Transcrição

Agora que já aprendemos a fazer consultas com a JPA, vamos aprofundar um pouco os conhecimentos das aulas anteriores estudando outros recursos de consultas. Nós havíamos criado o método `buscarTodos()` para carregar todos os produtos, que é um *select* sem filtro, isto é, carrega todos os registros do banco de dados. Mas, eventualmente, podemos querer limitar, criar um filtro com algum parâmetro.

Então, vamos criar um novo método chamado `buscarPorNome()`. Supondo que queremos buscar determinados produtos que tenham um determinado nome, nós receberemos, como parâmetro, qual é esse nome.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p";
        return em.createQuery(jpql, Produto.class).getResultList();

    }
```

COPIAR CÓDIGO

Agora, nossa *query* será um pouco diferente, não faremos mais `"SELECT p FROM Produto p"`, porque queremos filtrar. Para isso - semelhante ao SQL - adicionaremos, depois do `FROM Produto p`, `WHERE p.` e o nome do atributo, não da coluna na tabela, que, no caso, é `nome` (e, por coincidência, o mesmo nome da coluna).

Portanto, temos `"SELECT p FROM Produto p WHERE p.nome ="`, e precisamos passar o parâmetro que está chegando no método para essa parte. Uma maneira de fazer isso é adicionando dois pontos, para dizer à JPQL que passaremos um parâmetro

dinâmico na *query*, e, depois, dando um apelido para esse parâmetro. No nosso caso, chamaremos de `nome`, mas poderia ser qualquer outro nome.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
        return em.createQuery(jpql, Produto.class).getResultList();
```

COPIAR CÓDIGO

Antes de disparar essa *query*, temos que substituir o parâmetro que está em: `"SELECT p FROM Produto p WHERE p.nome = :nome"`, pelo que veio em: `buscarPorNome(String nome)`. Para isso, antes de chamar o `.getResultList()`, podemos chamar o método `.setParameter()`. Nele, informaremos qual o nome do parâmetro que, no caso, é `"nome"`, e qual é o valor que queremos substituir, que é o `nome` que está chegando em `buscarPorNome(String nome)`.

Portanto, nós substituiremos o parâmetro do método no parâmetro chamado `nome` que está na nossa *query*.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
        return em.createQuery(jpql, Produto.class)
                .setParameter("nome", nome)
                        .getResultList();
```

COPIAR CÓDIGO

Um detalhe importante é que em `setParameter("nome", nome)` nós não usamos dois pontos, como fizemos anteriormente no JPQL. Poderíamos ter quantos parâmetros quiséssemos. Por exemplo, para filtrar por outro parâmetro, poderíamos fazer `AND p.categoria =` e passaríamos o valor. Enfim, poderíamos ter vários parâmetros e usar `AND` ou `OR`, semelhante ao SQL.

Terminado, poderemos fazer uma *query* filtrando por algum atributo. Em `cadastroDeProduto.java`, no método `main()`, ao invés de `buscarTodos()`, vamos usar o `buscarPorNome()`, passar o nome que escolhemos, `"XIAOMI Redmi"`, e conferir se ele

fará a busca. Vamos rodar e olhar o Console. Analisando o *select* no Console, encontraremos "Where produto0_.nome=?", significa que ele filtrou corretamente pelo nome.

Essa maneira de passar o parâmetro é chamada de *Named parameter*, onde passamos o parâmetro pelo nome. Mas há outra forma de fazer que é passando `?1`, isto é, `"SELECT p FROM Produto p WHERE p.nome = ?1"`. E em `.setParameter("nome", nome)`, ao invés de passar um apelido, `"nome"`, passamos o `1`, isto é, `.setParameter(1, nome)`.

Ou seja, é possível ter um parâmetro posicional, com interrogação 1, 2, 3 e assim por diante, já que podemos passar vários parâmetros, cada qual com um número distinto. As duas abordagens funcionam.

Agora, vamos retornar à nossa entidade `Produto.java` e recordar que o `Produto` tem um atributo `Categoria`. Neste caso, `Categoria` é um relacionamento, outra entidade. É possível filtrar um produto pela categoria? Pelo nome, não do produto, mas da categoria? A resposta é sim.

Na classe `ProdutoDao.java` criaremos outro método e chamá-lo de `buscarPorNomeDaCategoria()`, significa que agora filtraremos pelo nome da categoria, não mais do produto.

```
public List<Produto> buscarPorNomeDaCategoria(String nome) {
    String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
    return em.createQuery(jpql, Produto.class)
            .setParameter("nome", nome)
            .getResultList();
}
```

COPIAR CÓDIGO

A nossa *query* mudará um pouco. Como se trata de um relacionamento, precisaremos fazer um *join*, isto é, fazer uma consulta por um *join* com a tabela de Categoria. Então, no JDBC, no SQL puro, seria por *join*. Na JPA, conseguimos encontrar uma maneira simplificada, se recordarmos que se trata de uma consulta orientada a objetos.

Portanto, basta fazer `"SELECT p FROM Produto p WHERE p.categoria.nome = :nome"`, sendo que `categoria` se refere ao relacionamento.

```
String jpql = "SELECT p FROM Produto p WHERE p.categoria.nome = :nome";
```

COPIAR CÓDIGO

A JPA entenderá que a `categoria` é um atributo da classe produto e, neste caso, um relacionamento. Então, ele quer filtrar por um atributo dentro do relacionamento, desta maneira, a JPA automaticamente gerará um *join*, isto é, ela já sabe que deve filtrar pelo relacionamento e faz o *join* automaticamente, evitando que seja necessário fazer manualmente, como seria no SQL.

O JPQL é um SQL simplificado, orientado a objetos, não ao modelo relacional. Vamos verificar se funciona indo na classe `CadastroDeProduto.java`. Nela, em lugar de `buscarPorNome()`, faremos `buscarPorNomeDaCategoria()`. Também precisamos trocar o nome do produto pelo nome da categoria, que é `"CELULARES"`.

```
List<Produto> todos = produtoDao.buscarPorNomeDaCategoria("CELULARES");
```

COPIAR CÓDIGO

Agora vamos rodar e conferir se ele encontrará os produtos. Analisando o Console, notaremos que ele fez o *select* e encontrou o "Xiaomi Redmi". Ele também gerou o *join* automaticamente: "cross join categorias", "where produto0_.categoria_id=categoria1", "and categoria1_.nome=?". Significa que ele está filtrando pelo nome da categoria, que é o relacionamento.

Anteriormente no `persistence.xml`, para o Hibernate imprimir o SQL, tivemos que colocar `property name="hibernate.show_sql" value="true"/>`. Existe outra propriedade parecida com essa, a `"hibernate.format_sql"`, que serve ele *identar* o código SQL quando for imprimir, principalmente em consultas. Portanto, podemos usar essa propriedade para que o Hibernate formate o SQL.

Em `CadastroDeProduto.java`, vamos rodar outra vez a nossa classe `main`. Observando o Console, perceberemos que agora o SQL está "quebrado". Ele faz o *select*, coloca as colunas que serão selecionadas, o *from*, o *join*, o *where*, enfim, com tudo *identado* fica mais fácil de visualizar. Portanto, podemos usar essa propriedade para facilitar a leitura dos comandos SQL quando o Hibernate for acessar o banco de dados.

No *insert* ele também quebrou a linha, dividindo em colunas, *values*, então, fica um pouco mais fácil de visualizar. Espero que tenham gostado da aula e aprendido a fazer consultas com JPQL e a filtrar os resultados com o atributo de um relacionamento.

Para isso, basta navegar, e se a categoria tivesse outro relacionamento, poderíamos continuar navegando: `p.categoria.xpto.nome`, isto é, ponto + nome do atributo, e o Hibernate gerará dois, três, cinquenta ou mais *joins* segundo o necessário para efetuar a consulta.

No próximo vídeo continuaremos discutindo mais detalhes relacionados a consulta. Vejo vocês lá!!