Artigo

Dominando a API de consultas do Hibernate

Durante o artigo serão explicados todos os meios de consulta disponíveis na implementação de referência do JPA. Abordaremos consultas nativas, JPQL, consultas nomeadas e a Criteria API, sempre com dicas de desempenho e de quando usar.

Do que se trata o artigo:

Durante o artigo serão explicados todos os meios de consulta disponíveis na implementação de referência do JPA. Abordaremos consultas nativas, JPQL, consultas nomeadas e a Criteria API, sempre com dicas de desempenho e de

Resumo DevMan:

Atualmente existem diversas maneiras de se consultar dados armazenados em um banco de dados. Sendo assim, é altamente recomendado conhecer esses meios e saber em qual situação empregá-los. Com base nesta necessidade, o artigo exibe não só as formas de consulta, mas também apresenta dicas para auxiliar na escolha da opção mais adequada.

É sabido que as aplicações, em sua grande maioria, têm como requisito não-funcional armazenar dados. Esse armazenamento é usualmente realizado pelos Sistemas Gerenciadores de Banco de Dados (SGBD), que, por sua vez, se comunicam com a aplicação em um nível mais baixo, trabalhando com as conexões do banco de dados por meio da manipulação de sockets. É nesse ponto que empregamos o Hibernate, abstraindo toda a comunicação de baixo nível entre a aplicação e o banco, fornecendo ainda uma excelente interface para consultas aos dados armazenados.

A experiência do autor tem mostrado que efetuar consultas em bancos de dados é a ação mais executada em sistemas. Levando em consideração esse fato, é importante conhecer as opções disponíveis para se recuperar informações previamente salvas. Com esse conhecimento em mãos, os desenvolvedores terão diferentes abordagens para solucionar seus problemas.

Neste contexto, este artigo apresentará os meios de se recuperar o que foi gravado em banco de dados usando o Hibernate. Esses meios foram criados para

os estudos de caso do artigo. Posteriormente, serão exibidos os meios disponíveis para recuperar as entidades no banco de dados.

Conceitos básicos

Neste tópico será relembrado de forma simples o que é um Sistema Gerenciador de Banco de Dados (SGBD), Object-Relational Mapping (ORM), chave primária, chave estrangeira e álgebra relacional. O leitor pode avançar para o próximo tópico caso já domine esses assuntos.

SGBD (Sistema Gerenciador de Bando de Dados)

Um Sistema Gerenciador de Banco de Dados é um software que gerencia toda a burocracia relacionada aos dados que uma aplicação precisa manter, como por exemplo, concorrência, dead-lock, quem tem permissão para executar tal comando e principalmente meios de organizar, armazenar e consultar dados. Esses sistemas são comumente chamados de **bancos de dados** ou **SGBDs**.

Entre outros, bancos de dados podem ser orientados a objetos, orientados a documentos ou relacionais. Para este artigo trabalharemos com bancos de dados relacionais. Esse será mais bem exemplificado no próximo subtópico.

ORM (Object-Relational Mapping)

É sabido que o tipo de banco de dados mais utilizado é o relacional. O autor

System – RDBMS) é um banco que armazena dados que se relacionam (ou não) entre si. O problema disso é que quando trabalhamos com linguagens orientadas a objetos, supostamente precisaríamos de um sistema que além de armazenar dados, também armazenasse operações.

Apresentado o problema, podemos avançar explicando o que é *Object-Relational Mapping* ou, em tradução livre, Mapeamento Objeto-Relacional, que significa resolver o problema citado no parágrafo anterior, ou seja, salvar os atributos (sejam eles de valor ou de relação) de um objeto em um banco de dados relacional, ao passo que mantém as operações no próprio objeto.

Apenas para exemplificar, a **Listagem 1** contém a definição de uma classe a partir da qual objetos são gerados.

Listagem 1. Código da classe Pessoa.

```
1  public class Pessoa {
2
3     private Integer id;
4     private String nome;
5     private Integer idade;
6
7     public String getDescricao() {
8      String descricao = “Identificação: “ + id.toString() + “\n”
9       “Nome: “ + nome + “\n”
10      “Idade: “ + idade.toString();
11
12      return descricao;
13    }
14
```

Listagem 2. Representação dos dados do objeto pessoa em um banco de dados relacional.

```
mysql> select * from pessoa;
  +----+----------------+-------+
  | id | nome           | idade |
  +----+----------------+-------+
  |  1 | Pedro da Silva |    22 |
  +----+----------------+-------+
```

Repare que apenas os valores dos atributos do objeto gerado a partir da classe Pessoa estão gravados no banco de dados. Sendo assim, uma ferramenta ORM ajuda a fazer a ligação entre os atributos dos objetos salvos em banco de dados e as operações dos objetos que estão em memória.

Além do problema clássico citado aqui, frameworks ORM costumam ter mais atribuições. Essas não serão abordadas no artigo, por não fazerem parte do escopo do mesmo, porém o leitor não terá problemas ao procurar por mais conhecimento em edições anteriores da Java Magazine.

Chave Primária e Chave Estrangeira

Para que uma entidade salva em banco de dados possa ser recuperada, é necessário que exista uma maneira única de identificá-la. Para tanto, geralmente é reservado um campo da tabela para essa função, e esse campo é comumente chamado de **id** ou **codigo**. Em alguns casos as entidades possuem chave primária

A **Listagem 3** exibe uma consulta SQL onde dado o valor **1**, da chave primária **id**, uma única linha da tabela *pessoa* é retornada.

Listagem 3. Resgatando a entidade pessoa a partir de sua chave primária.

```
mysql> select * from pessoa where id = 1;
  +----+----------------+-------+
  | id | nome           | idade |
  +----+----------------+-------+
  |  1 | Pedro da Silva |    22 |
  +----+----------------+-------+
```

Já a chave estrangeira é o que possibilita o relacionamento entre as entidades. Em um banco de dados relacional, uma chave estrangeira é a chave primária de uma tabela **A** contida entre os campos de outra tabela **B**. Cuidado ao ler a frase anterior, pois uma chave estrangeira na tabela **B** não necessariamente inibe a existência de uma chave primária na própria tabela **B**.

Ainda usando o exemplo da **Listagem 3**, suponha que uma entidade da tabela *pessoa* se relaciona com uma entidade da tabela *endereco*. Logo, a chave primária dessa suposta pessoa estará também na tabela *endereco*. Veja esse relacionamento na **Listagem 4**.

Listagem 4. Demonstrando o clássico relacionamento entre pessoa e endereco.

```
mysql> select * from pessoa where id = 1;
  +----+----------------+-------+
  | id | nome           | idade |
```

```
| 1 |         1 | Av. Paulista |  12345 | Consolação   |
+---+-----------+--------------+--------+--------------+
```

Escrever consultas no banco de dados que **relacionam as entidades** por meio das chaves primárias e chaves estrangeiras é conhecido como Álgebra Relacional.

Aplicação de exemplo

Como durante o artigo trabalharemos com várias consultas, foi selecionada uma aplicação para facilitar os exemplos e as explicações, além de tornar o artigo mais prático. Sendo assim, a aplicação idealizada é sobre compras coletivas. Um software que soluciona esse problema contém relacionamentos interessantes do ponto de vista didático, tais como: um para muitos e muitos para muitos.

Na **Figura 1** é possível visualizar o diagrama do Modelo de Domínio, que contempla as classes, seus atributos e relacionamentos. Neste diagrama existe a classe Usuario, que é composta por 1 ou mais Enderecos; a classe Cliente, que herda os atributos, operações e relacionamentos da classe Usuario, se relaciona com a classe Oferta por intermédio da relação **acessa** e também se relaciona com a classe Produto por meio da relação **compra**, especificando que um cliente pode acessar ofertas e comprar produtos; a classe Vendedor, que herda os atributos, operações e relacionamentos da classe Usuario e se relaciona com a classe Produto por meio da relação **cadastra**, indicando que um vendedor, além de possuir dados, operações e relacionamentos da classe Usuario, ainda poderá cadastrar produtos.

Com isso abordamos quase todas as classes do diagrama, exceto a classe Oferta. Essa, por sua vez, agrega objetos da classe Produto e se relaciona com objetos da classe Cliente por meio da relação inversa (implícita) **acessada por**, o que significa que uma oferta contém produtos e é acessada por clientes.

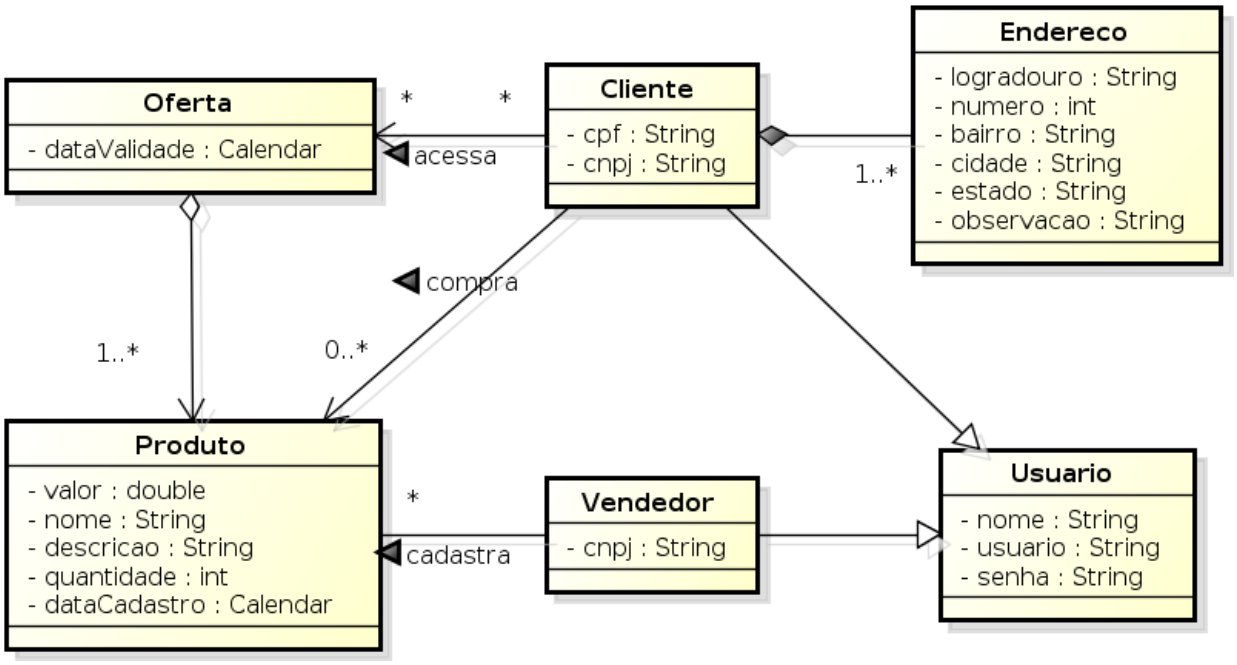

Figura 1. Diagrama do Modelo de Domínio da aplicação de exemplo.

Apresentado o diagrama, já é possível avançar para a arquitetura construída a partir das entidades exibidas no Modelo de Domínio. As classes contidas nesse Modelo (**Figura 1**) serão as nossas entidades "persistentes", também conhecidas como modelo da aplicação, ou seja, essas serão as entidades salvas em banco de dados.

Logo acima dessas classes existe uma camada que contém todas as operações referentes ao banco de dados. As classes que concentram tais operações são

Camada 2, que contém as classes expostas no diagrama da **Figura 1**.

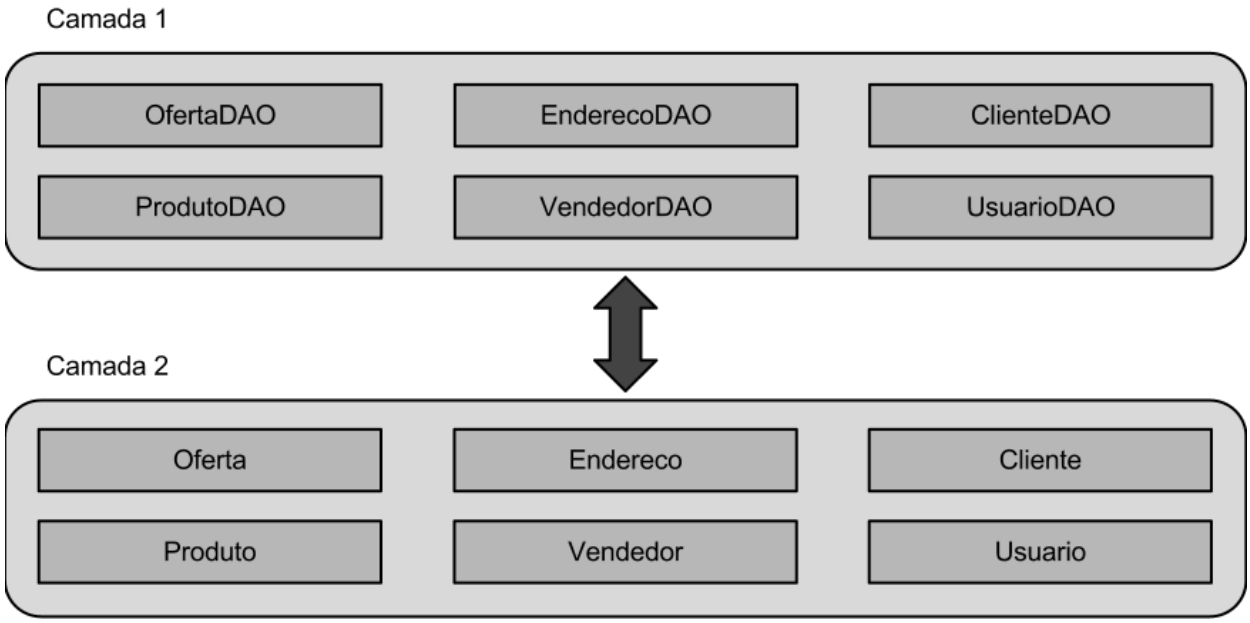

Figura 2. Representação das camadas da aplicação de exemplo.

Até este ponto foi apresentada a estrutura da aplicação exemplo, suas entidades persistentes e a divisão em camadas do software. Deste modo, avançaremos para a explicação da classe auxiliar EntityManagerUtils, que centraliza a criação de um objeto custoso utilizado em vários pontos da aplicação.

Mas para entender a utilidade dessa classe, antes é necessário explicar o que é um EntityManager e um EntityManagerFactory. Iniciaremos pela primeira, que como o próprio nome sugere, instâncias da classe EntityManager gerenciam entidades. Isso significa que essas instâncias contêm operações para manipular uma entidade persistente ou que está para ser persistida. Dentre elas, podemos citar a recuperação de entidades previamente salvas em banco de dados (que é o

isso em apenas um ponto da aplicação.

Devido ao fato de instanciar o EntityManagerFactory ser custoso, foi escrita a classe EntityManagerUtils, que centraliza essa operação. Veja na **Listagem 5** os detalhes de implementação dessa classe.

Listagem 5. Código da classe EntityManagerUtils.

```
1  package br.com.core;
2
3    import javax.persistence.EntityManager;
4    import javax.persistence.EntityManagerFactory;
5    import javax.persistence.Persistence;
6
7    public class EntityManagerUtils {
8
9     private static final EntityManagerFactory entityManagerFactory =
10      .createEntityManagerFactory("artigo");
11
12    private EntityManagerUtils() {
13     // Construtor vazio.
14    }
15
16    public static EntityManagerFactory getEntityManagerFactory() {
17     return entityManagerFactory;
18    }
19
20    public static EntityManager createEntityManager() {
21     return entityManagerFactory.createEntityManager();
22    }
23   }
```

tentará instanciá-la.

Efetuando consultas nativas

Efetuar consultas nativas é, sem dúvida, a forma mais antiga e conhecida entre os desenvolvedores mais experientes, mas isso não significa que eles a usem até os dias de hoje para resolver todos os problemas. Essas consultas são denominadas nativas porque elas podem ser escritas na linguagem nativa do banco de dados ou ainda usar funções próprias do SGBD em questão.

> Uma das justificativas para se escrever consultas nativas é o aumento de desempenho, já que o ORM não precisa interpretar a query antes de enviá-la para o banco. Além disso, em geral, as funções nativas possuem uma interface mais fácil de usar.

Apesar de ser possível aproveitar essas funções, o mais comum é escrever as consultas em uma linguagem conhecida como *Structured Query Language*, ou simplesmente SQL. Essa é uma linguagem padrão que possibilita a comunicação entre a aplicação e o banco de dados. Ademais, ela está disponível em todos os SGBDs mais conhecidos, como MySQL, Oracle, PostgreSQL, SQLite, entre outros.

> Seguir esse padrão e não usar funções específicas de determinado banco significa que será mais fácil portar a aplicação para outro banco de dados, caso seja necessário.

projetada (ou cresceu de forma desorganizada), então ela tem alguns problemas de relacionamento, onde só é possível recuperar as informações usando funções nativas do banco de dados.

O cenário explicado no parágrafo anterior, dependendo do nicho de atuação do leitor, não é muito comum, porém serve para exemplificar a utilidade das consultas nativas. Visto que para montar esse cenário seria necessário distorcer muito a estrutura da aplicação de exemplo, iremos demonstrar como utilizar uma consulta nativa com um exemplo simples.

Nesse exemplo teremos que exibir uma oferta para o usuário final. Para isso, precisamos escrever uma consulta que dado um código (**id**), uma oferta deve ser retornada. Veja como ficou a consulta na **Listagem 6**.

Listagem 6. Consultando uma oferta a partir de um id.

```
1.  public class OfertaDAO {
2.   public Oferta buscaOfertaPorId(Integer id) {
3.   try {
4.    return (Oferta) entityManager
5.      .createNativeQuery("SELECT * FROM oferta WHERE id = :id",
6.                         Oferta.class)
7.      .setParameter("id", id)
8.      .getSingleResult();
9.   } catch (NoResultException e) {
10.     return null;
11.  }
12.  }
13. }
```

linha 7.

Dentre as maneiras possíveis de passar parâmetros a uma consulta, essa é a mais eficiente, pois assim não é necessário escrever o parâmetro estaticamente na consulta ou ainda depender da ordem dos parâmetros, como no caso exibido na **Listagem 7**.

Esse método (da **Listagem 6**) é conhecido como **parâmetros nomeados**, e além das vantagens citadas acima, o leitor pode levar em consideração que eles são auto documentáveis. Observe que é fácil entender que a notação :id se refere ao campo id da tabela e que essa notação pode ser usada na query como se fosse uma variável, podendo ser repetida em várias partes da SQL. Assim, quando o método setParameter() for invocado, todos os lugares que possuírem a notação :id serão substituídos pelo valor correspondente.

Listagem 7. Passando parâmetro para uma consulta por posição.

```
1  public class OfertaDAO {
2     public Oferta buscaOfertaPorId(Integer id) {
3      try {
4       return (Oferta) entityManager
5         .createNativeQuery("SELECT * FROM oferta WHERE id = ?",
6           Oferta.class)
7         .setParameter(1, id)
8         .getSingleResult();
9      } catch (NoResultException e) {
10      return null;
11     }
12    }
13   }
```

dificultando a manutenção do código e ainda existiria a preocupação com a ordem dos parâmetros.

A documentação do Hibernate chama esse método de JDBC-style parameters ou, em tradução livre, estilo JDBC de parâmetros.

Analisando novamente o código da **Listagem 6**, a última observação a ser feita é referente ao método getSingleResult(), que dada uma consulta, retorna apenas um resultado. O único porém é quando nada é encontrado. Quando isso ocorre o método lança a exceção NoResultException.

Não se esqueça de fazer um cast do retorno desse método, pois ele retorna uma instância da classe Object e não uma instância da classe Oferta. Nesse caso, em especial, se o cast não for feito, um erro de compilação ocorre, já que a assinatura do método buscaOfertaPorId() diz que ele retorna uma Oferta.

Além dos métodos já apresentados para pesquisar uma entidade a partir de um **id**, o EntityManager fornece o método find(Object.class, id), que tem o mesmo efeito das consultas exibidas nas **Listagens 6** e **7**. Veja o código da **Listagem 8**.

Listagem 8. Consultando uma entidade por id por meio do método find().

```
1  public class OfertaDAO {
2     public Oferta buscaOfertaPorId(Integer id) {
3      try {
4       return entityManager.find(Oferta.class, id);
5      } catch (NoResultException e) {
6       return null;
7      }
```

contaremos com a Java Persistence Query Language (JPQL), especificada na *JSR 338 Java Persistence*. No Hibernate, ela é conhecida como Hibernate Query Language (HQL).

Essa linguagem não lida diretamente com as tabelas do banco de dados, e sim com objetos. Sendo assim, ela é capaz de compreender herança, polimorfismo e associações. Com essa abstração fornecida, cada linha de uma tabela, quando resgatada, gerará automaticamente um objeto. Portanto, não mais escreveremos consultas baseadas em tabelas, como descrito abaixo:

```
1 | SELECT * FROM oferta
```

Daqui para frente, devemos escrever consultas baseadas em entidades do nosso domínio. Veja abaixo um exemplo de consulta empregando a linguagem JPQL:

```
1 | FROM Oferta
```

Repare que não é obrigatório escrever “SELECT *” como na query nativa, e o mais importante, agora selecionamos entidades (Oferta) no lugar de tabelas (*oferta*). A **Listagem 9** ilustra o conceito explicado.

Listagem 9. Consulta que usa a JPQL.

```
1 public class OfertaDAO {
2    public Oferta buscaOfertaPorIdJPQL(Integer id) {
3     try {
4      return entityManager
```




Como verificado, a consulta se torna bastante simples, sendo importante ressaltar mais uma vez que ela é escrita de forma orientada a objetos. Devido a isso, o cast de Object para Oferta também não é mais necessário.

Outro detalhe interessante é que é possível definir um apelido para a entidade. Esse apelido é chamado de **alias**. Isso é importante caso exista a necessidade de referenciar a entidade em outras partes da query, como no exemplo a seguir:

```
1 | FROM Oferta oferta JOIN oferta.produtos WHERE oferta.id = 1
```

É uma boa prática nomear o alias com a primeira letra em minúscula, do mesmo modo como é feito ao se seguir a convenção de nomenclatura do Java. Prosseguindo com o artigo, também é possível relacionar entidades em uma consulta (igual ao que fizemos na **Listagem 4**). Esse relacionamento é efetuado com base nas anotações definidas nas entidades (@OneToMany, @ManyToOne, @OneToOne, etc.). As palavras-chave adotadas para montar essas consultas são: join, left join, right join e full join.

INNER JOIN

INNER JOIN pode ser abreviado apenas para JOIN e serve para trazer como resultado duas tabelas, baseando-se nos dados em comum entre ambas. Geralmente o relacionamento é feito por meio de chaves primárias e chaves estrangeiras. Veja na **Listagem 10** um exemplo.

Listagem 10. Exemplo de uso do INNER JOIN

resultados cuja condição endereco.usuario_id = usuario.id é satisfeita. O resultado da consulta da **Listagem 10** juntamente com o equivalente em JPQL pode ser visualizado na **Listagem 11**.

Listagem 11. Resultado da consulta com INNER JOIN e o equivalente em JQPL.

```
mysql> SELECT * FROM usuario INNER JOIN endereco ON (endereco.usuari
  +----+----------------+--------+---------+----+------------------+
  | id | nome           | senha  | usuario | id | logradouro       |
  +----+----------------+--------+---------+----+------------------+
  |  1 | Pedro da Silva | 123456 | pedro   |  1 | Rua Oscar Freire |
  +----+----------------+--------+---------+----+------------------+

  JPQL> FROM Usuario u JOIN u.enderecos
```

Apenas para deixar mais claro, caso a condição endereco.usuario_id = usuario.id não fosse satisfeita, ou seja, caso o usuário não possuísse endereço, a consulta não retornaria nenhum resultado.

LEFT JOIN

LEFT JOIN é na verdade a abreviação de LEFT OUTER JOIN. Isso significa que caso a condição de relação não seja satisfeita é retornado apenas a entidade da esquerda, pois a comparação é feita da esquerda para a direita. Veja na **Listagem 12** um exemplo de quando um usuário não possui endereço, com o equivalente em JPQL.

Listagem 12. Demonstra a pesquisa por um usuário que não possui endereço e o

```
7
8   JPQL> FROM Usuario u LEFT JOIN u.enderecos
```

Repare que todos os campos da tabela *endereco* estão com o valor NULL.

RIGHT JOIN

RIGHT JOIN tem o funcionamento parecido com o LEFT JOIN. A única diferença é que a comparação da condição é efetuada da direita para a esquerda. Logo, caso exista um endereço sem usuário, será exibido apenas o endereço. Veja na **Listagem 13** um exemplo.

Listagem 13. Demonstra a pesquisa por um endereço sem usuário e o equivalente em JQPL.

```
1   mysql> SELECT * FROM usuario RIGHT JOIN endereco ON (endereco.usuari
2     +------+------+-------+---------+----+------------------+--------
3     | id   | nome | senha | usuario | id | logradouro       | cidade
4     +------+------+-------+---------+----+------------------+--------
5     | NULL | NULL | NULL  | NULL    |  1 | Rua Oscar Freire | São Paul
6     +------+------+-------+---------+----+------------------+--------
7
8     JPQL> FROM Usuario u RIGHT JOIN u.enderecos
```

Note que todos os campos da tabela *usuario* estão com o valor NULL.

FETCH for usado, todos os endereços serão carregados no objeto da classe Usuario, inclusive quando se configura um relacionamento para trabalhar no modo LAZY. Veja na **Listagem 14** um exemplo.

Listagem 14. Exemplo do uso de FETCH em um relacionamento um-para-muitos.

```
1  public class UsuarioDAO {
2     public List<Usuario> buscarUsuariosEEnderecos() {
3      return entityManager.createQuery(
4        "FROM Usuario usuario JOIN FETCH usuario.enderecos")
5        .getResultList();
6     }
7    }
```

Repare na palavra-chave FETCH na consulta. Ela fará com que todos os endereços dos usuários da consulta sejam carregados. A **Listagem 15** demonstra como o Hibernate montou a consulta.

Listagem 15. Consulta gerada pelo Hibernate.

```
1  select
2        usuario0_.id as id3_0_,
3        enderecos1_.id as id1_1_,
4        usuario0_.nome as nome3_0_,
5        usuario0_.senha as senha3_0_,
6        usuario0_.usuario as usuario3_0_,
7        enderecos1_.bairro as bairro1_1_,
8        enderecos1_.cidade as cidade1_1_,
9        enderecos1_.estado as estado1_1_,
10       enderecos1_.logradouro as logradouro1_1_,
```

```
19      endereco enderecos1_
20          on usuario0_.id=enderecos1_.usuario_id
```

Note que ele selecionou todos os campos da tabela *usuario* e da tabela *endereço*. Em seguida, “por debaixo dos panos”, montou a lista de endereços de cada usuário e a atribuiu aos usuários retornados pela consulta.

Como verificado nos exemplos, a JPQL é poderosa e propicia total controle sobre as entidades a serem manipuladas. Além dos recursos que foram abordados, ela possui ainda muitos outros. Deste modo, caso o leitor deseje obter mais informações, na seção **Links** há um endereço correspondente a consultas efetuadas com HQL, que é a implementação do Hibernate.

Escrevendo consultas nomeadas

Consultas nomeadas, em geral, são aplicadas para organizar e separar as SQLs do código Java. Essa abordagem é interessante quando a aplicação tem poucas SQLs e as consultas são efetuadas com a API de critérios (abordada mais adiante neste artigo).

Sendo assim, as consultas nomeadas representam uma maneira simples de atribuir um nome a uma consulta, como se fosse um alias, para que em seguida seja possível referenciá-la em vários pontos da aplicação. A **Listagem 16** exibe um exemplo desse tipo de consulta.

```
4  public class Vendedor {
5   // …
6  }
7
8  public class VendedorDAO {
9
10  private EntityManager entityManager;
11
12  // …
13
14  public Vendedor buscaVendedorPorCNPJ(String cnpj) {
15   try {
16    return entityManager
17      .createNamedQuery("buscaVendedorPorCNPJ", Vendedor.class)
18      .setParameter("cnpj", cnpj).getSingleResult();
19   } catch (NoResultException e) {
20    return null;
21   }
22  }
23 }
```

Na classe Vendedor é definida uma consulta nomeada com a anotação @NamedQuery. Essa anotação tem como parâmetros obrigatórios os atributos name e query. Observe também que a consulta recebe um CNPJ por parâmetro explicitado pela notação :cnpj. Feito isso, a consulta agora pode ser referenciada a partir do nome buscaVendedorPorCNPJ.

Note que no método buscaVendedorPorCNPJ(), a variável cnpj foi informada como parâmetro para a query, assim como foi feito na **Listagem 6** (com createNativeQuery()), no método buscaOfertaPorId(), e na **Listagem 9** (com

Embora escrever consultas com a JQPL seja uma forma bastante poderosa, há desenvolvedores que preferem usar um meio mais dinâmico, tal como a consulta baseada em objetos (critérios) do Hibernate.

Até agora lidamos com a interface provida pela especificação do JPA. Entretanto, daqui para frente faremos uso direto do Hibernate. Para isso, precisamos de uma linha de código que dado um EntityManager, uma instância de uma classe que implemente a interface Session seja criada. Este recurso é provido pelo método unwrap(), definido na interface EntityManager.

O objeto obtido a partir do método unwrap() é equivalente a objetos das classes que implementam a interface EntityManager, empregados até o momento. Faremos esse *unwrap* porque os métodos definidos na interface Session são mais simples e em geral é necessário menos linhas de código para se trabalhar com a *Criteria API*, em comparação com a interface EntityManager.

Na **Listagem 17** expomos o código de como extrair um objeto Session a partir da instância de um EntityManager.

Listagem 17. Extraindo um objeto de uma classe que implemente a interface Session a partir de um EntityManager.

```
1 | Session session = entityManager.unwrap(Session.class);
```

Com o objeto session em mãos, podemos instanciar a API de critérios do Hibernate. Na **Listagem 18** é demonstrado um exemplo de uso da Criteria. O

```
1  public class VendedorDAO {
2
3     private Session session;
4
5     public VendedorDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
7     }
8
9     public Vendedor buscaVendedorPorCNPJ(String cnpj) {
10     Criteria criteria = session.createCriteria(Vendedor.class);
11     criteria.add(Restrictions.eq("cnpj", cnpj));
12     return (Vendedor) criteria.uniqueResult();
13    }
14
15   }
```

Na primeira linha do método buscaVendedorPorCNPJ(), instanciamos um objeto da classe Criteria a partir do objeto session, que por sua vez foi obtido do EntityManager (conforme o código da **Listagem 16**).

> Pode-se dizer que objetos da classe **Session**, além de outras atribuições, são também uma fábrica para instâncias da classe **Criteria**.

Na segunda linha do método, especificamos uma restrição de igualdade. Dessa forma, a API adicionará na consulta a ser efetuada no banco de dados a condição: where cnpj = <numero_do_cnpj>. Por fim, na última linha do método, pedimos ao Hibernate para procurar por um vendedor cujo CNPJ seja o mesmo fornecido por

O propósito da cláusula like é auxiliar em pesquisas onde seja necessário buscar por determinado texto que inicie, termine ou contenha determinados caracteres. Em conjunto com esta cláusula é usado o caractere %, que funciona como um coringa. No exemplo da **Listagem 19** buscamos por usuários cujo nome inicie com os caracteres informados por parâmetro.

Listagem 19. Buscando usuários que possuem o nome iniciado com o valor informado por parâmetro.

```
1  public class UsuarioDAO {
2
3     private Session session;
4
5     public UsuarioDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
7     }
8
9     public List<Usuario> buscaUsuarioPorNomeIniciadoCom(String nome)
10     Criteria criteria = session.createCriteria(Usuario.class);
11     criteria.add(Restrictions.like("nome", nome + "%"));
12     return criteria.list();
13    }
14
15   }
```

É importante notar a posição do coringa “%”. Se ele estivesse concatenado ao início do nome ao invés de estar no final (“%” + nome), o método pesquisaria por todos os usuários que possuíssem no final de seu nome os caracteres informados

A cláusula LIKE é bastante útil em pesquisas por String, porém é necessário usar com sabedoria e realmente quando necessário, pois ela consome muito processamento, visto que para cada linha da tabela é necessário comparar caractere por caractere até encontrar um diferente ou confirmar a validez do resultado encontrado. Esse cenário em uma tabela com muitos valores é crítico, por isso use com moderação.

Adicionando a cláusula BETWEEN na consulta

Suponha que na nossa aplicação de exemplo teremos que desenvolver um recurso que exiba para o administrador todas as ofertas que ocorreram há uma semana. Para isso, será necessário escrever uma consulta que pesquise entre domingo e sábado.

Justamente para esses casos existe a cláusula between, que pesquisa por números e datas dentro de um determinado período. O código da **Listagem 20** ilustra esse exemplo. Repare na segunda linha do método, onde é adicionada uma restrição com a cláusula between.

Listagem 20. Pesquisando por ofertas da semana passada.

```
1  public class OfertaDAO {
2
3     private Session session;
4
5     public OfertaDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
```

```
14    }
15
```

Conforme mencionado anteriormente, também é possível pesquisar por um intervalo de números. Por exemplo, suponha que em uma tela de exibição exista um filtro que permita ao usuário pesquisar por produtos informando uma faixa de preço. A **Listagem 21** exibe como implementar esse recurso.

Listagem 21. Pesquisando por produtos de acordo um intervalo de valores.

```
1  public class ProdutoDAO {
2
3     private Session session;
4
5     public ProdutoDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
7     }
8
9     public List<Produto> buscaProdutoPorFaixaDePreco(Double inicio, D
10     Criteria criteria = session.createCriteria(Produto.class);
11     criteria.add(Restrictions.between("valor", inicio, fim));
12     return criteria.list();
13    }
14
15  }
```

Repare na segunda linha do método buscaProdutoPorFaixaDePreco(). Assim como no código da **Listagem 20**, uma restrição foi inserida na query com a cláusula BETWEEN (Restrictions.between()), porém desta vez os parâmetros

Índices são campos marcados em uma tabela que quando devidamente selecionados aumentam consideravelmente o desempenho das consultas. Geralmente os SGBDs mantêm em um arquivo separado os índices e seus valores, então antes de pesquisar em uma tabela inteira pelos valores desejados, ele olha nesse arquivo, por isso a consulta fica mais rápida.

Adicionando a cláusula IN na consulta

Empregar a cláusula in nas consultas é geralmente algo raro. No entanto, o seu uso é bastante útil durante a depuração de aplicações (quando se está procurando a causa de um problema) e também para resgatar várias entidades pesquisando por **id**s.

O funcionamento do in é parecido com o = ou com o LIKE, entretanto, ao invés de pesquisar por coisas iguais ou que contenham determinado caractere, é pesquisado por várias coisas cujo campo comparado esteja entre os valores informados na cláusula. Por exemplo, é possível resgatar de uma só vez os produtos cujos **id**s sejam 1, 2, 7 e 10. O código da **Listagem 22** aplica esse exemplo com Criteria. Repare na segunda linha do método, onde temos a restrição in da classe Restrictions.

Listagem 22. Pesquisando por produtos usando a cláusula IN.

```
1  public class ProdutoDAO {
2
3     private Session session;
```

```
12      return criteria.list();
13    }
14
15  }
```

Este método é tão rápido quanto pesquisar por valores iguais; principalmente quando o campo pesquisado é um índice.

Organizando logicamente os parâmetros

Durante o desenvolvimento, esporadicamente surge à necessidade de agrupar os parâmetros passados a uma query, ou seja, verificar se um conjunto de comparações é verdadeiro **ou** se outro conjunto de comparações é verdadeiro.

Isso se assemelha a adicionar ordem de precedência em um if com os parênteses e operadores lógicos (&& e ||). Para melhor exemplificar essa organização lógica, veja o código da **Listagem 23**, onde será pesquisado por todos os produtos cujo valor seja menor ou igual a 10, **ou** trazer os resultados que satisfaçam às duas condições seguintes:

1. Verifica se o produto tem valor maior do que 10;

2. **Ou** quantidade igual a 100.

Listagem 23. Buscando produtos com parâmetros logicamente organizados.

```
1  public class ProdutoDAO {
2
```

```
    criteria.add(Restrictions.or(
     Restrictions.le("valor", 10.0),
     Restrictions.disjunction()
      .add(Restrictions.gt("valor", 10.0))
      .add(Restrictions.eq("quantidade", 100))
    ));

    return criteria.list();
   }

}
```

O que o Hibernate acaba fazendo é adicionar parênteses entre os grupos organizados logicamente. Entretanto existe um pequeno problema nessa organização, a complexidade inserida nas queries. Geralmente as queries começam simples e com o passar do tempo vão ficando complexas. Por isso, quanto mais níveis de organização lógica forem inseridos na query, mais difícil será de entender o que ela faz.

Uma query com vários níveis de separação lógica pode ser comparada a um método com vários ifs encadeados. Ambos se tornam complexos e consequentemente difíceis de entender e de manter.

Ordenando o resultado das consultas

Ordenar consultas é quase sempre necessário, seja para exibir os registros mais recentes primeiro (ordenação por data) ou para resgatar tudo em ordem alfabética. Enfim, os motivos são vários. O fato é que a ordenação é necessária

Dito isto, vamos a um exemplo. Na Criteria, o método que insere essa cláusula na query é o addOrder(). Este será invocado em nossa aplicação no cenário onde uma listagem de produtos de uma determinada oferta precisa ser ordenada do produto mais barato para o mais caro.

Veja como é feita esta ordenação na terceira linha do método buscaProdutosPorOferta(), apresentado da **Listagem 24**.

Listagem 24. Ordenando os produtos por preço.

```
1  public class ProdutoDAO {
2
3     private Session session;
4
5     public ProdutoDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
7     }
8
9     public List<Produto> buscaProdutosPorOferta(Integer ofertaId) {
10     Criteria criteria = session.createCriteria(Produto.class);
11     criteria.createCriteria("oferta").add(Restrictions.eq("id", ofer
12     criteria.addOrder(Order.asc("valor"));
13     return criteria.list();
14    }
15
16  }
```

Nesta consulta, o método addOrder() é informado de que é necessário ordenar a query pelo campo valor de modo crescente. Observe a declaração

```
1  public class ProdutoDAO {
2
3     private Session session;
4
5     public ProdutoDAO(EntityManager entityManager) {
6      session = entityManager.unwrap(Session.class);
7     }
8
9     public List<Produto> buscaProdutosPorOferta(Integer ofertaId) {
10     Criteria criteria = session.createCriteria(Produto.class);
11     criteria.createCriteria("oferta").add(Restrictions.eq("id", ofer
12     criteria.addOrder(Order.desc("valor"));
13     return criteria.list();
14    }
15
16  }
```

Ordenar o resultado das consultas facilita bastante a visualização da informação, no entanto, além de resgatar os dados, o banco de dados ainda precisará ordená-los. Portanto, haverá um custo a mais para a consulta. Apesar disso, dependendo da aplicação, esse custo a mais pode ser irrelevante, ou justificado pelo benefício proporcionado ao usuário.

Conclusão

No decorrer do artigo foram apresentadas as diferentes formas de consulta disponíveis na biblioteca ORM do Hibernate. Analisamos as consultas por meio de uma linguagem nativa, consultas com a JPQL (ou HQL, no Hibernate),

teve seus principais pontos ressaltados.

Foi mencionado ainda que a grande maioria das aplicações tem a necessidade de armazenar dados. Sendo assim, é muito importante relembrar que os desenvolvedores estão ligados diretamente com este requisito. Por isso, é interessante conhecer sobre o assunto e estar munido dos mais diferentes mecanismos para quando a necessidade de persistência surgir.

Cabe agora ao leitor mergulhar nos detalhes e peculiaridades da API, buscando mais informações sobre o assunto. Para isso, você pode tomar como ponto de partida os links citados na seção de referências.

Links

http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/ch18.html
Efetuando consultas nativas.

http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/ch16.html
Efetuando consultas com HQL.

http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/ch17.html
Documentação da API de critérios.

Tecnologias:

Banco de Dados Hibernate Java ORM UML