

MANUAL JPQL

Contenido:

1. [JPQL BASICO](#)
2. [SENTENCIAS CONDICIONALES](#)
3. [PARAMETROS DINAMICOS](#)
4. [ORDENAR LOS RESULTADOS](#)
5. [OPERACIONES DE ACTUALIZACION](#)
6. [OPERACIONES DE BORRADO](#)
7. [EJECUCION DE SENTENCIAS CON PARAMETROS DINAMICOS](#)
8. [CONSULTAS CON NOMBRE \(ESTATICAS\)](#)
9. [CONSULTAS NATIVAS SQL](#)

1. JPQL BASICO

Con `EntityManager` estamos limitados a realizar consultas en la base de datos proporcionando la identidad de la entidad que deseamos obtener, y solo podemos obtener una entidad por cada consulta que realicemos. JPQL nos permite realizar consultas en base a multitud de criterios, como por ejemplo propiedades de la entidad almacenada o condiciones booleanas, y obtener mas de un objeto por consulta. Veamos el ejemplo mas simple posible:

```
SELECT e FROM Empleado e
```

La linea anterior obtiene todas las instancias de la clase `Empleado` desde la base de datos. La expresion puede parecer un poco extraña la primera vez que se ve, pero es muy sencilla. Las palabras `SELECT` y `FROM` tienen un significado similar a las sentencias homonimas de Oracle, indicando que se quiere seleccionar (`SELECT`) cierta información desde (`FROM`) cierto lugar. La segunda `'e'` es un alias para la clase `Empleado`, y ese alias es usado por la primera `'e'` (llamada 'expresión') para acceder a la clase a la que refiere el alias o a sus propiedades. El siguiente ejemplo nos ayudara a comprender esto mejor:

```
SELECT e.vchEmplusuario FROM Empleado e
```

Facil, ¿verdad?. El alias `e` nos permite utilizar la expresión `e.vchEmplusuario` para obtener los usuarios de todas los empleados almacenadas en la base de datos. La expresiones JPQL utilizan la notación de puntos, convirtiendo tediosas consultas en algo realmente simple:

```
SELECT c.propiedad.subPropiedad.subSubPropiedad FROM Clase c
```

JPQL tambien nos permite obtener resultados referentes a mas de una propiedad:

```
SELECT e.vchEmplusuario,e.vchEmplclave FROM Empleado e
```

Todas las sentencias anteriores (que mas tarde veremos como ejecutar) devuelven o un unico valor o un conjunto de ellos. Podemos eliminar los resultados duplicados mediante la clausula DISTINCT:

```
SELECT DISTINCT e.vchEmplusuario FROM Empleado e
```

Así mismo, el resultado de una consulta puede ser el resultado de una función agregada aplicada a la expresión:

```
SELECT COUNT(e) FROM Empleado e
```

En el ejemplo anterior, COUNT() es una función agregada de JPQL que cuenta el numero de ocurrencias tras realizar la consulta. El valor devuelto por la función agregada es lo obtenemos. Otras funciones agregadas son AVG para la media aritmética, MAX para el valor máximo, MIN para el valor mínimo y SUM para la suma de todos los valores.

[inicio](#)

2. SENTENCIAS CONDICIONALES

Ahora que ya sabemos como realizar consultas basicas, vamos a introducir conceptos mas complejos (pero aun simples). El primero de ellos es el de consulta condicional, aplicado con la clausula WHERE, la cual restringe los resultados devueltos por una consulta en base a ciertos criterios lógicos (desde ahora la mayoría de los ejemplos constaran de varias líneas pero ten en cuenta que son unas únicas sentencias JPQL, no varias):

```
SELECT m FROM Movimiento m
```

```
WHERE m.decMoviimporte < 5000
```

La sentencia anterior obtiene todas las instancias de Movimiento almacenadas en la base de datos con un importe inferior a 5000. Las sentencias condicionales pueden contener varias condiciones:

```
SELECT m FROM Movimiento m
```

```
WHERE m.decMoviimporte < 5000 AND m.dttMovifecha = '01/08/2008 '
```

La sentencia anterior obtiene todas las instancias de Movimiento con con un importe inferior a 5000 y cuya fecha sea '01/08/2008 '. La otra sentencia condicional además de AND es OR (aplicado en el caso del ejemplo anterior solo una de las dos condiciones sería suficiente). Veamos mas ejemplos:

```
SELECT m FROM Movimiento m  
WHERE m.decMoviimporte BETWEEN 1000 AND 2000
```

La sentencia anterior obtiene todas las instancias de Movimiento con una importe entre (BETWEEN) 1000 y (AND) 2000 minutos. BETWEEN puede ser convertido en NOT BETWEEN en cuyo caso se obtendrían los que no se encuentre dentro del margen 1000-2000.

