TEMA

Tema 3. Persistencia BDR con ORM

Desarrollo de aplicaciones multiplataforma

Acceso a datos

Autora: Silvia Macho

Linkia FP

Tema 3: Persistencia en BDR con ORM

¿Qué aprenderás?

- Instalar y configurar Hibernate para su uso con Java.
- Crear una aplicación Java con BBDD usando Hibernate.

¿Sabías que…?

- Java es un lenguaje que trabaja con el paradigma de programación orientado a objetos mientras que SQL trabaja de forma relacional. Gracias a los ORM, podemos trabajar con los dos mundos de forma conjunta.
- Hibernate es un ORM que podemos añadir a una aplicación Java, pero que viene incorporado en el IDE NetBeans.

3.1. Introducción

El acceso a SGBD desde una aplicación Java con JDBC es una herramienta muy potente, ya que permite a la aplicación Java adaptarse a la BBDD con la que esté trabajando. A pesar de su potencia, es un sistema de acceso de bajo nivel que hacer que tengamos que utilizar un gran número de líneas de código para poder cubrir las necesidades de cada aplicación, ya que en el desarrollo de aplicaciones es muy común utilizar lenguajes OO y almacenar los datos en sistemas relacionales.

En estos casos, es necesario convertir los objetos con los que trabajamos a nivel de programación, a un modelo relacional, es decir, tenemos que convertir los atributos de los objetos a las tablas y modelo relacional. Utilizando ese sistema, los programadores tenemos que trabajar con el desfase objeto-relacional y conocer la estructura exacta de la base de datos con la que vamos a trabajar. Si la estructura de la BBDD cambia, tenemos que adaptar toda la aplicación a dichos cambios.

3.2. Mapeo Objeto-Relacional

Al trabajar con programación OO y BBDD relacionales, utilizamos paradigmas y formas de pensar diferentes. El modelo relacional, trabaja con relaciones y conjuntos de datos. La programación OO trabaja con clases de objetos, objetos, atributos, métodos y relaciones entre objetos. Un mapeo Objeto-Relacional tiene como objetivo evitar estas diferencias.

El mapeo Objeto-Relacional (ORM: Object-Relational Mapping) es una técnica de programación que nos permite convertir datos entre las aplicaciones orientadas a objetos y una base de datos relacional. Este mapeo se consigue porque esta técnica simula una BBDD Orientada a Objetos sobre la BBDD Relacional.
Las bases de datos relacionales solo pueden almacenar tipos simples, cosa que impide que puedan almacenar directamente los objetos con los que trabaja una aplicación Orientada a Objetos. Lo que hacemos con el ORM es transformar los Objetos (que son los elementos completos) en datos simples para que puedan almacenar en las correspondientes tablas de la BBDD relacional. Cuando queramos hacer el proceso contrario (recuperar los datos de las tablas para añadirlos a la aplicación), el mapeo ORM recogerá los datos simples de las tablas y los utilizará para crear el objeto correspondiente en la aplicación.

El objetivo fundamental del mapeo ORM es proporcionar una herramienta que ayude al programador a trabajar con dos mundos, de tal forma que la transformación datos relacionales a objetos y viceversa sea completamente transparente para él.

Ventajas del ORM:

- Rapidez en el desarrollo. Las herramientas de este tipo, nos permiten crear el modelo de objetos a partir del esquema de la BBDD.
- Abstracción de la BBDD. Con este sistema, la aplicación se abstrae del esquema de la BBDD y del SGBD. Si se cambia el esquema de la BBDD o el propio SGBD, la aplicación no se verá afectada.
- Reutilización. Podemos reutilizar los métodos de un objeto de la aplicación desde diferentes partes de ella, incluso desde diferentes aplicaciones.
- Lenguaje propio para las consultas. Las herramientas de mapeo ORM tienen su propio lenguaje para hacer consultas sobre la BBDD, haciendo que éstas sean más óptimas para la realización de búsquedas y consultas.

Desventajas del ORM:

- Tiempo. Al añadir una capa más a la implementación de la aplicación y siento herramientas complejas, incrementamos el tiempo de creación de la aplicación ya que el buen manejo de estas herramientas suele requerir tiempo.
- Velocidad. Al añadir una capa más a la aplicación, éstas se vuelven un poco más lentas por el proceso de traducción que tienen que realizar.

3.3. Herramientas ORM - Hibernate

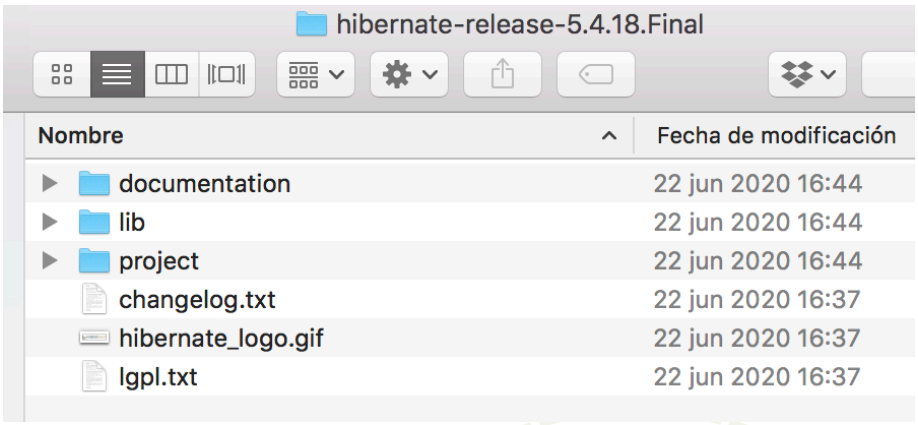
Existen muchas herramientas ORM en el mercado, pero nosotros utilizaremos Hibernate, ya que permite el mapeo objeto-relacional en Java. Hibernate es una aplicación de software libre, multiplataforma y optimizado para trabajar con aplicaciones creadas en Java. Está diseñado para adaptarse a la estructura de una base de datos, independientemente de cómo sea esta estructura. Hibernate además, tiene su propio lenguaje de consultas, HQL (Hibernate Query Language).

Para instalar Hibernate en el IDE de desarrollo, teniendo en cuenta que como IDE utilizamos NetBeans, podemos hacerlo de dos formas: manual o descargarlo con el propio IDE.

Para realizar la instalación de Hibernate de forma manual, seguiremos los siguientes pasos:

1. Acceder a la página oficial de Hibernate y descargar la última versión.
   https://hibernate.org/

   El archivo que tenemos que descargar tendrá el nombre hibernate-release-5.4.18.Final.zip (La numeración cambiará en función de la última versión estable que haya en el momento que hagamos la descarga).
2. Cuando finalice la descarga, descomprimiremos el fichero .zip.



Como podemos ver en la imagen, el ficher contiene las siguientes carpetas: lib, documentation y project.

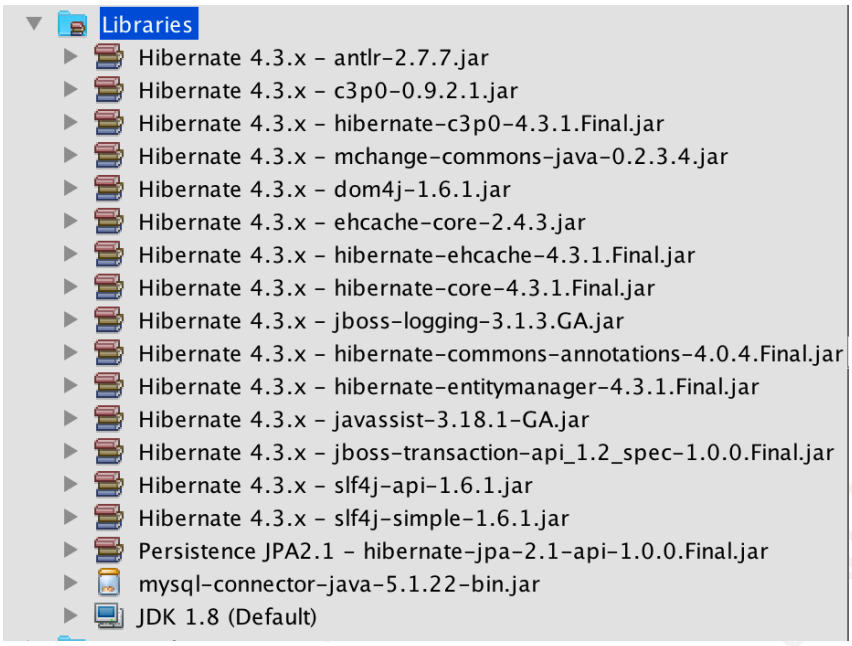
lib: agrupa las librerías con el propio código de Hibernate.

- lib\required: almacena los archivos .jar que usaremos en Hibernate. Estos archivos son los que tenemos que añadir en todas las aplicaciones Java donde queramos usar Hibernate.
- lib\jpa-metamodel-generator: almacena las librerías necesarias para usar JPA con Hibernate. JPA es una API para gestionar la persistencia en Java.

documentation: aquí tenemos la documentación de Hibernate.

project: agrupa el código fuente y ficheros de configuración que utilizaremos para configurar el acceso a cada una de las BBDD.

3. Copiaremos todos los ficheros .jar de la carpeta lib\required a la carpeta lib del proyecto que representa nuestra aplicación en Java.
4. Copiaremos el fichero hibernate-jpamodelgen-5.4.18.Final.jar de la carpeta lib\jpa-metamodel-generator en la carpeta lib del proyecto que representa nuestra aplicación en Java. La numeración del archivo dependerá de la versión de Hibernate con la que estemos trabajamdo.
5. Abriremos NetBeans y asociaremos todos los ficheros comentados en los puntos 3 y 4. Para hacerlo, los añadiremos como ficheros .jar al apartado Libraries del proyecto.



Como podemos ver en la imagen, además de todos los ficheros para el uso de Hibernate, también está el fichero mysql-connector-java-5.1.22-bin.jar. Este fichero lo hemos añadido porque la BBDD que utilizará la aplicación es una BBDD MySQL.

Una opción mucho más sencilla, es descargar la versión del IDE NetBeans que ya incluye el módulo de Hibernate y por lo tanto el propio IDE ya contiene los elementos necesarios para crear aplicaciones que utilizan Hibernate.

Una vez tenemos Hibernate añadido al NetBeans, ya podemos empezar a crear todos los elementos necesarios para crear la estructura que necesita una aplicación Java para trabajar con Hibernate. Estos elementos son:

- Clases Java. Estas clases representarán a las tablas de la BBDD en la aplicación Java.
- Ficheros de mapeo y configuración. Estos ficheros son necesarios para hacer la relación entre los campos de las tablas de la BBDD con los atributos de las clases, además de la configuración para poder acceder a la propia BBDD.

3.3.1. Clases Java

La aplicación Java necesita tener una serie de clases que representarán a las tablas en la BBDD. Estas clases son las que utilizará Hibernate para hacer la mapeo objeto-relacional.

Solo tenemos que crear clases de aquellas tablas de la BBDD con las que la aplicación vaya a trabajar. Si alguna tabla no se utiliza, no tendremos que crear la clase correspondiente. Cada una de estas clases, las marcaremos como clase POJO y deben cumplir con lo siguiente:

- Deben tener un constructor público sin ningún tipo de parámetro, es decir, el constructor por defecto.
- Para cada atributo de la clase que se corresponda con un campo de la tabla, tenemos que añadir los métodos get/set correspondientes, ya que los atributos serán privados, para cumplir con la encapsulación que marca la programación orientada a objetos.
- Debe implementar la interfaz Serializable, que es el elemento que permite marcar las clases para que podemos hacer persistentes sus objetos.

Los elementos de la lista son los mínimos que deben tener las clases que se correspondan con las tablas de la BBDD. Además de estos elementos, podemos añadirles todo lo necesario para el correcto funcionamiento de la aplicación.

Un ejemplo de tipo de clase que hemos descrito, sería la siguiente:

Album.java

```java
package accesohibernate;

public class Album  implements java.io.Serializable {


    private int id;
    private String titulo;
    private String autor;

    public Album() {
    }

    public Album(int id, String titulo, String autor) {
       this.id = id;
       this.titulo = titulo;
       this.autor = autor;
    }

    public Album(int id) {
       this.id = id;
    }

    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }
    public String getTitulo() {
        return this.titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getAutor() {
        return this.autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    @Override
    public String toString() {
        String cadena;
        cadena = "ID: " + this.id + " - Titulo: " + this.titulo +
                 " - Autor: " + this.autor;
        return cadena;
    }
}
```

Como podemos ver en el ejemplo, esta clase contiene los elementos mínimos del listado, más algunos elementos más, necesarios para la aplicación.

3.3.2. Ficheros de mapeo y configuración

Para cada una de las clases de las que queremos hacer objetos persistenten en la BBDD, Hibernate necesita un fichero para tener información de la correspondencia entre clase-tabla y atributo-campo. Este fichero tiene formato xml y es el que permitirá a Hibernate hacer el mapeo objeto-relacional. Este fichero estará ubicado en el mismo paquete del proyecto Java donde tenemos las clases de las que queremos hacer los objetos persistentes.

El fichero tendrá el nombre xxx.hbm.xml, donde xxx es el nombre de la tabla y clase que relaciona. Un ejemplo de este fichero que se correspondería con la clase del apartado anterior sería:

Album.hbm.xml

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 07-feb-2017 10:20:14 by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
    <class name="accesohibernate.Album" table="album" catalog="discografica" optimistic-lock="version">
        <id name="id" type="int">
            <column name="id" />
            <generator class="assigned" />
        </id>
        <property name="titulo" type="string">
            <column name="titulo" length="20" not-null="true" />
        </property>
        <property name="autor" type="string">
            <column name="autor" length="20" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Si nos fijamos en el fichero, veremos que tiene la estructura de cualquier fichero xml.

- La etiqueta class tiene los atributos para hacer la correspondencia entre el nombre de la clase (name) y la tabla (table).
- La etiqueta id, marca la relación entre la clave primaria de la tabla (column) y el atributo de la clase que la representa (name). También tiene un atributo para especificar el tipo de dato (type).
- La etiqueta property, marca la relación entre un campo de la tabla (name) y el atributo de la clase que representa (column). También tiene un atributo para especificar el tipo de dato (type). El archivo tendrá una etiqueta de este tipo por cada pareja campo-atributo que tengamos que relacionar.

Además de este fichero de mapeo, Hibernate necesita una serie de ficheros para poder unir aplicación Java y BBDD, y así poder completar el mapeo objeto-relacional. Estos ficheros son:

- Fichero de configuración hibernate.cfg

  Este archivo contiene la información necesaria para conectar la BBDD con la aplicación para así poder realizar la persistencia de los datos.
- Fichero de ingeniería inversa hibernate.reveng

  Este archivo contiene el esquema de la tabla a mapear y el nombre de la tabla.
- Fichero HibernateUtil.java

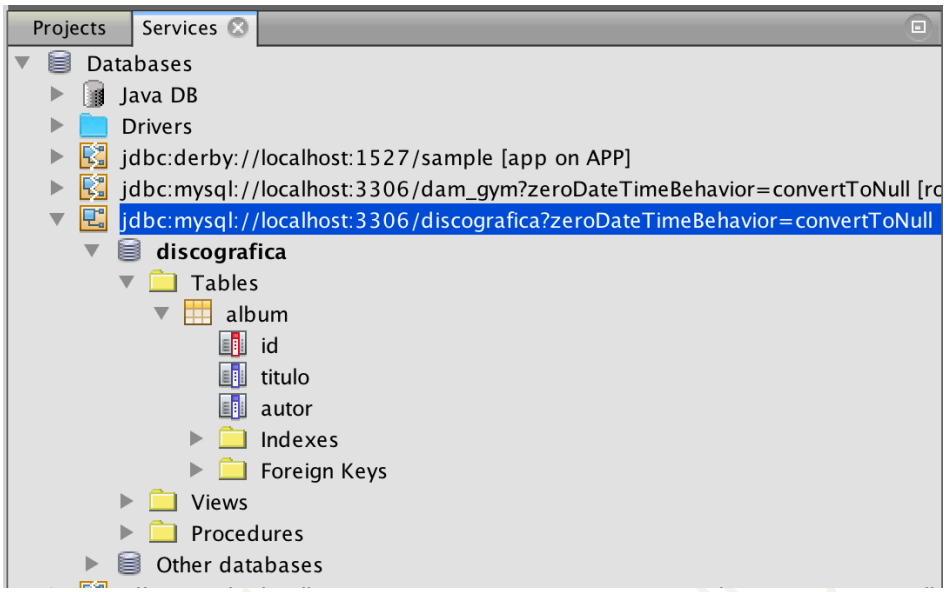
  Hibernate utiliza este fichero para gestionar las conexiones que se hacen a la BBDD y que permitirán mapear los objetos de la aplicación en los campos de la tabla correspondientes.

3.3.3. Creación de los ficheros

Desde el IDE NetBeans, podemos crear todos los ficheros necesarios para Hibernate de una forma más o menos automática. El proceso partirá de una BBDD relacional, con la que el IDE tiene conexión. A partir de aquí, creará las clases, los ficheros de mapeo y configuración comentados en la sección anterior.

Para poder empezar todo el proceso, necesitamos lo siguiente:

- Un proyecto Java.
- Una conexión a la BBDD con la que la aplicación vaya a trabajar. Esta conexión la tenemos que crear a nivel de NetBeans.



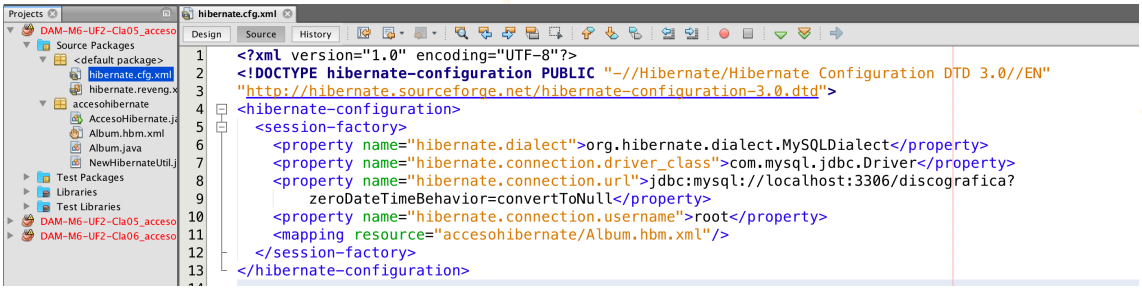
Fichero de configuración hibernate.cfg

El primer fichero que añadiremos, será el fichero de configuración de Hibernate. Este fichero contiene la información que la aplicación necesita para conectarse a la BBDD. El nombre del fichero es hibernate.cfg.xml y lo crearemos en la carpeta defaultpackage del proyecto.

Para crearlo, seguiremos los siguientes pasos:

1. Menú File – Nuevo archivo y seleccionar Otros: Hibernate Configuration Wizard.
2. El nombre por defecto del archivo es hibernate.cfg
3. De la lista desplegable, seleccionaremos la conexión a la BBDD que hemos creado con anterioridad.

Un ejemplo de este fichero, lo tenemos en la imagen siguiente:



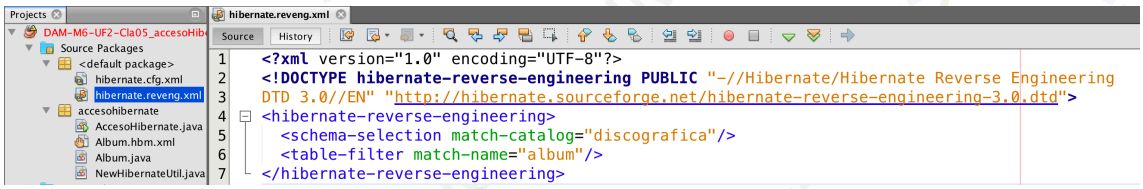
Fichero de ingeniería inversa hibernate.reveng

El siguiente fichero que crearemos será el fichero de ingeniería inversa. Este fichero contiene el esquema con la tabla a mapear y el nombre de la tabla. El nombre del fichero es hibernate.reveng.xml y lo crearemos en la carpeta defaultpackage del proyecto.

Para crearlo, seguiremos los siguientes pasos:

1. Seleccionando el default package, botón derecho y seleccionamos Nuevo – Otros – Hibernate Reverse Engineering Wizard.
2. El nombre por defecto será hibernate.reveng.
3. Seleccionamos la tabla o tablas que queremos mapear.

Un ejemplo de este fichero, lo tenemos en la imagen siguiente:



Clases y fichero de mapeo

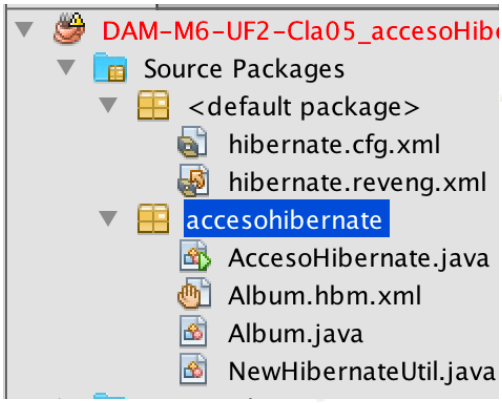
Los siguientes elementos que tenemos que crear son por un lado las clases que se corresponderán con las tablas de la BBDD y el fichero de mapeo donde tendremos la correspondencia entre clase/tabla y atributos/campos. Primero crearemos las clases y después el fichero de mapeo. El resultado será una clase por cada tabla.

Para crear las clases y el fichero de mapeo, seguiremos los siguientes pasos:

1. Seleccionando el default package, botón derecho y seleccionamos Nuevo – Otros – Hibernate mapping files and POJOS from Database.
2. Sólo es necesario indicar el paquete en el que vamos a almacenar los nuevos ficheros.

Si en un futuro queremos añadir más clases de las tablas existentes en la BBDD, solo tenemos que repetir este proceso para las nuevas tablas de las que queramos crear las clases correspondientes.

Durante este proceso, ponemos un nombre de paquete, es en este paquete donde se crearán las clases y el fichero de mapeo:



Ya hemos visto un ejemplo de clases y fichero de mapeo de este tipo en secciones anteriores.

Fichero HibernateUtil.java

El último elemento a crear para tener toda la infraestructura de Hibernate preparada, es el fichero HibernateUtil.java. Este fichero permite gestionar las conexiones que la aplicación hace a la BBDD.

Pasos para crear este fichero:

1. Seleccionando el paquete donde están las clases POJO, botón derecho y seleccionamos Nuevo – Otros – HibernateUtil.java.
2. Cambiamos el nombre por defecto NewHibernateUtil.java por HibernateUtil.java y seleccionamos el paquete de las clases POJO. Este cambio no es obligatorio, pero debemos tenerlo en cuenta a la hora de implementar la aplicación, para utilizar el nombre del archivo que corresponda.

Un ejemplo de este fichero, lo tenemos en la imagen siguiente:

```java
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * Hibernate Utility class with a convenient method to get Session Factory
 * object.
 *
 */
public class NewHibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from standard (hibernate.cfg.xml)
            // config file.
            sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

3.4. Creación de la aplicación

Una vez ya hemos creado toda la infraestructura de acceso a Hibernate, ya podemos empezar a implementar la aplicación. La idea es explotar los beneficios de utilizar el ORM de Hibernate, haciendo la aplicación lo más transparente posible al esquema de la BBDD. Esto la vamos a conseguir gracias a las clases que hemos generado de forma automática en los pasos anteriores para tener las tablas de la BBDD mapeadas en nuestra aplicación. Por lo tanto, los objetos que harán uso de la persistencia, serán precisamente los objetos de estas clases.

La clase más utilizada en Hibernate es la clase Session. Esta clase inicialmente la utilizaremos para crear una batería de sesiones de acceso a la BBDD. Esta batería la podremos crear gracias al fichero HibernateUtil creado también en los pasos anteriores. No olvidar que a este fichero podemos no ponerle exactamente ese nombre, por lo que tendremos que fijarnos en qué nombre tiene para poder gestionar esta batería de sesiones.

Para la gestión de estas sesiones, utilizaremos el siguiente código:

```
Session session = NewHibernateUtil.getSessionFactory().openSession();
```

Como ocurre con todos los elementos que abren una conexión o sesión con una BBDD, cuando ya no lo necesitamos, tenemos que cerrarlo, lo mismo pasa con este tipo de sesiones:

```
session.close();
```

La clase Session tiene una serie de métodos que son los que utilizaremos en la aplicación para hacer la gestión de la persistencia con Hibernate:

- beginTransaction(): con este método podremos crear transacciones. La transacción la representaremos con un objeto de la clase Transaction. Esta clase tiene métodos para poder confirmar o cancelar la transacción (commit, rollback).
- save(): con este método haremos persistente un nuevo objeto en la BBDD, es decir, sería el equivalente al INSERT de SQL.
- delete(): con este método eliminaremos los datos de un objeto en la BBDD, es decir, sería el equivalente al DELETE de SQL.
- update(): con este método podemos modificar los datos de un objeto en la BBDD, es decir, sería el equivalente al UPDATE de SQL.
- get(): con este método podemos recuperar los datos de un objeto de las BBDD, es decir, sería el equivalente al SELECT de SQL, pero parametrizado para recuperar un dato concreto.
- createQuery(): con este método podemos ejecutar sentencias de consulta HQL. Este lenguaje es un lenguaje propio de Hibernate para la creación de consultas. Para ello también utilizaremos la clase Query, para poder representar esa consulta y después ejecutarla.

3.4.1. El método save()

Para hacer persistente un objeto de la aplicación Java en la BBDD a través de Hibernate, el método que utilizaremos será save(). Este método recibe como parámetro el objeto que queremos hacer persistente. Los pasos que tenemos que seguir son:

1. Crear el objeto que queremos guardar en la BBDD.
2. Obtener el objeto sesión para conectarnos a la BBDD.
3. Iniciar una transacción sobre esa sesión.
4. Llamar al método save() para hacer persistente el objeto, pasando el objeto creado en el punto 1 como parámetro del método.
5. Confirmar la transacción.
6. Cerrar la sesión.

Un ejemplo de estos pasos, los tenemos en la imagen siguiente:

```
public static void annadir_album(int id, String tit, String aut)
{//añade un objeto nuevo a la base de datos (persistencia)
    Transaction tx=null;
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Album a = new Album(id, tit, aut);
    //a.setAutor(aut);
    //a.setTitulo(tit);
    //a.setId(id);
    session.save(a);//Guarda el objeto creado en la BBDD.
    tx.commit(); //Materializa la transacción
    session.close();

}
```

En el código del ejemplo, los datos para el objeto que queremos hacer persistente los pasamos como parámetro de la función. Para crear este objeto, podemos utilizar el constructor que tiene parámetros para inicializar los atributos o crear el objeto con el constructor por defecto y luego inicializar los atributos con los correspondientes métodos get (esta última parte está comentada en el ejemplo).

3.4.2. El método delete()

Para eliminar los datos de un objeto de la BBDD a través de Hibernate, el método que utilizaremos será delete(). Este método recibe como parámetro el objeto que queremos eliminar. Los pasos que tenemos que seguir son:

1. Crear el objeto que queremos borrar en la BBDD.
2. Obtener el objeto sesión para conectarnos a la BBDD.
3. Iniciar una transacción sobre esa sesión.
4. Llamar al método delete() para eliminar el objeto, pasando el objeto creado en el punto 1 como parámetro del método.
5. Confirmar la transacción.
6. Cerrar la sesión.

Un ejemplo de estos pasos, los tenemos en la imagen siguiente:

```
public static void borrar_album(int id)
{//Borrar un objeto cuyo id se pasa como parámentro
    Transaction tx=null;
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Album a = new Album(id,"One", "The Beatles");
    session.delete(a);
    System.out.println ("Objeto borrado");
    tx.commit(); //MAterializa la transacción
    session.close();
}
```

De nuevo, a la hora de crear el objeto que pasaremos al método para eliminar, lo podremos crear utilizando uno de los diferentes constructores en la clase, lo importante es que los atributos correspondientes estén inicializados de forma correcta.

3.4.3. El método update()

Para modificar los datos de un objeto de la BBDD a través de Hibernate, el método que utilizaremos será update(). Este método recibe como parámetro el objeto que queremos modificar. Los pasos que tenemos que seguir son:

1. Crear el objeto que queremos modificar en la BBDD.
2. Obtener el objeto sesión para conectarnos a la BBDD.
3. Iniciar una transacción sobre esa sesión.
4. Llamar al método update() para modificar el objeto, pasando el objeto creado en el punto 1 como parámetro del método.
5. Confirmar la transacción.
6. Cerrar la sesión.

Un ejemplo de estos pasos, los tenemos en la imagen siguiente:

```
public static void modificar_album(int id, String tit, String aut)
{//Modifica un objeto cuyo id se pasa como parámetro
    Transaction tx=null;
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Album a = new Album(id);
    a.setAutor(aut);
    a.setTitulo(tit);
    a.setId(id);
    session.update(a);//Modifica el objeto con Id indicado
    tx.commit(); //MAterializa la transacción
    session.close();

}
```

El atributo que se corresponde con la clave primaria de la tabla, no lo podremos modificar, ya que es el valor que utiliza Hibernate para localizar los datos del objeto a modificar.

3.4.4. El método get()

Para recuperar los datos de un objeto de la BBDD a través de Hibernate, el método que utilizaremos será get(). Este método recibe como parámetros el nombre de la clase a la que pertenece el objeto que queremos recuperar y el id que se corresponde con la clave primaria. Como retorno, el método devuelve un objeto de la clase que hemos pasado como primer parámetro. Este objeto estará creado a partir de los datos correspondientes a la clave primaria pasada como segundo parámetro.

Los pasos que tenemos que seguir son:

1. Obtener el objeto sesión para conectarnos a la BBDD.
2. Llamar al método get() para recuperar el objeto, pasando el nombre de la clase del objeto que queremos recuperar y la clave primaria para poder localizar los datos correspondientes. El retorno de este método, lo guardaremos en un objeto de la clase correspondiente, para ello tendremos que hacer un cast a la clase.
3. Cerrar la sesión.

Un ejemplo de estos pasos, los tenemos en la imagen siguiente:

```
public static void recuperar_album(int id)
{//recupera un objeto cuyo id se pasa como parámentro
    //Transaction tx=null;
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    Transaction tx=session.beginTransaction(); //Crea una transacción
    Album a ;
    a=(Album)session.get(Album.class,id);
    System.out.println ("Autor: " + a.getAutor());
    tx.commit(); //MAterializa la transacción
    session.close();
}
```

En el ejemplo, hemos hecho uso de transacciones aunque en este caso no harían falta ya que no modificamos nada en la BBDD.

3.4.5. Hibernate Query Language (HQL)

Hibernate además de permitir la recuperación de datos de un objeto a través del método get() de la clase Session, propociona un lenguaje propio para poder realizar consultas sobre una BBDD. Este lenguaje es el HQL, que son las siglas de Hibernate Query Language. Es un lenguaje muy parecido a SQL, pero optimizado para la gestión que realiza Hibernate y adaptada para devolver objetos en lugar de registros como hace SQL.

HQL se basa en que las consultas se realizan sobre los objetos Java existentes en la aplicación, aquellos elementos que hacemos persistentes en la BBDD. Este lenguaje se caracteriza por:

- Los tipos de datos con los que trabaja son los existentes en Java.
- Las consultas son independientes del lenguaje de consultas específico de la BBDD.
- Las consultas son independientes de la estructura de las tablas de la BBDD. No necesitamos conocer cómo está creada la BBDD, es decir, su estructura para ejecutar una consulta, hecho que resulta ventajoso.
- Podemos trabajar con las colecciones propias de Java.
- Podemos movernos por los objetos generados como resultado de la consulta.

A la hora de crear consultas con HQL, debemos tener en cuenta lo siguiente:

- HQL no es un lenguaje case-sensitive, por lo que no es sensible a las mayúsculas y minúsculas en el tratamiento de las palabras reservadas del lenguaje.
- Para crear criterios de selección en la consulta, utilizaremos el elemento WHERE. A la hora de montar la consulta con la cláusula WHERE, seguiremos las mismas normas que con SQL, pero teniendo en cuenta que las propiedades y el nombre de los objetos hacen referencia a los objetos de las clases y no a las tablas de la BBDD.

- La cláusula SELECT hace referencia a un objeto, aunque también podemos usar funciones para agrupar atributos de las clases.

Teniendo en cuenta todo esto, ya podemos ver cómo se ejecutan este tipo de consultas en una aplicación Java. Para ello lo primero que utilizaremos es la clase Query. Los objetos de esta clase representarán las consultas HQL que queremos ejecutar. Para crear estos objetos, utilizaremos el método createQuery() de la clase Session. Este método recibe un parámetro de tipo String que es la consulta que queremos ejecutar.
Después de crear el objeto Query, utilizaremos el método list() de la clase Query para obtener una colección Java con los objetos devueltos como resultado de la ejecución de la consulta.

Una vez tenemos los objetos en la colección, lo siguiente ya es procesar la colección de la forma en la que habitualmente hacemos la gestión del contenido de las colecciones.

El siguiente ejemplo ejecuta una consulta con HQL. El resultado son los objetos que devuelve la ejecución de la consulta. El código está dentro de una función a la que le pasamos como parámetro de tipo String la consulta que queremos realizar.

```
public static void consulta(String c)
{//Ejecuta una consulta cuando el resultado se devuelve como un objeto Albumes
    System.out.println ("Salida de consulta");
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    Query q= session.createQuery(c);
    List results = q.list();

    Iterator albumesiterator= results.iterator();
    while (albumesiterator.hasNext())
    {
        Album a2= (Album)albumesiterator.next();
        System.out.println ( a2.getId() + " - " + a2.getTitulo () );
    }
    session.close();

}
```

La consulta que pasamos es la siguiente:

```
consulta("select a from Album a where titulo like '%Black%'");
```

Si analizamos el código del ejemplo, la consulta devuelve todos y cada uno de los objetos de tipo Album que concuerdan con el criterio de búsqueda (que en este caso son todos aquellos cuyo título contiene el string Black). Como la consulta devuelve los objetos tal cual, el resto del código lo que hace es iterar sobre la colección y mostrar por pantalla dos atributos de estos objetos concatenados.

El siguiente ejemplo hace una consulta similar, pero el lugar de devolver el objeto completo que cumple con el criterio de búsqueda, solo devuelve el valor de uno de los atributos.

```
public static void consulta2(String c)
{ //Ejecuta una consulta cuando el resultado se devuelve como String
    System.out.println ("Salida de consulta");
    Session session = NewHibernateUtil.getSessionFactory().openSession();
    Query q= session.createQuery(c);
    List results = q.list();

    Iterator albumesiterator= results.iterator();
    while (albumesiterator.hasNext())
    {
        String a2= (String)albumesiterator.next();
        System.out.println ( " *-* " + a2 );
    }
    session.close();

}
```

Igual que en el ejemplo anterior, el código está encapsulado dentro de una función a la que le pasamos la consulta a ejecutar como parámetro. En este caso, la consulta es la siguiente:

```
consulta2("SELECT a.titulo from Album a WHERE a.titulo like '%B%' ");
```

Como podemos ver, lo que pedimos que devuelva la consulta no es el objeto completo, ya que después de la cláusula SELECT marcamos a.titulo, indicando que solo queremos el valor del atributo titulo.

A diferencia del primer ejemplo, como en este caso solo obtenemos el valor de un atributo, la colección resultado de la consulta, estará formada por todos y cada uno de los valores del atributo titulo de los objetos con coincidan con el criterio de búsqueda, haciendo que sea una colección formada por Strings. El resto del código lo que hace simplemente es recorrer la colección resultado y mostrar directamente el valor que guarda la colección, ya que este ya será el valor del atributo.

Recursos y Enlaces

- Java

- API Java 11

- NetBeans

- Hibernate

- API Hibernate

Conceptos clave

- **Hibernate**: ORM de código abierto optimizado para trabajar con aplicaciones Java.
- **HQL**: Hibernate Query Language. Lenguaje de consultas propio de Hibernate basado en SQL.
- **ORM**: herramienta que permite hacer el mapeo objeto-relacional, haciendo que una aplicación creada en un lenguaje orientado a objetos pueda acceder a una base de datos relacional sin preocuparse por su estructura.

Test de autoevaluación

1. Pon el significado de cada una de las siglas de HQL:
    a. H
    b. Q
    c. L

2. ¿Cuál es la clase más utilizada para crear una aplicación Java que utilice Hibernate para hacer persistentes sus objetos?
    a. Transaction
    b. Hibernate
    c. Session
    d. Query

3. ¿Qué método de la clase Session nos permite hacer persistente un objeto Java en una BBDD a través de Hibernate?
    a. save
    b. delete
    c. get
    d. createQuery

Ponlo en práctica

Actividad 1

Crea una aplicación Java que usando Hibernate obtenga un objeto de una determinada clase a partir de su clave primaria. La aplicación debe mostrar por pantalla el valor de uno de los atributos del objeto resultado.

SOLUCIONARIOS

Test de autoevaluación

1. Pon el significado de cada una de las siglas de HQL:
    a. H – **Hibernate**
    b. Q – **Query**
    c. L – **Language**

2. ¿Cuál es la clase más utilizada para crear una aplicación Java que utilice Hibernate para hacer persistentes sus objetos?
    a. Transaction
    b. Hibernate
    c. Session
    d. Query

3. ¿Qué método de la clase Session nos permite hacer persistente un objeto Java en una BBDD a través de Hibernate?
    a. **save**
    b. delete
    c. get
    d. createQuery

Ponlo en práctica

Actividad 1

Crea una aplicación Java que usando Hibernate obtenga un objeto de una determinada clase a partir de su clave primaria. La aplicación debe mostrar por pantalla el valor de uno de los atributos del objeto resultado.

Solución:

```
Session session = NewHibernateUtil.getSessionFactory().openSession();
Album a ;
a=(Album)session.get(Album.class,3);
System.out.println ("Autor: " + a.getAutor());
session.close();
```