ORM (II)

CRUD en Hibernate

Vamos a implementar un simple CRUD en Hibernate. Crea un proyecto Maven partiendo del fichero **EmpresaHibernateBase.zip**. A continuación, arranca el MySQL del XAMPP (o el que quieras). No hace falta que hagas nada con el MySQL, basta con que esté arrancado. Generaremos la Base de Datos gracias a Hibernate desde el Eclipse.

Notarás que al proyecto Java le falta código. Es lo que iremos completando poco a poco para hacer funcionar el CRUD.

A modo de resumen, los pasos para completar el ejemplo son:

1. Generar Fichero **hibernate.cfg.xml**
2. Crear **HibernateUtil**
3. Configura las **entidades**
4. Codificar los **DAO**
5. Usar los DAO

Fichero hibernate.cfg.xml

Este fichero es el archivo de configuración de Hibernate. Establece cómo van a hacerse las conexiones contra la base de datos.

```
<hibernate-configuration>
    <session-factory>

        <!-- Configuración de conexión -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/hibernateExample?createDatabaseIfNotExist=true
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>

        <!-- Dialecto para MySQL -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Mostrar el SQL generado -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- Actualizar el esquema automáticamente -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Registrar las entidades -->
        <mapping
            class="EmpresaHibernateExample.EmpresaHibernateExample.enterprise.ddbb.entity.Empresa" />
        <mapping
            class="EmpresaHibernateExample.EmpresaHibernateExample.enterprise.ddbb.entity.Empleado" />

    </session-factory>
</hibernate-configuration>
```

Este fichero TIENE que estar en la carpeta resources.

Entre las propiedades destacan:

- Configuración de conexión: similar a los datos suministrados en JDBC, informan del driver, de la url de la BBDD, su esquema… nótese que le estamos indicando que la BBDD se cree si no existe.

- Actualizar el esquema: esto es importante. Dependiendo de la opción que le indiquemos pueden suceder tres cosas diferentes:

    - **Create**: se crea la BD desde cero cada vez que arrancas la App a partir de las entidades. Se suele ejecutar una vez al inicio del proyecto, y luego se cambia por Update. Nótese que en el ejemplo en la URL hay puesta otra condición para la creación de la BBDD, luego no pasa nada si lo dejamos todo el update.

    - **Update**: actualiza las tablas, si hay cambios. Es la más habitual.

    - **None:** No se toca la BD hagas lo que hagas.

- Registrar entidades: Todas las entidades / tablas de la BDD deben de listarse aquí, o serán ignoradas por Hibernate.

- Otras opciones incluyen mostrar en consola las SQL que va ejecutando Hibernate, el dialecto, etc.

HibernateUtil

Esta clase se implementa normalmente como un Patrón Singleton. Se emplea para cargar el hibernate.cfg.xml y gestionar el SessionFactory. Puedes colocar esta clase donde quieras.

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            return new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Error inicializando SessionFactory: " + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

Las Entidades

El mapeo real entre entidad y tabla de BBDD se hace en las propias clases de Entidad. Lo habitual suele ser diseñar primero el esquema E/R de tablas, y después crear las propias Entidades. Una vez hechas, se el añaden las anotaciones de Hibernate.

Por ejemplo:

```
public class Empresa implements Serializable {

    private static final long serialVersionUID = -3524637996993798378L;

    private int id = 0;
    private String nombre = null;
    private String localizacion = null;
```

Partimos de la Entidad Empresa. Esta entidad "describe" a la tabla **empresas** de nuestra BBDD. Tiene tres atributos/columnas:

- Id, que es la PK entera auto incremental en el esquema de E/R
- Nombre, que es un campo varchar tamaño 15.
- Localización, que es un campo varchar tamaño 15.

A demás, **empresas** tiene una relación 1:N con **empleados**.

Enseguida te habrás dado cuenta de que no todos los detalles de la BBDD se especifican en la Entidad. No obstante, es posible indicar muchas cosas usando anotaciones, como por ejemplo @Nullable, que indica que un valor o atributo puede ser nulo.

Empecemos pues **anotando** la clase Empresa como una Entidad, y mapeándola con la tabla empresas. Y, a continuación, le indicamos la clave primaria y el criterio de generación de las claves: incremental.

```
@Entity
@Table(name = "empresas")
public class Empresa implements Serializable {

    private static final long serialVersionUID = -3524637996993798378L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id = 0;
    private String nombre = null;
    private String localizacion = null;
```

Ahora, nos queda únicamente la relación 1:N entre **empresas** y **empleados**. Una breve guía de cosas a tener en cuenta:

- Las relaciones NO son bidireccionales. Si estableces una relación 1:N, es una relación de una única dirección. De ti depende hacer la relación inversa N:1, si crees que tiene sentido en tu aplicación.

- Hacer bien las relaciones IMPORTA. Existen formas de hacer que, al cargar una entidad en memoria, Hibernate forzosamente cargue también todas las entidades de la relación 1:N (o cualquier otra). Esto es un problema porque te las deja en memoria, o no, o crees que las tienes pero no es así, etc.

- No, no estás obligado en una Entidad a registrar todos los atributos y relaciones de una tabla. Hibernate no te protestará.

En el caso de una relación 1:N, hacemos lo siguiente:

```
@Entity
@Table(name = "empresas")
public class Empresa implements Serializable {

    private static final long serialVersionUID = -3524637996993798378L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id = 0;
    private String nombre = null;
    private String localizacion = null;

    @OneToMany(mappedBy = "empresa",
            fetch = FetchType.EAGER)
    private List<Empleado> empleados;
```

Creamos un atributo nuevo que sea una Lista capaz de contener los posibles N Empleados de la relación. A continuación, la anotamos como una relación @OneToMany indicando el **nombre del atributo en Empleado** que mapea a esta Empresa (la FK). Finalmente, le indicamos cuándo cargar los datos relacionados:

- **Lazy**: No se carga la lista de Empleados, aunque creas que has hecho el equivalente a un “select empresa”. No obstante, en cualquier momento que hagas un empresa.getEmpleados () o similar Hibernate lanzará una select adicional para cargar esa lista. Esto puede dar error si cierras la sesión antes de cargar esa lista a mano.

- **Eager**: No quieres problemas, quieres cargar todos los datos de la lista de Empleados de golpe.

En el caso de una relación N:1, hacemos lo siguiente:

```
@Entity
@Table(name = "empleados")
public class Empleado implements Serializable {

    private static final long serialVersionUID = -6619513647860363379L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id = 0;
    private String nombre = null;
    private String apellido = null;
    private String oficio = null;
    private String direccion = null;
    private String fecha = null;
    private String salario = null;
    private String comision = null;

    @ManyToOne
    @JoinColumn(name = "empresa_id")
    private Empresa empresa;
```

En este caso, indicamos la anotación @ManyToOne y la columna que en Empleados va a ser la Clave Externa.

¿Y las relaciones N:M? Bueno… se pueden descomponer en dos relaciones 1:N con una nueva clase intermedia, ¿verdad? Pero sí, pueden definirse.

¿Y si tengo Claves Compuestas? Si tienes PK con dos o más atributos, se define otra clase con los atributos de la PK y se la anota no como @Entity sino como @Embebedable. En la Entidad original se crea una variable de esa clave compuesta y se la anota como @EmbeddedId.

Un último aviso: **NUNCA** pongas en el método **toString ()** de una entidad los atributos de una relación @ManyToOne, @OneToMany, o lo que sea. Te puedes encontrar conque el programa entra en bucle infinito y satura la memoria de Java, abortando el programa.

Esto se debe a que si intentas acceder a List <Empleados>, fuerzas a Hibernate a cargar todos los Empleados. Como éstos tienen el atributo empresa, también cargan la empresa. Y como empresa tiene List <Empleados>, entonces…

Se puede resolver con anotaciones. No merece la pena. No lo hagas y punto.

Los DAOs

En general (y sin entrar en detalles) las operaciones básicas de un CRUD son muy fáciles de hacer en Hibernate. Solamente cuando queramos hacer cosas complicadas tendremos que hacer uso extensivo del **HQL**.

Select

Una Select sencilla en base a la ID necesita una sola línea de código:

```
public Empresa get(int id) {
    Empresa ret = null;
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        ret = session.get(Empresa.class, id);

        transaction.commit();

        session.close();
    } catch (Exception e) {
        System.out.println("Error en busqueda: " + e.getMessage());
    }

    return ret;
}
```

Una select que no sea por ID, o que retorne varias Empresas, necesita HQL:

```
@Override
public List<Empresa> getAll() {
    List<Empresa> ret = null;
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // HQL -> Retorna todas las empresas
        ret = session.createQuery("FROM Empresa", Empresa.class).list();

        transaction.commit();

        session.close();
    } catch (Exception e) {
        System.out.println("Error en busqueda: " + e.getMessage());
    }

    return ret;
}
```

Insert

Un insert simple requiere generar el objeto Empresa, cargarlo con sus datos (excepto la ID, que se autogenera) y proceder a salvarlo. Nótese que en Hibernate 6 no se usa el método save (), sino persist ().

```
@Override
public void insert(Empresa empresa) {

    System.out.println("Insertando: " + empresa.toString());
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // Hibernate 6 no usa el metodo save ()
        session.persist(empresa);

        transaction.commit();

        session.close();

        System.out.println("Insertada");
    } catch (Exception e) {
        System.out.println("Error en Insercion: " + e.getMessage());
    }
}
```

Update

Esto es más complicado. Hay DOS formas de hacer un Update. La primera es cuando la Empresa que quieres crear está Detached. Es decir, que no la tiene sincronizada Hibernate. En este caso se usa merge ().

```
@Override
public void updateDetached(Empresa empresa) {
    System.out.println("Actualizando");
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // Si empresa viene Detached, usamos merge
        // Merge devuelve una instancia Persistent asociada a la sesion
        // Si empresa estaba Detached, Hibernate crea una copia y la sincroniza con BD
        session.merge(empresa);

        transaction.commit();

        session.close();

        System.out.println("Actualizada");
    } catch (Exception e) {
        System.out.println("Error en Actualizacion: " + e.getMessage());
    }
}
```

La otra opción es cuando extraemos la empresa de la BBDD, por lo tanto, ya está siendo sincronizada por Hibernate. En este caso, un commit () ya soluciona el update.

```
@Override
public void updatePersistent(int id, String name) {
    System.out.println("Actualizando");
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // Hacemos primero Persistant la Empresa, lo que la sincroniza con la BBDD
        Empresa empresa = session.get(Empresa.class, id);
        empresa.setNombre("Elorrieta Academy");

        // El commit hace el update() automaticamente
        transaction.commit();

        session.close();

        System.out.println("Actualizada");
    } catch (Exception e) {
        System.out.println("Error en Actualizacion: " + e.getMessage());
    }
}
```

Delete

De forma similar hay que considerar si está la Empresa Detached o no.

```
@Override
public void deleteDetached(Empresa empresa) {
    System.out.println("Eliminando");
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // Si empresa viene Detached, usamos merge
        // Merge devuelve una instancia Persistent asociada a la sesion
        // Si empresa estaba Detached, Hibernate crea una copia y la sincroniza con BD
        Empresa managed = session.merge(empresa);
        session.remove(managed);

        transaction.commit();

        session.close();

        System.out.println("Eliminada");
    } catch (Exception e) {
        System.out.println("Error en Eliminacion: " + e.getMessage());
    }
}
```

Si no lo está:

```
@Override
public void deletePersistent(int id) {
    System.out.println("Eliminando");
    try {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        Empresa empresa = session.get(Empresa.class, id);
        if (empresa != null) {
            session.remove(empresa);
        }

        transaction.commit();

        session.close();

        System.out.println("Eliminada");
    } catch (Exception e) {
        System.out.println("Error en Eliminacion: " + e.getMessage());
    }
}
```

Aclaraciones finales

Cuesta “pillarle el truco” a Hibernate. La forma más fácil de trabajar con ella es hacer pruebas aisladas y ver que las tablas se actualizan como quieres, y que te devuelve lo que quieres. Lleva un tiempo. Y hay que tener algo de mano para ir resolviendo las cosas raras que hace a veces.

Por último, recuerda que de la misma forma que en este ejemplo se generan las tablas de BBDD desde el Java, es posible hacerlo al revés.

Y por favor, no te olvides de cambiar la configuración para que sólo genere las tablas una única vez. Si no lo haces, cada vez que arrancas el programa Hibernate borrará las bases de datos y las creará vacías de nuevo.