HIBERNATE

El presente documento tiene por objetivo guiar a implemente el CRUD de dos tablas relacionadas en MySQL utilizando Hibernate con anotaciones en una aplicación de consola java.

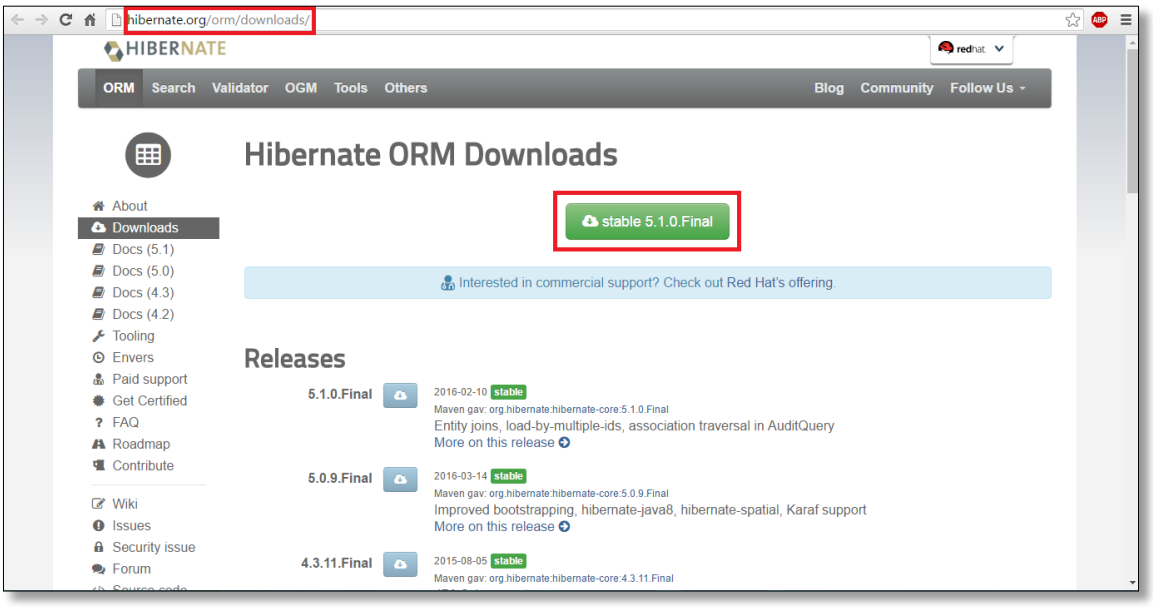
Hibernate es una herramienta de mapeo objeto/relacional. Además no solo se encarga del mapeo de clases Java a tablas de la base de datos (y de regreso), sino que también maneja los queries y recuperación de datos, lo que puede reducir de forma significativa el tiempo de desarrollo que de otra forma gastaríamos manejando los datos de forma manual con SQL y JDBC, encargándose de esta forma de alrededor del 95% de las tareas comunes relacionadas con la persistencia de datos, manejando todos los problemas relativos con la base de datos particular con la que estemos trabajando, de forma transparente para nosotros como desarrolladores. Entonces, si cambiamos el manejador de base de datos no será necesario que modifiquemos todo el SQL que ya teníamos para adaptarse al SQL que maneja la nueva base de datos. Solo será necesario modificar una línea en un archivo de configuración de Hibernate, y este se encargará del resto.

Existen dos formas de hacer los mapeos en Hibernate, la primera es a través de archivos de mapeo en XML y la otra forma es usando anotaciones, la cual usaremos ahora.

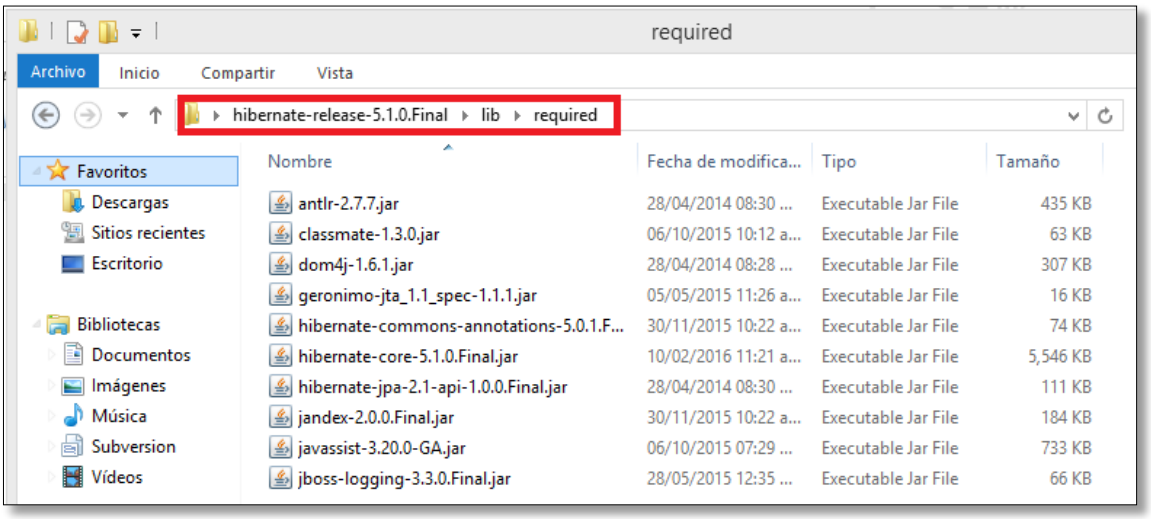
AÑADIENDO LIBRERÍA HIBERNATE

Lo primero 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 NetBeans ya tiene una biblioteca de Hibernate (dependiendo la versión y actualizaciones del IDE instaladas) pero en este caso añadiremos a la biblioteca la última versión de Hibernate. Así que descargamos la última versión del core de Hibernate desde la página de descarga de Hibernate.

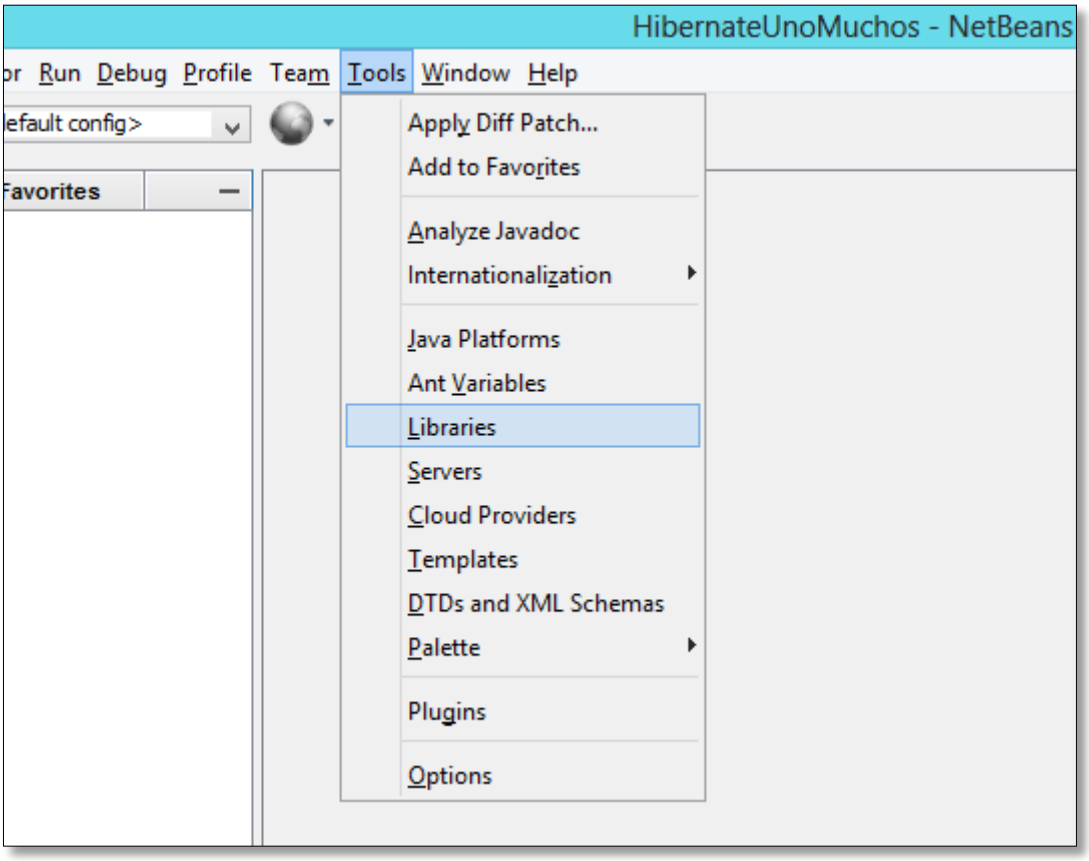
http://hibernate.org/orm/downloads/



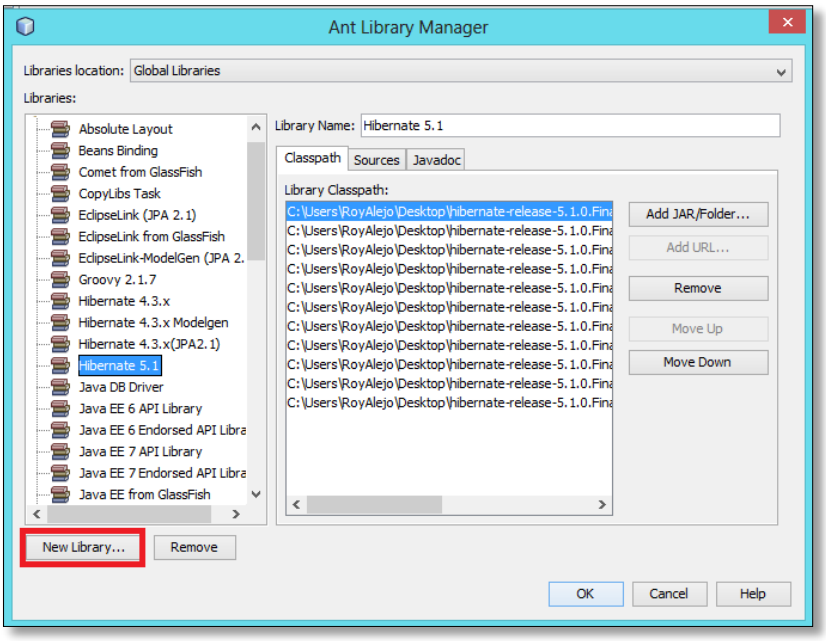
Una vez que descarguemos y descomprimamos el archivo correspondiente veremos que hay varios archivos, algunos son opcionales y otros son requeridos. Las clases obligatorias, que son las que nos interesan en este caso, se encuentran en el directorio "**lib\required**".

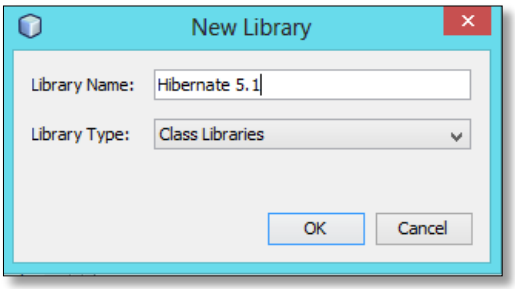


Bien, entonces para actualizar la biblioteca abrimos el NetBeans y nos dirigimos al menú "Tools -> Libraries":

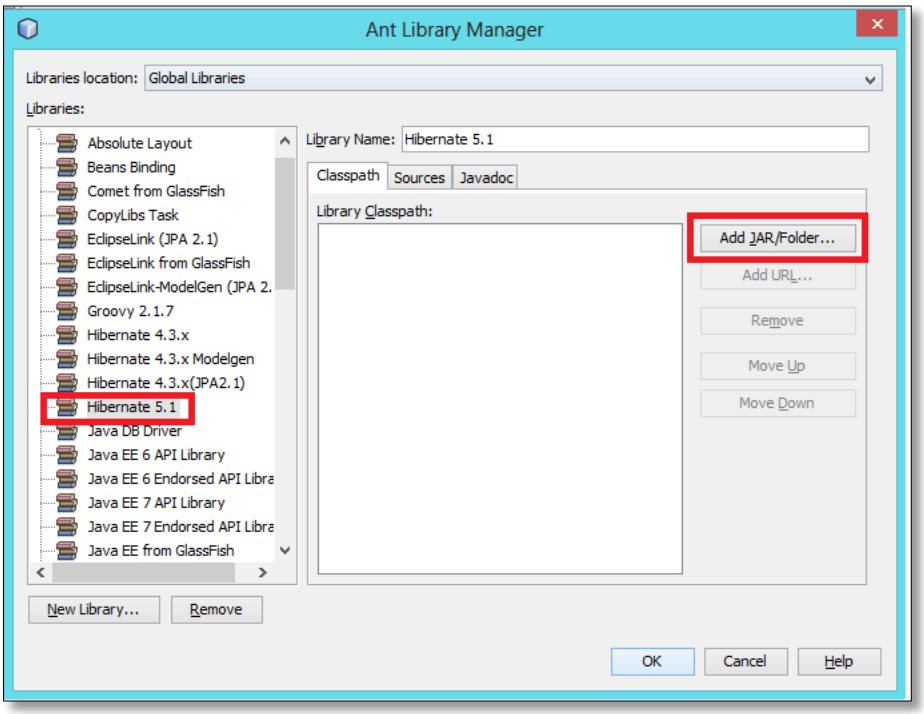


En la ventana que se abre clic en **New Library** y nombramos a nuestra librería en este caso **Hibernate 5.1**

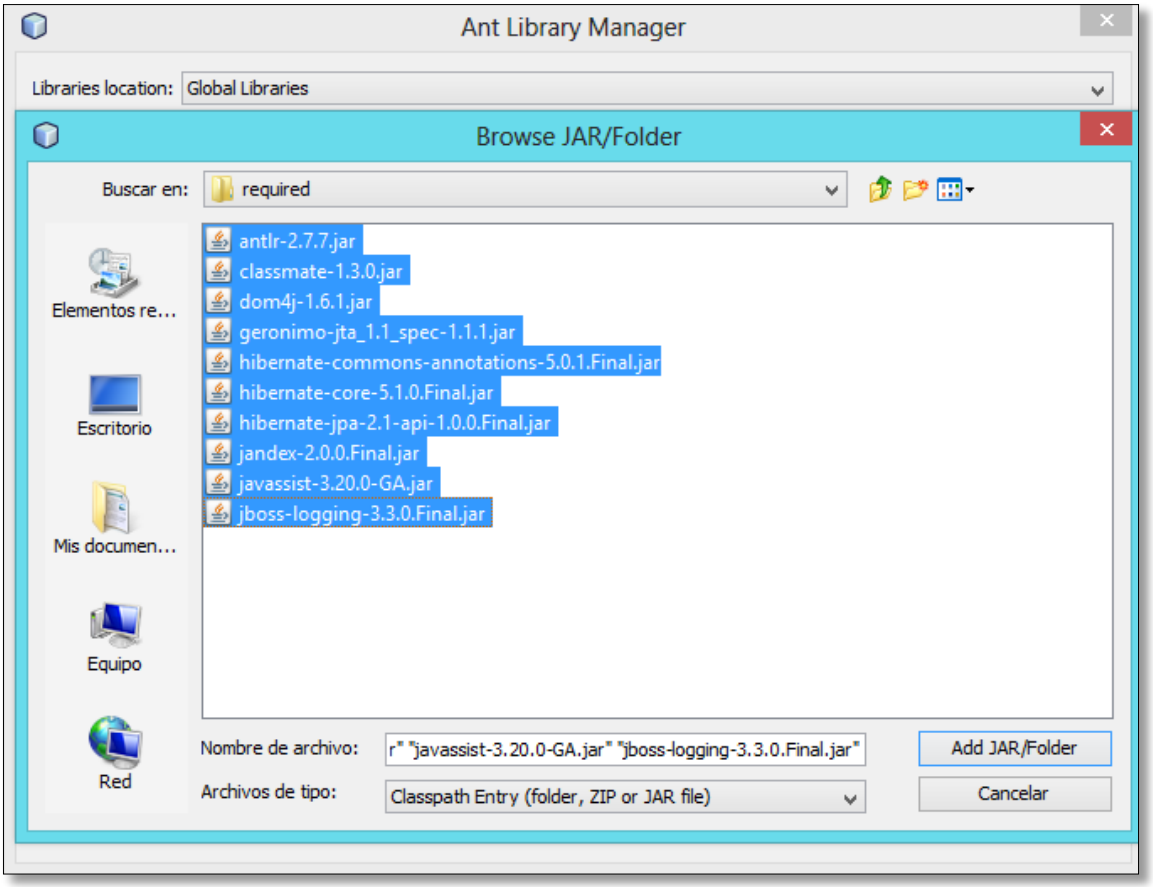




Seleccionamos la librería creada y presionamos el botón "Add JAR/Folder..." para agregar los nuevos archivos:



Seleccionamos estos archivos:

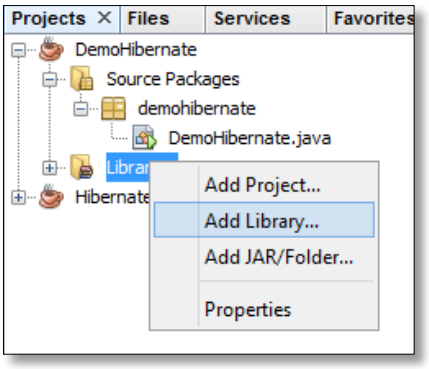


Presionamos el botón "OK" y nuestra biblioteca ya estará actualizada.

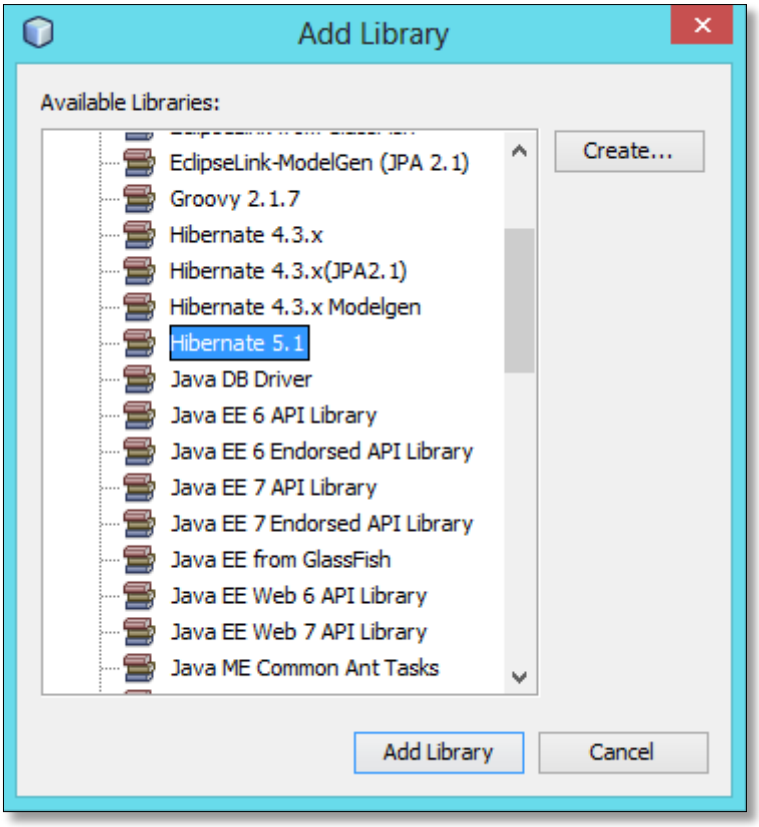
CREANDO EL PROYECTO

Ahora creamos un nuevo proyecto de NetBeans (menú "File -> New Project... -> Java -> Java Application"). Le ponemos por nombre DemoHibernate y una ubicación al proyecto y nos aseguramos de que la opción "Create Main Class" esté habilitada. Presionamos el botón "Finish" y veremos aparecer en el editor nuestra clase "Main".

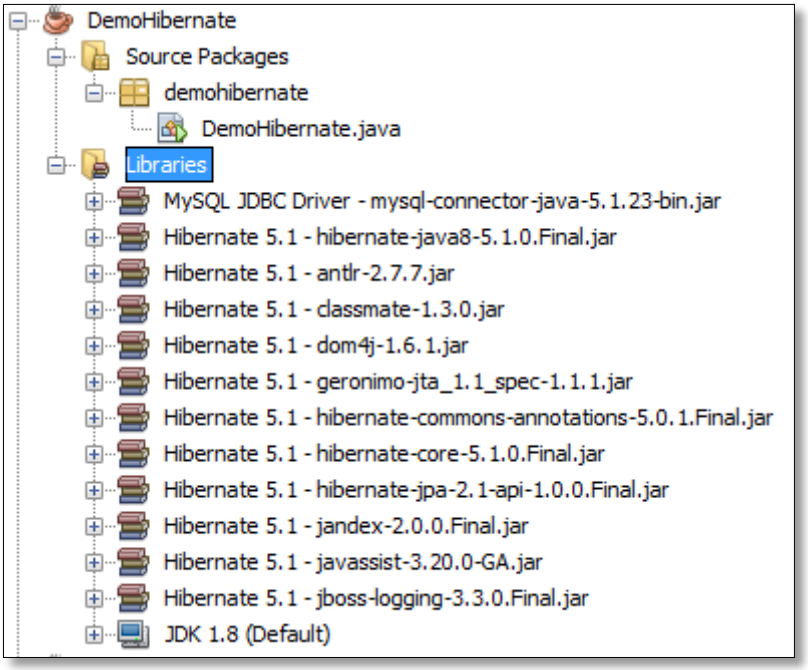
Agregamos la biblioteca de "Hibernate", que creamos hace unos momentos, a nuestro proyecto. Hacemos clic derecho en el nodo "Libraries" del proyecto. En el menú contextual que se abre seleccionamos la opción "Add Library...":



En la ventana que se abre seleccionamos la biblioteca "Hibernate 5.1":

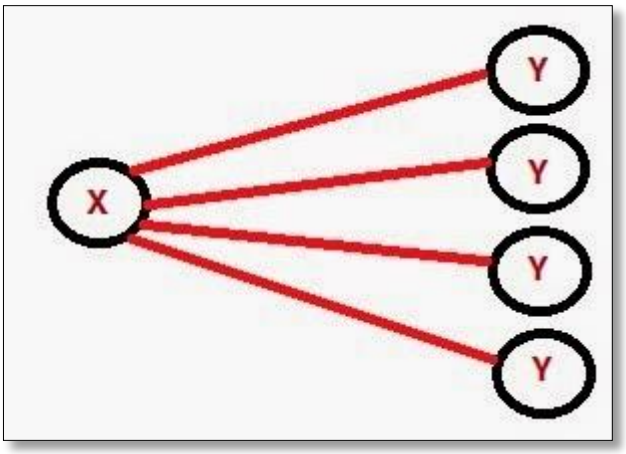


Presionamos el botón "Add Library" para que la biblioteca se agregue a nuestro proyecto. Aprovechamos también para agregar el conector de MySQL. Debemos tener los siguientes archivos en nuestro proyecto:

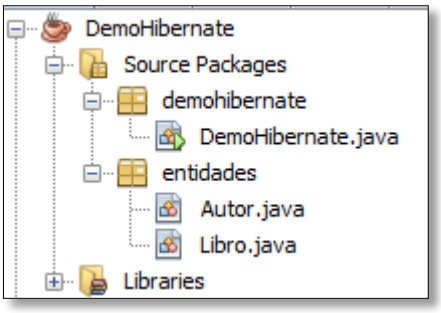


IMPLEMENTANDO

El siguiente paso es crear las clases cuyas instancias serán almacenadas como fila de una tabla en base de datos. Este tipo de objetos son llamados "Entidades". Para este caso las entidades serán **Autor(X)** y **Libro(Y)**. Un autor puede tener muchos libros pero un libro pertenece a un autor.



Crearemos el paquete **Entidades** y dentro las siguientes 2 clases: Autor y Libro. La estructura del proyecto quedará de la siguiente manera.



La clase **Libro** tendrá los atributos título y género. Además, por regla, necesitamos un atributo que funcione como identificador para cada una de las entidades.

En este caso la clase implementa la interface "java.io.Serializable", esto no es obligatorio pero es una buena práctica. La clase Libro quedaría de la siguiente manera:

```
package entidades;

import java.io.Serializable;

public class Libro implements Serializable {

   private long id;
   private String titulo;
   private String genero;

   public Libro() {
   }
```

```
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getTitulo() {
        return titulo;
    }

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

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

}
```

La clase **Autor** quedaría de la siguiente forma con los atributos nombres, apellidos, ciudad de nacimientos, país de nacimiento y la lista de libros que ha escrito:

```
package entidades;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class Autor implements Serializable {

    private long id;
    private String nombres;
    private String apellidos;
    private String ciudadNac;
    private String paisNac;
    private List<Libro> libros = new ArrayList<Libro>();

    public Autor() {
    }

    public long getId() {
        return id;
    }
```

```
    public void setId(long id) {
        this.id = id;
    }

    public String getNombres() {
        return nombres;
    }

    public void setNombres(String nombres) {
        this.nombres = nombres;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public String getCiudadNac() {
        return ciudadNac;
    }

    public void setCiudadNac(String ciudadNac) {
        this.ciudadNac = ciudadNac;
    }

    public String getPaisNac() {
        return paisNac;
    }

    public void setPaisNac(String paisNac) {
        this.paisNac = paisNac;
    }

    public List<Libro> getLibros() {
        return libros;
    }

    public void setLibros(List<Libro> libros) {
        this.libros = libros;
    }
}
```

Ahora es momento de indicarle a través de anotaciones las relaciones que existen.

En la clase Autor agregaremos las anotaciones Entity y Table de la siguiente manera:

```
@Entity
@Table(name = "autor")
```

```
public class Autor implements Serializable {
```

Cuando Hibernate genere la tabla para esta entidad lo hará en base al nombre de la misma. En este caso creará una tabla llamada "autor". La anotación "javax.persistence.Table" sirve para indicar cuál será el nombre de la tabla generada (o si ya tenemos una base de datos creada, cuál es el nombre de la tabla en la que se almacenarán estas entidades).

En el atributo id agregaremos las anotaciones Id y GeneratedValue:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
```

Podemos personalizar la forma en la que se generará el valor del identificador usando la anotación "javax.persistence.GeneratedValue" e indicando la "estrategia" que se usará para generarlo.

- AUTO
- IDENTITY
- SEQUENCE
- TABLE

Las más usadas son "AUTO" e "IDENTITY". La primera usa la estrategia por default de la base de datos, mientras que la segunda genera los identificadores en orden (1, 2, 3, etc.).

Al atributo libros agregar la anotación OneToMany

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
 private List<Libro> libros = new ArrayList<Libro>();
```

@OneToOne aquí definiremos qué operaciones serán realizadas en cascada. Además también podemos indicar el tipo de "fetch" o recuperación que tendrá la colección. Solo existen dos tipo de recuperación: tardado (lazy) e inmediato (eager). Si decidimos que la recuperación sea "lazy" entonces las entidades relacionadas (los Libros de la colección) no serán recuperados de la base de datos al momento que se recupera el Auto, sino hasta que se usen estos elementos (siempre y cuando estemos dentro de una transacción). Si la recuperación es "eager" entonces los Libros relacionados se recuperarán al mismo tiempo que el Autor. En este caso se usará un fetch de tipo eager.

La clase Autor con todas sus anotaciones quedaría de la siguiente manera:

```
package entidades;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
```

```
@Table(name = "autor")
public class Autor implements Serializable {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private long id;
   private String nombres;
   private String apellidos;
   private String ciudadNac;
   private String paisNac;

   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
   private List<Libro> libros = new ArrayList<Libro>();

   public Autor() {
   }

   public long getId() {
      return id;
   }

   public void setId(long id) {
      this.id = id;
   }

   public String getNombres() {
      return nombres;
   }

   public void setNombres(String nombres) {
      this.nombres = nombres;
   }

   public String getApellidos() {
      return apellidos;
   }

   public void setApellidos(String apellidos) {
      this.apellidos = apellidos;
   }

   public String getCiudadNac() {
      return ciudadNac;
   }

   public void setCiudadNac(String ciudadNac) {
      this.ciudadNac = ciudadNac;
   }

   public String getPaisNac() {
      return paisNac;
   }
```

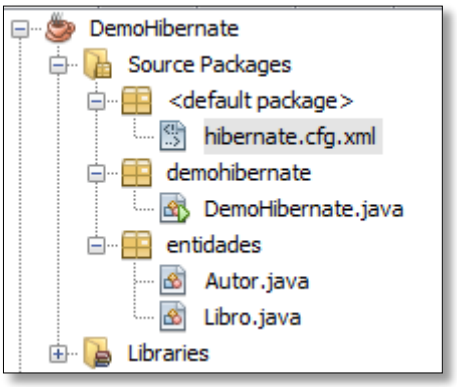
```
    public void setPaisNac(String paisNac) {
        this.paisNac = paisNac;
    }

    public List<Libro> getLibros() {
        return libros;
    }

    public void setLibros(List<Libro> libros) {
        this.libros = libros;
    }

    public void addLibro(Libro libro) {
        this.libros.add(libro);
    }
}
```

Ahora que tenemos claros estos conceptos de anotaciones, en la clase Libro agregar las anotaciones Entity. Al atributo id los atributos Id y GeneratedValue. Quedaría de la siguiente manera:

```
package entidades;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Libro implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titulo;
    private String genero;

    public Libro() {
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getTitulo() {
```

```
        return titulo;
    }

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

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

}
```

Hacemos clic derecho en **Source Packages** y agregamos un archivo XML y lo nombraremos: **hibernate.cfg**



Borramos todo el contenido del archivo generado y colocamos las siguientes líneas:

```
<?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>
</hibernate-configuration>
```

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:

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

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

Lo siguiente es configurar la conexión a nuestra base de datos. 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). Se usará una base de datos MySQL para este ejemplo, así que la configuración queda de la siguiente forma:

```
<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>
```

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.MySQL5Dialect":

```
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</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" (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, segun la documentación oficial (falta "none"), de "validate", "update", "create", y "create-drop" (aunque no todos los valores funcionan para todas las bases de datos). Serán colocados de la siguiente forma:

```
<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 los archivos se mapearás así:

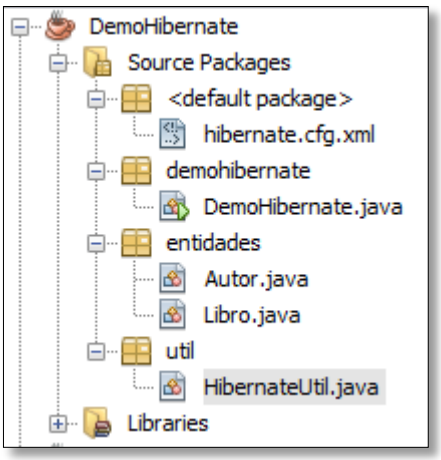
```
<mapping class="entidades.Libro" />
<mapping class="entidades.Autor" />
```

El contenido final de dicho archivo sería el siguiente:

```
<!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/hibernaterelaciones</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>

      <!-- Aqui iran los archivos de mapeo -->
      <mapping class="entidades.Libro" />
      <mapping class="entidades.Autor" />
   </session-factory>
</hibernate-configuration>
```

Ahora, para recuperar los objetos, crearemos 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.



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 cogemos 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:

```
package util;

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;
   }
}
```

Ahora, en **DemoHibernate.java** implementaremos lo siguiente:

```
package demohibernate;

import entidades.Autor;
import entidades.Libro;
import org.hibernate.Session;
import util.HibernateUtil;

public class DemoHibernate {

   public static void main(String[] args) {
      /*Primero creamos un autor y la asociamos con dos libros*/
      Libro libro1 = new Libro();
      libro1.setTitulo("20000 leguas de viaje submarino");
      libro1.setGenero("Novela");

      Libro libro2 = new Libro();
      libro2.setTitulo("La maquina del tiempo");
      libro2.setGenero("Novela");

      Autor autor1 = new Autor();
      autor1.setNombres("Nombre autor a eliminar");
      autor1.setApellidos("Apelido autor a elminar");
      autor1.setCiudadNac("Lima");
      autor1.setPaisNac("Peru");
      autor1.addLibro(libro1);
      autor1.addLibro(libro2);

      /*Creamos un segundo autor y lo asociamos con otros dos libros*/
      Libro libro3 = new Libro();
      libro3.setTitulo("El ingenioso hidalgo don Quijote de la Mancha");
      libro3.setGenero("Burlesque");

      Libro libro4 = new Libro();
      libro4.setTitulo("La Galatea");
      libro4.setGenero("Pastoral");

      Autor autor2 = new Autor();
      autor2.setNombres("Alex");
      autor2.setApellidos("Matias Soto");
      autor2.setCiudadNac("Paris");
      autor2.setPaisNac("Francia");
      autor2.addLibro(libro3);
      autor2.addLibro(libro4);

      /*En la primer sesion guardamos los dos autores (los libros
       correspondientes seran guardados en cascada*/
      Session sesion = HibernateUtil.getSessionFactory().openSession();
      sesion.beginTransaction();

      sesion.persist(autor1);
```

```
        sesion.persist(autor2);

        sesion.getTransaction().commit();
        sesion.close();

        /*En la segunda sesion eliminamos el autor1 (los dos primeros
         libros seran borrados en cascada)*/
        sesion = HibernateUtil.getSessionFactory().openSession();
        sesion.beginTransaction();

        sesion.delete(autor1);

        sesion.getTransaction().commit();
        sesion.close();
    }
}
```

Listo.

EJECUCION

Al ejecutar el programa, se insertan 2 autores con sus respectivos 2 libros en la base de datos. Por último, se elimina el primer autor y los 2 libros que le pertenecen se borran por la propiedad de cascada.

Autores

id	apellidos	ciudadNac	nombres	paisNac
2	Matias Soto	Paris	Alex	Francia

Libros

id	genero	titulo
3	Burlesque	El ingenioso hidalgo don Quijote de la Mancha
4	Pastoral	La Galatea