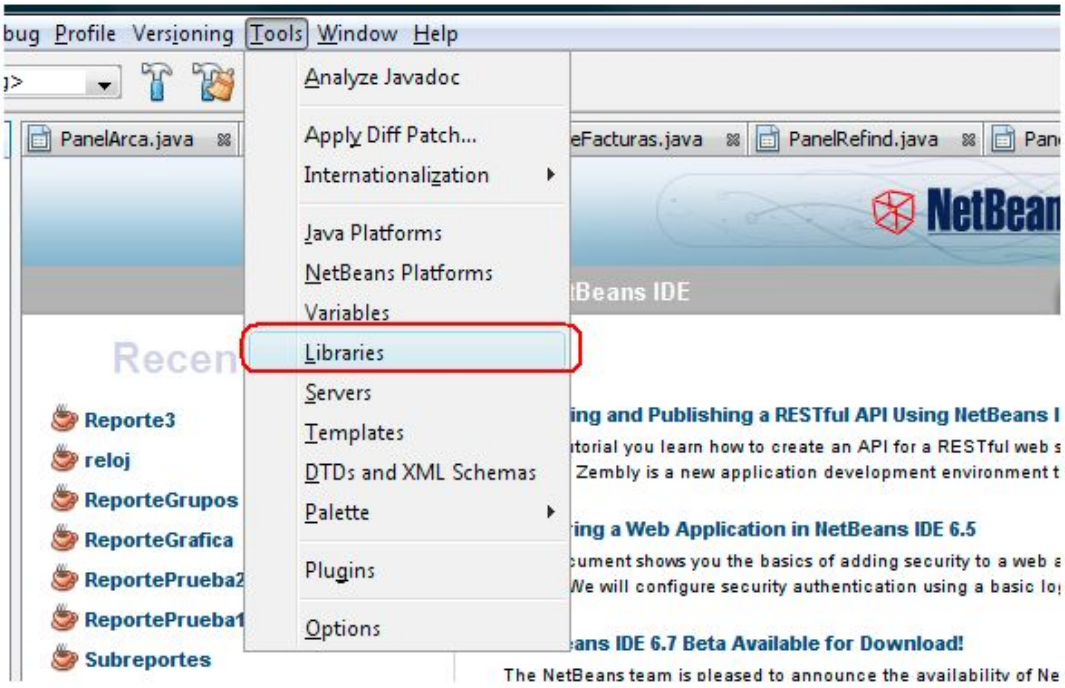
Persistiendo Objetos Simples usando Mapeos en XML

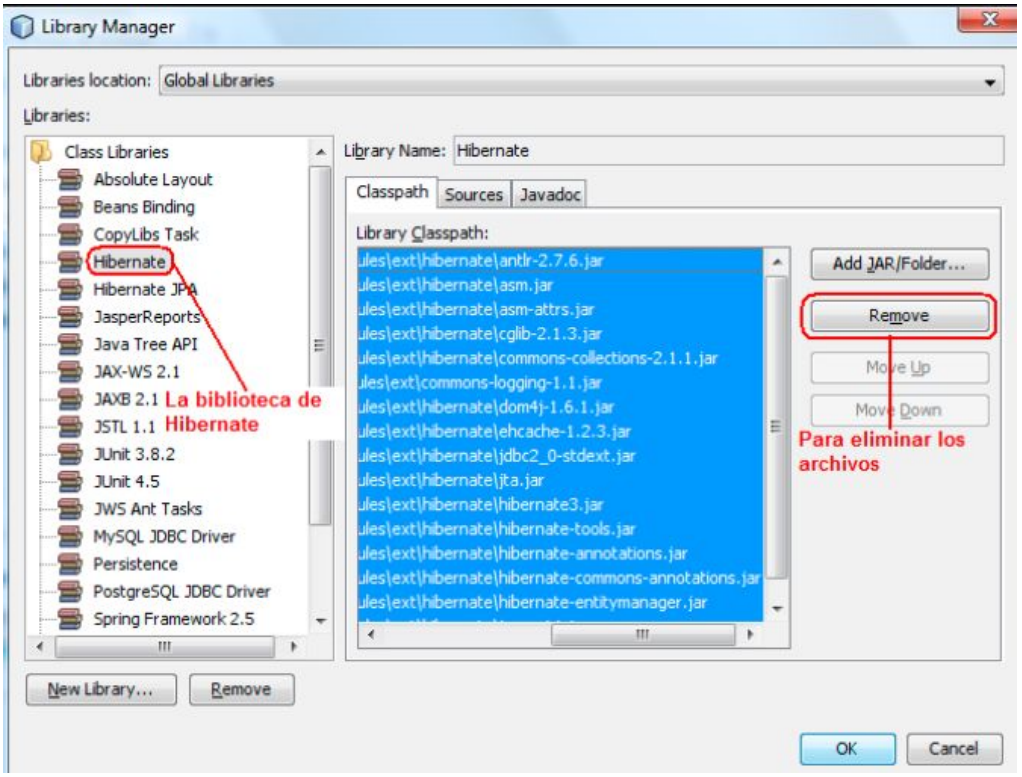
Primero debemos bajar el conector para la base de datos: conector para Java versión 5.1.23.
Luego lo que haremos, es crear una biblioteca de NetBeans con los jars básicos de Hibernate, de esta forma cada vez que queramos usar Hibernate en un proyecto solo tendremos que agregar esta biblioteca.
En realidad no crearemos esta biblioteca desde cero, ya que NetBeans ya tiene una biblioteca de Hibernate, solo actualizaremos la biblioteca con la última versión de Hibernate, la 5.2.10 Final. Así que descargamos la última versión del core de Hibernate desde la página de descarga de Hibernate.

Existen varias clases de soporte que son usadas por Hibernate para su funcionamiento normal. Algunas son opcionales y otras son requeridas. Afortunadamente estas se encuentran separadas en el mismo archivo que hemos descargado. Las clases obligatorias, que son las que nos interesan en este caso, se encuentran en el directorio "`lib\required`".

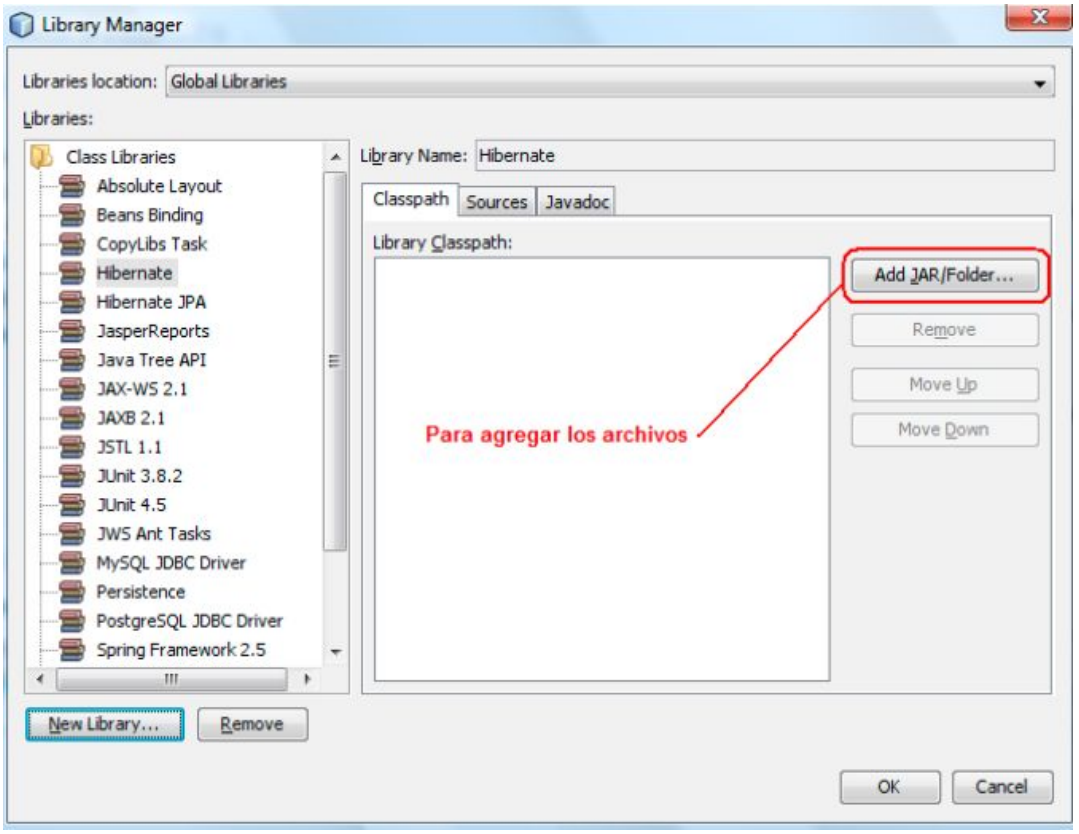
Para actualizar la biblioteca abrimos el NetBeans y nos dirigimos al menú "`Tools -> Libraries`":



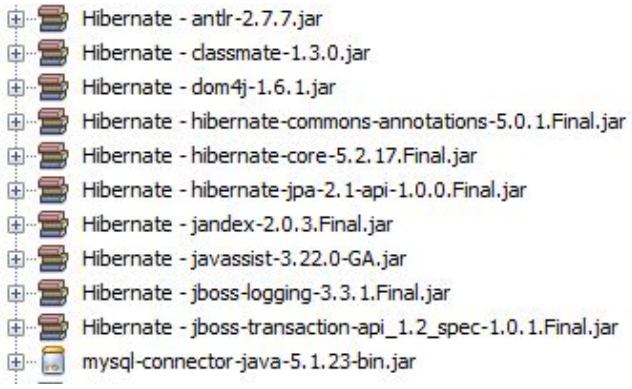
En la ventana que se abre buscamos la biblioteca "`Hibernate`" y la seleccionamos. (Si no existe la creamos) Con esto se mostrarán los archivos que la conforman. Seleccionamos todos los archivos y presionamos el botón "`Remove`" para eliminarlos de la biblioteca.



Una vez eliminados los archivos presionamos el botón "`Add JAR/Folder...`" para agregar los nuevos archivos:

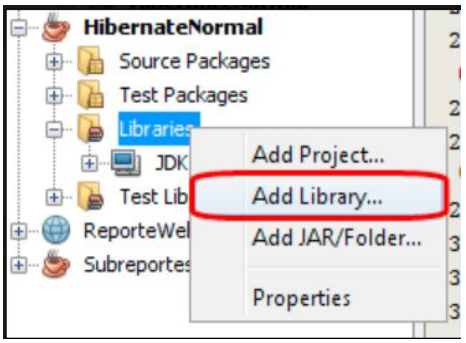


Agregamos a la biblioteca los siguientes archivos, y del directorio de jars requeridos (ver las versiones):

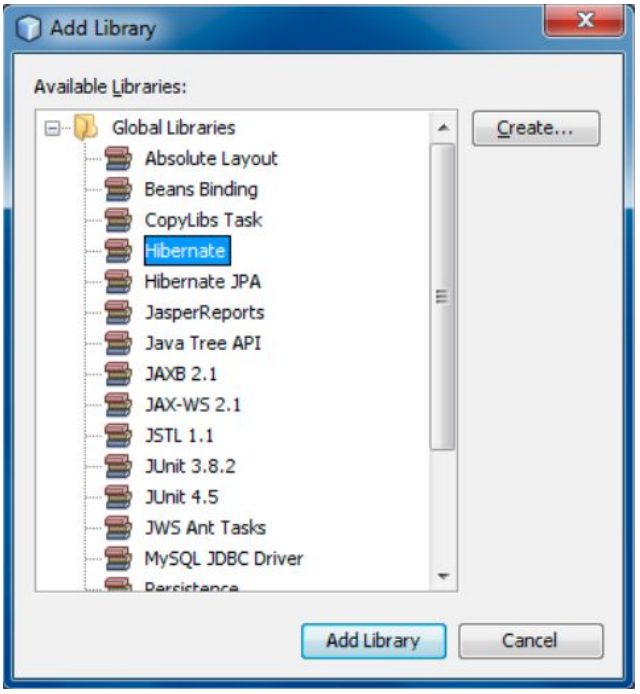


Presionamos el botón "`OK`" y nuestra biblioteca ya estará actualizada.

Ahora creamos un nuevo proyecto de NetBeans, agregamos la biblioteca de "`Hibernate`", que creamos hace unos momentos, o agregamos los archivos a un proyecto ya existente. Hacemos clic derecho en el nodo "`Libraries`" del proyecto. En el menú contextual que se abre seleccionamos la opción "`Add Library...`":



En la ventana que se abre seleccionamos la biblioteca "`Hibernate`":



Presionamos el botón "`Add Library`" para que la biblioteca se agregue a nuestro proyecto. Aprovechamos también para agregar el conector de MySQL.

Nota: debemos asegurarnos que todas nuestras entidades sean JavaBean.
Para este ejercicio vamos a usar la siguiente entidad: `Contacto`.

La clase "`Contacto`" usa la convención de nombres JavaBeans para los `getters` y `setters` y visibilidad privada para los atributos. Este también es un diseño recomendado pero no obligatorio. Hibernate puede acceder a los campos o atributos de las entidades directamente.

El constructor sin argumentos si es obligatorio ya que Hibernate creará instancias de esta clase usando reflexión cuando recupere las entidades de la base de datos. (más adelante veremos que es reflexión)
Este constructor puede ser privado (si es que no quieren permitir que alguien más lo utilice), pero usualmente el nivel de acceso más restrictivo que usaremos es el de paquete (el default), ya que esto hace más eficiente la creación de los objetos.

La propiedad "`id`" mantendrá, como dije antes, un valor único que identificará a cada una de las instancias de "`Contacto`".

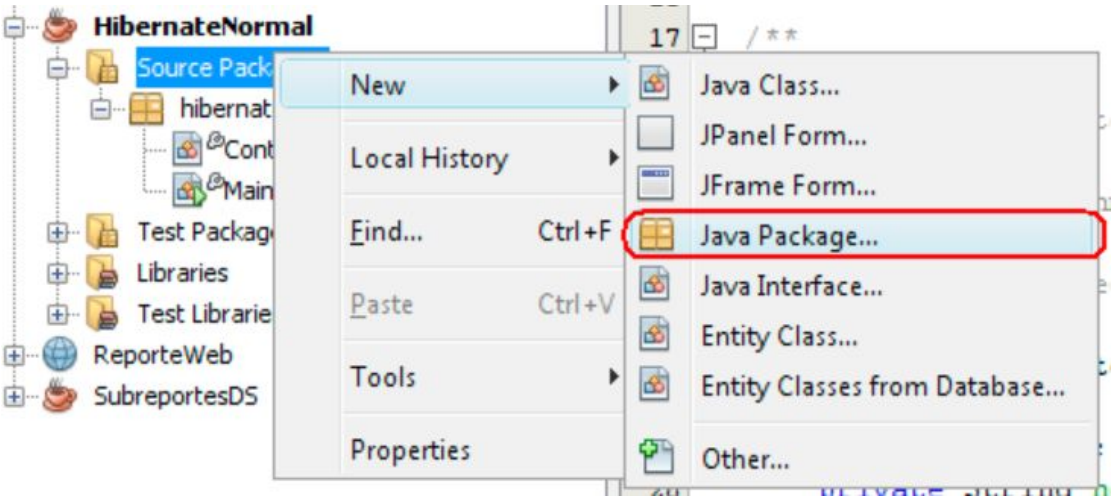
Todas las clases de entidades persistentes deben tener una propiedad que sirva como identificador si queremos usar el conjunto completo de funcionalidades que nos ofrece Hibernate (Por lo tanto verificaremos una vez más que todas nuestras clases de entidades tengan un atributo id). De todas formas, la mayoría de las aplicaciones necesitan poder distinguir objetos de la misma clase mediante algún tipo de identificador.
Usualmente no manipulamos directamente este identificador (dejamos que sea la base de datos quien lo genere, cuando la entidad sea guardada, y Hibernate quien lo asigne al objeto), por lo tanto el `setter` del "`id`" es privado (fíjense cómo lo he puesto en la clase "`Contacto`").

Hibernate puede acceder a los campos `private` directamente, así que no debemos preocuparnos por el hecho de que el identificador no pudiera ser establecido.

El siguiente paso es crear el archivo de mapeo. Si recuerdan, dijimos que Hibernate es una herramienta de mapeos Objeto/Relacional, o sea que mapea los atributos de los objetos con las columnas de una tabla de una base de datos relacional. También dijimos que hay dos formas de hacerlo: mediante un archivo de mapeo en XML y mediante anotaciones y que en esta ocasión veríamos solo como usar los archivos de mapeo.

Primero entonces, crearemos un nuevo paquete que contendrá los archivos de mapeo, no es obligatorio tener los archivos de mapeo en un paquete separado ya que más adelante indicaremos donde se encuentra cada uno de los archivos, pero nos ayudará a mantener un poco de orden en nuestro proyecto.

Hacemos clic derecho sobre el nodo "`Source Packages`" del proyecto para mostrar un menú contextual. De ese menú seleccionamos la opción "`New -> Java Package..`":



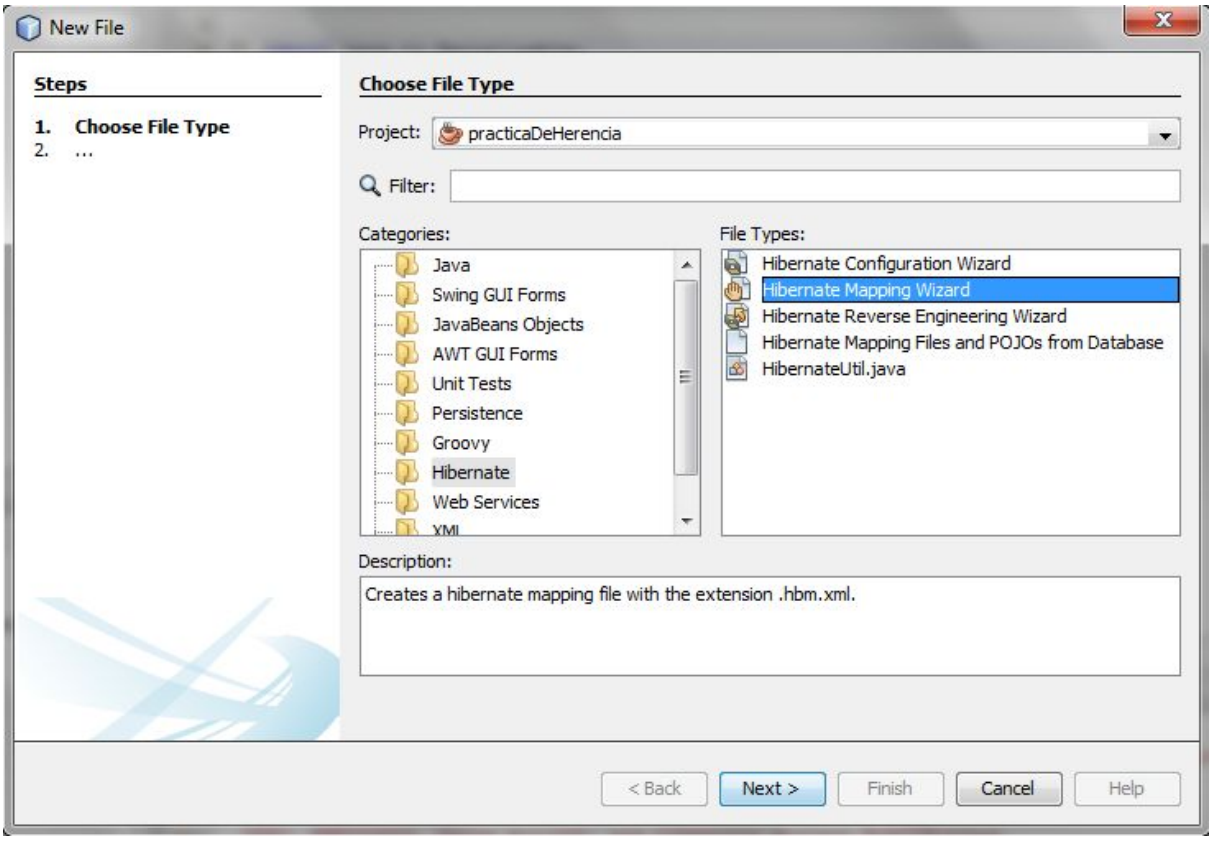
Nombramos a este paquete como "`resources.hbm`" y presionamos el botón "`Finish`" para agregar el nuevo paquete.

Ahora creamos un nuevo archivo XML que contendrá el mapeo de la clase "`Contacto`" por ejemplo, con la tabla "`contactos`" de la base de datos.

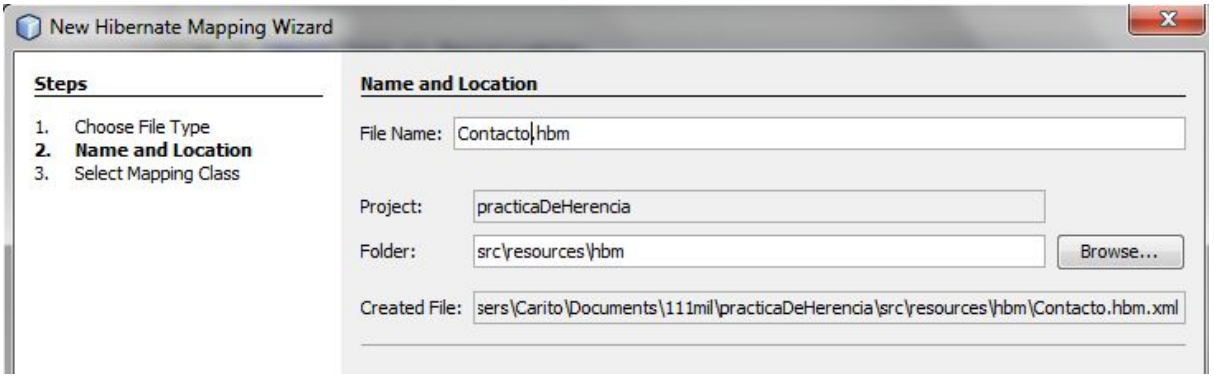
Por convención la extensión de estos archivos es "`.hbm.xml`" y tiene el mismo nombre de la entidad que está mapeando (aunque eso tampoco es necesario ya que podemos mapear más de una clase o entidad dentro del mismo archivo).

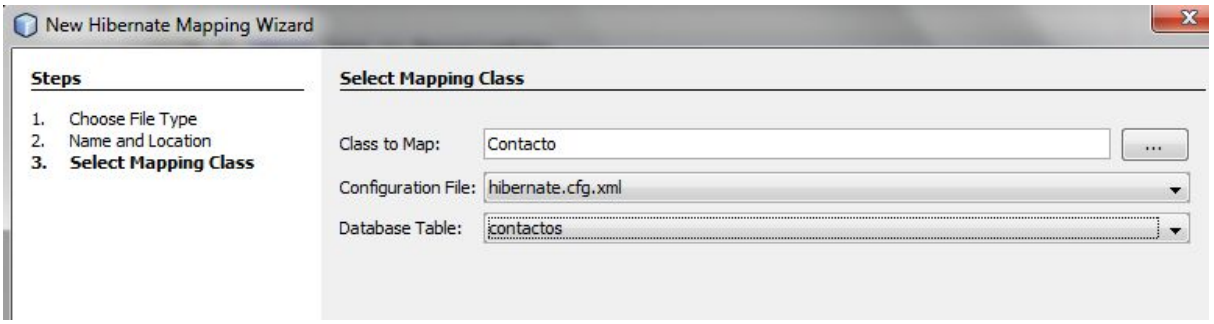
Creando .hbm.xml:

Hacemos clic derecho sobre el paquete que acabamos de crear. En el menú contextual que se abre seleccionamos la opción "`New -> Other`". Si no se muestra está opción seleccionamos "`Other...`" y en la ventana que se abre seleccionamos "`Hibernate ->`":



Le damos al archivo el nombre "`Contacto.hbm`" (el asistente se encargará de agregar de forma automática el "`.xml`" al final):





Presionamos el botón "`Finish`" para que se nos muestre en el editor el nuevo archivo XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Contacto" table="contactos"/>
</hibernate-mapping>
```

El elemento "`<hibernate-mapping>`" es el nodo raíz del documento y, por lo tanto, el resto de los elementos irán entre estas dos etiquetas.

El elemento con el que indicamos qué clase es la que estamos mapeando es el elemento "`<class>`" este elemento tiene los atributos "`name`" y "`table`" que nos permiten indicar el nombre completo de la clase y la tabla con la que será mapeada, respectivamente.

Con esto indicamos que las entidades de la clase "`Contacto`" serán almacenadas en la tabla "`contactos`". Ahora debemos indicar cuál de los elementos de la clase entidad es el identificador.
Este identificador será mapeado con la llave primaria de la tabla, además como nosotros no manejaremos el valor del identificador, le indicamos a Hibernate cómo queremos que este valor sea generado (la estrategia de generación). Para esto usamos el elemento "`<id>`", indicando el nombre del atributo de la clase entidad que representa el identificador (que en este caso también se llama "`id`"). Opcionalmente en este elemento (como en el resto de los elementos de mapeo de propiedades) podemos indicar con qué columna queremos que se mapee usando el atributo "`column`":

```
<hibernate-mapping>
   <class name="Contacto" table="CONTACTOS">
      <id name="id" column="ID">
         <generator class="identity" />
      </id>
   </class>
</hibernate-mapping>
```

El elemento "`<generator>`" es el que nos permite indicar la estrategia de generación del identificador usando su atributo "`class`". Existen varias estrategias que están explicadas en esta página, pero las que usaremos más frecuentemente son "`identity`" (con la que Hibernate se encarga de generar el query necesario para que el nuevo identificador sea igual a el último identificador + 1) y "`native`" (con el que se usa la estrategia por default del manejador que estemos utilizando (dialecto)).

Para terminar con el mapeo incluimos las declaraciones para el resto de las propiedades persistentes (las que queremos que sean almacenadas) de la clase entidad ("`nombre`", "`email`", y "`telefono`") usando el elemento "`<property>`" en el cual indicamos el nombre de la propiedad como aparece en la clase entidad y, opcionalmente, el tipo de la propiedad y la columna con la que será mapeada usando los atributos "`type`" y "`column`" respectivamente:

```
<hibernate-mapping>
   <class name="Contacto" table="contactos">
      <id name="id" column="ID">
         <generator class="identity" />
      </id>
      <property name="nombre" type="string" column="nombre" />
      <property name="email" />
      <property name="telefono" />
   </class>
</hibernate-mapping>
```

El archivo "`Contacto.hbm.xml`" final debe verse así:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="hibernatenormal.Contacto" table="CONTACTOS">
        <id name="id" column="ID">
            <generator class="identity" />
        </id>
        <property name="nombre" type="string" column="NOMBRE" />
        <property name="email" />
        <property name="telefono" />
    </class>
</hibernate-mapping>
```

¿Qué ocurre si no indicamos el tipo de la propiedad? En ese caso Hibernate utiliza nuevamente reflexión en nuestra clase para determinar el tipo del atributo y así elegir el tipo más adecuado para la columna de la tabla (en caso de que dejemos que Hibernate genere las tablas).

¿Qué ocurre si no indicamos el nombre de la columna con la que mapea una propiedad? Esto depende de si Hibernate está generando las tablas de la base de datos o las hemos generado nosotros mismos. Si es Hibernate quien está generando las tablas no hay problema, simplemente dará a la columna correspondiente el mismos nombre de la propiedad. Por ejemplo, para la propiedad "`email`" creará una columna "`email`". Sin embargo, si nosotros hemos creado las tablas hay un riesgo potencial de un error. Hibernate buscará en las consultas una columna con el mismo nombre de la propiedad, pero si hemos usado un nombre distinto, por ejemplo en vez de "`email`" hemos llamado a la columna "`E_MAIL`" ocurrirá una excepción indicando que no se ha encontrado la columna "`email`", así que en ese caso debemos tener cuidado de indicar el nombre de las columnas como están en la base de datos.

Siguiente paso: debemos configurar Hibernate (hibernate.cfg.xml)

Hibernate está en la capa de nuestra aplicación que se conecta a la base de datos (la capa de persistencia), así que necesita información de la conexión.
La conexión se hace a través de un pool de conexiones JDBC que también debemos configurar. La distribución de Hibernate contiene muchos pools de conexiones JDBC Open Source, como por ejemplo `C3P0`, pero en esta ocasión usaremos el pool de conexiones que Hibernate trae integrado.

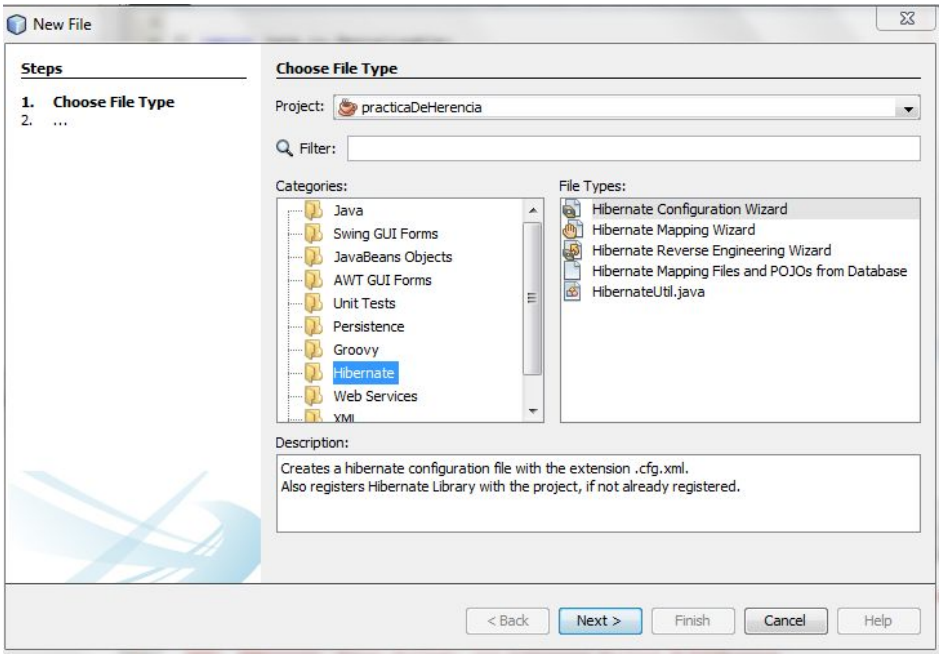
La configuración de Hibernate puede hacerse en tres lugares:

- Un archivo de propiedades llamado "`hibernate.properties`".
- Un archivo XML llamado "`hibernate.cfg.xml`".
- En código dentro de la misma aplicación.

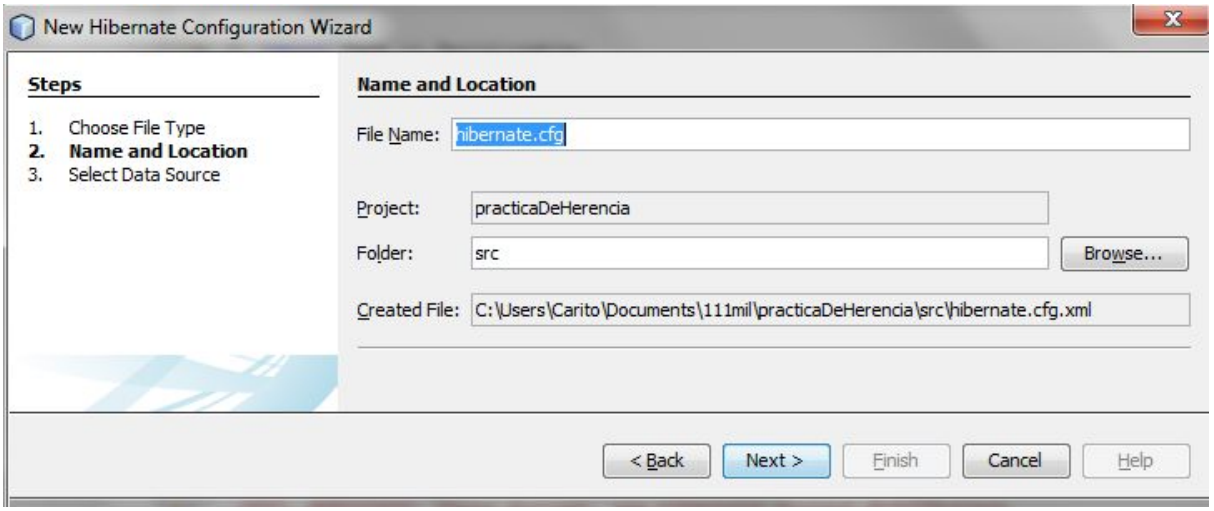
En realidad los archivos pueden tener cualquier nombre, pero Hibernate buscará por default los archivos con los nombres que he mencionado, en una ubicación predeterminada (la raíz del classpath de la aplicación). Si queremos usar archivos con otros nombres deberemos especificarlo en el código.

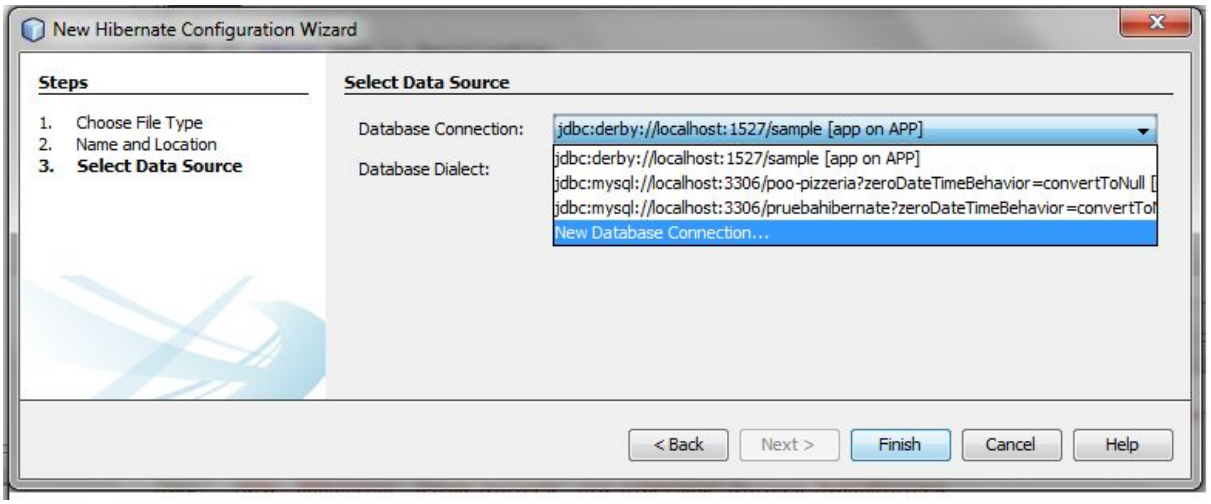
Nosotros vamos a usar el archivo XML ya que, entre otras cosas, los IDEs nos ayudan a su creación.

Crearemos este archivo en nuestro proyecto. Hacemos clic derecho en el paquete donde lo vamos a agregar: "`resources`" del proyecto (si no existe lo creamos) En el menú contextual que se abre seleccionamos:

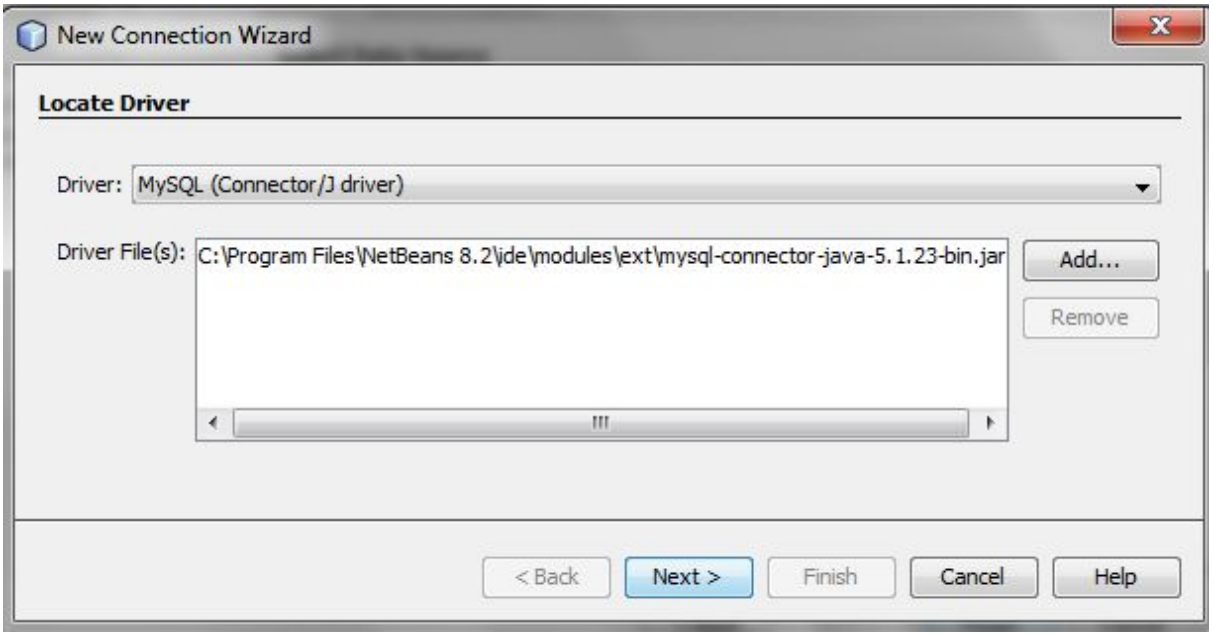


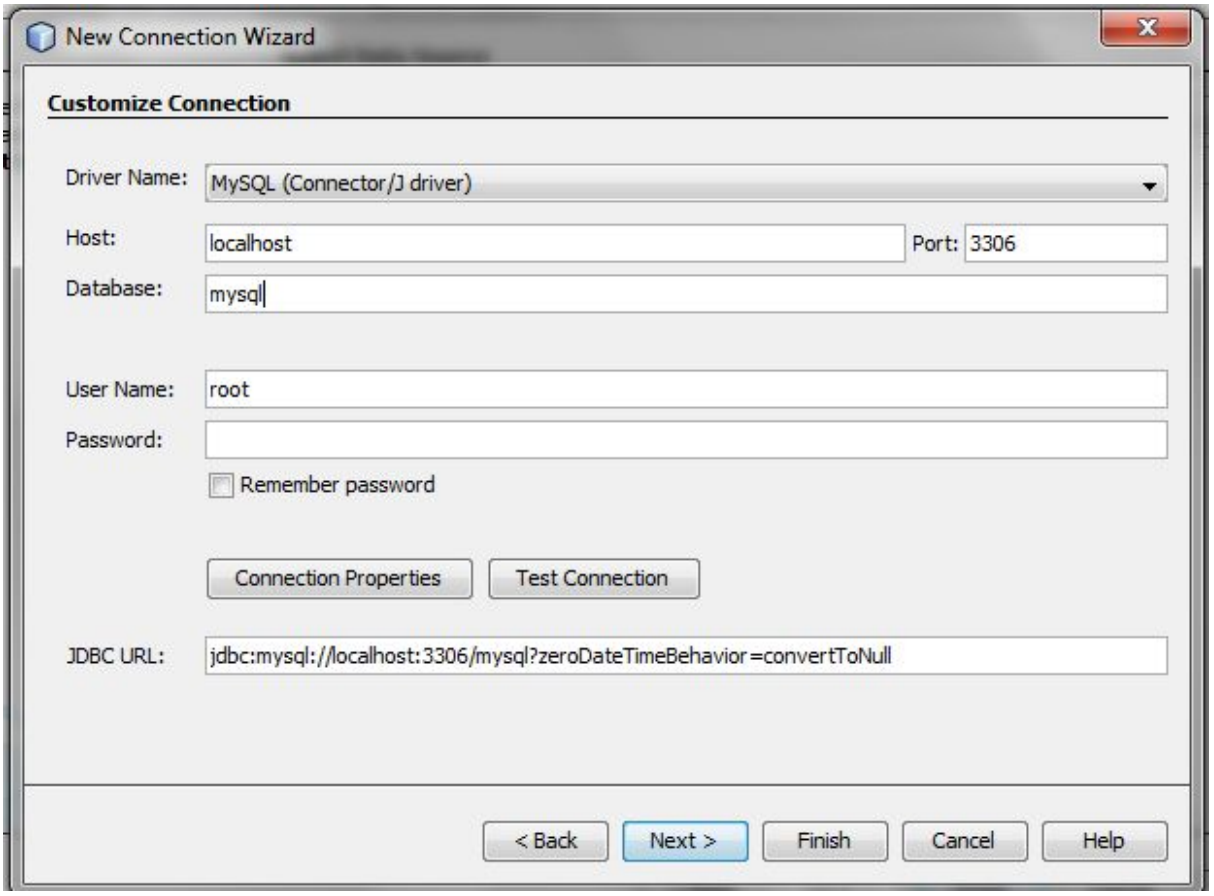
Nombramos al archivo "`hibernate.cfg`", el IDE se encargará de colocarle la extensión "`.xml`".





Le damos Next y seleccionamos la opción <u>New Database Connection</u>,





Configuramos en nombre del host, el puerto y el nombre de la BD. Además agregamos el user name y el password, y probamos la conexión a la base de datos con el "Test Connection", para verificar que todo está bien.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourc

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/pruebahibernate</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
  </session-factory>
</hibernate-configuration>
```

Como podemos ver, el elemento raíz del archivo de configuración es "`<hibernate-configuration>`" y, por lo tanto, todos los elementos los colocaremos entre estas dos etiquetas.

Lo primero que debemos hacer es configurar un "`session-factory`", que básicamente es lo que le dice a Hibernate cómo conectarse y manejar la conexión a la base de datos. Podemos tener más de un "`session-factory`" en el archivo de configuración (por si quisiéramos conectarnos a más de una base de datos), pero por lo regular solo ponemos uno.

En este archivo se configuran los parámetros básicos y típicos para una conexión (la URL, nombre de usuario, contraseña, driver, etc.). Cada uno de estos parámetros se configura dentro de una etiqueta "`<property>`" (al igual que casi todos los elementos del archivo de configuración). Como dije antes, usaré una base de datos MySQL para este ejemplo.

Después, configuramos el pool de conexiones de Hibernate. En este caso como es un ejemplo muy simple, solo nos interesa tener una conexión en el pool, por lo que colocamos la propiedad "`connection.pool_size`" con un valor de "1":

```
<property name="connection.pool_size">1</property>
```

El siguiente paso es muy importante. Debemos indicar el "dialecto" que usará Hibernate para comunicarse con la base de datos. Este dialecto es la variante de `SQL` que usa la base de datos para ejecutar queries. Indicamos el dialecto con el fully qualified class name, o el nombre completo de la clase incluyendo el paquete. En el caso de MySQL 5 usamos "`org.hibernate.dialect.MySQLDialect`". En esta página pueden encontrar una lista más o menos completa de los dialectos soportados por Hibernate, pero siempre es mejor revisar la documentación de la versión que estén usando para estar seguros:

```
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

Otras dos propiedades importantes que podemos configurar son: "`show_sql`" que indica si queremos que las consultas SQL generadas sean mostradas en el `stdout` (normalmente la consola), y "`hbm2ddl.auto`", que indica si queremos que se genere automáticamente el esquema de la base de datos (las tablas). "`show_sql`" puede tomar valores de "`true`" o "`false`", yo lo colocaré en "`true`" (lo que puede ser bueno mientras estamos en etapas de desarrollo o pruebas, pero querrán cambiar su valor cuando su aplicación pase a producción). Por otro lado "`hbm2ddl.auto`" puede tomar los valores, según la documentación oficial (falta "`none`"), de "`validate`", "`update`", "`create`", y "`create-drop`" (aunque no todos los valores funcionan para todas las bases de datos). Ejemplo:

```
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create-drop</property>
```

Con el valor "`create-drop`" hacemos que cada vez que se ejecute la aplicación Hibernate elimine las tablas de la base de datos y las vuelva a crear. Para terminar con este archivo de configuración, debemos indicar dónde se encuentra cada uno de los archivos de mapeo que hemos creado, usando el elemento "`<mapping>`". En nuestro caso solo hemos creado un archivo de mapeo, pero debemos colocar un elemento "`<mapping>`" por cada uno de los archivos que hayamos creado, por ejemplo:

```
<mapping resource="mapeos/Contacto.hbm.xml"/>
```

En el elemento "`resource`" debemos colocar la ubicación de los archivos de mapeo dentro de la estructura de paquetes de la aplicación. El archivo de configuración "`hibernate.cfg.xml`" final debe verse así:

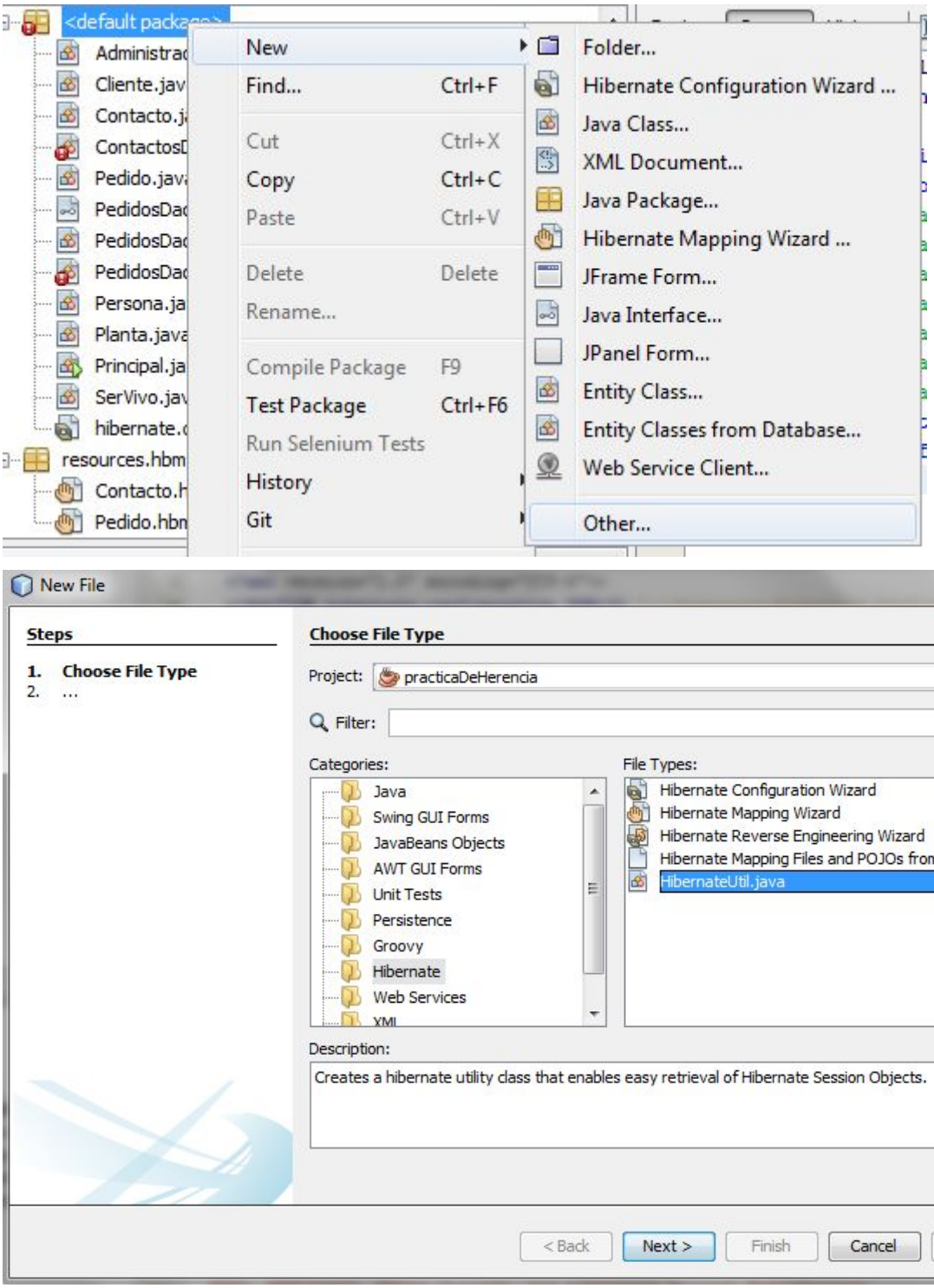
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>

   <!-- parametros para la conexion a la base de datos -->
   <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
   <property
name="connection.url">jdbc:mysql://localhost/pruebahibernate</property
>
   <property name="connection.username">usuario</property>
   <property name="connection.password">password</property>

   <!-- Configuracion del pool interno -->
   <property name="connection.pool_size">1</property>

   <!-- Dialecto de la base de datos -->
   <property
name="dialect">org.hibernate.dialect.MySQL5Dialect</property>

   <!-- Otras propiedades importantes -->
   <property name="show_sql">true</property>
   <property name="hbm2ddl.auto">create-drop</property>

   <!-- Archivos de mapeo -->
   <mapping resource="resouces/hbm/Contacto.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

Creando `HibernateUtil.java`

Ahora veremos el código necesario para guardar y recuperar objetos "`Contacto`" de la base de datos.

Lo primero que haremos es crear una clase ayudante o de utilidad llamada "`HibernateUtil`", que se hará cargo de inicializar y hacer el acceso al "`org.hibernate.SessionFactory`" (el objeto encargado de gestionar las sesiones de conexión a la base de datos que configuramos en el archivo "`hibernate.cfg.xml`") más conveniente.

En nuestro paquete seleccionamos agregar New -> Other -> Hibernate .



Seleccionamos: HibernateUtil.java

Dentro de esta clase declaramos un atributo `static` de tipo "`SessionFactory`", así nos aseguraremos de que solo existe una instancia en la aplicación. Además lo declararemos como `final` para que la referencia no pueda ser cambiada después de que la hayamos asignado.

```
private static final SessionFactory sessionFactory;
```

Después usamos un bloque de inicialización estático para inicializar esta variable en el momento en el que la clase sea cargada en la JVM.

Para realizar esta inicialización lo primero que se necesita es una instancia de la clase "`org.hibernate.cfg.Configuration`" que permite a la aplicación especificar las propiedades y documentos de mapeo que se usarán (es aquí donde indicamos todo si no queremos usar un archivo XML o de propiedades). Si usamos el método "`configure()`" que no recibe parámetros entonces Hibernate busca el archivo "`hibernate.cfg.xml`" que creamos anteriormente. Una vez que tenemos este objeto, entonces podemos inicializar la instancia de "`SessionFactory`" con su método "`buildSessionFactory()`". Además como este proceso puede lanzar "`org.hibernate.HibernateException`" (que extiende de "`RuntimeException`") la cachamos y lanzamos como un "`ExceptionInInitializarError`" (que es lo único que puede lanzarse desde un bloque de inicialización). El bloque de inicialización queda de la siguiente forma:

```
static
{
   try
   {
      sessionFactory = new Configuration().configure().buildSessionFactory();
   } catch (HibernateException he)
   {
      System.err.println("Ocurrió un error en la inicialización de la
SessionFactory: " + he);
      throw new ExceptionInInitializerError(he);
   }
}
```

Finalmente creamos un método `static` llamado "`getSessionFactory()`" para recuperar la instancia de la "`SessionFactory`":

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

La clase "`HibernateUtil`" queda de la siguiente forma:

```
import org.hibernate.HibernateException;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil
{
   private static final SessionFactory sessionFactory;
```

```
    static
    {
        try
        {
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (HibernateException he)
        {
            System.err.println("Ocurrió un error en la inicialización de la SessionFactory: " + he);
            throw new ExceptionInInitializerError(he);
        }
    }

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

Bien ha llegado el momento, ahora escribiremos una clase DAO (no seguiremos el patrón al pie de la letra, es solo para mostrar cómo trabajar con Hibernate) que nos permitirá realizar operaciones de base de datos.

Pero antes vamos a ver que es un patrón de diseño:

QUÉ ES UN PATRÓN DE DISEÑO?

Un patrón de diseño es una solución probada que resuelve un tipo específico de problema en el desarrollo de software referente al diseño.

Existen una infinidad de patrones de diseño los mismos que se dividen en categorías por ejemplo: de creación, estructurales, de comportamiento, interacción etc.

Cada uno se especializa en resolver un problema específico.

POR QUÉ UTILIZAR UN PATRÓN DE DISEÑO?

Ahora, cuales son las ventajas?, bueno son algunas, entre las principales es que permiten tener el código bien organizado, legible y mantenible, además te permite reutilizar código y aumenta la escalabilidad en tu proyecto.

En sí proporcionan una terminología estándar y un conjunto de buenas prácticas en cuanto a la solución en problemas de desarrollo de software.

EL PATRÓN MODEL VIEW CONTROLLER O MVC

En español Modelo Vista Controlador, este patrón permite separar una aplicación en 3 capas, una forma de organizar y de hacer escalable un proyecto, a continuación una breve descripción de cada capa.

Modelo: Esta capa representa todo lo que tiene que ver con el acceso a datos: guardar, actualizar, obtener datos, además todo el código de la lógica del negocio, básicamente son las clases Java y parte de la lógica de negocio.

Vista: La vista tiene que ver con la presentación de datos del modelo y lo que ve el usuario, por lo general una vista es la representación visual de un modelo (POJO o clase java).

Por ejemplo el modelo usuario que es una clase en Java y que tiene como propiedades, nombre y apellido debe pertenecer a una vista en la que el usuario vea esas propiedades.

Controlador: El controlador es el encargado de conectar el modelo con las vistas, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encarga de direccionar al modelo la petición respectiva.

Por ejemplo el usuario quiere ver los clientes con apellido Álvarez, la petición va al controlador y el se encarga de utilizar el modelo adecuado y devolver ese modelo a la vista.

QUE GANÓ UTILIZANDO ESTE PATRÓN?

Lo importante de este patrón es que permite dividir en partes, que de alguna manera son independientes, con lo que si por ejemplo hago algún cambio el modelo no afectaría a la vista o si hay algún cambio sería mínimo.

EL PATRÓN DATA ACCES OBJECT (DAO)

El problema que viene a resolver este patrón es el acceso a los datos, que básicamente tiene que ver con la gestión de diversas fuentes de datos y además abstrae la forma de acceder a ellos.

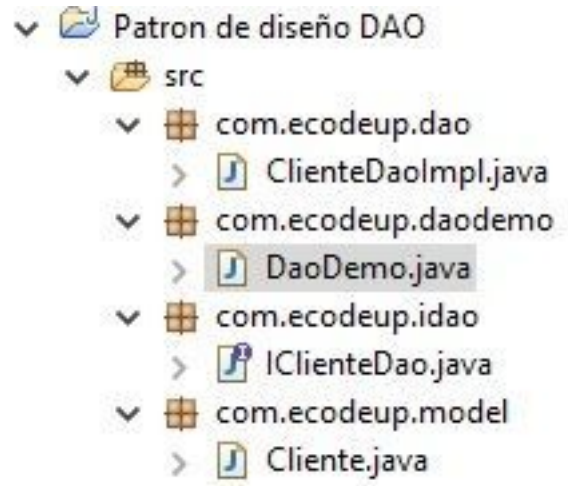
Imagínate que tienes un sistema montado en producción con una base de datos MySQL y de pronto lo debes cambiar a PostgreSQL o a cualquier otro motor de base de datos.

Eso puede ser un verdadero problema.

Y precisamente esto lo que soluciona este patrón, tener una aplicación que no esté ligada al acceso a datos, que si por ejemplo la parte de la vista pide encontrar los clientes con compras mensuales mayores $ 200, el *DAO* se encargue de traer esos datos independientemente si está en un archivo o en una base de datos.

La capa DAO contiene todos los métodos CRUD (create, read, update, delete), por lo general se tiene un DAO para cada tabla en la base de datos.

La implementación se la realiza de la siguiente manera:



Se crea una clase *Cliente.java* únicamente con sus constructores, getters y setters.

```
1  public class Cliente {
2          private int id;
3          private String nombre;
4          private String apellido;
5
6
7          public Cliente() {
8                  super();
9          }
10         public Cliente(int id, String nombre, String apellido) {
11                 super();
12                 this.id = id;
13                 this.nombre = nombre;
14                 this.apellido = apellido;
15         }
16         public int getId() {
17                 return id;
18         }
19         public void setId(int id) {
20                 this.id = id;
21         }
22
23         public String getNombre() {
24                 return nombre;
25         }
26         public void setNombre(String nombre) {
27                 this.nombre = nombre;
28         }
29
30         public String getApellido() {
31                 return apellido;
32         }
```

```
33          public void setApellido(String apellido) {
34                  this.apellido = apellido;
35          }
36
37          @Override
38          public String toString() {
39                  return this.getNombre()+" "+this.getApellido();
40          }
41  }
42
43
```

Se crea el acceso a los datos a través de una interface ***IClienteDao.java***, aquí se declara todos los métodos para acceder a los datos.

```
1  import java.util.List;
2  import com.ecodeup.model.Cliente;
3  public interface IClienteDao {
4          //declaración de métodos para acceder a la base de datos
5          public List<Cliente> obtenerClientes();
6          public Cliente obtenerCliente(int id);
7          public void actualizarCliente(Cliente cliente);
8          public void eliminarCliente(Cliente cliente);
9  }
```

Se implementa en la clase ***ClienteDaoImpl.java*** haciendo un *implements* de la interface ***IClienteDao.java***, lo que se hace aquí, no es más que implementar cada método de la interface.

```java
package com.ecodeup.dao;
import java.util.ArrayList;
import java.util.List;

import com.ecodeup.idao.*;
import com.ecodeup.model.Cliente;

public class ClienteDaoImpl implements IClienteDao {
	//lista de tipo cliente
	List<Cliente> clientes;

	//inicializar los objetos cliente y añadirlos a la lista
	public ClienteDaoImpl() {
		clientes = new ArrayList<>();
		Cliente cliente1 = new Cliente(0,"Javier", "Molina");
		Cliente cliente2 = new Cliente(1,"Lillian","Álvarez");
		clientes.add(cliente1);
		clientes.add(cliente2);
	}
	//obtener todos los clientes
	@Override
	public List<Cliente> obtenerClientes() {
		return clientes;
	}
	//obtener un cliente por el id
	@Override
	public Cliente obtenerCliente(int id) {
		return clientes.get(id);
	}
	//actualizar un cliente
	@Override
	public void actualizarCliente(Cliente cliente) {
		clientes.get(cliente.getId()).setNombre(cliente.getNombre());

clientes.get(cliente.getId()).setApellido(cliente.getApellido());
		System.out.println("Cliente con id: "+cliente.getId()+"
actualizado satisfactoriamente");
	}

	//eliminar un cliente por el id
	@Override
	public void eliminarCliente(Cliente cliente) {
		clientes.remove(cliente.getId());
		System.out.println("Cliente con id: "+cliente.getId()+" elimnado
satisfactoriamente");
	}
}
```

Por último se prueba el patrón DAO a través de la clase DaoDemo.java

```java
package com.ecodeup.daodemo;

import com.ecodeup.dao.ClienteDaoImpl;
import com.ecodeup.idao.IClienteDao;
import com.ecodeup.model.Cliente;

public class DaoDemo {

	public static void main(String[] args) {
		// objeto para manipular el dao
		IClienteDao clienteDao = new ClienteDaoImpl();

		// imprimir los clientes
		clienteDao.obtenerClientes().forEach(System.out::println);

		// obtner un cliente
		Cliente cliente = clienteDao.obtenerCliente(0);
		cliente.setApellido("Pardo");
		//actualizar cliente
		clienteDao.actualizarCliente(cliente);

		// imprimir los clientes
		System.out.println("*****");
		clienteDao.obtenerClientes().forEach(System.out::println);
	}
}
```

Preguntas??

Después de haber visto el patrón de diseño DAO, retomamos nuestro ejemplo para conectar a la base de datos.

Creamos una clase llamada "`ContactosDAO`" y agregamos dos atributos, uno llamado "`sesion`" de tipo "`org.hibernate.Session`", y otro llamado "`tx`" de tipo "`org.hibernate.Transaction`".

```java
private Session sesion;
private Transaction tx;
```

Estos atributos nos servirán para mantener la referencia a la sesión a base de datos, y a la transacción actual, respectivamente. Ahora agregaremos dos métodos de utilidad. El primero nos ayudará a iniciar una sesión y una transacción en la base de datos. Llamaremos a este método "`iniciaOperacion`", y la implementación es la siguiente:

```
private void iniciaOperacion() throws HibernateException
{
   sesion = HibernateUtil.getSessionFactory().openSession();
   tx = sesion.beginTransaction();
}
```

En el método anterior obtenemos una referencia a "`SessionFactory`" usando nuestra clase de utilidad "`HibernateUtil`". Una vez que tenemos la "`SessionFactory`" creamos una conexión a la base de datos e iniciamos una nueva sesión con el método "`openSession()`". Una vez teniendo la sesión iniciamos una nueva transacción y obtenemos una referencia a ella con "`beginTransaction()`".

Ahora el segundo método de utilidad (llamado "`manejaExcepcion`") nos ayudará a manejar las cosas en caso de que ocurra una excepción. Si esto pasa queremos que la transacción que estamos ejecutando se deshaga y se relance la excepción (o podríamos lanzar una propia). Por lo que el método queda así:

```
private void manejaExcepcion(HibernateException he) throws
HibernateException
{
   tx.rollback();
   throw new HibernateException("Ocurrió un error en la capa de acceso a
datos", he);
}
```

Ahora crearemos los métodos que nos permitirán realizar las tareas de persistencia de una entidad "`Contacto`", conocidas en lenguaje de base de datos como CRUD: guardarla, actualizarla, eliminarla, buscar un entidad "`Contacto`" y obtener todas los contactos que existen en la base de datos, así que comencemos.

Afortunadamente Hibernate hace que esto sea fácil ya que proporciona métodos para cada una de estas tareas. Primero veamos como guardar un objeto "`Contacto`". Para esto Hibernate proporciona el método "`save`" en el objeto de tipo "`org.hibernate.Session`", que se encarga de generar el "`INSERT`" apropiado para la entidad que estamos tratando de guardar. El método "`guardaContacto`" queda de la siguiente forma:

```
public long guardaContacto(Contacto contacto)
{
   long id = 0;

   try
   {
      iniciaOperacion();
      id = (Long)sesion.save(contacto);
      tx.commit();
   }catch(HibernateException he)
   {
      manejaExcepcion(he);
      throw he;
   }finally
   {
      sesion.close();
   }
   return id;
}
```

Regresamos el "`id`" generado al guardar el "`Contacto`" solo por si queremos usarlo más adelante en el proceso (como lo haremos nosotros), o si queremos mostrarle al usuario, por alguna razón, el identificador del "`Contacto`".

Ahora veremos cómo actualizar un "`Contacto`". Para eso usamos el método "`update`" del objeto "`sesion`" en nuestro método "`actualizaContacto`":

```
public void actualizaContacto(Contacto contacto) throws HibernateException
{
   try
   {
      iniciaOperacion();
      sesion.update(contacto);
      tx.commit();
   }catch (HibernateException he)
   {
      manejaExcepcion(he);
      throw he;
   }finally
   {
      sesion.close();
   }
}
```

Como podemos ver, el método para actualizar es muy similar al método para guardar la entidad. Lo mismo ocurre con el método para eliminarla, "`eliminaContacto`":

```
public void eliminaContacto(Contacto contacto) throws HibernateException
{
   try
   {
      iniciaOperacion();
      sesion.delete(contacto);
      tx.commit();
   } catch (HibernateException he)
   {
      manejaExcepcion(he);
      throw he;
   }finally
   {
      sesion.close();
   }
}
```

Ahora veremos unos métodos un poco más interesantes.

Cuando queremos buscar una entidad podemos usar varios criterios. La forma más fácil es buscar una entidad particular usando su "`id`". La clase "`org.hibernate.Session`" proporciona dos métodos para esto: "`load`" y "`get`". Los dos hacen prácticamente lo mismo: en base al identificador y tipo de la entidad recuperan la entidad indicada, con la diferencia de que "`load`" lanza una excepción en caso de que la entidad indicada no sea encontrada en la base de datos, mientras que "`get`" simplemente regresa "`null`". Pueden usar el que prefieran, en lo personal me gusta más "`get`", así que lo usaré para el método "`obtenContacto`":

```
public Contacto obtenContacto(long idContacto) throws HibernateException
{
   Contacto contacto = null;

   try
   {
      iniciaOperacion();
      contacto = (Contacto) sesion.get(Contacto.class, idContacto);
   } finally
   {
      sesion.close();
   }
   return contacto;
}
```

Finalmente veremos el método "`obtenListaContactos`" que recupera todos los `Contacto`s que estén guardados en la base de datos. Como en este caso

regresaremos una lista de elementos deberemos crear una consulta. Cuando tenemos que crear una consulta con JDBC lo hacemos en SQL, sin embargo con Hibernate tenemos varias opciones:

- Usar una query en SQL nativo.
- Crear una query en HQL (Hibernate Query Language).
- Crear una query usando Criteria que es un API para crear queries de una forma más "orientada a objetos".

Critera es, a mi parecer, la forma más fácil de crear las queries. Sin embargo, también en mi opinión, HQL es más poderosa y tenemos más control sobre lo que está ocurriendo, así que será esta forma la que usaremos.

Creando la consulta en HQL Hibernate la transformará al SQL apropiado para la base de datos que estemos usando. La consulta en realidad es muy simple: indicamos que queremos obtener datos (generar un "SELECT"), en este caso la clausula "SELECT" no es necesaria (aunque si existe) porque vamos a regresar todos los datos del objeto (podemos indicar solo unos cuantos atributos, eso es llamado proyección, pero se maneja de una forma distinta). Lo que si debemos indicar es de cuál clase queremos recuperar las instancias, por lo que necesitamos la clausula "FROM" y el nombre de la clase (recuerden que en base al nombre de la clase Hibernate sabrá en cuál tabla están almacenadas las entidades). Espero que esta explicación se haya entendido, se que es un poco enredado, pero quedará más claro con el ejemplo:

```
public List<Contacto> obtenListaContactos() throws HibernateException
{
   List<Contacto> listaContactos = null;

   try
   {
      iniciaOperacion();
      listaContactos = sesion.createQuery("from Contacto").list();
   }finally
   {
      sesion.close();
   }

   return listaContactos;
}
```

Eso es todo, nuestra consulta para recuperar todos los Contactos que tenemos en la base de datos solo debemos usar la clausula "`FROM Contacto`". Si quieren una referencia más amplia de HQL pueden encontrarla en el tutorial sobre HQL.

Bien, eso es todo. La clase "`ContactosDAO`" queda de la siguiente forma (omitiendo los `imports`):

Ahora, y para terminar el ejemplo, haremos uso de "`ContactosDAO`" para crear y recuperar algunas instancias de "`Contacto`". . La clase "`Main`" queda de la siguiente forma: