FIAP ON

FRAMEWORKS JAVA, .NET & WEBSERVICES

AMALGÁMA ENTRE **JAVA E SQL**

3A

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 AMÁLGAMA ENTRE JAVA E SQL

Além do uso do Java para acessar o banco de dados e da utilização de SQL para fazer consultas, existe uma terceira opção que é a utilização de uma linguagem chamada JPQL (Java Persistence Query Language). Essa linguagem é um híbrido do Java e do SQL criada para ser utilizada em conjunto com o mapeamento objeto-relacional do JPA/Hibernate. Com ela, é possível fazer consultas de forma mais intuitiva e prática, utilizando objetos Java ao invés de tabelas do banco de dados.

1.1 Introdução

No capítulo anterior, vimos como concentrar operações básicas CRUD numa superclasse que chamamos de GenericDAO. Descobrimos que é possível criar novas funcionalidades nos DAOs que o 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**).

1.2 O QUE É A JPQL?

JPQL (Java Persistence Query Language) é uma linguagem de consulta do JPA (Java Persistence API) usada para executar consultas em bancos de dados relacionais. Ela é muito parecida com SQL, mas, ao invés de usar tabelas e colunas, JPQL usa entidades e atributos, permitindo assim que as consultas sejam escritas em termos de objetos.

Os principais tópicos que você precisa conhecer para usar JPQL são:

- Sintaxe básica: JPQL usa uma sintaxe parecida com SQL, mas com algumas diferenças importantes, como o uso de entidades e atributos em vez de tabelas e colunas.
- Consultas simples: você pode usar JPQL para buscar objetos de uma entidade específica, por exemplo, "SELECT e FROM Employee e".

- Consultas com cláusulas WHERE: é possível filtrar os resultados das consultas usando a cláusula WHERE. Por exemplo, "SELECT e FROM Employee e WHERE e.salary > 5000".
- Consultas com junções: JPQL permite juntar os resultados de várias entidades em uma consulta. Por exemplo, "SELECT e FROM Employee e JOIN e.department d WHERE d.name = 'IT'".
- Consultas com funções de agregação: é possível usar funções de agregação como COUNT, MAX, MIN, AVG e SUM em JPQL. Por exemplo, "SELECT COUNT(e) FROM Employee e".
- Consultas dinâmicas: JPQL permite que você crie consultas dinamicamente em tempo de execução, usando parâmetros em vez de valores literais. Por exemplo, "SELECT e FROM Employee e WHERE e.salary > :minSalary".
- Ordenação de resultados: você pode ordenar os resultados de uma consulta JPQL usando a cláusula ORDER BY. Por exemplo, "SELECT e FROM Employee e ORDER BY e.salary DESC".
- Paginação de resultados: JPQL permite que você faça paginação de resultados, limitando o número de resultados retornados e especificando o início da página. Por exemplo, "SELECT e FROM Employee e ORDER BY e.salary DESC LIMIT 10 OFFSET 20".

Exemplos reais de uso de JPQL incluem:

- Buscar todos os clientes com um determinado nome: "SELECT c FROM Customer c WHERE c.name = 'John'".
- Buscar todos os pedidos realizados em um determinado período: "SELECT o FROM Order o WHERE o.orderDate BETWEEN :startDate AND :endDate".
- Buscar o total de vendas por categoria de produto: "SELECT p.category, SUM(p.price) FROM Product p GROUP BY p.category".

JPQL é uma poderosa ferramenta para consultar bancos de dados relacionais usando o JPA, permitindo que as consultas sejam escritas em termos de objetos, em vez de tabelas e colunas. Com o conhecimento dos tópicos acima e alguns exemplos reais de uso, você pode começar a usar JPQL em suas próprias aplicações.

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 fazer inclusão de registros usando JPQL.

Ser **orientada a objetos** significa que devemos utilizar **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 precise se preocupar com isso. Isso 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.

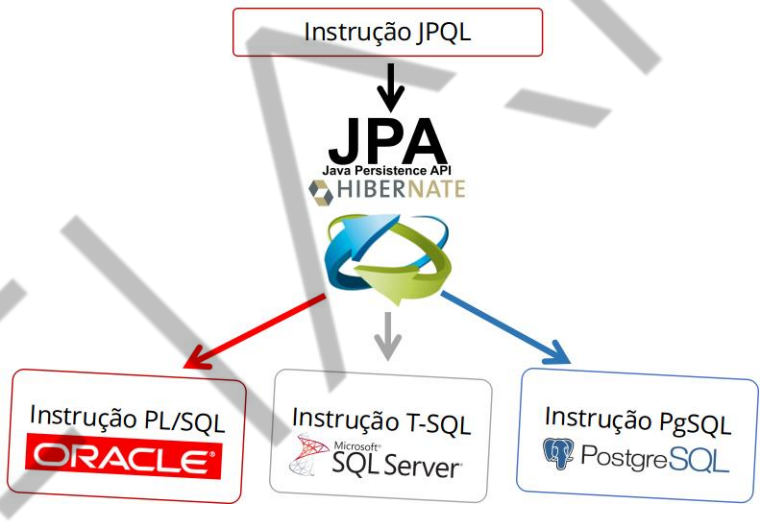

Figura 1 – Tradução de JPQL para SQL de diferentes fabricantes de banco de dados
Fonte: FIAP (2017)

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. São as entidades mapeadas para as tabelas da Figura “DER das tabelas ‘estabelecimento’ e ‘tipo_estabelecimento’”:

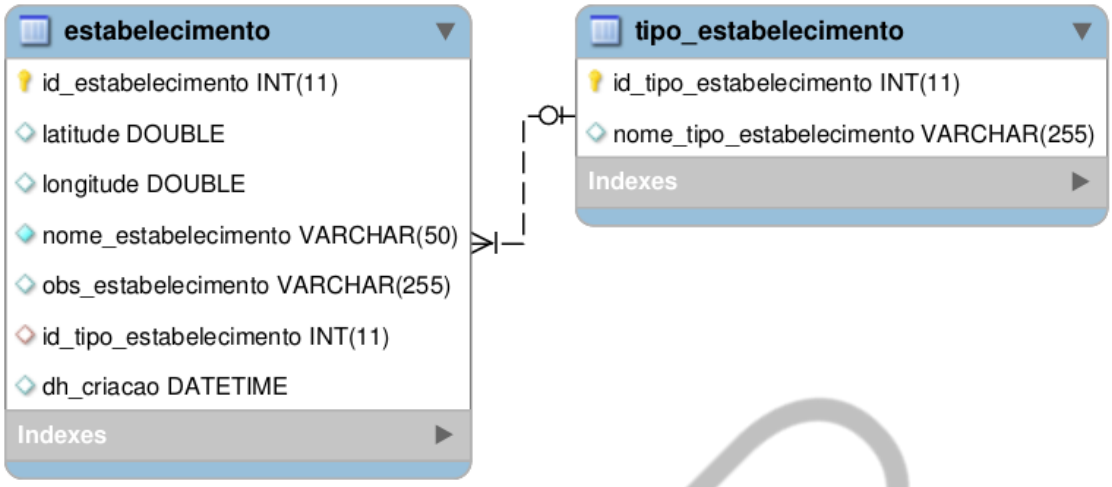

Figura 2 – DER das tabelas “estabelecimento” e “tipo_estabelecimento”
Fonte: Elaborado pelo autor (2017)

Para conhecer melhor o mapeamento do objeto relacional entre as Tabelas e as Entidades, observe a tabela para **Estabelecimento** e a tabela para **TipoEstabelecimento:**

	Tabela	Entidade
Nome tabela / classe	estabelecimento id_estabelecimento latitude longitude	Estabelecimento Id Latitude Longitude
Campos / Atributos	nome_estabelecimento obs_estabelecimento id_tipo_estabelecimento dh_criacao	Nome Obs Tipo dataCriacao

Tabela 1 – ORM da entidade “Estabelecimento”
Fonte: FIAP (2017)

	Tabela	Entidade
Nome tabela / classe	Tipo_estabelecimento id_tipo_estabelecimento	TipoEstabelecimento Id
Campos / Atributos	nome_tipo_estabelecimento	Nome

Tabela 2 – ORM da entidade “TipoEstabelecimento
Fonte: FIAP (2017)

2.1 Consultas com JPQL

JPQL permite que você escreva consultas que operam sobre objetos e suas propriedades, em vez de operar diretamente com as tabelas e colunas do banco de dados. As consultas JPQL retornam objetos do tipo da classe que está sendo consultada, em vez de um conjunto de resultados.

2.1.1 Consultas básicas

Uma consulta simples, que corresponde àquele famoso "*select * from tabela*", é muito parecida. Sendo o "*select*" opcional, 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 utilizada 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 após o 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 1 – Sobrescrevendo o método "listar()" na "TipoEstabelecimentoDAO" usando JPQL
Fonte: Elaborado pelo autor (2017)

Para usar JPQL, invocamos o método **createQuery()** a partir de nosso **em**, que é uma instância de **EntityManager**. Depois de 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 t from TipoEstabelecimento t"*. Vale ressaltar que usamos o nome da Entidade e não da Tabela.

2.1.2 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 organizados de acordo com a ordem alfabética do atributo **nome:**

```
package br.com.fiap.smartcities.dao;

import java.util.List;
import javax.persistence.EntityManager;
import br.com.fiap.smartcities.domain.TipoEstabelecimento;

public class TipoEstabelecimentoDAO extends
GenericDAO<TipoEstabelecimento, Integer> {
```

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

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

Código-fonte 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. Lembre-se que 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 proceder 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 3 – Ordenando por mais de um atributo com JPQL
Fonte: Elaborado pelo autor (2017)

2.1.3 Consultas limitadas

Algumas tabelas ou consultas podem conter muito mais registros do que o 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 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 5 – Limitação de resultados em diferentes servidores de banco de dados
Fonte: Elaborado pelo autor (2017)

2.1.4 Consultas paginadas

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

Vamos imaginar que você esteja criando uma página de resultados de pesquisa, na qual é preciso exibir uma grande quantidade de informações ao usuário. A ideia da paginação é permitir que você divida essa grande quantidade de informações em pequenas partes, ou "páginas", para que o usuário possa navegar de forma mais fácil e rápida.

O conceito é bem simples: você recupera uma quantidade de registros por vez. Cada "página" é composta por uma quantidade fixa de registros, por exemplo, 10 registros por página. Para exibir a primeira página, basta selecionar os primeiros 10 registros. Para exibir a segunda página, é necessário pular os 10 primeiros registros e buscar os próximos 10. E assim por diante.

Isso é útil porque, se houver muitos registros, pode ser impraticável recuperá-los todos de uma só vez. Além disso, o usuário pode se sentir sobrecarregado com a quantidade de informações exibidas. Com a paginação, ele pode se concentrar em um subconjunto de informações de cada vez.

Para implementar a paginação em seu código, é necessário criar uma consulta que recupere apenas um subconjunto de registros por vez. Para isso, você pode usar a cláusula "LIMIT" em SQL ou "setMaxResults" em JPQL, dependendo do banco de

dados ou ORM que estiver usando. Em seguida, basta calcular o intervalo correto de registros que você precisa recuperar para cada página.

Por exemplo, se você quiser exibir 10 registros por página e estiver na página 2, precisará pular os primeiros 10 registros e buscar os próximos 10. Para fazer isso, você pode usar a cláusula "OFFSET" em SQL ou "setFirstResult" em JPQL para pular os primeiros 10 registros. Em seguida, use a cláusula "LIMIT" ou "setMaxResults" para buscar os próximos 10 registros.

Em resumo, a paginação é uma técnica importante para tornar grandes conjuntos de dados mais gerenciáveis e acessíveis ao usuário final. Implementá-la requer algumas modificações no código da consulta, mas é um recurso valioso para melhorar a experiência do usuário em aplicações que envolvem grandes quantidades de dados.

Para exemplificar, criamos um novo método na TipoEstabelecimentoDAO que recupera todos os tipos de estabelecimentos, agora paginados. Como parâmetro, apontaremos 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 6 – Método “listarPaginado()” na “TipoEstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

O código utiliza o método createQuery da classe EntityManager do JPA para criar uma consulta JPQL que busca os registros da entidade TipoEstabelecimento, ordenando-os pelo nome.

Os parâmetros itensPorPagina e página definem quantos registros serão retornados por página e qual é a página atual, respectivamente. O cálculo da posição do primeiro registro a ser retornado é feito através da fórmula (página - 1) * itensPorPagina.

A consulta é, então, configurada com os valores de itensPorPagina e primeiro através dos métodos setMaxResults e setFirstResult, respectivamente. Por fim, a lista de resultados é retornada através do método getResultList.

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, pois consideramos que as páginas começam em 1 e não em 0, e multiplicamos pelo valor de **itensPorPagina**. Chamamos o resultado desse cálculo de **primeiro** e o usamos invocando o método **setFirstResult()** do **em**. Assim, com poucas linhas de código, podemos ter resultados “paginados”.

2.1.5 Consultas parametrizadas

Até agora, todas as nossas consultas usavam uma instrução JPQL estática. Porém, é muito comum que consultas sejam realizadas de acordo com valores informados pelo usuário. Pense num formulário de pesquisa de alunos por nome num sistema escolar, por exemplo. Ou até mesmo 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 observada 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 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 o Código-fonte “Método ‘listarPorNome()’ da ‘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 = :n"
                    ).setParameter("n", nome)
                    .getResultList();

     }
}
```

Código-fonte 8 – Método “listarPorNome()” da “EstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

Repare que usamos a cláusula ***select*** de forma explícita, além de dar um *alias* para a Entidade (**e**). Essa 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 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()’ da ‘EstabelecimentoDAO’” possui apenas um parâmetro, configurado no trecho **“*:n*”**.

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

Veremos 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()' da '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) {

          String jpqlQuery = "select e from Estabelecimento e
where e.nome = :n and e.dataCriacao > :criacao";

          return
this.em.createQuery(jpqlQuery).setParameter("n",
nome).setParameter("criacao", criacaoApos)
                    .getResultList();
     }

}
```

Código-fonte 9 – Método "listarPorNome()" da "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

No exemplo do Código-fonte "Método 'listarPorNome()' da '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.

É possível ainda usar outra Entidade como parâmetro de uma pesquisa. Vamos supor que queremos recuperar todos os **Estabelecimento** de um certo tipo de **TipoEstabelecimento**. Veja no Código-fonte "Método 'listarPorTipo()' da '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 = :t"
                    ).setParameter("t", tipo)
                    .getResultList();

     }
}
```

Código-fonte 10 – Método "listarPorTipo()" da "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Observe que usamos diretamente uma instância de **TipoEstabelecimento** como valor para o parâmetro "*t*". 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()' da '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 11 – Instrução SQL gerada pelo método "listarPorTipo()" da "EstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

2.1.6 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 1 estabelecimento de acordo com uma latitude e uma longitude. Observe o Código-fonte “Método ‘listarPorLocalizacao()’ da ‘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 12 – Método “listarPorLocalizacao()” da “EstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

Perceba que não houve muitas novidades. A diferença mais relevante é que usamos o método **getSingleResult()** do **em**. Esse método vai transformar 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 os parâmetros suficientes para trazer somente uma linha na consulta.

3 ATUALIZAÇÃO DE REGISTROS COM JPQL

3.1 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'", temos um exemplo de como poderíamos alterar todos os registros de **Estabelecimento**, alterando seus **tipo** para um mesmo **TipoEstabelecimento:**

```
     // pacote e imports

     public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

     // construtor e outros métodos

                                          public           void
alterarTipoTodos(TipoEstabelecimento tipo) {

               em.getTransaction().begin();

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

               em.getTransaction().commit();

          }
     }
```

Código-fonte 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. Não podemos esquecer de utilizar as transações (*commit*) para efetivar as operações no banco de dados.

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**.

3.2 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 mostra a quantidade de linhas afetadas. Assim, bastaria uma pequena alteração no método **alterarTipoTodos()**, conforme pode ser observado no Código-fonte "Método 'alterarTipoTodos()' retornando à quantidade de registros alterados":

```
// pacote e imports

public class TipoEstabelecimentoDAO extends
 GenericDAO<TipoEstabelecimento, Integer>{

     // construtor e outros métodos

                                    public                  int
alterarTipoTodosComContador(TipoEstabelecimento tipo) {

                em.getTransaction().begin();

                int    total    =    this.em.createQuery("update
Estabelecimento e set e.tipo = :tipo")
                     .setParameter("tipo", tipo)
                     .executeUpdate();

                em.getTransaction().commit();

                return total;

          }
}
```

Código-fonte 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()**.

3.3 Exclusão de registros com JPQL

3.3.1 Exclusões básicas

Para realizar exclusões explícitas com JPQL, usamos a instrução "*delete*", que é muito parecida 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)
      {
     em.getTransaction().begin();
this.em.createQuery(
               "delete from Estabelecimento e "
               + "where dataCriacao < :data"
               ).setParameter("data", data)
                .executeUpdate();
em.getTransaction().commit();
     }
}
```

Código-fonte 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 necessária para lidar com a data do tipo **Calendar**.

3.3.2 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 exibe 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 excluirAntesDeContador(Calendar data) {
               em.getTransaction().begin();

               int total = this.em.createQuery("delete from
Estabelecimento e where dataCriacao < :data")
                    .setParameter("data", data)
                    .executeUpdate();

               em.getTransaction().commit();

               return total;
          }
}
```

Código-fonte 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()**.

A solução completa com a implementação das operações em JPQL pode ser baixada em:

Git: https://github.com/FIAP/smartcities-orm/tree/06-jpql

Zip: https://github.com/FIAP/smartcities-orm/archive/refs/heads/06-jpql.zip

https://github.com/profflaviomoreni/smartcities-orm/tree/06-jpqlhttps://github.com/profflaviomoreni/smartcities-orm/tree/05-generic-dao

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: 12 jan. 2021.

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: 12 jan. 2021.

JENDROCK, Eric. **Persistence – The Java EE5 Tutorial**. [s.d.]. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 12 jan. 2021.

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, NY: Manning Publications, 2014.