Acceso a Datos

UT4. CONSULTAS

1. Introducción

De poco servirían nuestras aplicaciones si no pudiéramos consultar los datos que tenemos almacenados en nuestras bases de datos. JPA e Hibernate nos proveen de un lenguaje de consulta a nivel de entidades (JPA propone **JPQL** (Java Persistence Query Language), e Hibernate **HQL** (Hibernate Query Language)).

Una consulta JPQL se considera una consulta HQL válida, pero no todas las consultas HQL son consultas JPQL válidas.

Las consultas HQL son traducidas por *Hibernate* a consultas SQL convencionales.

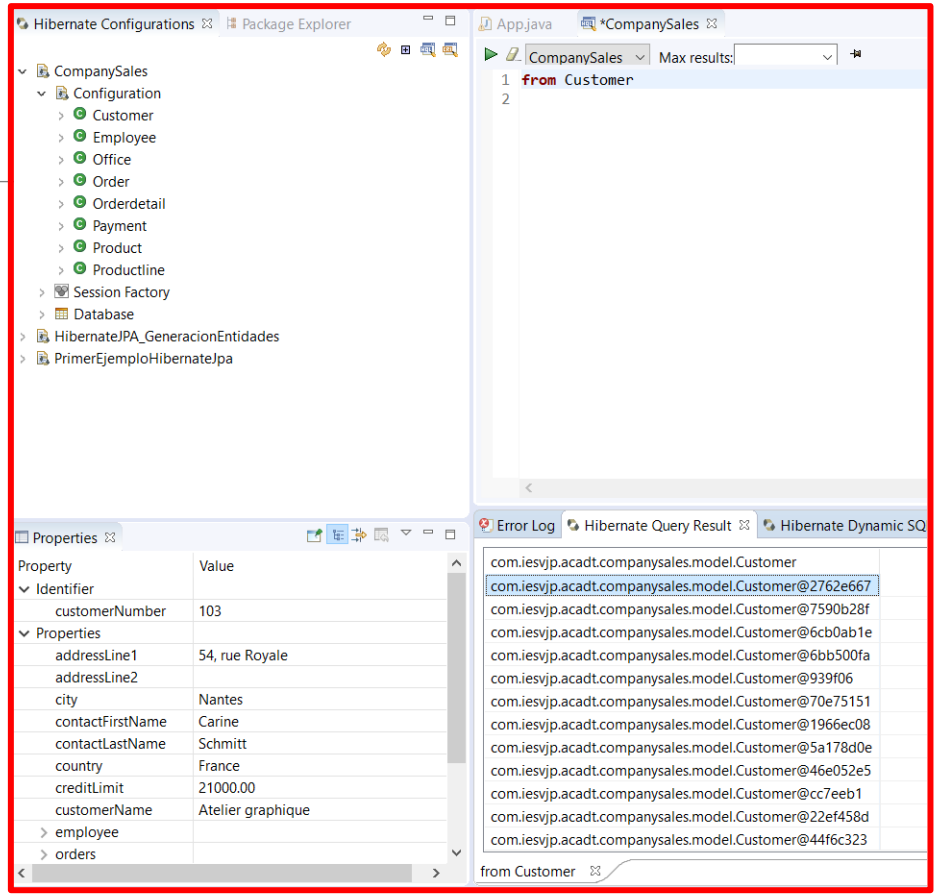
Vamos a partir de la base de datos **company-sales** que ya tenemos mapeada del ejercicio anterior:

1. Introducción

1. Introducción

Antes de ejecutar las consultas en java es recomendable probarlas antes, comprobar que funciona correctamente y pegarla en el código java, para ello **Hibernate Tools** nos proporciona una consola para probar consultas HQL/JPQL, no se permiten sentencias UPDATE, INSERT O DELETE. Para ello será necesario abrir una nueva perspectiva:

Window→Perspective→Open Perspetive→Other y buscamos Hibernate. Una vez abierta elegimos nuestro archivo de configuración de persistencia y podemos ya probar las consultas:



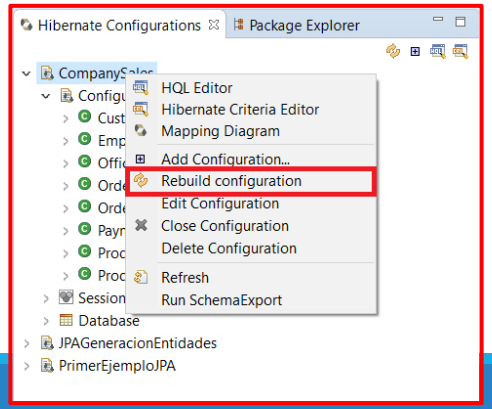
1. Introducción

Puede que al ejecutar la consulta nos dé el siguiente error, eso es debido a que la relación es bidireccional y tenemos que poner **insertable=false, updatable=false**.

