03

Montando as consultas dinâmicas

Transcrição

Como nosso objetivo é conhecer o básico da **API da Criteria**, vamos começar fazendo uma busca simples de todas os produtos cadastrados no banco. Para essa busca com JPQL teríamos o seguinte:

```
select p from Produto p
```

Mas, queremos utilizar **Criteria** e antes de criarmos a **query** em si, precisamos de um objeto que será o responsável pela criação delas - um `CriteriaBuilder`. Para recuperar esse objeto, vamos usar o `EntityManager`. Para isso, adicione o seguinte código:

```
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
```

`CriteriaBuilder` é uma fábrica auxiliar para criar expressões sobre as funções que utilizaremos na busca. A fábrica não executa a `query`, ela apenas ajuda a criá-la.

O `CriteriaBuilder` possui métodos que definem operações da busca, como, por exemplo:

```
equal(), greaterThan(), lesserThan(), like() ...
sum(), max(), min(), avg(), count(), desc(), distinct() ...
```

Com um `CriteriaBuilder` criado, podemos começar a criar um critério de consulta. Inicialmente invocamos o método `createQuery` passando para ele a classe que é o tipo de retorno da nossa consulta. No caso, se buscamos os `Produtos`, devemos colocar como parâmetro `Produto.class`. Assim, recuperaremos um objeto do tipo `CriteriaQuery`. Com isso, temos o seguinte código:

```
CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
```

Utilizando a `CriteriaQuery` descrevemos a busca como algo parecido ao JPQL. Quando escrevermos JPQL, usaremos palavras chaves como `select`, `from`, `where` ou `groupBy`. As mesmas palavras chaves aparecem na `CriteriaQuery`, mas aqui são nomes de `métodos`. Resumindo, encontramos os seguintes métodos na interface `CriteriaQuery`:

```
select
from
where
orderBy
groupBy
having
```

Aqui, vale um outro destaque, o `CriteriaQuery` que é *type safe*, ou seja, ele garante que os objetos retornados por esta consulta serão do tipo definido.

Perceba que agora criamos um critério de busca informando que o tipo de retorno é `Produto`. Se tentarmos criar uma `query` baseado apenas nesta definição, do jeito que está, vamos receber uma `exception` informando que ele não sabe de onde deve ser feita a consulta.

Temos que, explicitamente, informar de `onde` deve ser feito o `select`. Já vimos que existe um método no `CriteriaQuery` chamado `from`. Através deste método, podemos passar a classe que vai servir de base na consulta. Vamos acrescentar o seguinte trecho de código:

```
query.from(Produto.class);
```

Com o critério mais simples criado, podemos agora pedir para o `EntityManager` criar uma `query` (no caso uma `TypedQuery`) baseada nesse critério e executá-la com o método `getResultList` que nos retornará uma lista de objetos do tipo e da origem definidos no critério:

```
TypedQuery<Produto> typedQuery = em.createQuery(query);
typedQuery.getResultList();
```

Podemos verificar que tudo deu certo fazendo uma simples interação na lista e imprimindo alguns dados, como por exemplo, o nome e o preço do produto:

```
for (Produto p : produtos) {
    System.out.println("\Nome ..: " + p.getNome());
    System.out.println("Preco ......: R$ " + p.getPreco());
}
```

Vamos executar esse código e ver como ficará a saída no console do eclipse.

Não podemos esquecer que, por enquanto, estamos listando TODOS os produtos do banco, independente de qual a conta em que eles estão vinculados. Mas, você pode estar se perguntando: E, se em algum momento precisarmos listar apenas os produtos de uma determinada categoria? Ou a soma dos valores de todas os produtos? Com a `Criteria` da JPA isso é bem simples. Vamos testar então!

##Usando Predicates

Agora que sabemos como buscar todos os produtos, vamos colocar alguns filtros na nossa query. Observe o código que temos por enquanto:

```
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
query.from(Produto.class);

TypedQuery<Produto> typedQuery = em.createQuery(query);
typedQuery.getResultList();
```

É interessante colocarmos esse código dentro de uma classe específica, para que isso trabalharemos com os dados do nosso produto. Portanto, podemos utilizar a classe `ProdutoDao` e o método `getProdutos`. Teremos:

```
@Repository
public class ProdutoDao {

    @PersistenceContext
    private EntityManager em;

    public List<Produto> getProdutos(String nome, Integer categoriaId, Integer lojaId) {

        CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
        CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
        query.from(Produto.class);

        TypedQuery<Produto> typedQuery = em.createQuery(query);

        return typedQuery.getResultList();
    }
}
```

Agora, através do método `where`, do `CriteriaQuery`, vamos colocar algumas regras para a busca. Esse método recebe como parâmetro um `Predicate` ou mais. O `Predicate` (aplicável ao `where`), assim como `Expression`, podem ser obtidos através de métodos como o `CriteriaBuilder`. Sendo assim, para dizer que queremos os produtos faremos o seguinte:

```
**(1)** Vamos aproveitar o retorno do método `from` e guardar a raiz (`Root`) da pesquisa:


Root<Produto> root = query.from(Produto.class);
```

(2) A raiz é usada para definir os caminhos (`Path`) até os atributos do objeto. Por exemplo, se quisermos o caminho até o atributo `nome` do produto selecionado, usaremos:

```
public List<Produto> getProdutos(String nome, Integer categoriaId, Integer lojaId) {

    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
    CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
    Root<Produto> root = query.from(Produto.class);

    Path<String> nomePath = root.<String> get("nome");

    TypedQuery<Produto> typedQuery = em.createQuery(query);

    return typedQuery.getResultList();
}
```

ATENÇÃO: Como parâmetro do método `get` passamos o nome do atributo da `Entity` e para garantir o tipo do retorno, colocamos ele antes do método e entre os símbolos "< >".

```
(3)  Como já temos a raiz da pesquisa e o caminho até o atributo que será usado no teste, preci
```

Mas, como comparar o nome? Podemos dizer ao JPA que queremos buscar o produto que tenha o nome exatamente igual ao entrado pelo usuário. O objeto `CriteriaBuilder` possui um método para criar um `Predicate` de igualdade, o método `equal`. Mas só criaremos esse `Predicate` caso o nome não seja vazio. Podemos fazer isso da seguinte forma:

```
public List<Produto> getProdutos(String nome, Integer categoriaId, Integer lojaId) {

    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
    CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
    Root<Produto> root = query.from(Produto.class);

    Path<String> nomePath = root.<String> get("nome");

    if (!nome.isEmpty()) {
        Predicate nomeIgual = criteriaBuilder.equal(nomePath , nome);
    }

    TypedQuery<Produto> typedQuery = em.createQuery(query);

    return typedQuery.getResultList();
}
```

O problema dessa abordagem é que o nome será comparado de uma maneira exatamente igual ao nome contido no bando de dados. Explicando... Não podemos garantir que todo o texto preenchido pelo usuário será igual ao nome de algum produto disponível no banco de dados. O que ocorre, frequentemente, é que o usuário ao buscar algum produto insere o nome de forma incompleta ou parcial. Por exemplo, escreve "ivro" ou "vro" ao em vez de "livro".

E, como não existem produtos chamados "ivro" ou "vro" no banco, a busca não retornará nenhum resultado! Portanto, não podemos buscar por nomes que sejam **estritamente iguais**. Devemos abrir a brecha para alguns possíveis erros de digitação do usuário.

Diante disso, uma solução seria buscarmos determinados produtos utilizando nomes que sejam parecidos. Podemos dizer que queremos buscar por um produto que tenha um nome **parecido** com o que foi inserido pelo usuário. Vamos utilizar o método `like` do `CriteriaBuilder` para fazer isso. Passaremos o nome e um padrão para busca.

```
Path<String> nomePath = root.<String> get("nome");

if (!nome.isEmpty()) {
    Predicate nomeIgual = criteriaBuilder.like(nomePath, "%" + nome + "%");
}
```

Desta forma, estamos dizendo ao banco que mesmo que algum caractere seja adicionado, antes ou depois do nome, por engano, isso não irá acarretar em problemas na busca. :)

Então, passamos o `Predicate` para o método `where`:

```
public List<Produto> getProdutos(String nome, Integer categoriaId, Integer lojaId) {

    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
    CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);
    Root<Produto> root = query.from(Produto.class);
```

```
        Path<String> nomePath = root.<String> get("nome");

        if (!nome.isEmpty()) {
            Predicate nomeIgual = criteriaBuilder.like(nomePath, "%" + nome + "%");
            query.where(nomeIgual);
        }


        TypedQuery<Produto> typedQuery = em.createQuery(query);

        return typedQuery.getResultList();
    }
```

Pronto, agora nossa consulta aos produtos - usando um determinado nome - está pronta!

##Adicionando mais filtros à consulta

Até o momento a única forma de consulta aos produtos é buscando por seu nome. O que desejamos inserir, ainda, é que a consulta possa ser realizada também pela `loja` e `categoria`.

Vamos adicionar "categoria" e "loja" à consulta e faremos isso da mesma maneira que fizemos com o nome, ou seja, através de `Predicates`. Antes disso, precisamos do caminho para chegar no `id` da loja. Como `id` é um atributo da classe `Loja` precisamos encontrar a loja, e a partir desta, encontrar o seu `id`:

```
Path<Integer> lojaPath = root.<Loja> get("loja").<Integer> get("id");
```

Agora que já temos o caminho, precisamos criar o `Predicate` para fazer a comparação. Caso o id não seja nulo (`id != null`) vamos mais uma vez usar o `CriteriaBuilder` para criá-lo, dizendo que queremos que o `id` da loja seja igual ao `id` passado no método:

```
Path<Integer> lojaPath = root.<Loja> get("loja").<Integer> get("id");

if (lojaId != null) {
    Predicate lojaIgual = criteriaBuilder.equal(lojaPath, lojaId);
}
```

Faremos o mesmo com "categoria". Com a pequena diferença de que o relacionamento de "categoria" é `@ManyToMany`. Ou seja, precisaremos de um `join` - a partir do produto - como já vimos. Só parar refrescar a memória:

```
"join fetch p.categorias c where c.nome = :pCategoria and
```

Ou seja, precisamos fazer um `join` a partir do produto e se o id da categoria não for `null`, criamos o `Predicate`. Teremos o seguinte:

```
Path<Integer> categoriaPath = root.join("categorias").<Integer> get("id");
if (categoriaId != null) {
    Predicate categoriaIgual = criteriaBuilder.equal(categoriaPath, categoriaId);
}
```

Agora que terminamos todos os filtros, podemos criar a `query`. Teremos:

```
query.where(...);

TypedQuery<Produto> typedQuery = em.createQuery(query);
```

Mas qual deve ser a condição para passar pro `where`? Vimos que este recebe um `Predicate`, mas agora temos três (nome, categoria e loja). É comum usarmos conectivos com `AND` ou `OR` para agrupar várias condições e, com a `Criteria API`, não será diferente. Uma das formas de fazer isso é usar o conectivo `AND` e passar um `Array` de `Predicates` para o método `where`. Então, vamos criar um `ArrayList` para guardar os `Predicates` que serão usados e, depois, transformaremos esse `ArrayList` em um `Array`:

```
Path<String> nomePath = root.<String> get("nome");
Path<Integer> categoriaPath = root.join("categorias").<Integer> get("id");
Path<Integer> lojaPath = root.<Loja> get("loja").<Integer> get("id");

List<Predicate> predicates = new ArrayList<Predicate>();

if (!nome.isEmpty()) {
    Predicate nomeIgual = criteriaBuilder.like(nomePath, "%" + nome + "%");
    predicates.add(nomeIgual);
}

if (categoriaId != null) {
    Predicate categoriaIgual = criteriaBuilder.equal(categoriaPath,
            categoriaId);
    predicates.add(categoriaIgual);
}

if (lojaId != null) {
    Predicate lojaIgual = criteriaBuilder.equal(lojaPath, lojaId);
    predicates.add(lojaIgual);
}
```

Agora, precisamos converter essa lista em um `Array` e passa-la no método `where`:

```
query.where((Predicate[]) predicates.toArray(new Predicate[0]));
TypedQuery<Produto> typedQuery = em.createQuery(query);
```

Por último, chamaremos o método `getResultList()` da `typedQuery` para realizarmos a busca. O método ficará assim:

```
public List<Produto> getProdutos(String nome, Integer categoriaId,
        Integer lojaId) {

    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
    CriteriaQuery<Produto> query = criteriaBuilder
            .createQuery(Produto.class);
    Root<Produto> root = query.from(Produto.class);

    Path<String> nomePath = root.<String> get("nome");
    Path<Integer> categoriaPath = root.join("categorias").<Integer> get("id");
```

```
        Path<Integer> lojaPath = root.<Loja> get("loja").<Integer> get("id");

        List<Predicate> predicates = new ArrayList<Predicate>();

        if (!nome.isEmpty()) {
            Predicate nomeIgual = criteriaBuilder.like(nomePath, "%" + nome + "%");
            predicates.add(nomeIgual);
        }

        if (categoriaId != null) {
            Predicate categoriaIgual = criteriaBuilder.equal(categoriaPath,
                    categoriaId);
            predicates.add(categoriaIgual);
        }

        if (lojaId != null) {
            Predicate lojaIgual = criteriaBuilder.equal(lojaPath, lojaId);
            predicates.add(lojaIgual);
        }

        query.where((Predicate[]) predicates.toArray(new Predicate[0]));

        TypedQuery<Produto> typedQuery = em.createQuery(query);

        return typedQuery.getResultList();

    }
```

Tudo bem? :)

Este foi um capítulo verdadeiramente extenso! Não se preocupe se alguma informação não ficou clara pois, iremos fixar todo esse conteúdo nos exercícios. Vamos por a mão na massa?! Nos vemos lá!