

# UT3. MAPEO OBJETO-RELACIONAL

## Proyectos ejemplo

### Contenido

1.	DEMO 3.1 PARTE1. PRIMER PROYECTO HIBERNATE: INGENIERÍA INVERSA .....	2
1.1.	Crear base de datos y tabla en MySQL Workbench .....	2
1.2.	Crear proyecto Maven en NetBeans .....	2
1.3.	Añadir dependencias para MySQL e Hibernate: fichero pom.xml .....	4
1.4.	hibernate.cfg.xml.....	8
1.5.	Fichero hibernate.reveng.xml (Reverse Engineering) .....	14
1.6.	Mapeo de tablas: generación de los POJOS y los ficheros de mapeo .....	16
	Libro.java.....	18
	Libro.hbm.xml.....	19
	Fichero <b>hibernate.cfg.xml</b> modificado .....	20
1.7.	Clase HibernateUtil.java.....	21
1.8.	Programa principal .....	24
	Principal.java .....	25
2.	DEMO 3.1 PARTE 2. OPERACIONES CRUD CON HIBERNATE.....	27
2.1.	Métodos cerrarSessionFactory y cerrarSesion (HibernateUtil.java).....	27
2.2.	Nueva clase Operaciones.java (CRUD) .....	27
	Método para inicializar los valores de Session y Transaction .....	28
	Método para insertar (crear) un Libro .....	28
	Método para leer un Libro (facilitando el ISBN).....	29
	Método para modificar un Libro (facilitando el ISBN) .....	29
	Método para borrar un Libro (facilitando el ISBN).....	30
2.4.	Programa principal ejemplo .....	31

# 1. DEMO 3.1 PARTE1. PRIMER PROYECTO HIBERNATE: INGENIERÍA INVERSA

## 1.1. Crear base de datos y tabla en MySQL Workbench

Creamos una base de datos denominada ut3demo1 y en ella una tabla LIBRO:

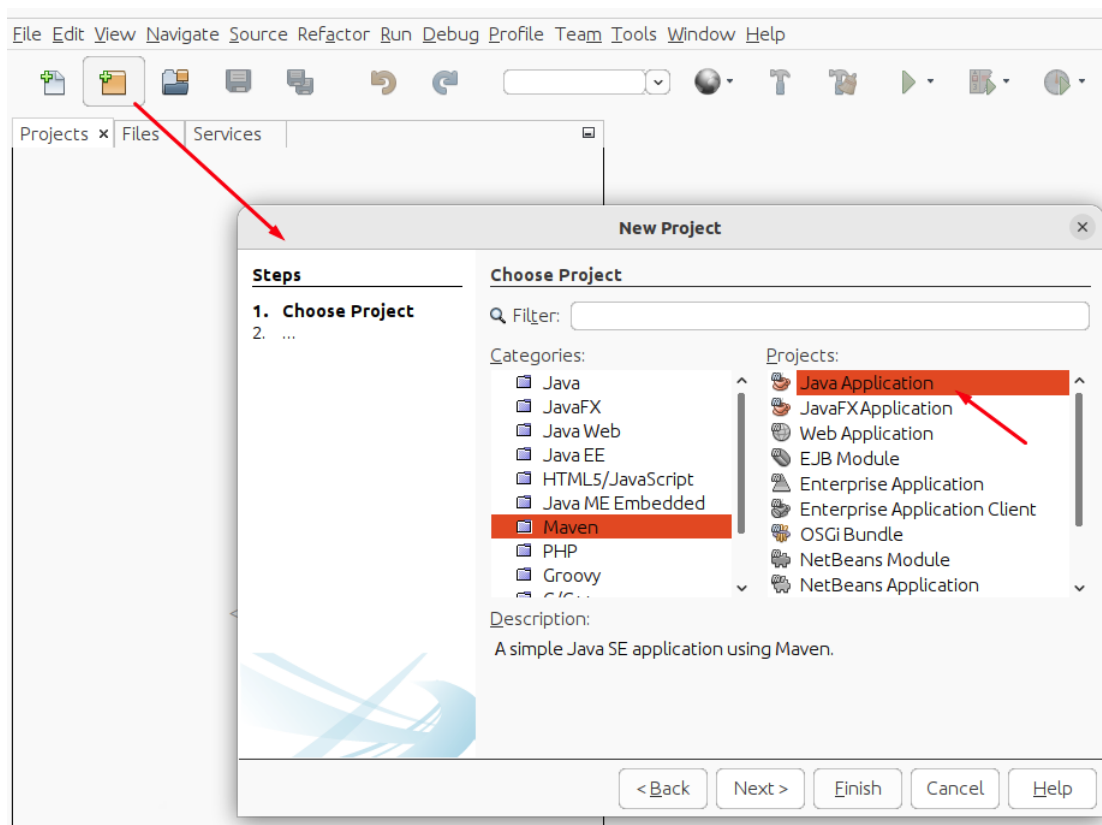
```
CREATE DATABASE ut3demo1;
USE ut3demo1;
CREATE TABLE LIBRO (
    isbn INTEGER PRIMARY KEY,
    titulo VARCHAR(100),
    autor VARCHAR(100)
);
```