1. Introducción

```
//bi-directional many-to-one association to Customer
@ManyToOne
@JoinColumn(name="customerNumber", insertable=false, updatable=false)
private Customer customer;
```

Una vez hechos los cambios, será necesario volver a reconstruir la configuración:



2. JPQL básico

Por lo que hemos trabajado anteriormente, solo hemos podido obtener, a través del *EntityManager*, entidades a partir de su identificador, y solamente una entidad por consulta. Para aquellos que ya conozcan SQL esto resultará un enfoque muy pobre. JPQL nos va a permitir realizar consultas en base a muchos criterios, así como obtener más de un valor por consulta.

La consulta JPQL más básica tiene la siguiente estructura:

```
SELECT c FROM Customer c
```

2. JPQL básico

Esta sentencia obtiene todas las instancias de Customer que existan en la base de datos. La expresión puede parecer un poco extraña la primera vez que se ve, pero es muy sencilla de entender. Las palabras SELECT y FROM tienen un significado similar a las sentencias homónimas del lenguaje SQL, indicando que se quiere seleccionar (SELECT) cierta información desde (FROM) cierto lugar. La segunda c es un alias para la clase Customer, y ese alias es usado por la primera c (llamada expresion) para acceder a la clase (tabla) a la que hace referencia el alias, o a sus propiedades (columnas). El siguiente ejemplo nos ayudará a comprender esto mejor:

```
SELECT c.customerName FROM Customer c
```

2. JPQL básico

El alias c nos permite utilizar la expresión *c.CustomerName* para obtener los nombres de todos los clientes almacenados en la base de datos. Las expresiones JPQL utilizan la notación de puntos, pudiendo acceder a objetos anidados:

```
SELECT e.propiedad.anidada.masanidada FROM Entidad e
```

Al igual que en SQL, JPQL nos permite obtener más de una propiedad:

```
SELECT c.customerName, c.city, c.country FROM Customer c
```

2.1. Consultas con parámetros

JPQL nos permite la inclusión de parámetros en base a un índice y en base a un nombre:

```
SELECT e FROM Employee e WHERE e.jobTitle = :jobTitle
```

```
SELECT o FROM Order o WHERE o.orderDate BETWEEN ?1 AND ?2 AND status= ?3
```

2.2. Consultas con salida ordenada

También nos permite la ordenación de los resultados, al estilo SQL. Este orden puede ser ascendente (ASC) o descendente (DESC).

```
SELECT o FROM Order o ORDER BY o.orderDate DESC
```

3. JPA Query

JPA nos provee de dos interfaces, *javax.persistence.Query* y *javax.persistence.TypedQuery*, que se obtienen directamente desde el *EntityManager*. Para ello, podemos usar el método *EntityManager#createQuery*. Para consultas con nombre (las veremos más adelante), podemos usar el método *EntityManager#createNamedQuery*.

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

EntityManager em = emf.createEntityManager();

 //Consulta 1 – Básica
System.out.println("\n\nConsulta 1 - Básica");
Query query = em.createQuery("select c from Customer c");

List<Customer> listCustomer = (List<Customer>) query.getResultList();

System.out.println("LISTADO DE TODOS LOS CLIENTES");
System.out.println("============================");
   for(Customer c : listCustomer)
        System.out.println(c.getCustomerName());

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

3. JPA Query

Para aquellas consultas que devuelven más de un resultado, tenemos a nuestra disposición el método *getResultList();* si la consulta devuelve un solo resultado, tendremos entonces que llamar a *getSingleResult().*

En HQL no existe el comando LIMIT, por lo tanto para recuperar el primer resultado será necesario utilizar el método *setMaxResults(1).*

```
em.createQuery(“….”).setMaxResults(1);
```

3. JPA Query

Para asignar los parámetros dinámicos a las consultas, tenemos a nuestra disposición diferentes versiones del método *setParameter*, que acepta tanto un índice como un nombre de parámetro, y un objeto para utilizar en la consulta.

```
Query query = em.createQuery( "SELECT e FROM Employee e WHERE
e.jobTitle = :jobTitle");

query.setParameter("jobTitle", "Sales Rep");

Query query = em.createQuery( "SELECT o FROM Order o WHERE o.orderDate
BETWEEN ?1 AND ?2 AND status = ?3 ORDER BY o.orderDate DESC" );
```

```
LocalDate ld1= LocalDate.of(2003, 1, 1);
LocalDate ld2= LocalDate.of(2003, 5, 31);


query.setParameter(1, Date.valueOf(ld1));
query.setParameter(2, Date.valueOf(ld2));
query.setParameter(3, "Shipped");
```

4. Joins con JPQL

La operación JOIN es una operación básica en SQL, y sirve para realizar una reunión conveniente entre dos tablas a partir de una clave externa.

4.1. Joins explícitos

Como no podía ser de otra manera, JPQL permite realizar los mismos JOIN que en SQL (JOIN, INNER JOIN, LEFT | RIGHT JOIN), si bien uno de los más interesantes el *JOIN FETCH*.

```
Query query = em.createQuery(
     "select c "
     + "from Customer c "
     + "left join fetch c.employee "
);
```

4.1. Joins explícitos

Para que un JOIN FETCH sea válido, la asociación debe estar en alguna de las clases que estén descritas en la cláusula SELECT

Comparativa de JOIN y JOIN FETCH: https://www.arquitecturajava.com/jpa-join-fetch-uso

Otro ejemplo con JOINS sobre varias tablas sería: *Muestra aquellos empleados que hayan atendido a clientes que hayan comprado el producto 1969 Ford Falcon*:

4.1. Joins explícitos

```
Query query = em.createQuery(
                        "select o.orderNumber, e.employeeNumber,
                         e.firstName, e.lastName\n" +
                        "from Product p\n" +
                        "inner join p.orderdetails d\n" +
                        "inner join d.order o\n" +
                        "inner join o.customer c\n" +
                        "inner join c.employee e\n" +
                        "where p.productName= '1969 Ford Falcon'\n"  );


        List<Object[]> list = (List<Object[]>) query.getResultList();
        System.out.println("LISTADO DE EMPLEADOS (Order, Empleado
                                    Number, Nombre, Apellidos)");
        System.out.println("============================");
        for(Object[] o : list) {
                System.out.print((Integer)o[0] + ", ");
                System.out.print((Integer)o[1] + ", ");
                System.out.print((String)o[2]+ "-");
                System.out.println((String)o[3]);
         }
```

4.2. Joins implícitos

Un JOIN implícito se realiza siempre a través de alguna de las asociaciones de una entidad, navegando a través de sus propiedades.

```
TypedQuery<Customer> query = em.createQuery(
    "select c "
  + "from Customer c "
  + "where c.employee = :employee", Customer.class);

query.setParameter("employee", em.find(Employee.class, 1370));
Query query = em.createQuery("select p.customer.customerName,
                              sum(p.amount) " +
                             "from Payment p " +
                             "group by p.customer");

List<Object[]> list = (List<Object[]>) query.getResultList();
System.out.println("LISTADO DEL GASTO TOTAL DE LOS CLIENTES (Nombre,
total gastado)");

System.out.println("============================");
   for(Object[] cliente : list) {
        System.out.print((String) cliente[0] + "--");
        System.out.println((BigDecimal) cliente[1] + "$");
                                                  }
```

5. Consultas de actualización

JPQL también nos permite lanzar consultas de actualización de datos. La estructura sintáctica de las consultas es muy similar a la de SQL nativo:

```
UPDATE Entity e
SET e.prop1 = newValue1, e.prop2 = newValue2, ...
WHERE ...
```

```
DELETE FROM Entity e
WHERE ...
```

5. Consultas de actualización

Para invocarlas, tenemos que utilizar el método *executeUpdate* de la interfaz *Query*.

```
em.getTransaction().begin();

//UPDATE

//Incremento del 10% en el límite de crédito a todos los clientes
int numUpdateResults = em.createQuery(
    "update Customer c "
    + "set c.creditLimit = c.creditLimit * 1.1")
    .executeUpdate();
System.out.println("Número de registros afectados: " +
numUpdateResults);
```

(Continúa en la siguiente diapositiva)

5. Consultas de actualización

```
//DELETE

Date date = null;
try {
date = new SimpleDateFormat("dd/MM/yyyy").parse("06/06/2003");

  int numDeleteResults = em.createQuery(
      "delete from Payment p "
      + "where p.paymentDate = :fecha")
      .setParameter("fecha", date)
      .executeUpdate();

  System.out.println("Número de registros afectados: " +
numDeleteResults);


} catch (ParseException e) {
System.err.println("Error en el parseo de la fecha");
}

em.getTransaction().commit();
```

5. Consultas de actualización

En la sentencia UPDATE solo puede haber una única entidad nombrada. No se pueden especificar JOINS, ni implícitos ni explícitos. Pero si se pueden utilizar subconsultas en la cláusula WHERE, donde podemos incluir JOINS.

```
update User set code = (select c.code from City c where c.id = 1000)
```

JPQL no permite la ejecución de sentencias INSERT. Sin embargo, HQL si que lo permite:

```
int insertedEntities = session.createQuery(
     "insert into Partner (id, name) " +
     "select p.id, p.name " +
     "from Person p ")
.executeUpdate();
```

6. NamedQueries

Las consultas con nombre son un tipo de consultas especiales ya que, una vez que son definidas, no pueden ser modificadas. Son leídas y transformadas en SQL durante la inicialización del contexto de persistencia. Por ello, son más eficientes y ofrecen un rendimiento mayor. Se suelen definir mediante anotaciones en las clases entidad, si bien también pueden declararse en XML.

El asistente de Eclipse que nos ha generado las entidades desde las tablas nos ha incluido una sentencia por defecto en cada una de ellas:

```
@Entity
@Table(name="customers")
@NamedQuery(name="Customer.findAll", query="SELECT c FROM Customer c")
     public class Customer implements Serializable {
     //Resto del código
}
```

6. NamedQueries

Como podemos comprobar, la anotación @NamedQuery requiere de dos argumentos: el nombre que le vamos a dar a la consulta, y la definición de la consulta en sí.

Para ejecutar esta consulta, tenemos que utilizar el método *createNamedQuery* de *EntityManager*.

```
Query query = em.createNamedQuery("Customer.findAll");
```

Se pueden añadir más de una consulta con nombre, a través de la anotación @NamedQueries. Las consultas también pueden recibir parámetros.

6. NamedQueries

```
@Entity
@Table(name="customers")
@NamedQueries({
        @NamedQuery(name="Customer.findAll", query="SELECT c FROM
Customer c"),
        @NamedQuery(name="Customer.findByName", query="SELECT c FROM
Customer c WHERE c.customerName LIKE :name"),
        @NamedQuery(name="Customer.findByEmployee", query="SELECT c
FROM Customer c WHERE c.employee = :employee"),
})
public class Customer implements Serializable {

}
```

7. Consultas con SQL nativo

JPA e Hibernate también permiten la ejecución de SQL nativo (en particular, el dialecto del RDBMS que estemos usando). Esto es muy útil cuando nuestro sistema tiene funcionalidades específicas, o si nuestra experiencia en SQL nos permite implementar sentencias que sean realmente eficientes.

7.1. Consultas escalares

La consulta más básica es aquella en la que obtenemos una lista de valores en un Object[].

```
List<Object[]> employees = entityManager.createNativeQuery(
    "SELECT employeeNumber, firstName, lastName FROM
employees").getResultList();

for(Object[] employee : employees) {
    Number employeeNumber = (Number) employee[0];
    String firstName = (String) employee[1];
    String lastName = (String) employee[2];
}
```

7.2. Consultas de entidades

También podemos obtener entidades desde consultas nativas:

```
List<Customer> customers = entityManager.createNativeQuery(
     "SELECT * FROM customers", Customer.class ).getResultList();
```

Dado que la entidad Customer tiene asociaciones y algunas de ellas están mapeadas con fetch EAGER, al ejecutar este ejemplo, JPA+Hibernate rescatan el resto de datos necesarios a través de consultas JPQL.

7.3. Consultas SQL con nombre

Por último, decir que también podemos definir consultas con nombre, como ocurría con JPQL. Estas se definen a través de la anotación *@NamedNativeQuery*.

```
@Entity
@Table(name="employees")
@NamedQuery(name="Employee.findAll", query="SELECT e FROM Employee e")
@NamedNativeQuery(name="Employee.nativeFindAll", query="SELECT * FROM
employees", resultClass=Employee.class)
public class Employee implements Serializable {
  //Resto de código
}
```

Su uso también es muy sencillo:

```
List<Employee> employeesList =
em.createNamedQuery("Employee.nativeFindAll").getResultList();
```

8. Expresiones para consultas HQL

- Las consultas en HQL no son sensibles a mayúsculas, **a excepción de los nombres de las clases y propiedades Java**. Podemos escribir FROM, from, SELECT, sElect, etc.

- La cláusula más simple que existe en Hibernate es **from**, que obtiene todas las instancias de una clase, por ejemplo **from Empleados** obtiene todas las instancias de la clase Empleados (en SQL, obtiene todas las filas de la tabla EMPLEADOS). La cláusula order by ordena los resultados de la consulta.

- La cláusula **where** permite refinar la lista de instancias retornadas y **order by** ordena la lista. Ejemplos:

```
from Empleados where departamentos.deptNo=20 order by apellido
from Empleados as em where departamentos.deptNo=30 order by 1 desc
from Empleados as em where em.departamentos.deptNo=20
```

8. Expresiones para consultas HQL

- Podemos asignar alias a las clases usando la cláusula AS: `from Empleados as em`.
- Para obtener determinadas propiedades (columnas) en una consulta utilizamos la cláusula SELECT: `select apellido, salario from Empleados`, obtiene los atributos apellido y salario de la clase empleados.
- Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo **Object[],** una lista, un map o una clase. En apartados anteriores vimos algunos ejemplos.

8. Expresiones para consultas HQL

- Las funciones de grupo soportadas son las siguientes, la semántica es similar a SQL:
  - `avg(...), sum(...), min(...), max(...)`
  - `count(*)`
  - `count(...), count(distinct ...), count (all...)`

- Se puede utilizar alias para nombrar los atributos y expresiones. Se pueden utilizar operadores aritméticos, de concatenación y funciones SQL reconocidas en la cláusula SELECT. Veamos algunos ejemplos:

```
select avg (salario) as med, count (empNo) as num from Empleados
select avg(salario) , count(empNo) from Empleados
select apellido ||'--OFICIO--' || oficio as campo from Empleados
select count(distinct departamentos.loc)from Empleados
```

8. Expresiones para consultas HQL

Las expresiones utilizadas en la cláusula WHERE incluyen lo siguiente:

- operadores matemáticos: `+, -, *, /`
- operadores de comparación binarios: `=, >=, <=, <>, ! = , like`
- operadores lógicos `and, or, not`
- Paréntesis ( ) que indican agrupación
- `in, not in, between, is null, is not null, is empty, is not empty…`
- concatenación de cadenas … | | … o `concat (...,...)`
- `current_date(), current_time() y current_timestamp()`
- `minute(...), hour (...), day(...), month(...), y year(...)`
- Cualquier función u operador definido por EJB-QL 3.0: `substring (), trim (), lower (), upper(), length(), abs(), sqrt(), mod(),coalesce() y nullif()`
- `str ()` para convertir valores numéricos o temporales a una cadena legible.
- Cualquier función escalar SQL soportada por la base de datos como `sign(), trunc(), rtrim() y sin()`
- parámetros posicionales JDBC ? Y parámetros con nombre: `:name, : start_date y :xl`

Fuente: https://docs.jboss.org/hibernate/stable/core.old/reference/en/html/queryhql-expressions.html

8. Expresiones para consultas HQL

Ejemplos:

```
from Empleados where departamentos.deptNo in (10,20)
from Empleados where departamentos.deptNo not in (10,20)
from Empleados where salario between 2000 and 3000
select lower(apellido),coalesce(comision, 0) from Empleados
```

Se pueden agrupar consultas usando group by y having. Las funciones SQL y las funciones de agregación SQL están permitidas en las cláusulas having y order by, si están soportadas por la base de datos subyacente. Ni la cláusula group by ni la cláusula order by pueden contener expresiones aritméticas. Ejemplos:

```
select departamentos.dnombre, avg(salario) from Empleados
group by departamentos.dnombre
having avg(salario) > 2000
```

8. Expresiones para consultas HQL

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis. Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior). Ejemplos:

```
from Empleados as em where em.salario >= (select avg(salario) from Empleados)
```

Dudas y preguntas