Otro operador de comparación muy útil es [NOT] LIKE (NOT es opcional, como en el ejemplo anterior), el cual nos permite comparar una cadena de texto con comodines con las propiedades de una entidad almacenada en la base de datos. Veamos un ejemplo para comprenderlo:

```
SELECT e FROM Empleado e  
WHERE e.vchEmplpaterno LIKE 'cast's%'
```

La sentencia anterior obtiene todas las instancias de Empleado cuyo apellido paterno sea como (LIKE) 'cast...', osea la palabra 'cast' seguida de mas caracteres (entre cero y varios).

El comodin que representa 'cero o mas caracteres' es el simbolo de porcentaje (%). El otro comodin aceptado por LIKE es el caracter de barra baja (_) el cual representa un solo caracter. JPQL dispone de muchos operadores de comparación y estos son solo un par ejemplos.

[inicio](#)

3. PARAMETROS DINAMICOS

Podemos añadir parámetros dinámicamente a nuestras sentencias JPQL de dos maneras: por posición y por nombre. La sentencia a continuación indica parámetros por posición:

```
SELECT e FROM Empleado e  
WHERE e.vchEmplusuario = ?1
```

Y la siguiente, parámetros por nombre:

```
SELECT e FROM Empleado e  
WHERE e.vchEmplusuario = :vchEmplusuario
```

En el momento de realizar la consulta, podemos pasar los parámetros a la sentencia JPQL. Esto lo veremos mas adelante.

[inicio](#)

4. ORDENAR LOS RESULTADOS

Cuando realizamos una consulta a la base de datos, podemos ordenar los resultados mediante la clausula ORDER BY (ordenar por), la cual admite ordenamiento ascendente mediante la clausula ASC (comportamiento por defecto) u en orden descendiente mediante la clausula DESC:

```
SELECT m FROM Movimiento m  
  
ORDER BY m. decMoviimporte  DECC
```

[inicio](#)

5. OPERACIONES DE ACTUALIZACION

JPQL puede realizar operaciones de actualizacion en la base de datos mediante la sentencia UPDATE

```
UPDATE Cuenta c  
  
SET c.decCuensaldo = 1000  
  
WHERE c.chrCuencodigo = '00200001'
```

La sentencia anterior actualiza (UPDATE la instancia de Cuenta con un número de cuenta 00200001) SET un saldo nuevo de 1000.

[inicio](#)

6. OPERACIONES DE BORRADO

JPQL puede realizar operaciones de borrado en la base de datos mediante la sentencia DELETE

```
DELETE FROM Movimiento m  
  
WHERE m. decMoviimporte < 5000
```

La sentencia anterior elimina (DELETE) todas las instancias de Movimiento cuyo importe sea menor de 5000.

[inicio](#)

7. EJECUCION DE SENTENCIAS JPQL

JPQL es integrado a través de implementaciones de la interface Query. Dichas implementaciones se obtienen a través de nuestro querido amigo EntityManager mediante diversos métodos de factoría. Los tres más típicamente usados (y los que explicaremos aquí) son:

```
createQuery(String jpql)
```

```
createNamedQuery(String name)
```

```
createNativeQuery(String sql)
```

Veamos cómo crear un objeto Query y realizar una consulta a la base de datos:

```
package mjllanos.main;
```

```
import java.util.List;
```

```
import javax.persistence.EntityManager;
```

```
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.Persistence;
```

```
import mjllanos.entity.Empleado;
```

```
public class Main {
```

```
    public List consulta() {
```

```
        List results = null;
```

```
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("simpleJAPU");
```

```
        EntityManager em = emf.createEntityManager();
```

```
        javax.persistence.Query q = em.createQuery("SELECT p FROM Empleado p");
```

```
        results = (List) q.getResultList();
```

```
        em.close();
```

```

        emf.close();

        return results;
    }

    public static void main(String[] args) {

        Main m = new Main();

        List results = null;

        results = m.consulta();

        for (Empleado empl : (List<Empleado>) results) {

            System.out.println("Empleado: (" + empl.getChrEmplcodigo()

                + " ; " + empl.getVchEmplpaterno()

                + " ; " + empl.getVchEmplmaterno()

                + " ; " + empl.getVchEmplnombre()

                + " ; " + empl.getVchEmplciudad() + ")");

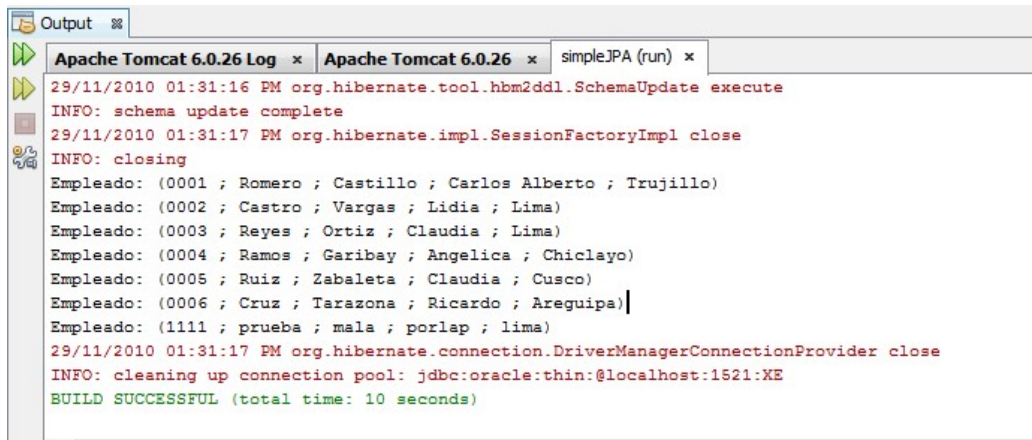
        }

    }

}

```

En el ejemplo anterior obtenemos una implementacion de Query mediante el metodo createQuery(String) de EntityManager, al cual le pasamos una sentencia JPQL en forma de cadena de texto. Con el objeto Query ya inicializado, podemos realizar la consulta a la base de datos llamando a su metodo getResultList() el cual devuelve un objeto List con todas las entidades alcanzadas por la sentencia JPQL. Esta sentencia es una sentencia dinamica, ya que es generada cada vez que se ejecuta. En el ejemplo obtenenos:



```
Output
Apache Tomcat 6.0.26 Log x Apache Tomcat 6.0.26 x simpleJPA (run) x
29/11/2010 01:31:16 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: schema update complete
29/11/2010 01:31:17 PM org.hibernate.impl.SessionFactoryImpl close
INFO: closing
Empleado: (0001 ; Romero ; Castillo ; Carlos Alberto ; Trujillo)
Empleado: (0002 ; Castro ; Vargas ; Lidia ; Lima)
Empleado: (0003 ; Reyes ; Ortiz ; Claudia ; Lima)
Empleado: (0004 ; Ramos ; Garibay ; Angelica ; Chiclayo)
Empleado: (0005 ; Ruiz ; Zabaleta ; Claudia ; Cusco)
Empleado: (0006 ; Cruz ; Tarazona ; Ricardo ; Arequipa)
Empleado: (1111 ; prueba ; mala ; porlap ; lima)
29/11/2010 01:31:17 PM org.hibernate.connection.DriverManagerConnectionProvider close
INFO: cleaning up connection pool: jdbc:oracle:thin:@localhost:1521:XE
BUILD SUCCESSFUL (total time: 10 seconds)
```

[inicio](#)

8. EJECUCION DE SENTENCIAS CON PARAMETROS DINAMICOS

Ya se vio como escribir sentencias JPQL con parámetros dinámicos. Ahora que sabemos como funciona Query veamos como insertar dichos parámetros:

```
String jpql = "SELECT e FROM Empleado e WHERE e.vchEmplusuario = ?1";
```

```
Query q = em.createQuery(jpql);
```

```
q.setParameter(1,"lcastro");
```

```
List<Empleado> results = q.getResultList();
```

En el ejemplo anterior insertamos dinámicamente (mediante el metodo `setParameter(int, Object)`) los valores deseados para `e.vchEmplusuario`, que en la sentencia JPQL se corresponden al parámetro de posición `?1`. Si el valor que pasamos no se corresponde con el valor esperado, la aplicación lanzara una excepción de tipo `IllegalArgumentException`. Esto también ocurrirá si ajustamos una posición inexistente.

Así mismo, podemos utilizar parámetros por nombre en nuestras sentencias JPQL:

```
String jpql = "SELECT e FROM Empleado e WHERE e.vchEmplusuario = :usuario";
```

```
Query q = em.createQuery(jpql);
```

```
q.setParameter("usuario","lcastro");
```

```
List<Empleado> results = q.getResultList();
```


Ahora en vez de utilizar ? hemos utilizado : usuario. Al ajustar este valor mediante el metodo setParameter(String, Object) hemos incluido los nombres del parámetro (sin los dos puntos iniciales) y sus valores correspondientes, haciendo la aplicación mas sencilla de leer y mantener.

[inicio](#)

9. CONSULTAS CON NOMBRE (ESTATICAS)

Las consultas con nombre son diferentes de las sentencias dinamicas que hemos visto hasta ahora en el sentido en que no pueden cambiar: son leidas y transformadas en sentencias SQL cuando el programa arranca en lugar de cada vez que son ejecutadas. Este comportamiento estatico las hace mas eficientes y por tanto ofrecen un mejor rendimiento. Las consultas con nombre son definidas mediante metadatos (recuerda que los metadatos se definen mediante anotaciones o configuracion XML) definidos en las propias entidades. Veamos un ejemplo:

```
@Entity
```

```
@Table(name = "EMPLEADO")
```

```
@NamedQuery(name = "Empleado.findAll", query = "SELECT e FROM Empleado e")
```

```
public class Empleado implements Serializable {
```

El código anterior define una consulta con nombre a traves de la anotacion @NamedQuery. Esta anotacion necesita dos atributos: name que define el nombre de la consulta, y query que define la sentencia JPQL a ejecutar. El nombre de la consulta debe ser unico dentro de su unidad de persistencia, de manera que no puede existir otra entidad definiendo una consulta con nombre que se llame findAll. Para evitar que podamos modificar por error la sentencia, es una buena idea utilizar una constante definida dentro de la propia entidad como nombre de la consulta:

```
@Entity
```

```
@Table(name = "EMPLEADO")
```

```

@NamedQuery(name = Empleado.BUSCAR_TODOS, query = "SELECT e FROM Empleado e")

public class Empleado implements Serializable {

public static final String BUSCAR_TODOS= "Empleado.findAll";

...

}

```

Tanto de la primera manera, como de esta ultima, una vez definida una consulta con nombre podemos ejecutarla desde nuestra aplicación mediante el segundo método de la lista: `createNamedQuery()`:

```

Query query = em.createNamedQuery(Empleado.findAll);

List<Empleado> results = query.getResultList();

```

`createNamedQuery()` requiere un parametro de tipo `String` que contenga el nombre de la consulta (el cual hemos definido en `@NamedQuery(name=...)`). Una vez obtenida la coleccion de resultados, podemos trabajar con ellos de la manera habitual.

[inicio](#)

10. CONSULTAS NATIVAS SQL

El tercer y ultimo tipo de consultas que vamos a ver requiere una sentencia SQL en lugar de una JPQL:

```

String sql = "SELECT * FROM EMPLEADO";

Query query = em.createNativeQuery(sql);

...

```

Las consultas nativas SQL pueden ser definidas de manera estática como las consultas con nombre, obteniendo los mismos beneficios de eficiencia y rendimiento. Para ello, necesitamos utilizar de nuevo metadatos:

```
@Entity
```

```
@Table(name = "EMPLEADO")
```

```
@NamedQuery(name = "Empleado.findAll", query = "SELECT e FROM Empleado e")
```

```
public class Empleado implements Serializable {
```

```
@Entity
```

```
@NamedNativeQuery(name = Empleado.BUSCAR_TODOS, query="SELECT * FROM  
EMPLEADO")
```

```
public class Empleado implements Serializable {
```

```
public static final String BUSCAR_TODOS= "Empleado.findAll";
```

```
...
```

```
}
```

Tanto las consultas con nombre como las consultas nativas SQL estáticas son muy útiles para definir consultas inmutables (por ejemplo buscar todas las instancias de una entidad en la base de datos, como hemos hecho en los ejemplos anteriores), aquellas que se mantienen entre distintas ejecuciones del programa y que son usadas frecuentemente, etc.

[inicio](#)