

Mapeo Objeto-Relacional. Hibernate

Índice

Introducción	2
Desfase objeto-relacional.....	2
¿Qué es el mapeo objeto-relacional?.....	3
Herramientas ORM. Características.....	4
Hibernate	5
Instalación de Hibernate en Netbeans	5
Estructura de los ficheros de mapeo	19
Clases persistentes	19
Sesiones y objetos hibernate.....	20
Transacciones	21
Estados de un objeto Hibernate.....	21
Carga de objetos	22
Almacenamiento, modificación y borrado de objetos	22
EJEMPLOS 1	23
Consultas	24
EJEMPLOS 2	24

Introducción

En este tema aprenderemos a acceder a una base de datos relacional utilizando el lenguaje orientado a objetos de Java. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se llama **ORM (Object Relational Mapping)** y es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de bases de datos relacionales a representaciones de objetos; es decir, las tablas de nuestra base de datos pasan a ser clases y las filas de las tablas (o registros) objetos que podemos manejar con facilidad.

Desfase objeto-relacional

El *desfase objeto-relacional* surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. Hay que tener en cuenta que esta situación se da porque tanto los lenguajes orientados a objetos como las bases de datos relacionales están ampliamente extendidas.

En cuanto al desfase, ocurre que en nuestra aplicación Java (como ejemplo de lenguaje Orientada a Objetos) tendremos, por ejemplo, la definición de una clase cualquiera con sus atributos y métodos:

```
public class Personaje {
    private int id;
    private String nombre;
    private String descripcion;
    private int vida;
    private int ataque;

    public Personaje(. . .) {
        . . .
    }

    // getters y setters
}
```

Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

```
CREATE TABLE personajes (
    id INT PRIMARY KEY AUTO_INCREMENT;
    nombre VARCHAR(50) NOT NULL,
    descripcion VARCHAR(50),
    vida INT DEFAULT 10,
    ataque INT DEFAULT 10;
);
```

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia `SQL` para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.

¿Qué es el mapeo objeto-relacional?

El mapeo objeto-relacional (más conocido por su nombre en inglés, **Object-Relational mapping**, o sus siglas O/RM, **ORM**, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto **crea una base de datos orientada a objetos virtual**, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

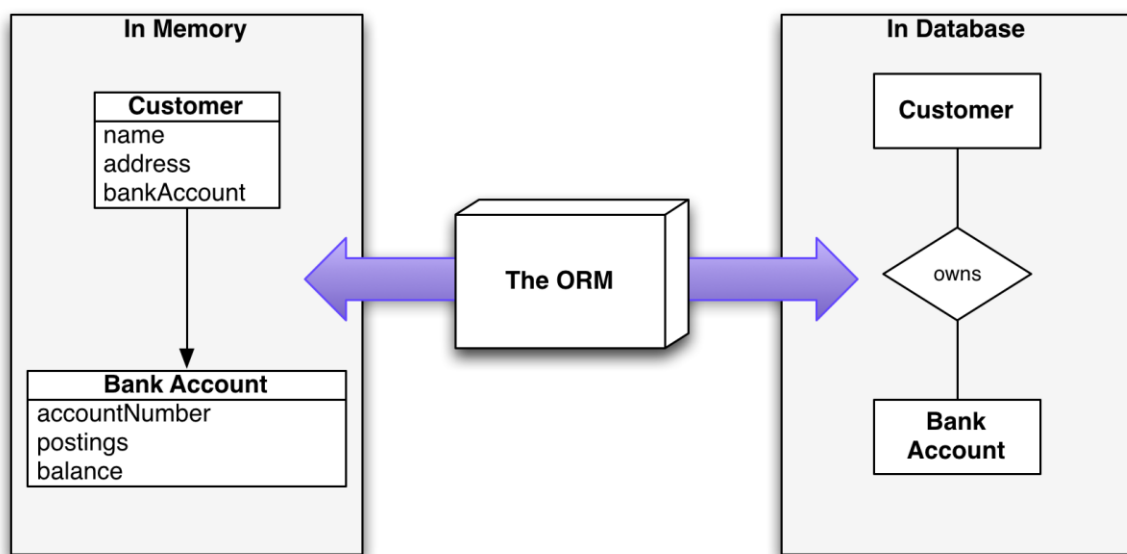


Figure 1: Mapeo Objeto-Relacional (ORM)

Por ejemplo, si trabajamos directamente con JDBC tendremos que descomponer el objeto para construir la sentencia `INSERT` del siguiente ejemplo

```
...
String sentenciaSql = "INSERT INTO personajes (nombre, descripcion, vida,
ataque)" +
    ") VALUES (?, ?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sentenciaSql);
sentencia.setString(1, personaje.getNombre());
sentencia.setString(2, personaje.getDescripcion());
sentencia.setInt(3, personaje.getVida());
sentencia.setInt(4, personaje.getAtaque());
sentencia.executeUpdate();
```

```
if (sentencia != null)
    sentencia.close();
. . .
```

Si contamos con un framework como *Hibernate*, esta misma operación se traduce en unas pocas líneas de código en las que **podemos trabajar directamente con el objeto Java**, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican a éste con que tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

```
@Entity
@Table(name="personajes")
public class Personaje {
    @Id // Marca el campo como la clave de la tabla
    @GeneratedValue(strategy = IDENTITY)
    @Column(name="id")
    private int id;
    @Column(name="nombre")
    private String nombre;
    @Column(name="descripcion")
    private String descripcion;
    @Column(name="vida")
    private int vida;
    @Column(name="ataque")
    private int ataque;

    public Personaje(. . .) {
        . . .
    }

    // getters y setters
}
```

Así, podemos simplemente establecer una sesión con la Base de Datos y enviarle el objeto, en este caso invocando al método `save` que se encarga de registrarlo en la Base de Datos donde convenga según sus propias anotaciones.

```
. . .
sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
sesion.save(personaje);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Herramientas ORM. Características

Las herramientas ORM nos permiten crear una capa de acceso a datos; una forma sencilla y válida de hacerlo es crear una clase por cada tabla de la base de datos y mapearlas una a una.

Estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, solo será necesario cambiar alguna línea en el fichero de configuración. Existen muchas herramientas ORM, en este tema estudiaremos Hibernate que es uno de los ORM más populares.

Hibernate

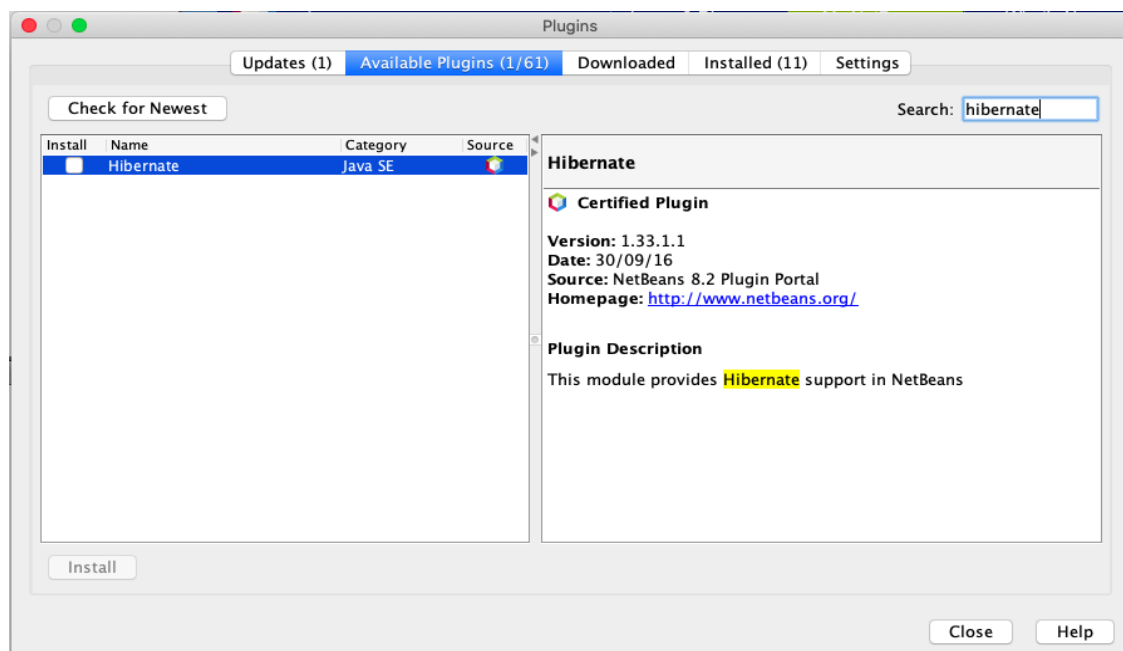
Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java (disponible también para .Net) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML) que permiten establecer estas relaciones.

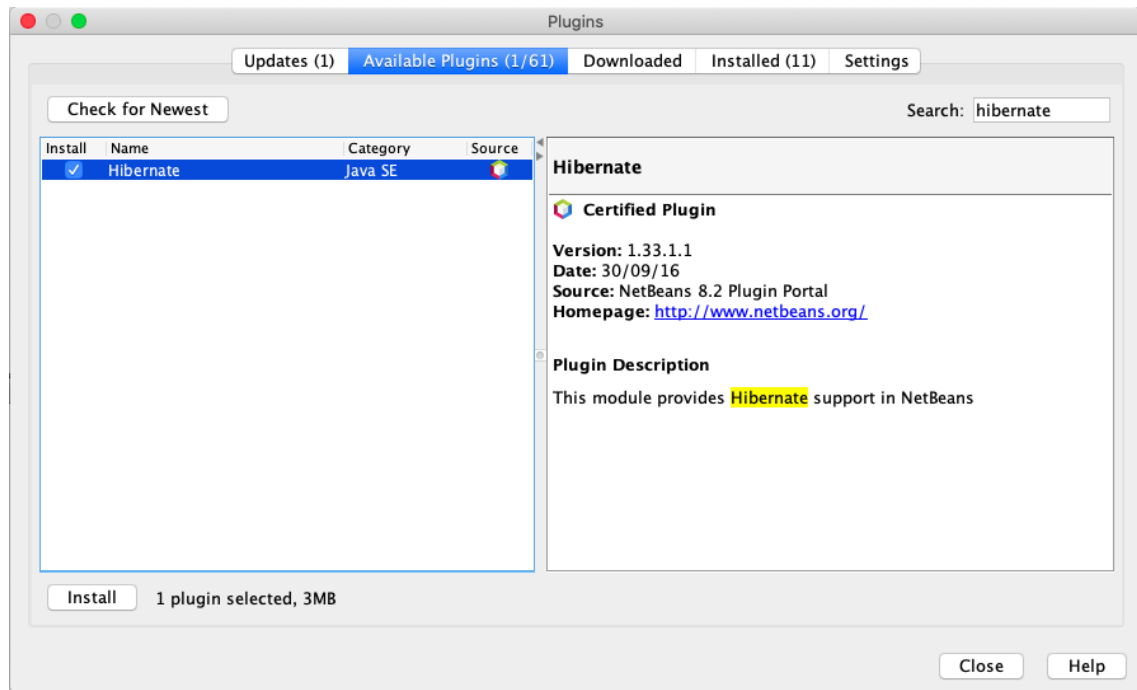
Con Hibernate no emplearemos habitualmente SQL para acceder a datos, sino que el propio motor de Hibernate, mediante el uso de factorías (patrón de diseño **Factory**) y otros elementos de programación construirá esas consultas para nosotros. Hibernate pone a disposición del diseñador un lenguaje llamado **HQL (Hibernate Query Language)** que permite acceder a datos mediante POO.

Instalación de Hibernate en Netbeans

Instalamos el plugin de Hibernate, para ello vamos a la pestaña Tools/Plugins, buscamos el plugin hibernate y lo instalamos. Debemos usar una versión de Netbeans inferior a la 12. Por ejemplo Netbeans 8.2 o Netbeans 11.3.

Si no lo tenemos disponible deberemos asegurarnos de que en Settings tenemos habilitado Netbeans Plugin Portal y actualizar los plugins disponibles desde la pestaña Update.



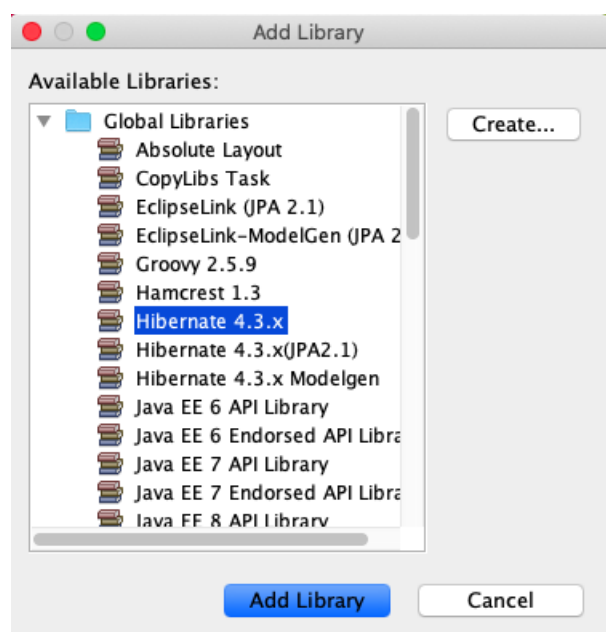


En este apartado vamos a crear un proyecto en java y a configurar Hibernate para que actúe entre la base de datos MySQL, que tenemos funcionando mediante el puerto 3306, y nuestro código en Java. Gracias a Hibernate trataremos la Base de Datos relacional como si fueran objetos.

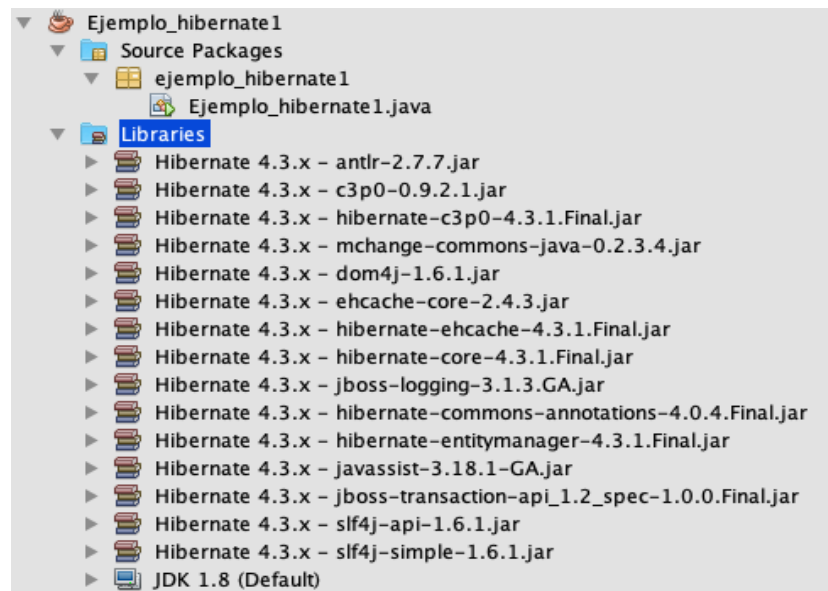
Con *Hibernate*, la forma normal de trabajar es generar unos ficheros de extensión **.hbm.xml** en los que se define de alguna manera las tablas de base de datos, asociaciones entre ellas y los beans java asociados a dichas tablas. Luego, desde la aplicación java, usando *hibernate* y estos ficheros de configuración, se pueden crear automáticamente las tablas en la base de datos.

Puesto que esta es la forma normal de trabajar, se conoce como **ingeniería inversa** en *Hibernate* al proceso contrario, es decir, a partir de las tablas ya creadas de base de datos, generar los ficheros .hbm.xml. Vamos a ver aquí cómo realizar este proceso con la base de datos ejemplo que tenemos creada en MySQL.

Creamos un nuevo proyecto
Ejemplo_hibernate_1, botón
derecho en la carpeta Libraries
y añadimos los JAR de
Hibernate

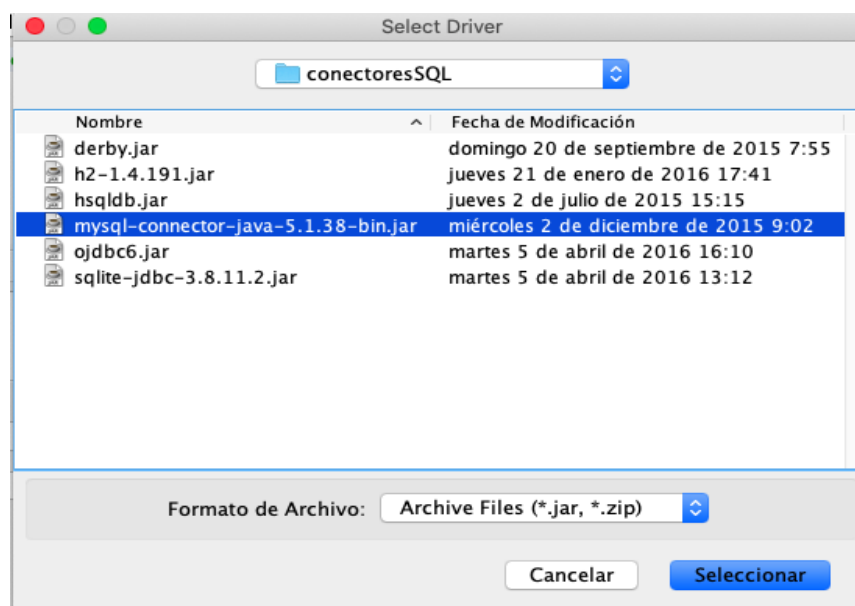
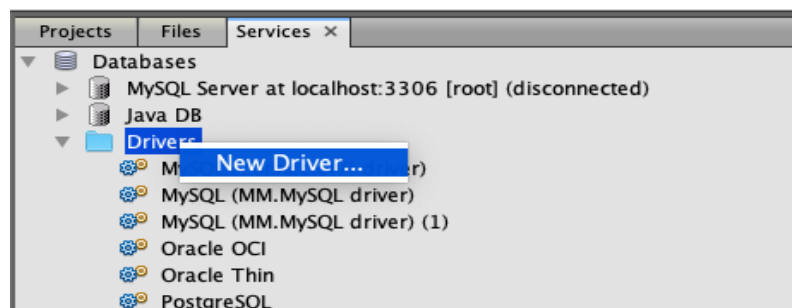


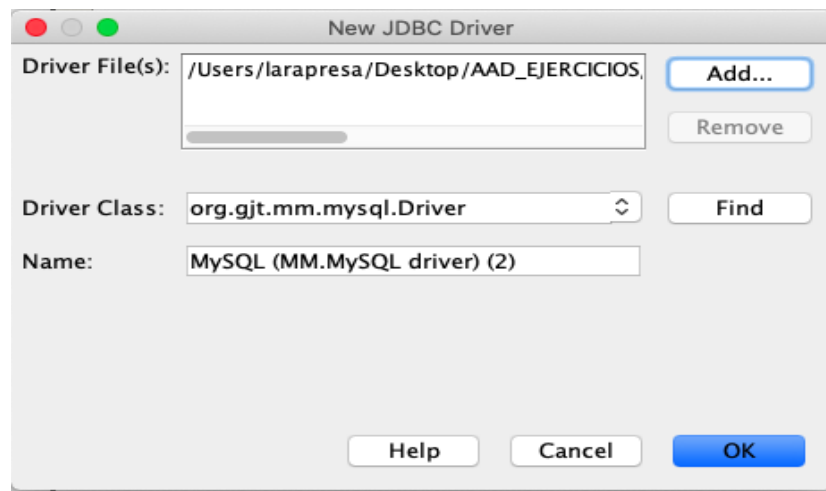
El proyecto ya tiene las librerías cargadas



También debemos añadir al proyecto el jar del conector mysql.

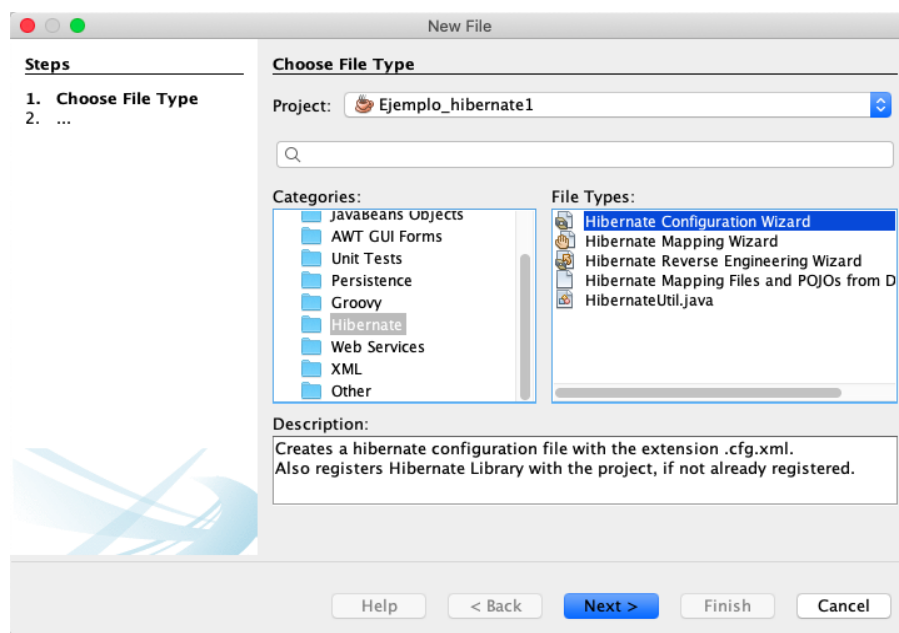
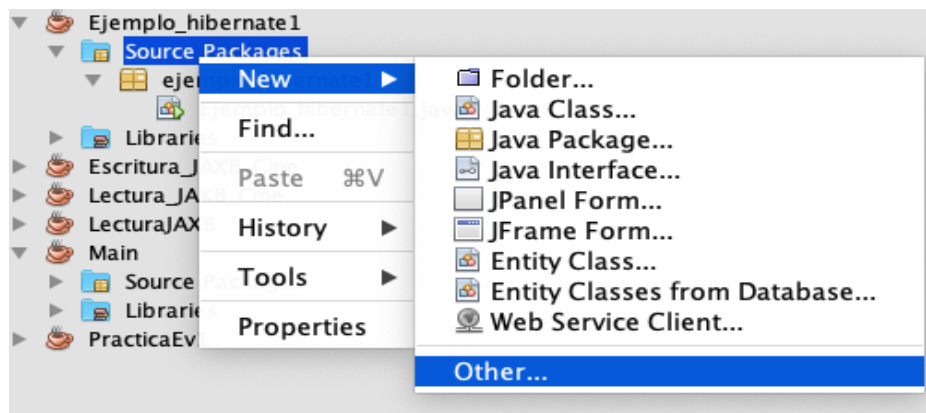
Antes de continuar con la configuración de Hibernate, vamos a configurar la conexión a la base de datos desde Netbeans. Para ello vamos a la pestaña Services.



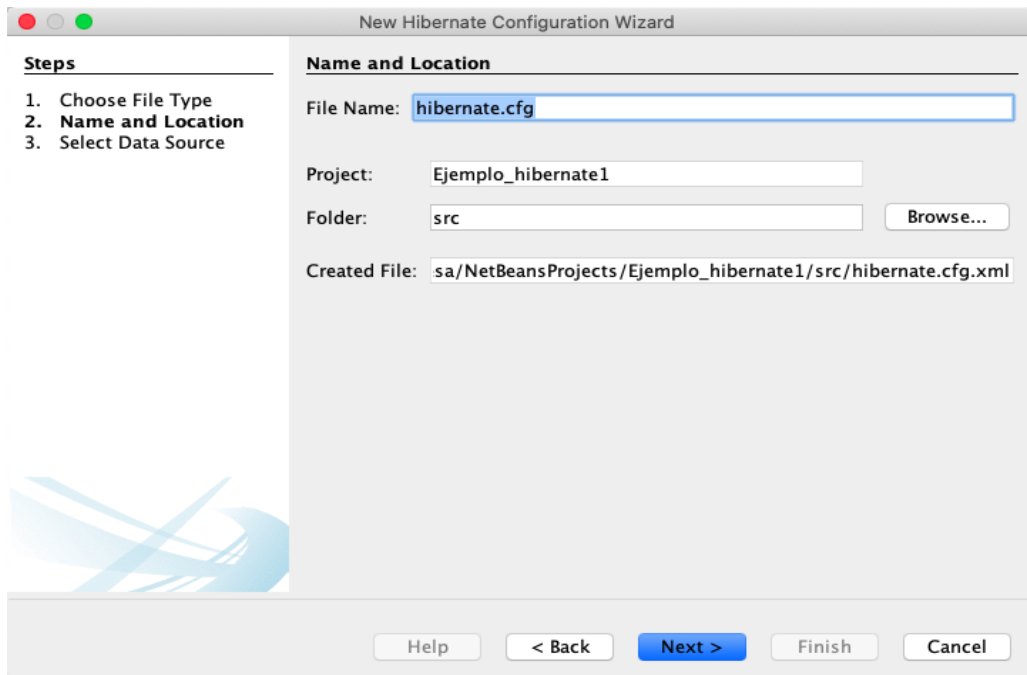


De este modo añadimos el driver correcto de MySQL que emplearemos para la conexión.

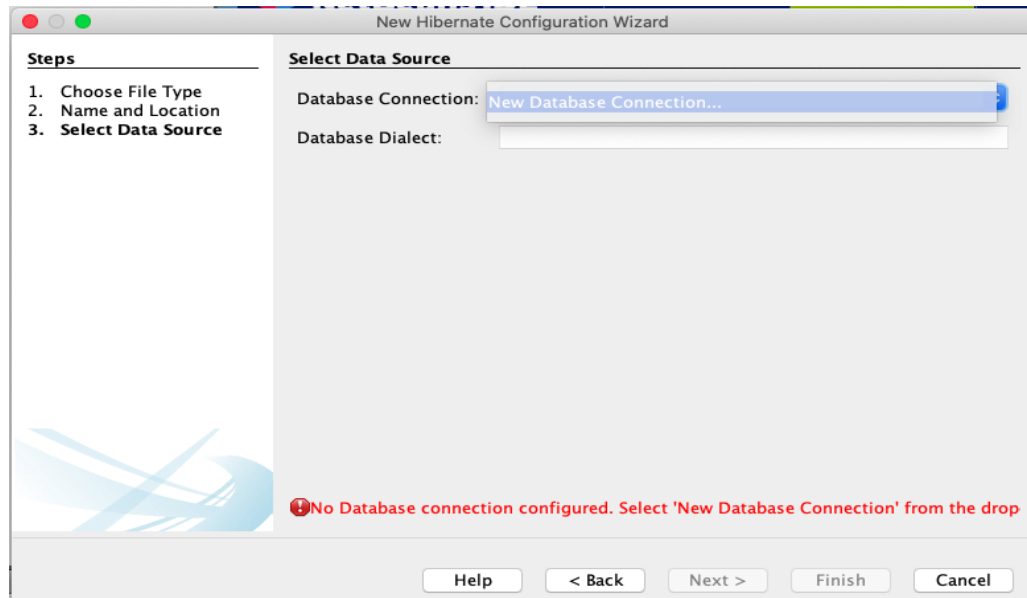
Empezamos a crear los archivos de configuración Hibernate, el asistente de configuración nos va a ayudar a crear el fichero .cfg.



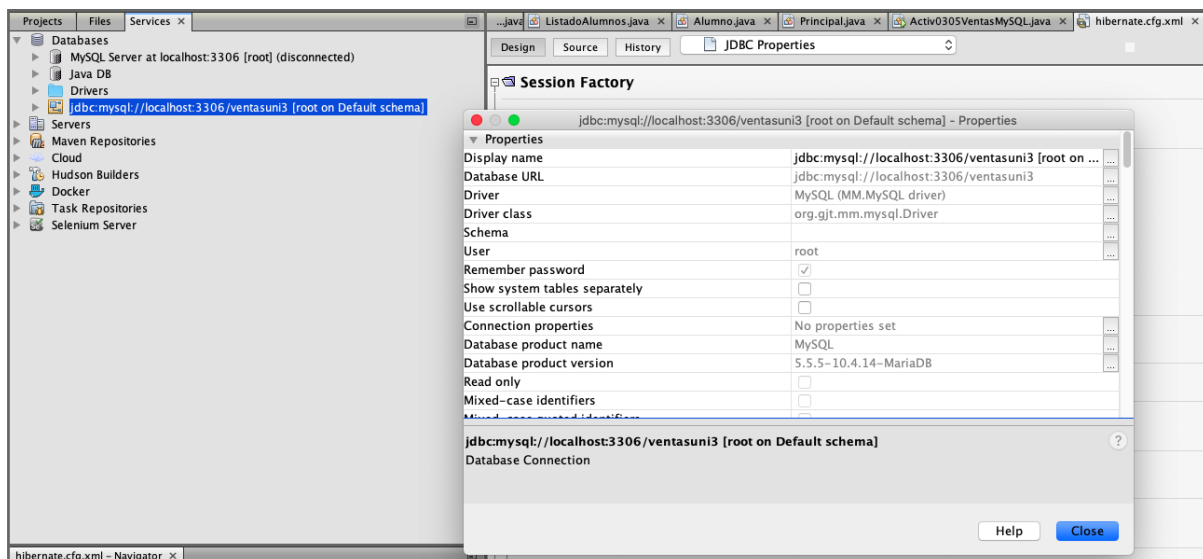
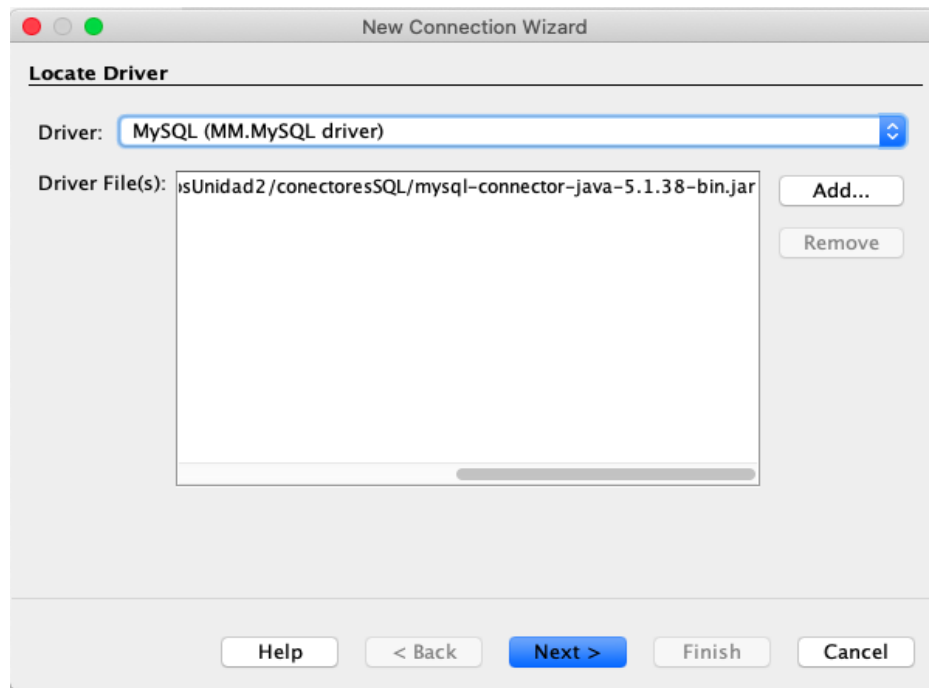
Comprobamos nombre y carpeta de destino



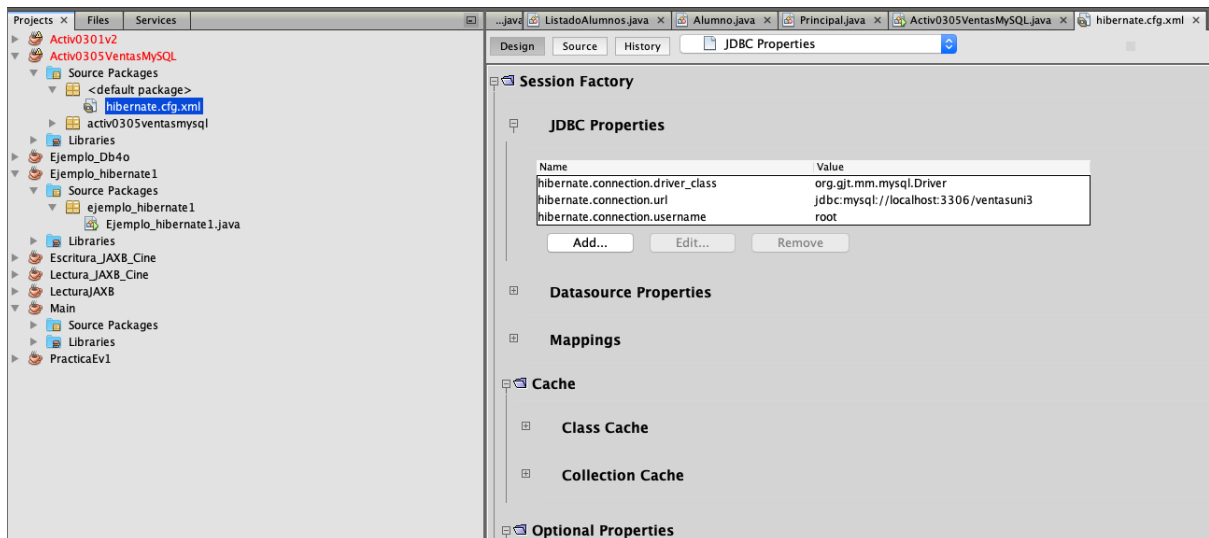
The screenshot shows the 'New Hibernate Configuration Wizard' dialog box, specifically the 'Name and Location' step. On the left, the 'Steps' list shows '1. Choose File Type', '2. Name and Location' (which is the current step), and '3. Select Data Source'. The main area contains the following fields: 'File Name:' with the text 'hibernate.cfg' entered; 'Project:' with 'Ejemplo_hibernate1' entered; 'Folder:' with 'src' entered and a 'Browse...' button to its right; and 'Created File:' showing the path 'sa/NetBeansProjects/Ejemplo_hibernate1/src/hibernate.cfg.xml'. At the bottom, there are five buttons: 'Help', '< Back', 'Next >' (highlighted in blue), 'Finish', and 'Cancel'.



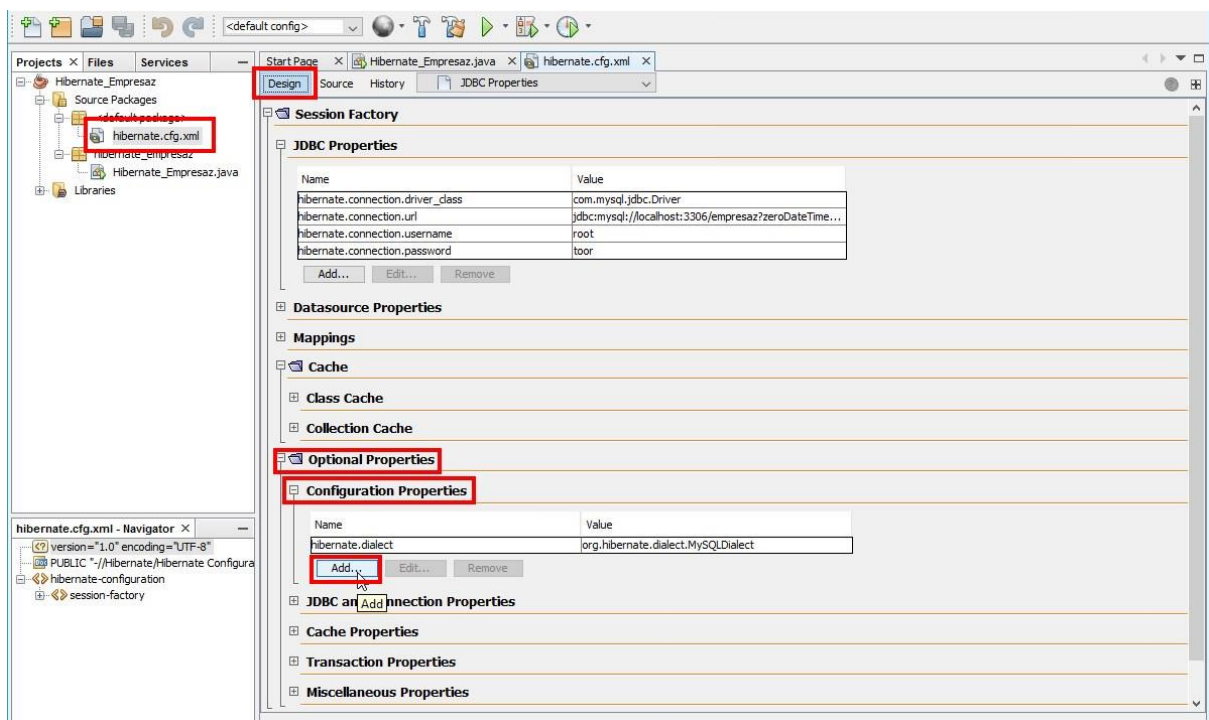
The screenshot shows the 'New Hibernate Configuration Wizard' dialog box, specifically the 'Select Data Source' step. On the left, the 'Steps' list shows '1. Choose File Type', '2. Name and Location', and '3. Select Data Source' (which is the current step). The main area contains the following fields: 'Database Connection:' with a dropdown menu showing 'New Database Connection...' selected; and 'Database Dialect:' with an empty text field. At the bottom, there are five buttons: 'Help', '< Back', 'Next >', 'Finish', and 'Cancel'. A red error message is displayed at the bottom of the main area: 'No Database connection configured. Select 'New Database Connection' from the drop'.



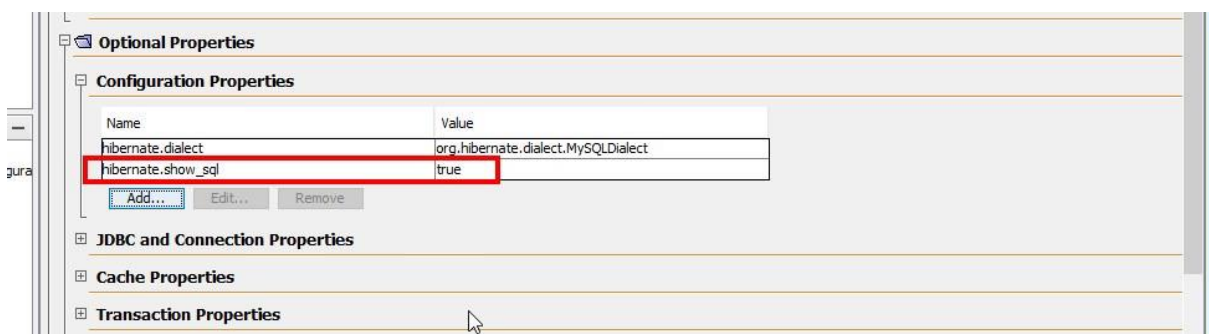
Ya tenemos el archivo creado, en formato xml, que Netbeans nos muestra en modo gráfico para mayor comodidad. Si se desea editar el archivo en modo texto se puede hacer sin problema desde la pestaña Source.



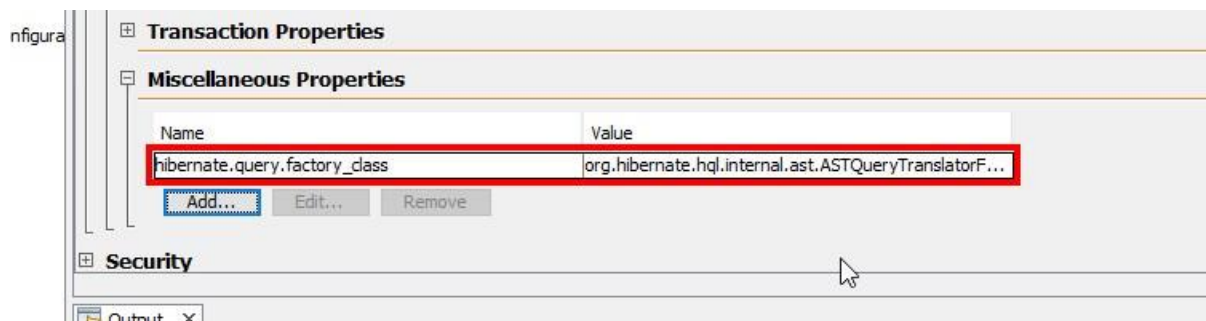
Vamos a editar una propiedad. Para ello primero añadimos.



En este caso queremos ver todo el log de depuración resultante de ejecutar operaciones SQL por consola. Añadimos también el dialecto MySQL si no lo tenemos.



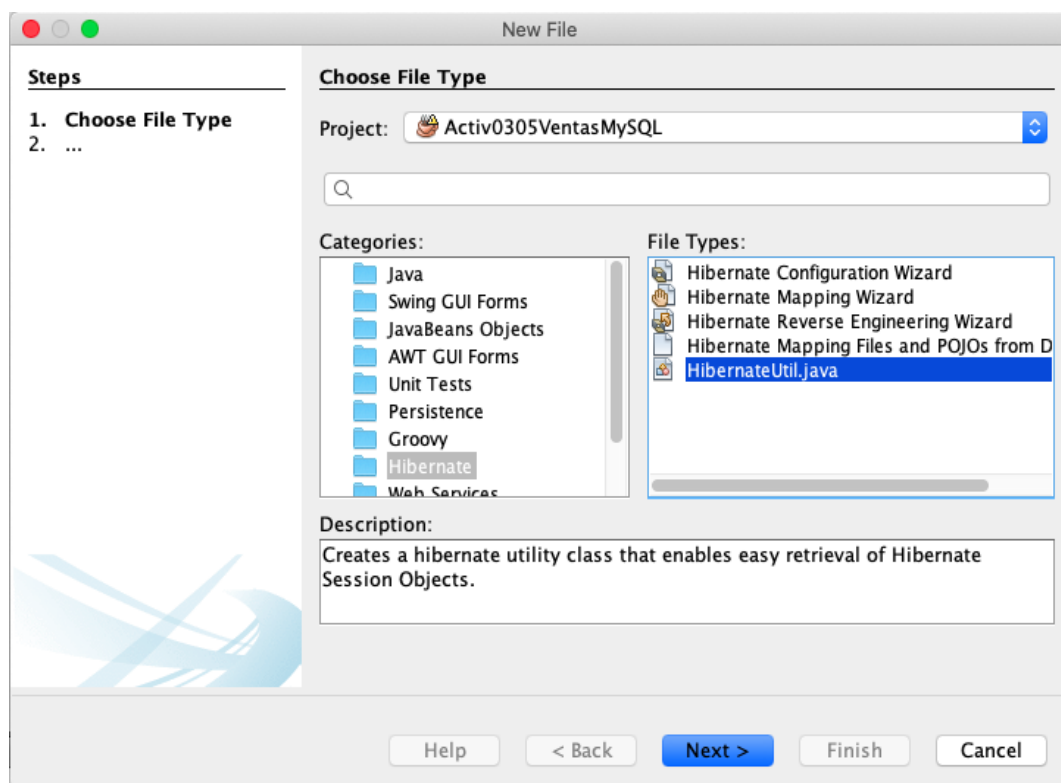
Esta opción de parser, en teoría viene implementada en Hibernate por defecto desde la versión 4.1. Aun así lo podemos activar por aquí.



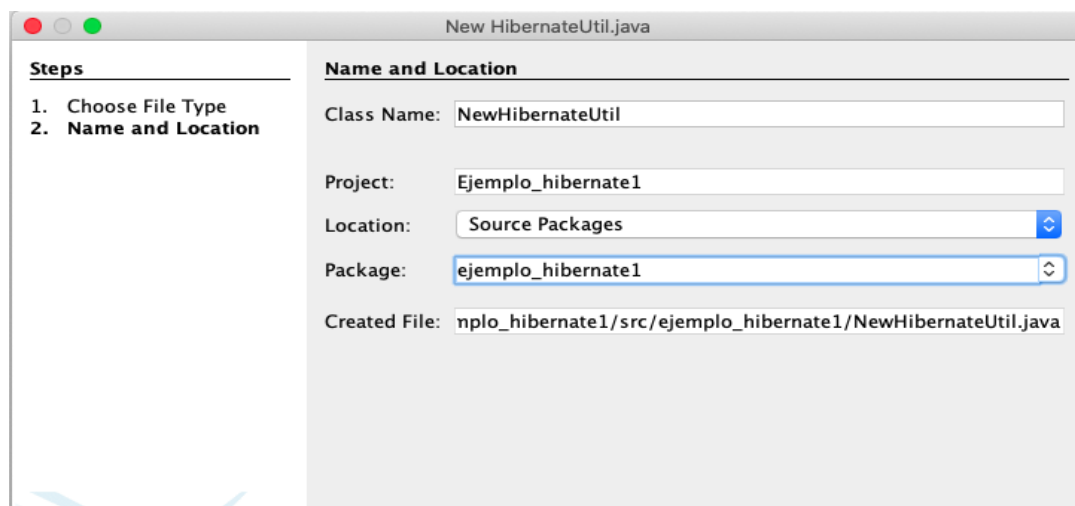
Seguimos creando archivos Hibernate

En este caso el **HibernateUtil**, que nos proporciona una referencia al objeto **SessionFactory** para que cualquier clase tenga acceso al objeto sesión en la aplicación. La clase HibernateUtil la debemos crear nosotros ya que no está incluida en Hibernate, contiene código estático que inicializa Hibernate y crea el objeto SessionFactory. Se incluye además un método estático que da acceso al objeto SessionFactory que se ha creado.

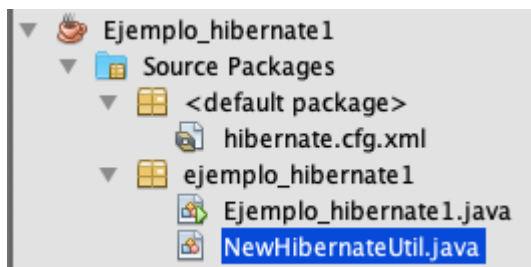
Esta clase es tan *famosa* que hasta el propio NetBeans tiene una opción para crearla, si vamos a la opción de menú "File --> New File ..." y seleccionamos en el árbol "Hibernate" veremos a la derecha la opción "HibernateUtil.java".



Seleccionamos nombre y paquete:



Ya tenemos nuestro archivo creado automáticamente



A partir de Hibernate 4.x hay algunos métodos de HibernateUtil que aparecen con la anotación *deprecated*. Si es así modificaremos este fichero y pondremos el siguiente que sí que está actualizado:

```
package ejemplo_hibernate1;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class NewHibernateUtil {
    private static SessionFactory sessionFactory;

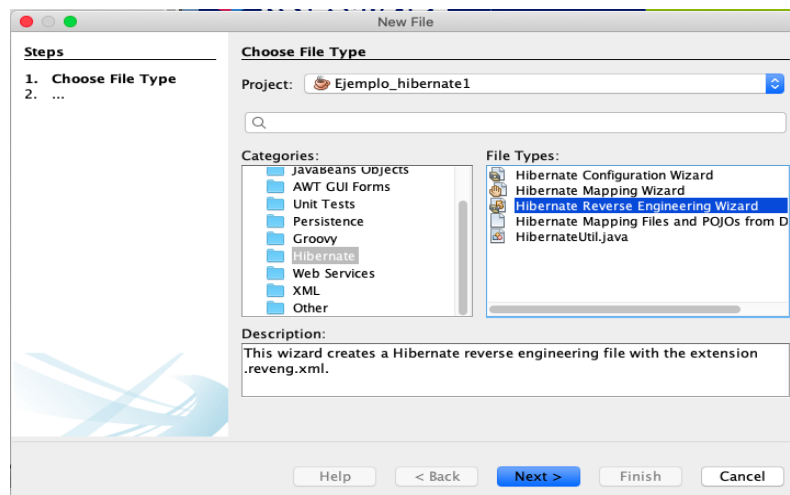
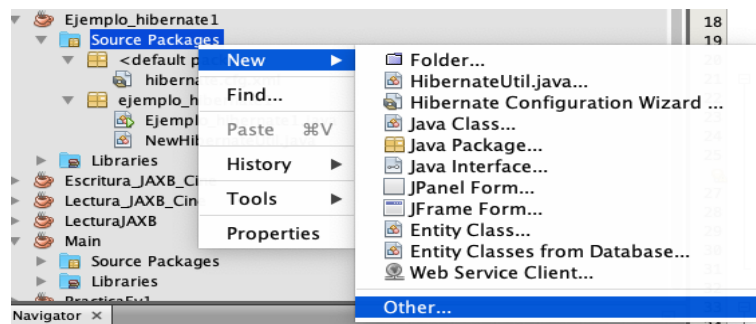
    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            // loads configuration and mappings
            Configuration configuration = new Configuration().configure();
            ServiceRegistry serviceRegistry
                = new StandardServiceRegistryBuilder()
```

```
.applySettings(configuration.getProperties()).build();

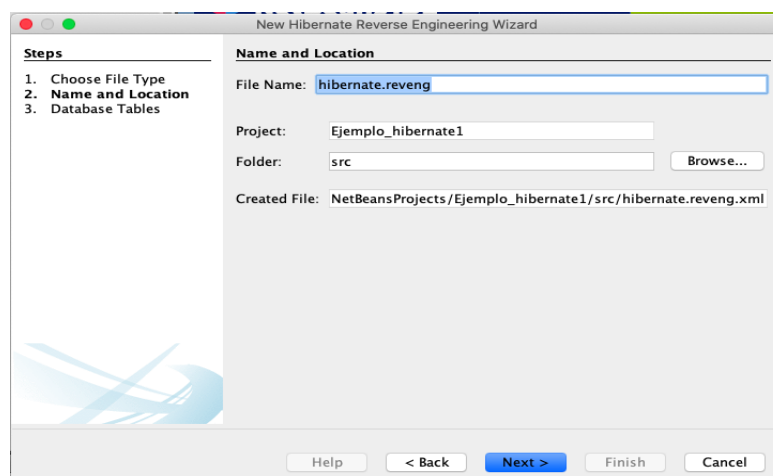
// builds a session factory from the service registry
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
}

return sessionFactory;
}
}
```

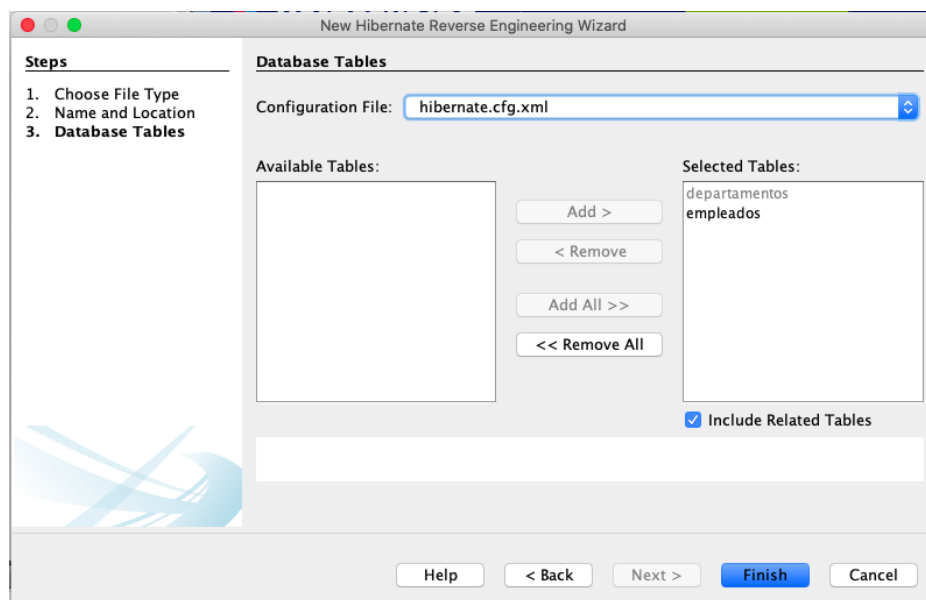
Ahora vamos a crear la **ingeniería inversa** de nuestra base de datos



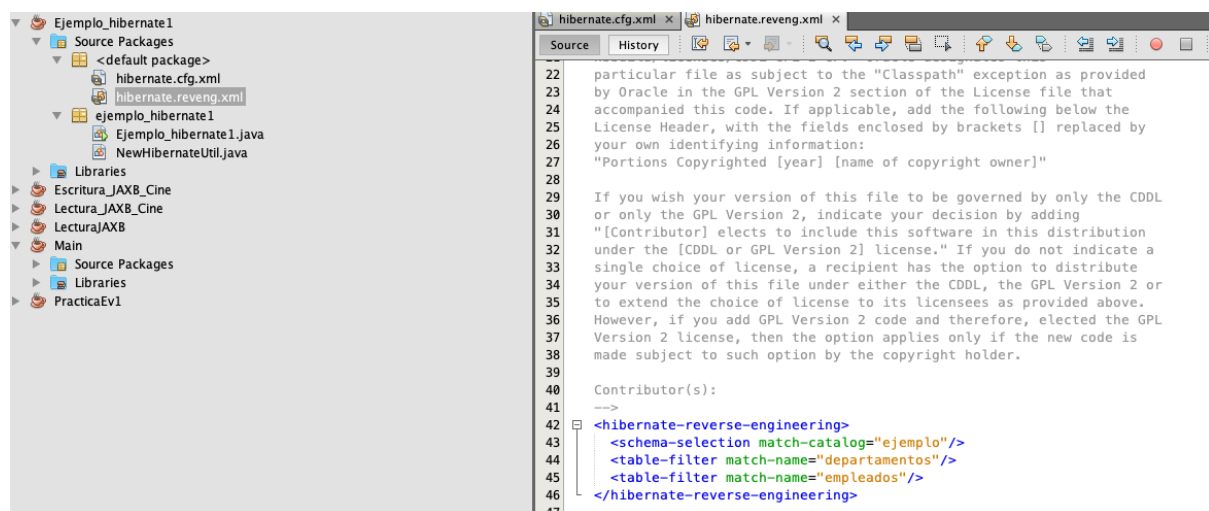
Comprobamos que el nombre y el directorio son los correctos:



Seleccionamos las tablas de la base de datos que queramos añadir para trabajar con ellas:



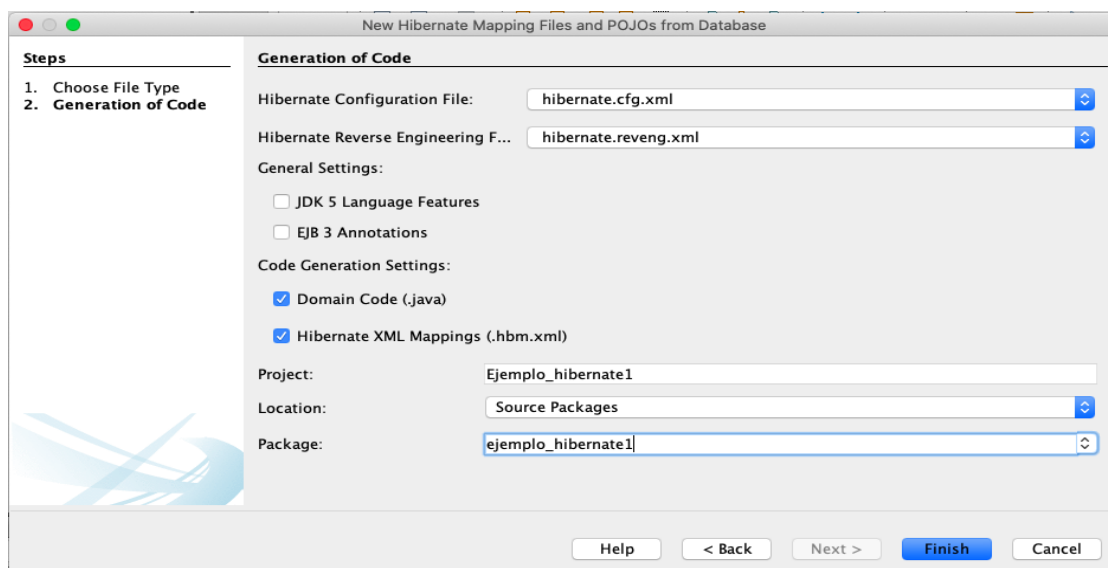
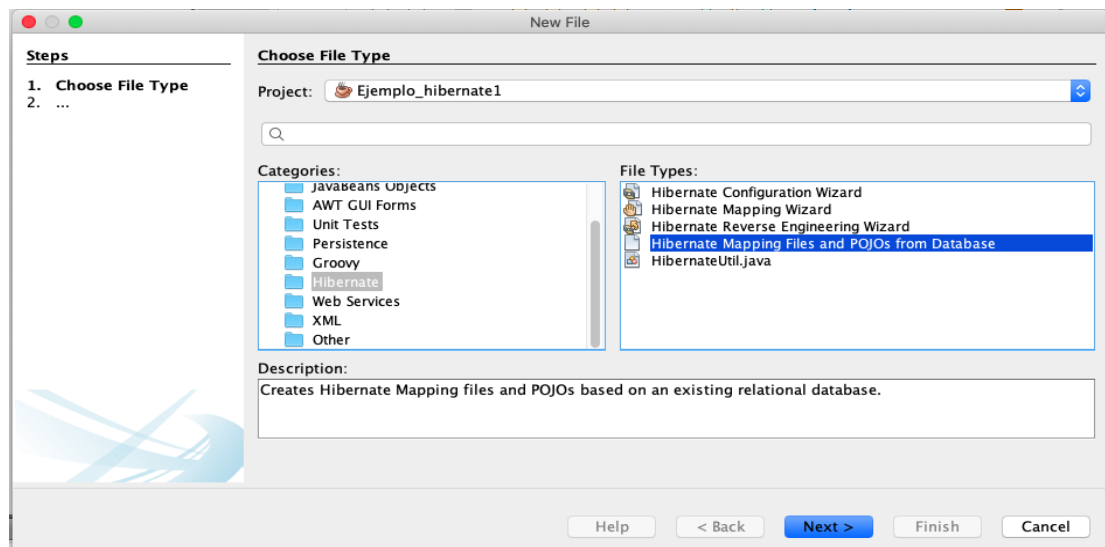
Ya tenemos creado el fichero **hibernate.reveng.xml** del cual podemos ver su contenido.



Por último, creamos los POJOs y los archivos de mapeado.

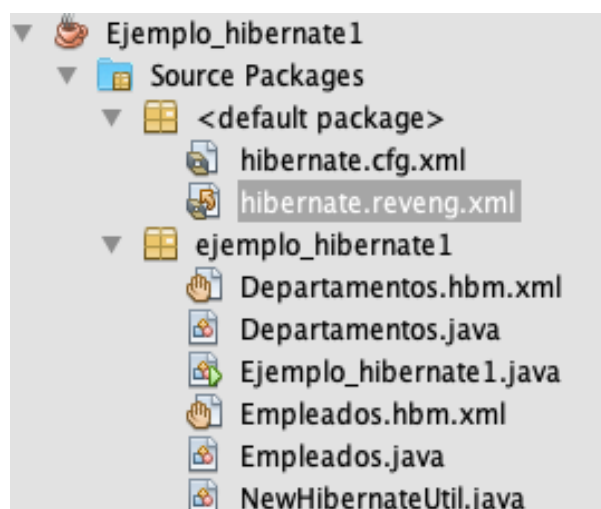
Hibernate funciona asociando a cada tabla de la base de datos un **Plain Old Java Object (POJO)**, a veces llamado Plain Ordinary Java Object). Un POJO es similar a una Java Bean, con propiedades accesibles mediante métodos setter y getter.

Volvemos a File, NewFile

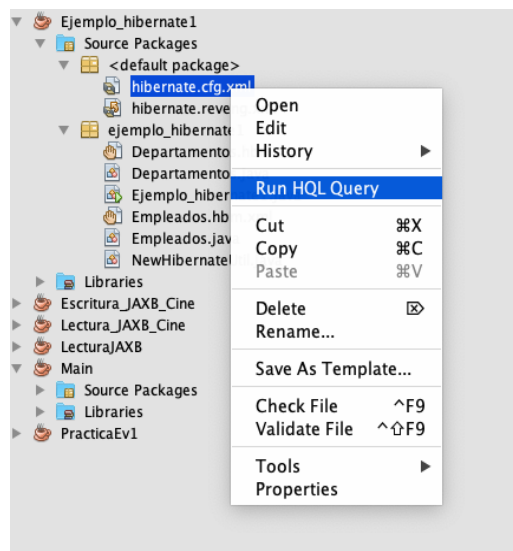


Definimos su propio paquete ya que nos generará una serie de archivos. Entre ellos, las clases java con sus atributos, constructores y métodos de get y set. También los archivos .xml de mapeado hibernate.

Este binomio xml-ClasesDeJava mapea la base de datos relacional y genera una capa intermedia que trata la tabla de datos relacional como una tabla objeto-relacional. De esta forma nos podemos relacionar con la tabla como si fueran objetos Java. Hibernate puede generar automáticamente los constructores y los métodos get y set, después podemos implementar los métodos que creamos conveniente.



Ahora ya estamos listos para empezar a probar y crear métodos. Se pueden hacer consultas HQL desde Netbeans.



Probamos algo sencillo y comprobamos tanto los datos de la tabla empleados como la tabla departamentos.

Session: hibernate.cfg

from Departamentos|

Result SQL

0 row(s) updated.; 5 row(s) selected.

Dnombre	Loc	Empleados...	DeptNo
POSTVENTA	LOGROÑO	[]	5
CONTABIL...	SEVILLA	[ejemplo_...	10
INVESTIG...	MADRID	[ejemplo_...	20
VENTAS	BARCELONA	[ejemplo_...	30
PRODUCC...	BILBAO	[]	40

Session: hibernate.cfg

from Empleados|

Result SQL

0 row(s) updated.; 14 row(s) selected.

Departamentos	Apellido	Comision	Oficio	FechaAlt	Salario	Dir	EmpNo
ejemplo_hibernate1.Departamentos@190dff34	SINCHEZ	NULL	EMPLEADO	1990-12...	1040.0	7902	7369
ejemplo_hibernate1.Departamentos@5197aa3b	ARROYO	390.0	VENDEDOR	1990-02...	1500.0	7698	7499
ejemplo_hibernate1.Departamentos@5197aa3b	SALA	650.0	VENDEDOR	1991-02...	1625.0	7698	7521
ejemplo_hibernate1.Departamentos@190dff34	JIM...NEZ	NULL	DIRECTOR	1991-04...	2900.0	7839	7566
ejemplo_hibernate1.Departamentos@5197aa3b	MARTON	1020.0	VENDEDOR	1991-09...	1600.0	7698	7654
ejemplo_hibernate1.Departamentos@5197aa3b	NEGRO	NULL	DIRECTOR	1991-05...	3005.0	7839	7698
ejemplo_hibernate1.Departamentos@3f5a3232	CEREZO	NULL	DIRECTOR	1991-06...	2985.0	7839	7782
ejemplo_hibernate1.Departamentos@190dff34	GIL	NULL	ANALISTA	1991-11...	3000.0	7566	7788
ejemplo_hibernate1.Departamentos@3f5a3232	REY	NULL	PRESIDENTE	1991-11...	4200.0	NULL	7839
ejemplo_hibernate1.Departamentos@5197aa3b	TOVAR	0.0	VENDEDOR	1991-09...	1350.0	7698	7844
ejemplo_hibernate1.Departamentos@190dff34	ALONSO	NULL	EMPLEADO	1991-09...	1430.0	7788	7876
ejemplo_hibernate1.Departamentos@5197aa3b	JIMENO	NULL	EMPLEADO	1991-12...	1335.0	7698	7900
ejemplo_hibernate1.Departamentos@190dff34	FERNANDEZ	NULL	ANALISTA	1991-12...	3000.0	7566	7902
ejemplo_hibernate1.Departamentos@3f5a3232	MUÑOZ	NULL	EMPLEADO	1992-01...	1790.0	7782	7934

Desde este entorno también se pueden realizar consultas al estilo SQL, respetando los nombres de las clases y de los atributos de las mismas, por ejemplo:

```
select dnombre, loc, deptNo from Departamentos
```

Ten en cuenta que el * no se puede utilizar a la derecha del SELECT ni tampoco se puede emplear ninguna sentencia que no respeten los nombres de las clases y los atributos.

ACTIVIDAD 3.1

Crea un nuevo proyecto Java para acceder a la base de datos ejemplo de MySQL usando Hibernate. Sigue los pasos indicados previamente y comprueba que todo se ha generado correctamente listando los departamentos y los empleados de la base de datos realizando consultas HQL desde Netbeans.

ACTIVIDAD 3.2

Realiza consultas HQL con las tablas mapeadas. Prueba estas consultas:

```
from Empleados as e where e.departamentos.deptNo = 10  
from Departamentos where deptNo=10  
from Departamentos as d join d.empleadoses  
from Departamentos as d left outer join d.empleadoses
```

Estructura de los ficheros de mapeo

Hibernate utiliza unos ficheros de mapeo para relacionar las tablas de la base de datos con los objetos Java. Estos ficheros están en formato XML y tienen la extensión **.hbm.xml**.

En el proyecto anterior se han creado los ficheros **Empleados.hbm.xml** y **Departamentos.hbm.xml** asociados a la tabla empleados y departamentos de nuestra base de datos ejemplo.

Estos ficheros se guardan en el mismo directorio que las clases Java Empleados.java y Departamentos.java.

```
<?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 15-nov-2020 10:52:31 by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
  <class name="ejemplo_hibernate1.Departamentos" table="departamentos" catalog="ejemplo" optimistic-lock="version">
    <id name="deptNo" type="byte">
      <column name="dept_no" />
      <generator class="assigned" />
    </id>
    <property name="dnombre" type="string">
      <column name="dnombre" length="15" />
    </property>
    <property name="loc" type="string">
      <column name="loc" length="15" />
    </property>
    <set name="empleadoses" table="empleados" inverse="true" lazy="true" fetch="select">
      <key>
        <column name="dept_no" not-null="true" />
      </key>
      <one-to-many class="ejemplo_hibernate1.Empleados" />
    </set>
  </class>
</hibernate-mapping>
```

Veamos el significado del contenido del fichero XML:

<hibernate-mapping> todos los ficheros de mapeo comienzan y acaban con esta etiqueta.

<class> engloba a la clase con sus atributos

<id> name(atributo clave en la clase)

<column> su nombre sobre la tabla

<generator> indica la naturaleza del campo clave

<set> se utiliza para mapear colecciones

<key> define el nombre de la columna identificadora de la asociación

<one-to-many> define la relación, en este caso uno-a-muchos.

Resumiendo, este mapeo indica que la clase Departamentos.java tiene un atributo de nombre empleadoses que es una lista de instancias de la clase ejemplo_hibernate1.Empleados.

Clases persistentes

En nuestro proyecto se han creado las clases *Empleados.java* y *Departamentos.java*. A estas clases se les llama clases persistentes, son las clases que implementan las entidades del problema, deben implementar la interfaz Serializable.

Equivalen a una tabla de la base de datos, y un registro o fila es un objeto persistente de esa clase. Además tienen unos atributos y unos métodos get y set para acceder a los mismos.

Utilizan convenciones de nombrado estándares de JavaBean para los métodos de propiedades getter y setter así como también visibilidad privada para los campos. Al ser los atributos de los objetos privados se crean métodos públicos para retornar un valor de un atributo o para cargar un valor de un atributo (p.e. getDnombre(), setDnombre()). A estas reglas se las llama modelo de programación POJO (Plain Old Java Object).

Para completar el nombre de un método getter o setter, solo hay que poner la primera letra que los une en mayúsculas. Si nos fijamos en el fichero **Departamento.hbm.xml**, el elemento **id** es la declaración de la propiedad identificadora, el atributo de mapeo name="deptNo" declara el nombre de la propiedad JavaBean y le dice a Hibernate que utilice los métodos getDeptNo() y setDeptNo() para acceder a la propiedad.

```
<id name="deptNo" type="byte">
```

Al igual que con el elemento **id**, el atributo **name** del elemento **property** le dice a Hibernate qué métodos getter y setter utilizar.

```
<property name="dnombre" type="string">
```

Sesiones y objetos hibernate

Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto Session utilizando la clase **SessionFactory** de Hibernate. El siguiente fragmento de código ilustra este proceso:

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

Donde NewHibernateUtil es la clase que hemos creado con el configurador y que contiene el siguiente código:

```
public class NewHibernateUtil {
    private static SessionFactory sessionFactory;

    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            // loads configuration and mappings
            Configuration configuration = new Configuration().configure();
            ServiceRegistry serviceRegistry
                = new StandardServiceRegistryBuilder()
                    .applySettings(configuration.getProperties()).build();

            // builds a session factory from the service registry
            sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        }
        return sessionFactory;
    }
}
```

Transacciones

Un objeto Session de Hibernate representa una única unidad de trabajo para un almacén de datos dado y lo abre un ejemplar de SessionFactory. Al crear la sesión se crea la transacción para dicha sesión. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción.

El siguiente código ilustra una sesión de persistencia de Hibernate:

```
Session sesion = sessionFactory.openSession(); // crea la sesión
Transaction tx = sesion.beginTransaction(); // crea la transacción
// Código de persistencia
.....
tx.commit(); // valida la transacción
sesion.close(); // finaliza la sesión
```

El método **beginTransaction()** marca el comienzo de la transacción. El método **commit()** valida la transacción y **rollback()** deshace la transacción.

Estados de un objeto Hibernate

Hibernate define y soporta los siguientes estados de objeto:

- *Transitorio (Transient)* - un objeto es transitorio si ha sido recién instanciado utilizando el operador new, y no está asociado a una Session de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más una referencia. Podemos hacer una instancia transitoria persistente asociándola con una sesión.
- *Persistente (Persistent)* - una instancia persistente tiene una representación en la base de datos y un valor identificador. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una Session. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo.
- *Separado (Detached)* - una instancia separada es un objeto que se ha hecho persistente, pero su Session ha sido cerrada. La referencia al objeto todavía es válida, por supuesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser asociada a una nueva Session más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

Carga de objetos

Para la carga de objetos usaremos los siguientes métodos de **Session**:

MÉTODO	DESCRIPCIÓN
<code><T> T load (Class<T> Clase, Serializable id)</code>	Devuelve la instancia persistente de la clase indicada con el identificador dado. La instancia tiene que existir, si no existe el método lanza una excepción (<code>ObjectNotFoundException</code>).
<code>Object load (String nombreClase, Serializable id)</code>	Similar al método anterior pero en este caso indicamos en el primer parámetro el nombre de la clase en formato String
<code><T> T get (Class<T> Clase, Serializable id)</code>	Devuelve la instancia persistente de la clase indicada con el identificador dado. La instancia tiene que existir, si no existe devuelve null
<code>Object get (String nombreClase, Serializable id)</code>	Similar al método anterior pero en este caso indicamos en el primer parámetro el nombre de la clase en formato String

Ejemplo de load:

Visualiza los datos del departamento 10:

```
Departamentos dep = new Departamentos();
try{
    dep = (Departamentos) sesion.load(Departamentos.class, (byte) 10);
    System.out.println("Nombre Dep:" + dep.getDnombre());
    System.out.println("Localidad:" + dep.getLoc());
} catch (ObjectNotFoundException o) {
    System.out.println("NO EXISTE EL DEPARTAMENTO!!");
}
```

Usando el segundo formato de load() quedaría así:

```
dep = (Departamentos) sesion.load("ejemplo_hibernate1.Departamentos", (byte) 10);
```

Si no tenemos la certeza de que la fila exista debemos utilizar el método get() que devuelve null si no existe la fila correspondiente.

Almacenamiento, modificación y borrado de objetos

Para almacenamiento, modificación y borrado de objetos usamos los siguientes métodos de Session.

MÉTODO	DESCRIPCIÓN
Serializable save (Object obj)	Guarda el objeto en la base de datos. Hace que la instancia transitoria del objeto sea persistente.
void update (Object objeto)	Actualiza en la base de datos el objeto que se pasa como argumento. El objeto a modificar debe ser cargado con el método load() o get()
void delete (Object objeto)	Elimina de la base de datos el objeto que se pasa como argumento. El objeto a eliminar debe ser cargado con el método load() o get().

EJEMPLOS 1

ACTIVIDAD 3.3

Añade al proyecto los siguientes ejemplos. Analiza el código y ejecutalos comprobando que se realizan las modificaciones correspondientes en la base de datos.

El siguiente listado de ejemplos se puede descargar de los recursos de la Unidad 3.

1. EjL01MainEmpleado. Inserta un empleado en la tabla Empleados, en el departamento 10
 2. EjL02ListadoDep. Lista los datos del departamento 10 y sus empleados.
 3. EjL03BorradoDep. Borra del departamento 10.
 4. EjL04ModificarEmpleado. modifica el salario y el departamento del empleado 7369. le asigna el departamento 30 y le sube el salario 1000Euros.
-

ACTIVIDAD 3.4

Sube el salario a todos los empleados del departamento 10. Muestra el apellido y el salario del empleado antes y después de actualizar.

Consultas

Hibernate soporta un lenguaje de consulta orientado a objetos denominado HQL (Hibernate Query Language) fácil de usar pero potente a la vez. Este lenguaje es una extensión orientada a objetos de SQL. Las consultas HQL y SQL nativas son representadas con una instancia de **org.hibernate.Query**. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto resultado y para la ejecución de la sentencia real. Siempre obtiene una **Query** utilizando el objeto **Session** actual.

Para realizar una consulta usaremos el método **createQuery()** de la interface **SharedSessionContract**, se le pasará en un String la consulta HQL.

```
Query q = sesion.createQuery("from Departamentos");
```

Para recuperar los datos de la consulta usaremos el método **list()** o el método **iterate()**

```
List<Departamentos> lista = q.list(); // devuelve en una colección todos los resultados de la consulta
```

Iterator iter = q.iterate(); // devuelve un iterador Java para recuperar los datos de la consulta. En este caso Hibernate ejecuta la consulta obteniendo solo los ids de las entidades, y en cada llamada al método `Iterator.next()` ejecuta la consulta propia para obtener la entidad completa. Esto implica mayor cantidad de accesos a la base de datos y, por tanto, mayor tiempo de procesamiento total. La ventaja es que no se requiere que todas las entidades estén cargadas en memoria simultáneamente. Se puede utilizar el método `setFetchSize()` para fijar la cantidad de resultados a recuperar en cada acceso a la base de datos.

EJEMPLOS 2

ACTIVIDAD 3.5

Añade al proyecto los siguientes ejemplos de consultas. Analiza el código y ejecutalos comprobando que se realizan las modificaciones correspondientes en la base de datos.

El siguiente listado de ejemplos emplean consultas. Se pueden descargar de los recursos de la Unidad 3.

5. EjL05ListaDepartamentos. Realiza una consulta de todas las filas de la tabla departamentos empleando el método `list()`
6. EjL06ListaDepartamentosIterator. Realiza una consulta de todas las filas de la tabla departamentos empleando el método `iterate()`
7. EjL07ListaEmpleados20. Lista los empleados del departamento 20

8. EjL08EjemploUniqueResult. El método **uniqueResult()** ofrece un atajo si sabemos que la consulta devolverá un objeto.
9. EjL09ConsultasParametros. Hibernate soporta parámetros con nombre y parámetros de estilo JDBC (?) en las consultas HQL. Los parámetros con nombre son identificadores de la forma **:nombre** en la cadena de la consulta. Hibernate numera los parámetros desde cero. Para asignar nombre a los parámetros se emplean los métodos **setXXX**. La sintaxis más simple de utilizar es usando el método **setParameter()**.
10. EjL10ClasesNoasociadas. Consultas sobre clases no asociadas. Si queremos recuperar los datos de una consulta en la que intervienen varias tablas y no tenemos asociada a ninguna clase los atributos que devuelve esa consulta podemos utilizar la clase **Object**. Los resultados se devuelven en un array de objetos, donde el primer elemento del array se corresponde con la primera clase que ponemos a la derecha de FROM, el siguiente elemento con la siguiente clase y así sucesivamente. Este ejemplo realiza una consulta para obtener los datos de los empleados y de sus departamentos. El resultado de la consulta se recibe en un array de objetos donde el primer elemento del array pertenece a la clase Empleados y el segundo a la clase Departamentos.
11. EjL11UsoClaseTotales. Anteriormente vimos cómo se pueden tratar los resultados obtenidos por un SELECT que no está asociada a ninguna entidad. Supongamos que a partir de las tablas empleados y departamentos quiero obtener una consulta en la que aparezcan en nombre del departamento, su número, el número de empleados y el salario medio. Como los datos de esta consulta no están asociados a ninguna clase, puedo crear una y utilizarla sin necesidad de mapearla. Cada fila devolverá un objeto de esa clase.

Por ejemplo creo la clase Totales con 4 atributos: número, cuenta, media y nombre, y los constructores getter y setter correspondientes. En el ejemplo se puede ver cómo hacemos uso de la clase Totales.
12. EjL12UsoUPDATE_DELETE. Tanto el UPDATE como el DELETE se realizan con **executeUpdate()**. En este ejemplo se modifica el salario de GIL y eliminamos los empleados del departamento 20. **tx.commit()** valida la transacción. **tx.rollback()** deshace la transacción.
13. EjL13UsoINSERT. Hay que crear previamente la tabla en SQL, modificar el fichero hibernate.reveng.xml y generar la nueva clase. Ver página 189 del libro Acceso a Datos.

ACTIVIDAD 3.6

Realiza una consulta con `createQuery()` para obtener los datos del departamento 20 y visualiza también el apellido de sus empleados.
