - `SELECT pf FROM PessoaFisica pf WHERE pf.endereco = 'Santos' OR pf.endereco = 'São Paulo' OR pf.endereco = 'Natal'`

Sintaxe das *queries* JPQL (8/8)

- **`[INNER] JOIN, LEFT [OUTER] JOIN`**: a cláusula `JOIN` do JPQL funciona de forma semelhante ao `JOIN` do SQL, entretanto, opera sobre entidades que estejam relacionadas
  - Vamos supor que queremos todas as movimentações da categoria Lazer
    - `select m from Movimentacao m` **`join`** `m.categorias c on c.nome = 'Lazer'`

Parâmetros de *queries*

- Queries podem ter parâmetros
  - Os parâmetros podem ser
    - Posicionais
      - ? seguido de um número
      - `select c from Conta c where c.numero=?1`
    - Nomeados
      - : seguindo de um nome
      - `select c from Conta c where c.numero=:pNumero`

Funções JPQL (1/5)

- **AVG**: média
  - Exemplo: obter a média de todas as movimentação de um tipo (entrada ou saída) de uma conta
    - `select avg(m.valor) from Movimentacao m where m.conta = ?1 and m.tipo = ?2`

Funções JPQL (2/5)

- **COUNT**: número de ocorrências
  - Exemplo: obter o número de movimentações de uma conta
    - `select count(m) from Movimentacao m where m.conta = ?1`

Funções JPQL (3/5)

- **MAX**: valor máximo
  - Exemplo: obter o valor da maior movimentação de um tipo de uma conta
    - `select max(m.valor) from Movimentacao m where m.conta=?1 and m.tipo=?2`

Funções JPQL (4/5)

- **MIN**: valor mínimo
  - Exemplo: obter o valor da menor movimentação de um tipo de uma conta
    - `select min(m.valor) from Movimentacao m where m.conta=?1 and m.tipo=?2`

Funções JPQL (5/5)

- **SUM**: soma
  - Exemplo: obter a soma das movimentações de um tipo de uma conta num período
    - `select **sum(m.valor)** from Movimentacao m where m.conta = ?1 and m.tipo = ?2 and m.data between ?3 and ?4`

Named Queries JPQL (1/5)

- Uma *named query* (consulta nomeada) é uma consulta predefinida que criamos e associamos a uma entidade
- Para isso, usamos a anotação **`@NamedQuery`**

Named Queries JPQL (2/5)

- A anotação `@NamedQuery` possui 2 elementos obrigatórios:
  - `name`: nome da consulta
  - `query`: string com a consulta JPQL
- Por exemplo, para definir uma consulta para devolver todas as contas de uma agencia faríamos:

```
@Entity
@NamedQuery(name="Conta.listarPorAgencia",
 query="select c from Conta c where c.agencia=?1")
public class Conta implements Serializable {
   ...
```

Named Queries JPQL (3/5)

- Para associar várias consultas nomeadas a uma entidade é necessário empacotá-las em uma anotação **@NamedQueries**
- Por exemplo:

```
@Entity
@NamedQueries({
   @NamedQuery(name="Conta.listarTodas",
       query="select c from Conta c order by c.numero"),
   @NamedQuery(name="Conta.listarPorAgencia",
       query="select c from Conta c where c.agencia=?1")
})
public class Conta implements Serializable {
...
```

Named Queries JPQL (4/5)

- Para usar uma *named query* JPQL em um método do repositório, basta que o método tenha nome da *query*
- Vamos supor as seguintes *named queries* na classe `Conta`

```
@NamedQueries({
   @NamedQuery(name = "Conta.listarPorAgencia",
      query = "select c from Conta c where c.agencia=?1"),
   @NamedQuery(name = "Conta.listarPorAgenciaESaldo",
      query = "select c from Conta c where c.agencia=?1 and"
              + " c.saldo between ?2 and ?3"),
   @NamedQuery(name = "Conta.listarPorNomeCliente",
      query = "select c from Conta c join Cliente cc on"
              + " cc.conta = c where cc.nome like ?1")
})
```

Named Queries JPQL (5/5)

- Para criar métodos na interface `ContaRepository` associados a essas *named queries* bastaria fazer

```
@Repository
public interface ContaRepository extends
                        JpaRepository<Conta, Long> {

   List<Conta> listarPorAgencia(Integer agencia);

   List<Conta> listarPorAgenciaESaldo(Integer agencia,
                                Float from, Float to);

   List<Conta> listarPorNomeCliente(String nome);

}
```

ContaService.java

- Acrescentar esses métodos

```
public List<Conta> listarPorAgencia(Integer agencia) {
    return repository.listarPorAgencia(agencia);
}

public List<Conta> listarPorAgenciaESaldo(
                Integer agencia, Float from, Float to) {
    return repository.listarPorAgenciaESaldo(agencia,
                                             from, to);
}

public List<Conta> listarPorNomeCliente(String nome) {
    return repository.listarPorNomeCliente('%' + nome
                                           + '%');
}
```

ContaController.java (1/2)

- Acrescentar esses métodos

```
@GetMapping(value = "/agencia/{agencia}")
public ResponseEntity<List<Conta>> getByAgencia(
            @PathVariable("agencia") Integer agencia) {
   return ResponseEntity.ok(
                  service.listarPorAgencia(agencia));
}
```

ContaController.java (2/2)

```
@GetMapping(value = "/agencia/{agencia}/{from}/{to}")
public ResponseEntity<List<Conta>> getByAgenciaESaldo(
        @PathVariable("agencia") Integer agencia,
        @PathVariable("from") Float from,
        @PathVariable("to") Float to) {
   return ResponseEntity
      .ok(service.listarPorAgenciaESaldo(agencia, from,
                                         to));
}

@GetMapping(value = "/cliente/{nome}")
public ResponseEntity<List<Conta>> getByNomeCliente(
              @PathVariable("nome") String nome) {
   return ResponseEntity
      .ok(service.listarPorNomeCliente(nome));
}
```

@Query (1/3)

- Também é possível associar uma *query* JPQL a um método do repositório através da anotação `@Query`

`@Query` tem prioridade sobre *named queries*

```
@Repository
public interface ContaRepository extends
                                    JpaRepository<Conta, Long> {
    @Query("select c from Conta c where c.agencia=?1")
    List<Conta> listarPorAgencia(Integer agencia);

    @Query("select c from Conta c where c.agencia=?1 and "
                                + "c.saldo between ?2 and ?3")
    List<Conta> listarPorAgenciaESaldo(Integer agencia,
                                    Float from, Float to);

    @Query("select c from Conta c join Cliente cc on "
                    + " cc.conta = c where cc.nome like %?1%")
    List<Conta> listarPorNomeCliente(String nome);
}
```

@Query (2/3)

- Se a *query* tem parâmetros nomeados, então é necessário usar a anotação `@Param` para identificá-los

```
@Query(
    "select c from Conta c where c.agencia=:pAgencia"
     + " and c.saldo between :pInicio and :pFinal")
   List<Conta> listarPorAgenciaESaldo(
           @Param("pAgencia") Integer agencia,
           @Param("pInicio") Float from,
           @Param("pFinal") Float to);
```

@Query (3/3)

- A anotação `@Query` permite a execução de queries nativas definindo a flag `nativeQuery` como `true`

```
@Query(
 "select * from tb_conta where nr_agencia=:pAgencia"
  + " and vl_saldo between :pInicio and :pFinal",
  nativeQuery = true)
List<Conta> listarPorAgenciaESaldo(
        @Param("pAgencia") Integer agencia,
        @Param("pInicio") Float from,
        @Param("pFinal") Float to);
```

Query methods (1/5)

- Baseados na API Criteria do JPA
- Os nomes dos métodos não mapeados em *queries*
- Normalmente o nome do método tem o seguinte formato
  - `find`**`By`**`AtributoPalavrachave`

Query methods (2/5)

Palavras-chaves Suportadas		
Palavra-chave	Exemplo	JPQL equivalente
And	findByAgenciaAndNumero	… where c.agencia=?1 **and** c.numero=?2
Or	findByAgenciaOrNumero	… where c.agencia=?1 **or** c.numero=?2
Between	findByIdBetween	… where c.id **between** ?1 and ?2
LessThan	findBySaldoLessThan	… where c.saldo **<** ?1
LessThanEqual	findBySaldoLessThanEqual	… where c.saldo **<=** ?1
GreaterThan	findBySaldoGreaterThan	… where c.saldo **>** ?1

Query methods (3/5)

Palavras-chaves Suportadas		
Palavra-chave	Exemplo	JPQL equivalente
GreaterThanEqual	findBySaldoGreaterThanEqual	… where c.saldo **>=** ?1
Null, IsNull	findByAgencia[Is]Null	… where c.agencia **is null**
NotNull, IsNotNull	findByAgencia[Is]NotNull	… where c.agencia **not null**
Like	findByNumeroLike	… where c.numero **like** ?1
NotLike	findByNumeroNotLike	… where c.numero **not like** ?1
StartingWith	findByNumeroStartingWith	… where c.numero **like** ?1**%**

Query methods (4/5)

Palavras-chaves Suportadas		
Palavra-chave	Exemplo	JPQL equivalente
EndingWith	findByNumeroEndingWith	… where c.numero **like** %?1
Containing	findByNumeroContaining	… where c.numero like **%**?1**%**
OrderBy	findByAgenciaOrderBySaldo	… where c.agencia=?1 **order by** c.saldo
Not	findByAgenciaNot	… where c.agencia **<>** ?1
In	findByAgenciaIn	… where c.agencia **in** ?1
NotIn	findByAgenciaNotIn	… where c.agencia **not in** ?1

Query methods (5/5)

- ContaRepository com *query methods*

```
@Repository
public interface ContaRepository extends
                        JpaRepository<Conta, Long> {

   List<Conta> findByAgencia(Integer agencia);

   List<Conta> findByAgenciaAndSaldoBetween(
               Integer agencia, Float from, Float to);

   @Query("select c from Conta c join Cliente cc on "
        + "cc.conta = c where cc.nome like ?1")
   List<Conta> listarPorNomeCliente(String nome);

}
```

Paginação no Spring (1/6)

- Quando sua base de dados tem muitos registros, não é uma boa prática trazê-los todos de uma vez
- Ao invés disso, trazemos os dados paginados
- Para adicionar suporte a paginação aos seus repositórios, é necessário estender a interface `PagingAndSortingRepository<T,ID>`

`JpaRepository` estende esta interface

Paginação no Spring (2/6)

- A interface `PageAndSortingRepository` define o método
  - `public` **`Page`**`<T> findAll(`**`Pageable`**` pageable)`
    - Devolve uma página (`Page`) de entidades que satisfazem às restrições de paginação providas pelo objeto `Pageable`

Paginação no Spring (3/6)

- Vamos definir um método na camada de serviço (ContaService.java) que devolva objetos do tipo `Conta` paginados

```
public Page<Conta> findAll(Pageable pageable) {
    return repository.findAll(pageable);
}
```

Paginação no Spring (4/6)

- Na camada de controle (ContaController.java) vamos definir um *end point* que devolva uma reposta paginada

```
@GetMapping(value = "/page")
public ResponseEntity<Page<Conta>> getAll(Pageable pageable)
{
    return ResponseEntity.ok(service.findAll(pageable));
}
```

A paginação *default* é de 20 registros por página

Paginação no Spring (5/6)

- JSON de resposta


```
{
    "content": [
        {
            "id": 4,
            "agencia": 431,
            "numero": "1234-X",
            "saldo": 4000.0
        },
        ...
    ],
    "pageable": {
        "sort": {
            "sorted": false,
            "unsorted": true,
            "empty": true
        },
        "pageNumber": 0,
        "pageSize": 20,
        "offset": 0,
        "paged": true,
        "unpaged": false
    },
    "last": true,
    "totalPages": 1,
    "totalElements": 3,
    "sort": {
        "sorted": false,
        "unsorted": true,
        "empty": true
    },
    "first": true,
    "size": 20,
    "number": 0,
    "numberOfElements": 3,
    "empty": false
}
```

Paginação no Spring (6/6)

- Para mudar o tamanho da página, acrescente um parâmetro **`size`** na URL
  - GET: http://localhost:8080/contas/page?**size=5**
- Paga recuperar uma página específica, acrescente um parâmetro page
  - GET: http://localhost:8090/contas/page?size=5&**page=1**
- Para ordenar, acrescente um parâmetro **`sort`**
  - GET: http://localhost:8090/contas/page?size=5&page=1&**sort=saldo**

Referenciando uma *stored procedure*

- A forma mais usual de referenciar uma *stored procedure* é usando a anotação `@Procedure` no repositório
- Vamos supor a existência da seguinte *stored procedure*

```
CREATE PROCEDURE COUNT_ACCOUNTS_BALANCE(IN saldo_in FLOAT,
                                        OUT contas INT)
BEGIN
    SELECT COUNT(*) FROM tb_conta WHERE vl_saldo >= saldo_in;
END
```

Referenciando uma *stored procedure*

- Há 2 formas diferentes e equivalentes de referenciar essa *stored procedure*
  - Definindo um método com o mesmo nome da *stored procedure*

```
@Procedure
int COUNT_ACCOUNTS_BALANCE(Float saldo);
```

  - Definindo o nome da *stored procedure* como um elemento da anotação `@Procedure`

```
@Procedure("COUNT_ACCOUNTS_BALANCE")
int countContasPorSaldo(Float saldo);
```

Referência

- SPRING. Query Methods. Disponível em: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods