FIAP ON

FUNDAMENTOS JAVA

PERSISTÊNCIA OO (JPQL)

JOSÉ YOSHIRIRO

10

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

10 PERSISTÊNCIA OO (JPQL)

10.1 Introdução

No capítulo anterior, vimos como concentrar operações básicas CRUD numa superclasse que chamamos de Generic DAO. E, como explicado, é possível criar novas funcionalidades nos DAOs que os estendem. Neste capítulo, veremos como realizar consultas personalizadas, alterações e exclusões de registros usando uma linguagem de acesso a dados muito parecida com **SQL**, a **JPQL** (**Java Persistence Query Language**).

10.2 O que é a JPQL?

A **JPQL** é uma linguagem de acesso a dados **orientada a objetos** e **independente de banco de dados** usada pelo JPA para realizar operações de consulta, alteração e exclusão de registros no banco de dados. Não é possível realizar inclusão de registros usando JPQL.

Ser orientada a objetos significa que devemos usar os nomes das classes e atributos das Entidades e não das tabelas nas instruções JPQL.

Ser **independente de banco de dados** significa que o JPA traduz instruções JPQL para o SQL específico de cada fabricante de banco de dados, sem que o desenvolvedor se preocupe com isso. O que é um grande elemento facilitador, pois há muitas diferenças entre os SQLs, como conversão de tipos e paginação de resultados. A Figura *Tradução de JPQL para SQL de diferentes fabricantes de banco de dados* ilustra esse processo para alguns servidores de banco.

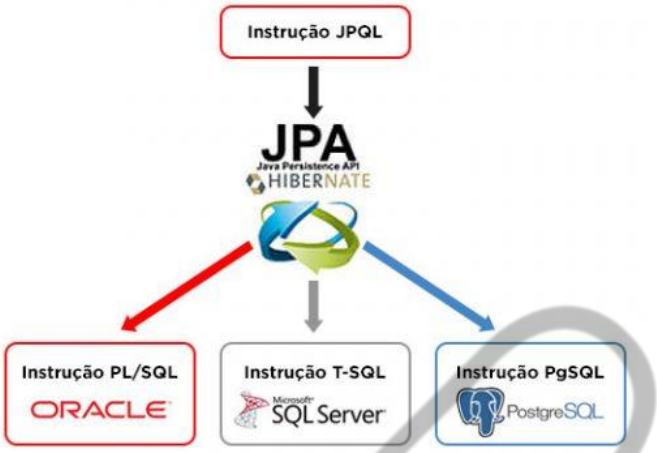

Figura 10.1 – Tradução de JPQL para SQL de diferentes fabricantes de banco de dados
Fonte: FIAP (2017)

Outra grande vantagem dessa independência de banco de dados é que o projeto pode acessar, por exemplo, o ***PostgreSQL*** durante o desenvolvimento e o ***Oracle*** em produção, ambos sendo acessados pelas mesmas instruções JPQL.

A seguir, veremos como realizar operações com JPQL considerando duas entidades, **Estabelecimento** e **TipoEstabelecimento**, que já estudamos em capítulos anteriores, mapeadas nas tabelas da Figura *DER das tabelas "estabelecimento" e "tipo_estabelecimento".*

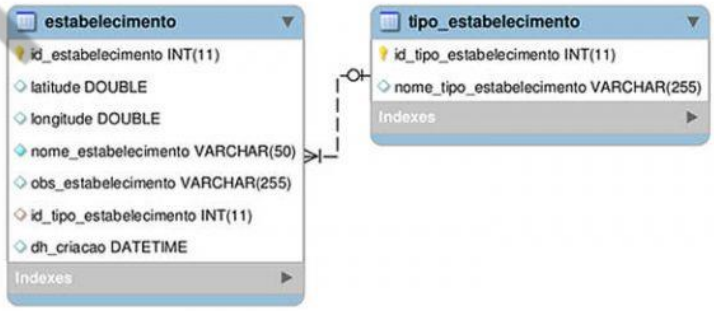

Figura 10.2 – DER das tabelas "estabelecimento" e "tipo_estabelecimento"
Fonte: FIAP (2017)

Para conhecer melhor o mapeamento objeto relacional entre as Tabelas e as Entidades, observe os quadros para **Estabelecimento** e para **TipoEstabelecimento**, *ORM da entidade "Estabelecimento"* e *ORM da entidade "TipoEstabelecimento"*, respectivamente.

	Tabela	**Entidade**
Nome Tabela/Classe	estabelecimento	Estabelecimento
Campos/Atributos	id_estabelecimento	id
	latitude	latitude
	longitude	longitude
	nome_estabelecimento	nome
	obs_estabelecimento	obs
	id_tipo_estabelecimento	tipo
	dh_criacao	dataCriacao

Quadro 10.1 – ORM da entidade "Estabelecimento"
Fonte: FIAP (2017)

	Tabela	**Entidade**
Nome Tabela / Classe	Tipo_estabelecimento	TipoEstabelecimento
Campos / Atributos	id_tipo_estabelecimento	id
	nome_tipo_estabelecimento	nome

Quadro 10.2 – ORM da entidade "TipoEstabelecimento"
Fonte: FIAP (2017)

10.3 Consultas com JPQL

10.3.1 Consultas básicas

Uma consulta simples é muito parecida com aquele famoso "*select * from tabela*", mas o "*select*" é opcional e usamos o nome da Entidade em vez do nome da Tabela. Veja como seria a instrução JPQL para obter todos os registros de **TipoEstabelecimento** e como ela poderia ser usada se sobrescrevêssemos o método **listar()** da **TipoEstabelecimentoDAO** (a mesma do capítulo anterior) no Código-fonte *Sobrescrevendo o método "listar()" na "TipoEstabelecimentoDAO" usando JPQL*. A explicação detalhada da consulta encontra-se logo em seguida ao código-fonte.

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{
// construtor
@Override
public List<TipoEstabelecimento> listar() {
```

```
            return this.em.createQuery(
               "from TipoEstabelecimento"
            ).getResultList();
    }
    }
```

Código-fonte 10.1 – Sobrescrevendo o método “listar()” na “TipoEstabelecimentoDAO” usando JPQL
Fonte: Elaborado pelo autor (2017)

Para usar o JPQL, invocamos o método **createQuery()** a partir de nosso **em**, que é uma instância de **EntityManager**. Após indicar a instrução desejada, para solicitar um resultado que pode retornar de 0 a vários registros, usamos o método **getResultList()**. Como foi dito antes, o *“select”* é opcional. Logo, se você preferir, a instrução poderia ser *“select from TipoEstabelecimento”*. E, ratificando, usamos o nome da Entidade e não o da Tabela.

10.3.2 Consultas ordenadas

Para ordenar o resultado de uma consulta, usamos a cláusula **order by**, que é muito parecida com a cláusula de mesmo nome da SQL, inclusive no que diz respeito aos seus complementos **asc** e **desc**. A única diferença é que usamos o nome de atributos das Entidades e não de campos das tabelas. No Código-fonte *Método “listarOrdenadoNome()” na “TipoEstabelecimentoDAO*”, criamos um novo método na **TipoEstabelecimentoDAO** chamado **listarOrdenadoNome()**, que traz todos os registros, porém ordenados pela ordem alfabética do atributo **nome**.

```
    // pacote e imports

    public class TipoEstabelecimentoDAO extends
     GenericDAO<TipoEstabelecimento, Integer>{

    // construtor

    // método listar()

    public List<TipoEstabelecimento> listarOrdenadoNome() {
             return this.em.createQuery(
                "from TipoEstabelecimento order by nome"
             ).getResultList();
    }
    }
```

Código-fonte 10.2 – Método “listarOrdenadoNome()” na “TipoEstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

Assim como na SQL, se não usarmos **asc** nem **desc**, a ordem-padrão é **asc**. Por isso, a consulta traria os registros pela ordem alfabética. Se após *"nome"*, usássemos **desc**, a listagem viria na ordem inversa à alfabética. E, ratificando, usamos o nome do atributo da Entidade (**nome**) e não do campo da Tabela (**nome_tipo_estabelecimento**).

Caso seja necessária a ordenação por mais de um campo, basta fazer como se faz em SQL. Por exemplo, poderíamos alterar a consulta do Código-fonte *Método "listarOrdenadoNome()" na "TipoEstabelecimentoDAO"* para ordenar por **nome** crescente e por **id** decrescente. A instrução JPQL ficaria como a do Código-fonte *Ordenando por mais de um atributo com JPQL*.

```
"from TipoEstabelecimento order by nome, id desc"
```

Código-fonte 10.3 – Ordenando por mais de um atributo com JPQL
Fonte: Elaborado pelo autor (2017)

10.3.3 Consultas limitadas

Algumas tabelas ou consultas podem conter muito mais registros do que aquilo que se deseja apresentar por vez ao usuário. Um exemplo disso seria uma consulta em um sistema escolar que retorne os 10 alunos com as melhores médias entre todos ou os 5 alunos com maior peso entre os matriculados numa academia.

Ainda na **TipoEstabelecimentoDAO**, vamos supor que precisamos dos três últimos registros cadastrados no sistema. Nesse caso, pediríamos uma consulta de todos, ordenada de forma decrescente pelo atributo **id** (afinal, ele cresce a cada novo registro). Vamos criar, então, o método **listarTresUltimos()**, conforme o Código-fonte *Método "listarTresUltimos()" na "TipoEstabelecimentoDAO*".

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

// construtor

// outros métodos

public List<TipoEstabelecimento> listarTresUltimos() {
     return this.em.createQuery(
          "from TipoEstabelecimento order by id desc"
          ).setMaxResults(3).getResultList();
```

```
    }
    }
```

Código-fonte 10.4 – Método “listarTresUltimos()” na “TipoEstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

O código que limitou o retorno em **até 3** registros é o **setMaxResults(3)** antes do **getResultList()**. Esse recurso do JPA é muito poderoso, pois abstrai algo que é bem diferente entre os fabricantes de banco de dados. Quando executada, essa consulta será traduzida pelo JPA e usará o comando SQL próprio do banco de dados configurado para fazer essa limitação de linhas no resultado. Para se ter noção da diferença entre os fabricantes, veja no Código-fonte *Limitação de resultados em diferentes servidores de banco de dados* como essa consulta ficaria para os servidores **Oracle**, **MS SQL Server** e **PostgreSQL**.

```
-- Em PL/SQL (Oracle)
select * from (
   select * from tipo_estabelecimento
) where rownum <= 3

-- Em T-SQL (SQL Server)
select top 3 * from tipo_estabelecimento

-- Em PgSQL (PostgreSQL)
select * from tipo_estabelecimento limit 3
```

Código-fonte 10.5 – Limitação de resultados em diferentes servidores de banco de dados
Fonte: Elaborado pelo autor (2017)

10.3.4 Consultas paginadas

Você já deve ter feito pesquisas em sites de e-commerce e, ao navegar pelo resultado, notou que era exibido certo número de resultados agrupados em algumas páginas, a numeração delas indicada, normalmente, ao fim da página. Quando vemos isso, dizemos que foi feita uma **consulta paginada**.

O conceito de paginação é simples: pretende-se recuperar uma quantidade X de registros por vez. A “página”, na verdade, é a posição do primeiro registro a partir do qual serão recuperados X registros. Por exemplo, você está programando o resultado de uma pesquisa e precisa exibir 10 registros por página: nesse caso, limitamos sempre a 10 a quantidade de registros a ser recuperados. Então, para a página 1, temos que configurar que o primeiro registro do resultado deve ser o 1º encontrado, consequentemente, o último acabará sendo o 10º. Para a página 2, o

primeiro registro deve ser o 11º, consequentemente, o último acabará sendo o 20º. E assim por diante.

Para exemplificar, criamos um método na TipoEstabelecimentoDAO que recupera todos os tipos de estabelecimentos, paginados. Como parâmetro, indicaremos a "página" e a quantidade de itens por página (vide o Código-fonte *Método "listarPaginado()" na "TipoEstabelecimentoDAO"*).

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

// construtor

// outros métodos

     public List<TipoEstabelecimento>
           listarPaginado(int itensPorPagina, int
pagina) {

          int primeiro = (pagina - 1) * itensPorPagina;

          return this.em.createQuery(
            "from TipoEstabelecimento order by nome"
          ).setMaxResults(itensPorPagina)
           .setFirstResult(primeiro)
           .getResultList();
     }
}
```

Código-fonte 10.6 – Método "listarPaginado()" na "TipoEstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

O código limitou o retorno em até o valor do argumento **itensPorPagina**. Para calcularmos o primeiro registro, apenas reduzimos o valor da página em 1, poisconsideramos que as páginas começam em 1 e não em 0 e multiplicamos pelo valor de **itensPorPagina**. O resultado desse cálculo é chamado **primeiro** e o usamos invocando o método **setFirstResult()** do **em**. Assim, com poucas linhas de código, podemos ter resultados "paginados".

10.3.5 Consultas parametrizadas

Até agora, todas as nossas consultas usavam uma instrução JPQL estática. Porém, é muito comum que consultas sejam feitas de acordo com valores informados pelo usuário. Pense num formulário de pesquisa de alunos por nome num sistema escolar, por exemplo. Ou na busca que fazemos em sites do governo com nosso CPF para saber nossa situação fiscal. Em situações como essas, precisamos usar o recurso de parâmetros de consulta do JPA.

Para exemplificar esse tipo de consulta, vamos trabalhar com a Entidade Estabelecimento. Logo, precisamos criar a classe **EstabelecimentoDAO**, que pode ser vista no Código-fonte *Classe "EstabelecimentoDAO"*.

```
package br.com.fiap.smartcities.dao;
import javax.persistence.EntityManager;
import br.com.fiap.smartcities.domain.Estabelecimento;

public class EstabelecimentoDAO
     extends GenericDAO<Estabelecimento, Integer> {

     public EstabelecimentoDAO(EntityManager em) {
          super(em);
     }
}
```

Código-fonte 10.7 – Classe "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Vamos supor que precisamos de uma consulta de Estabelecimento por latitude e longitude. Nesse caso, criamos um novo método, **listarPorNome()**, que recebe o nome de estabelecimento a ser pesquisado (vide Código-fonte *Método "listarPorNome()" na "EstabelecimentoDAO"*).

```
// pacote e imports

public class EstabelecimentoDAO
     extends GenericDAO<Estabelecimento, Integer> {

     // construtor

     public List<Estabelecimento> listarPorNome(String
nome)
     {

          return this.em.createQuery(
                    "select e from Estabelecimento e "
```

```
                              +"where e.nome = :nome"
                              ).setParameter("nome", nome)
                              .getResultList();

        }
    }
```

Código-fonte 10.8 – Método "listarPorNome()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Note que usamos a cláusula ***select*** de forma explicita, além de dar um *alias* para a Entidade (**e**). Esta é uma boa prática quando a consulta possui parâmetros, pois evita algumas exceções em tempo de execução. Usamos o *alias* **e** simplesmente por ser a primeira letra de **Estabelecimento**, facilitando, assim, a leitura e a análise humana da instrução JPQL.

Dentro da instrução, indicamos a parametrização quando temos um termo precedido por dois pontos (**:**). Logo, a instrução do Código-fonte *Método "listarPorNome()" na "EstabelecimentoDAO"* possui apenas um parâmetro, configurado no trecho "***:nome***".

O preenchimento do parâmetro foi feito com o método **setParameter()**, do **em**. Este é um recurso muito poderoso, não precisamos nos preocupar com detalhes, como conversões e uso ou não de aspas nas instruções SQL, o JPA faz isso por nós. Basta, como no exemplo, indicar o nome do parâmetro e o valor a ser usado.

Vamos ver agora outro exemplo: uma consulta com mais de um parâmetro. Vamos criar uma consulta por nome e data de criação e chamá-la de **listarPorNomeCriacaoApos()** (vide o Código-fonte *Método "listarPorNome()" na "EstabelecimentoDAO"*).

```
    // pacote e imports
    import java.util.Calendar;

    public class EstabelecimentoDAO
         extends GenericDAO<Estabelecimento, Integer> {

         // construtor e outros métodos

         public List<Estabelecimento>
    listarPorNomeCriacaoApos(String nome, Calendar
    criacaoApos)
         {

              return this.em.createQuery(
                        "select e from Estabelecimento e "
```

```
                              +"where e.nome = :nome and "
                              +"where e.dataCricao > :criacao "
                              ).setParameter("nome", nome)
                               .setParameter("cricao", cricaoApos)
                              .getResultList();

          }
     }
```

Código-fonte 10.9 – Método "listarPorNome()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

No exemplo do Código-fonte *Método "listarPorNome()" na "EstabelecimentoDAO"*, temos uma consulta com dois parâmetros indicados por "*:nome*" e "*:criacao*", logo, são os parâmetros "*nome*" e "*criacao*". Mais uma vez, não precisamos nos preocupar em como a data deve ser enviada ao banco de dados. O JPA faz a tradução correta por nós.

Existe ainda a possibilidade de usar outra Entidade como parâmetro de uma pesquisa. Vamos supor que queremos recuperar todos os **Estabelecimentos** de um certo tipo de **TipoEstabelecimento**. Veja no Código-fonte *Método "listarPorTipo()" na "EstabelecimentoDAO"* como faríamos isso.

```
     // pacote e imports
     import
     br.com.fiap.smartcities.domain.TipoEstabelecimento;

     public class EstabelecimentoDAO
          extends GenericDAO<Estabelecimento, Integer> {

          // construtor e outros métodos

          public List<Estabelecimento>
                         listarPorTipo(TipoEstebelecimento
     tipo) {

               return this.em.createQuery(
                         "select e from Estabelecimento e "
                         +"where e.tipo = :tipo"
                         ).setParameter("tipo", nome)
                         .getResultList();

          }
     }
```

Código-fonte 10.10 – Método "listarPorTipo()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Observe que usamos diretamente uma instância de **TipoEstabelecimento** como valor para o parâmetro "*tipo*". O JPA identificará a chave estrangeira na tabela e fará a tradução para a criação de uma instrução SQL, provavelmente como a do Código-fonte *Instrução SQL gerada pelo método "listarPorTipo()" na "EstabelecimentoDAO"*.

```
select
   estabeleci0_.id_estabelecimento as id_estab1_4_,
   estabeleci0_.dh_criacao as dh_criac2_4_,
   estabeleci0_.nome_estabelecimento as nome_est3_4_,
   estabeleci0_.id_tipo_estabelecimento as id_tipo_4_4_
from
   estabelecimento estabeleci0_
where
   estabeleci0_.id_tipo_estabelecimento=?
```

Código-fonte 10.11 – Instrução SQL gerada pelo método "listarPorTipo()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

10.3.6 Consultas com JPQL que trazem apenas 1 linha

Algumas consultas, mesmo não sendo pela chave primária, trazem somente um registro. Exemplos disso seriam: o aluno com a maior média de uma turma ou o vencedor de um campeonato.

Para exemplificar esse tipo de consulta, vamos criar um método que retorna apenas um estabelecimento, de acordo com uma latitude e uma longitude. Veja no Código-fonte *Método "listarPorLocalizacao()" na "EstabelecimentoDAO"*.

```
// pacote e imports

public class EstabelecimentoDAO
     extends GenericDAO<Estabelecimento, Integer> {

     // construtor e outros métodos

     public Estabelecimento
      listarPorLocalizacao(double latitude, double
longitude)
     {

          return (Estabelecimento) this.em
                    .createQuery(
                    "select e from Estabelecimento e
where "
                   + "e.latitude = :latitude and
```

```
                        + "e.longitude = :longitude")
                        .setParameter("latitude", latitude)
                        .setParameter("longitude",
    longitude)
                        .getSingleResult();

        }
    }
```

Código-fonte 10.12 – Método "listarPorLocalizacao()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Note que não houve muitas novidades. A diferença mais relevante é que usamos o método **getSingleResult()** do **em**. Esse método transformará o resultado da consulta em um único objeto.

Como o **getSingleResult()** retorna um **Object**, é necessário fazer um ***cast***, como foi feito logo após a instrução **return**.

Importante: Caso a consulta configurada na instrução JPQL não retorne apenas uma linha do banco, o método **getSingleResult()** lança uma **NonUniqueResultException**. Portanto, tenha certeza de que sua consulta possui parâmetros suficientes para trazer somente uma linha na consulta.

10.4 Atualização de registros com JPQL

10.4.1 Atualizações básicas

Para realizar atualizações explícitas com JPQL, usamos a instrução "*update*", que é muito parecida com a "*update*" da SQL, a única diferença é que usamos os nomes das Entidades e seus atributos em vez dos nomes das Tabelas e campos.

No Código-fonte *Método "alterarTipoTodos()" na "EstabelecimentoDAO"*, damos um exemplo de como poderíamos alterar todos os registros de **Estabelecimento**, modificando seus **tipos** para um mesmo **TipoEstabelecimento**.

```
    // pacote e imports

    public class TipoEstabelecimentoDAO extends
     GenericDAO<TipoEstabelecimento, Integer>{

    // construtor e outros métodos

    public void alterarTipoTodos(TipoEstabelecimento tipo)
          {
```

```
                this.em.createQuery(
                     "update Estabelecimento e set e.tipo =
    :tipo"
                     ).setParameter("tipo", novoTipo)
                     .executeUpdate();
    }
    }
```

Código-fonte 10.13 – Método “alterarTipoTodos()” na “EstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

A novidade aqui é o método **executeUpdate()** do **em**. Ele é usado quando, em vez de realizar uma consulta, queremos fazer alterações ou exclusão de registros.

A questão de parâmetros aqui é exatamente como vimos nas consultas com JPQL. E, mais uma vez, o JPA fará a tradução de acordo com o relacionamento existente entre as tabelas **tipo_estabelecimento** e **estabelecimento**.

10.4.2 Recuperando a quantidade de linhas atualizadas

Caso seja necessário saber quantas linhas sofreram alteração com a execução da instrução JPQL de atualização, basta usar o retorno do **executeUpdate()**. Seu retorno é um valor inteiro, que indica a quantidade de linhas afetadas. Assim, bastaria uma pequena alteração no método **alterarTipoTodos()**, conforme pode ser visto no Código-fonte *Método “alterarTipoTodos()” retornando a quantidade de registros alterados*.

```
    // pacote e imports

    public class TipoEstabelecimentoDAO extends
     GenericDAO<TipoEstabelecimento, Integer>{

    // construtor e outros métodos

    public int alterarTipoTodos(TipoEstabelecimento tipo)
          {
                Return this.em.createQuery(
                     "update Estabelecimento e set e.tipo =
    :tipo"
                     ).setParameter("tipo", novoTipo)
                     .executeUpdate();
    }
    }
```

Código-fonte 10.14 – Método “alterarTipoTodos()” retornando a quantidade de registros alterados
Fonte: Elaborado pelo autor (2017)

Alteramos o método **alterarTipoTodos()** para retornar um valor inteiro, aproveitando o próprio valor de retorno do **executeUpdate()**.

10.5 Exclusão de registros com JPQL

10.5.1 Exclusões básicas

Para realizar exclusões explícitas com JPQL, usamos a instrução "*delete*", que é muito semelhante com a "*delete*" da SQL, a única diferença é que usamos os nomes das Entidades e seus atributos em vez dos nomes das Tabelas e campos.

No Código-fonte *Método "excluirAntesDe()" na "EstabelecimentoDAO"*, temos um exemplo de como poderíamos excluir todos os registros de **Estabelecimento** criados antes de uma determinada data.

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

// construtor e outros métodos

public void excluirAntesDe(Calendar data)
      {
          this.em.createQuery(
               "delete from Estabelecimento e "
               + "where dataCriacao < :data"
               ).setParameter("data", data)
                .executeUpdate();
}
}
```

Código-fonte 10.15 – Método "excluirAntesDe()" na "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Novamente, aqui usamos o método **executeUpdate()** do **em**. Afinal, a instrução JPQL é de exclusão de registros.

A questão de parâmetros aqui é exatamente como vimos nas consultas com JPQL. E, mais uma vez, o JPA fará a tradução para a instrução SQL, que é necessária para lidar com a data do tipo **Calendar**.

10.5.2 Recuperando a quantidade de linhas atualizadas

Caso seja necessário saber quantas linhas foram excluídas com a execução da instrução JPQL de exclusão, basta usar o retorno do **executeUpdate()**. Seu retorno é um valor inteiro, que indica a quantidade de linhas excluídas. Assim, bastaria uma pequena alteração no método **excluirAntesDe()**, conforme pode ser visto no Código-fonte *Método "excluirAntesDe()" retornando a quantidade de registros alterados.*

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

// construtor e outros métodos

public int excluirAntesDe(Calendar data)
      {
          return this.em.createQuery(
               "delete from Estabelecimento e "
               + "where dataCriacao < :data"
               ).setParameter("data", data)
                .executeUpdate();
}
}
```

Código-fonte 10.16 – Método "excluirAntesDe()" retornando a quantidade de registros alterados
Fonte: Elaborado pelo autor (2017)

Alteramos o método **excluirAntesDe()** para retornar um valor inteiro, aproveitando o próprio valor de retorno do **executeUpdate()**.

REFERÊNCIAS

JANSSEN, Thorben. **Ultimate Guide to JPQL Queries with JPA and Hibernate**. 18 jan. 2017. Disponível em: <https://www.thoughts-on-java.org/jpql/>. Acesso em: 29 nov. 2017.

JBOSS.ORG. **Hibernate ORM 5.2.12. Final User Guide**. [s.d.] Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 25 nov. 2017.

JENDROCK, Eric. **Persistence – The Java EE5 Tutorial**. [s.d.] Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 25 nov. 2017.

NETO, Oziel Moreira. **Entendendo e dominando o Java**. 3. ed. São Paulo: Universo dos Livros, 2012.

PANDA, Debu; RAHMAN, Reza; CUPRAK, Ryan; REMIJAN, Michael. **EJB 3 in Action**. 2. ed. Shelter Island: Manning Publications, 2014.