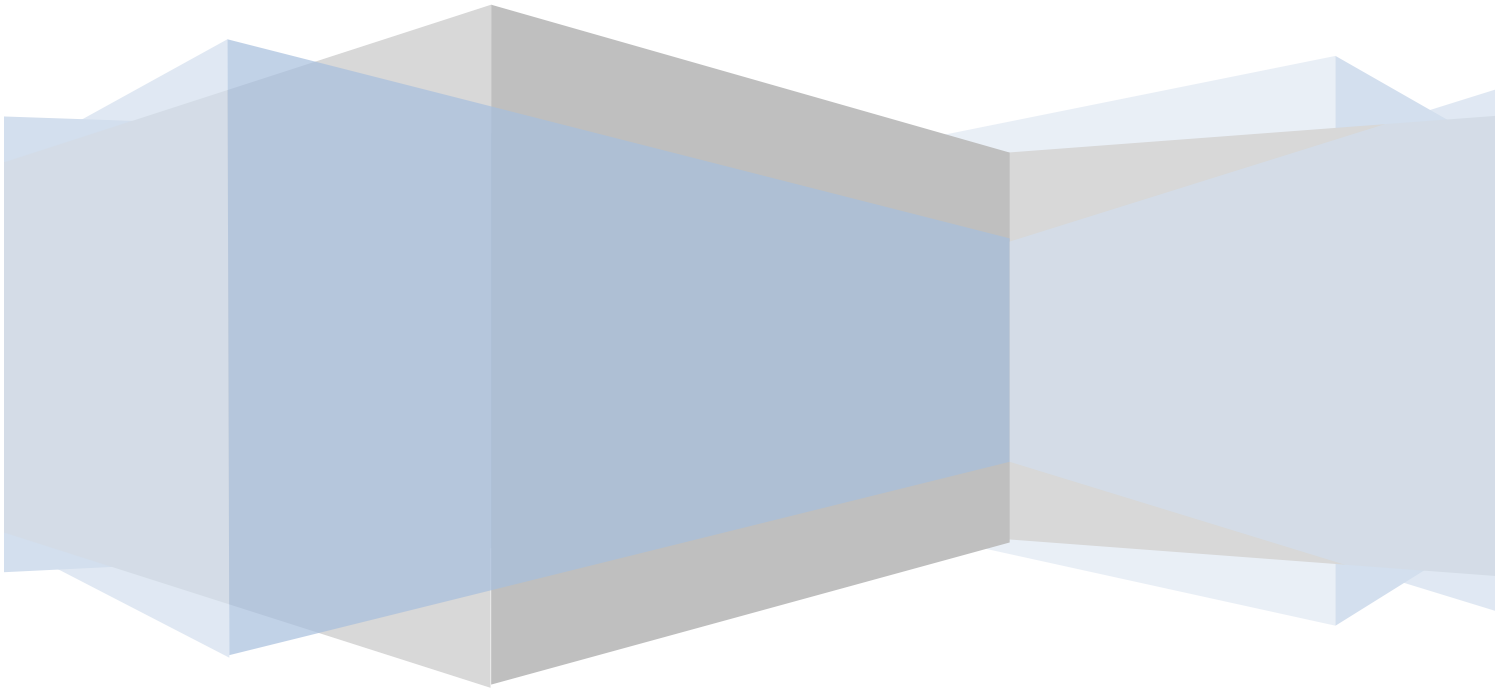
Universidad de Zaragoza

Práctica 4:

Explotación de una base de datos utilizando Java Persistance API (JPA)

Adrian Casans (590114) Sergio Pedrero (627669) Diego Sánchez (628279) Cristian Simón (611487)



ÍNDICE

Esfuerzos invertidos:

Tabla que recoge un desglose con las tareas y esfuerzos realizados por cada uno de los miembros del grupo junto con el tiempo invertido:

Tarea	Encargados	Horas
1 - Desarrollo Clases java	Cristian	2 horas
1 - Generar esquema con Hibernate	Diego	4 horas
2 – Desarrollo ficheros XML para mapeo O/R	Sergio	4 horas
2 – Prueba correcta correspondencia	Diego	
3 - Formular consultas	Adrián	
Memoria	Cristian/Sergio	2 horas

1. Objetivos:

El objetivo del desarrollo de la practica consistirá en la utilización de Java Persistance API (JPA), concretamente Hibernate como middleware, para realizar un mapeo objeto/relacional.

- Se deberá generar automáticamente un esquema relacional a partir de un modelo de clases en Java.
- Se establecerán las correspondencias entre un modelo de clases Java y un esquema relacional existente.
- Se realizaran una serie de consultas utilizando alguna o varias de las siguientes opciones: JPQL, Criteria API y SQL nativo.

2. Generación automática del esquema de BD:

Para la generación automática del esquema de la base de datos a partir de un modelo en Java lo primero que hemos realizado es el desarrollo de las clases que posteriormente serán las tablas de nuestra BD. Para ello se debe incluir en la cabecera de cada clase Java la siguiente línea o anotación:

@Entity

A continuación , lo primero que tendremos que hacer será definirnos la clave primaria de cada tabla mediante la notación:

@Id

Si tuviéramos que incluir claves compuestas se pueden definir en una clase a parte, marcada con la anotación **@Embbedable** y utilizar la siguiente anotación e n la definición del identificador correspondiente:

@EmbebbedId

Despues de esto, deben definirse todos los atributos que van a representar a las diferentes columnas dentro de la tabla, y que deben ser definidos con los tipos de datos que se van a implementar en la BD.

Acto seguido nos hemos definido las relaciones entre las clases Java para que posteriormente se pueda hacer el mapeo correctamente. Para ello se han utilizado las siguientes notaciones:

@ManyToOne

@OneToOne

@OneToMany

@ManyToMany

Por último hemos definido un constructor sin argumentos, ya que toda clase JPA debe tener uno, además de otro con los diferentes atributos de la clase, así como los métodos necesarios para la interacción con los atributos. Es importante y útil redefinir el método que codifica los atributos de la clase en una cadena de caracteres(habitualmente toString).

El resultado de esto se puede visionar en el directorio /fuentes/java-gen-esquema.zip de esta misma practica.

3. Esquema BD preexistente:

En el caso de integrar la base de datos mediante JPA partiendo de un esquema de BD preexistente, se deben seguir una serie de pasos distintos a los expuesto anteriormente.

En este caso además hemos optado por separar la implementación en clases Java y ficheros XML para realizar la correcta correspondencia entre objetos y base de datos.

Las clases Java únicamente contienen los atributos y los métodos necesarios para tratar esos atributos. En este caso, las anotaciones no se hacen dentro del mismo fichero de clase que representa a un objeto, sino que para cada una de ellas se ha generado un fichero XML que contiene las sentencias necesarias para relacionar los atributos con la correspondiente función que desempeñan en la base de datos.

Es decir, por ejemplo para el caso de los Clientes, tenemos una clase Cliente.java perteneciente al package "parte2" que tendría la estructura clásica de una clase orientada a objetos en Java(sin anotaciones) y además tendríamos el fichero Cliente.hbm.xml que es el que mapea esta información mediante Hibernate para la base de datos Oracle.

Usando los atributos **"table"** de los XML se hace la correspondencia entre tablas, **"colum"** para los atributos y usando los elementos descritos en el apartado anterior realizamos la correspondencia de claves, tipos de datos y relaciones.

Además se pueden establecer ciertas propiedades como **"not-null",** la longitud de cada columna con **"length",** entre otros.

Para ver el resultado final de esto, en la práctica se dispone de un fichero llamado "java-previo-esquema.zip" en el que se ve con detalle todo lo descrito anteriormente.

4. Consultas:

Se han formulado una serie de consultas siguiendo las opciones JPQL, Criteria API y SQL Nativo. Estás consultas se han realizado sobre la parte 2(correspondencia):

Consulta 1:

Obtener la oficina a la que pertenece una cuenta corriente:

```
private static Oficina consulta1(EntityManager em, String numCuenta) {
            String consulta1 = "SELECT cc.oficina FROM parte2.Cuenta cc
                  WHERE cc.nCuenta = :numCuenta";

            return (Oficina)
em.createQuery(consulta1).setParameter("numCuenta",
                        numCuenta).getSingleResult();
}
```

Consulta 2:

Obtener las cuentas que posean alguna operacion con importe mayor que una cantidad dada:

```
private static List<Cuenta> consulta2(EntityManager em, float
      cantidad) {
            String consulta2 = "SELECT c FROM parte2.Cuenta c,
                        parte2.Operacion o WHERE c.nCuenta = o.cuenta
                  AND o.cantidad > :cant";

            return em.createQuery(consulta2).setParameter("cant",
                        cantidad).setMaxResults(100).getResultList();
}
```

Consulta 3:

Obtener los clientes con un nombre dado que entre sus cuentas tengan alguna con una determinada cantidad:

```
private static List<Cliente> consulta3(EntityManager em, String
      nombre, double cantidad) {
            String consulta3 = "SELECT c FROM parte2.Cliente c JOIN
                        c.listaCuentas lc WHERE c.nombre LIKE :nom
                  AND lc.saldo > :cant";

            return em.createQuery(consulta3).setParameter("cant",
                        cantidad).setParameter("nom",

      nombre).setMaxResults(100).getResultList();
}
```

Consulta 4:

Obtener las cuentas que posean un saldo superior a una cantidad dada:

```
private static List<Cuenta> consulta4(EntityManager em, double
      cantidad) {

      CriteriaBuilder cb = em.getCriteriaBuilder();

      CriteriaQuery<Cuenta> q = cb.createQuery(Cuenta.class);
      Root<Cuenta> c = q.from(Cuenta.class);
      CriteriaQuery<Cuenta> select = q.select(c);

      ParameterExpression<Double> p = cb.parameter(Double.class);
      select.where(cb.gt((Expression) c.get("saldo"), p));

      TypedQuery<Cuenta> typedQuery = em.createQuery(select);
      typedQuery.setParameter(p, cantidad);

      return typedQuery.getResultList();
}
```

Consulta 5:

Obtener la operación que más dinero a movido:

```
private static List<Operacion> consulta5(EntityManager em) {

      CriteriaBuilder cb = em.getCriteriaBuilder();

      CriteriaQuery<Operacion> q = cb.createQuery(Operacion.class);
      Root<Operacion> c = q.from(Operacion.class);
      CriteriaQuery<Operacion> select = q.select(c);

      select.orderBy(cb.desc(c.get("cantidad")));

      TypedQuery<Operacion> typedQuery = em.createQuery(select);

      return typedQuery.setMaxResults(1).getResultList();
}
```