**NOTA:** En una base de datos MySQL, la sentencia USE se utiliza para seleccionar la base de datos sobre la que se van a ejecutar las operaciones posteriores. Es decir, cambia el contexto a la base de datos especificada, para que no sea necesario anteponer el nombre de la base de datos a cada operación (como ut3demo1.LIBRO en este caso).

## 1.2. Crear proyecto Maven en NetBeans

Para el primer proyecto de esta unidad vamos a utilizar NetBeans 8.2 y la versión 8 del JDK de Java.

1. File > New Project. Elegimos Maven > Java Application:



2. Pulsamos Next.

3. Rellenamos los campos de la ventana 'Name and Location':

- **Artifact id** es el nombre del proyecto, sin el nombre del paquete. Es un nombre único que identifica el proyecto.
- **Group id** es el nombre del grupo u organización que creó el proyecto. Es un nombre único que identifica al grupo u organización.
- **Package** es el nombre del paquete del proyecto. Es un nombre que identifica el conjunto de clases que forman el proyecto.

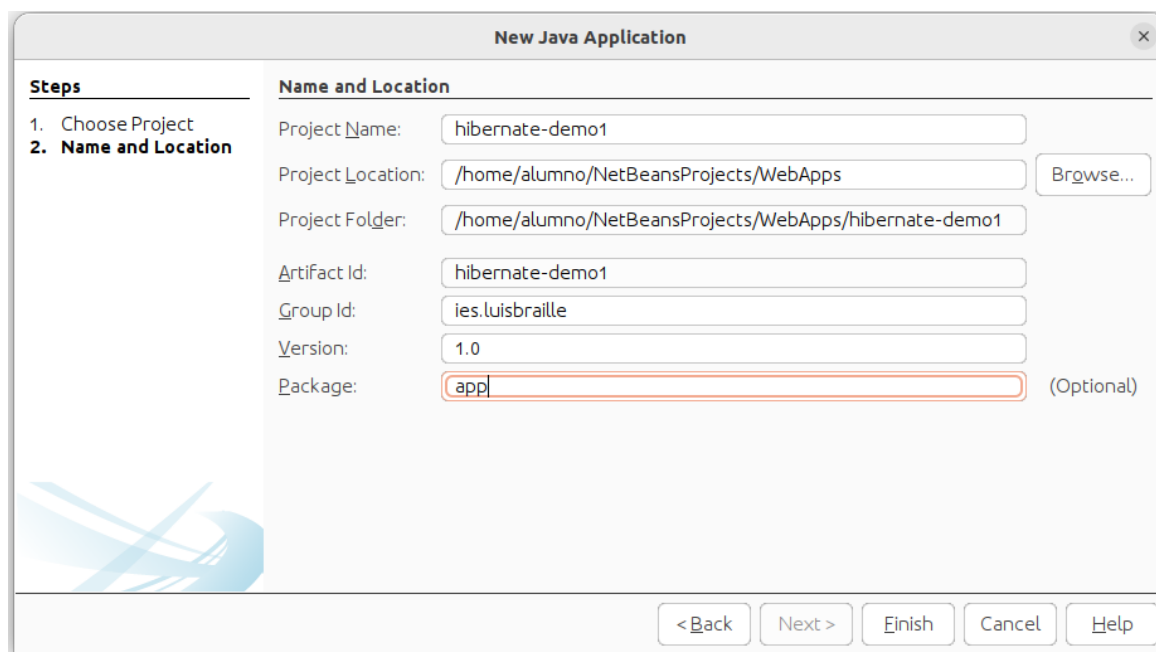
Los valores de estos campos se utilizan para crear un nombre de archivo JAR para el proyecto. El nombre del archivo JAR tiene el siguiente formato:

```
groupId:artifactId:version
```

Por ejemplo, si el artifact id es "hibernate-demo1", el group id es "ies.luisbraille" y la versión es "1.0", el nombre del archivo JAR sería:

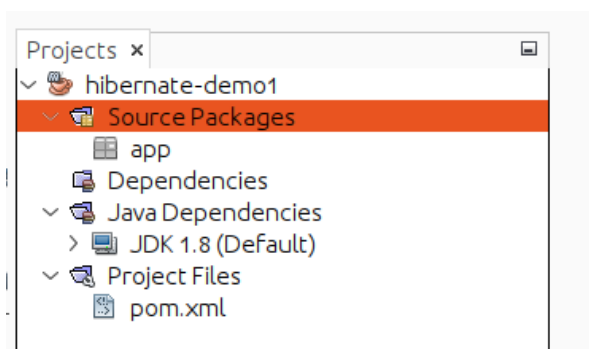
```
ies.luisbraille:hibernate-demo1:1.0
```

Estos campos también se utilizan para identificar el proyecto en el repositorio central de Maven. El repositorio central es un repositorio de código abierto que contiene millones de proyectos Java.



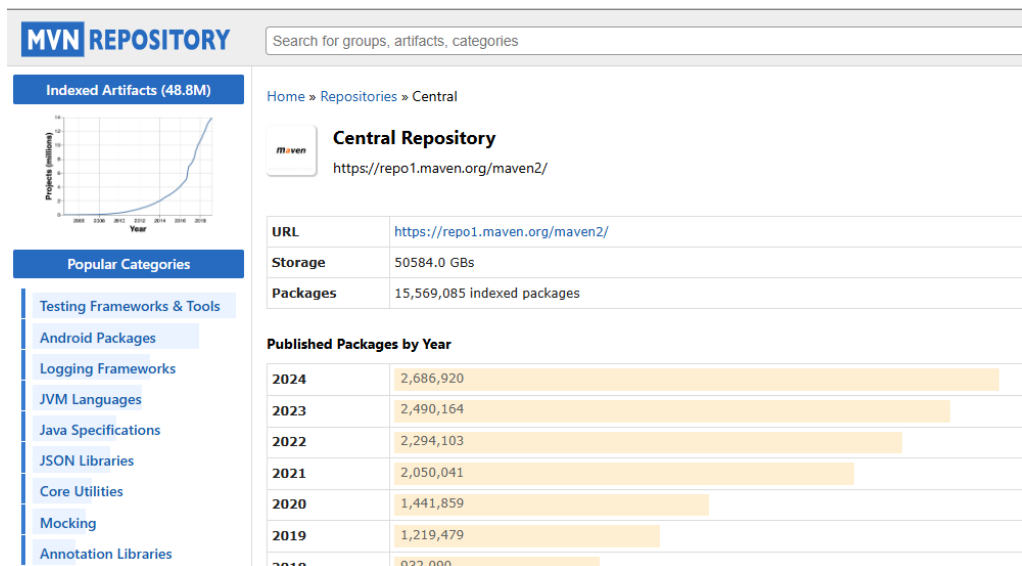
4. Pinchamos en Finish y se creará nuestro proyecto Java/Maven.

Se creará la siguiente estructura de proyecto:



### 1.3. Añadir dependencias para MySQL e Hibernate: fichero pom.xml

Tenemos que añadir en nuestro proyecto las dependencias para trabajar con Hibernate y MySQL (Connector J - JDBC). Las dependencias se incluyen el fichero Project Files > pom.xml, y las obtenemos del repositorio Maven central: <https://mvnrepository.com/repos/central>.



### Dependencia para trabajar con MySQL

En el repositorio Maven central Buscamos "conector jdbc mysql" y elegimos el primer resultado:

**Repository**

- Central 1.1k
- Sonatype 96
- JCenter 51
- Mulesoft 35
- Clojars 34
- BT OpenHAB 23
- XWiki Releases 22
- Spring Plugins 18

**Found 1426 results**


Sort: **relevance** | popular | newest

**1. MySQL Connector/J**  
com.mysql » mysql-connector-j **945 usages**

MySQL Connector/J is a JDBC Type 4 driver, which means that it is pure Java implementation of the MySQL protocol and does not rely on the MySQL client libraries. This driver supports auto-registration with the Driver Manager, standardized validity checks, categorized SQLExceptions, support for large update counts, support for local and offset date-time variants from the java.time package, support for JDBC-4.x XML processing, support for per connection client information and support for the NCHAR, NVARCHAR ...

Last Release on Oct 15, 2024

Seleccionamos la última versión disponible en la pestaña Central:



### MySQL Connector/J

MySQL Connector/J is a JDBC Type 4 driver, which means that it is pure Java implementation of the MySQL protocol and does not rely on the MySQL client libraries. This driver supports auto-registration with the Driver Manager, standardized validity checks, categorized SQLExceptions, support for large update counts, support for local and offset date-time variants from the java.time package, support for JDBC-4.x XML processing, support for per connection client information and support for the NCHAR, NVARCHAR ...

Categories	JDBC Drivers
Tags	database sql jdbc driver connector rdbms mysql connection
HomePage	http://dev.mysql.com/doc/connector-j/en/
Ranking	#579 in MvnRepository (See Top Artifacts) #7 in JDBC Drivers
Used By	945 artifacts

Central (9)Redhat GA (1)Redhat EA (1)

Version		Vulnerabilities	Repository	Usages	Date
9.1.x	9.1.0		Central	119	Oct 15, 2024
9.0.x	9.0.0		Central	115	Jul 02, 2024
8.4.x	8.4.0		Central	141	Apr 30, 2024

Pinchamos en el nombre de la versión 9.1.0 y en la página que se abre encontramos la dependencia para Maven:



### MySQL Connector/J » 9.1.0

MySQL Connector/J is a JDBC Type 4 driver, which means that it is pure Java implementation of the MySQL protocol and does not rely on the MySQL client libraries. This driver supports auto-registration with the Driver Manager, standardized validity checks, categorized SQLExceptions, support for large update counts, support for local and offset date-time variants from the java.time package, support for JDBC-4.x XML processing, support for per connection client information and support for the NCHAR, NVARCHAR ...

Categories	JDBC Drivers
Tags	database sql jdbc driver connector rdbms mysql connection
Organization	Oracle Corporation
HomePage	http://dev.mysql.com/doc/connector-j/en/
Date	Oct 15, 2024
Files	pom (3 KB) jar (2.5 MB) View All
Repositories	Central Fit2Cloud WSO2 Public
Ranking	#579 in MvnRepository (See Top Artifacts) #7 in JDBC Drivers
Used By	945 artifacts
Vulnerabilities	Vulnerabilities from dependencies: CVE-2024-7254

MavenGradleGradle (Short)Gradle (Kotlin)SBTLvyGrapeLeiningenBuildr

```
<!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <version>9.1.0</version>
</dependency>
```

Este código xml tendremos que copiarlo en el fichero pom.xml (lo veremos más adelante).

## Dependencia para trabajar con Hibernate

Vamos a trabajar con la **versión 5.6.14 Final** de Hibernate. Para encontrar la dependencia que tenemos que añadir a nuestro proyecto procedemos igual que para la dependencia de MySQL.

En el repositorio Maven central Buscamos "Hibernate core 5.6.14" y elegimos el primer resultado: "Hibernate Core Relocation"

The screenshot shows the Maven Repository search interface. The search bar contains 'hibernate core 5.6.14'. The results show 55268 results. The first result is 'Hibernate Core Relocation' by org.hibernate, with 4,641 usages. The description is 'Hibernate's core ORM functionality'. The last release is on Dec 18, 2024. A relocation notice indicates it was moved from org.hibernate.org to hibernate-core.

Buscamos 5.6.14 en la lista de versiones hasta encontrar la 5.6.14 Final:

5.6.15.Final	Central	342	Feb 06, 2023
5.6.14.Final	Central	165	Nov 04, 2022
5.6.13.Final	Central	33	Nov 04, 2022
5.6.12.Final	Central	131	Sep 27, 2022

Y obtenemos la dependencia Maven pinchando en el nombre de la versión:

The screenshot shows the detailed page for 'Hibernate Core Relocation 5.6.14.Final'. It includes the license (LGPL 2.1), categories (Object/Relational Mapping), tags (persistence, mapping, orm, hibernate, relational), organization (Hibernate.org), homepage (https://hibernate.org/orm), date (Nov 04, 2022), files (pom (5 KB), jar (7.2 MB)), repositories (Central), ranking (#119 in MvnRepository, #1 in Object/Relational Mapping), and used by (4,641 artifacts). A note indicates a new version (7.0.0.Beta3) is available. The page also shows build tool integrations for Maven, Gradle, SBT, Ivy, Grape, Leiningen, and Buildr. The Maven dependency XML snippet is provided at the bottom.

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.14.Final</version>
</dependency>
```

## Incluir dependencias en el archivo pom.xml

- Abrimos el fichero del proyecto Project Files > pom.xml, que contiene la configuración Maven de nuestro nuevo proyecto. Añadimos las dependencias para trabajar con Hibernate y MySQL (Connector J - JDBC), dentro de una etiqueta "dependencies":

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.1.0</version>
  </dependency>

  <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.14.Final</version>
  </dependency>
</dependencies>
```

- También necesitamos añadir al archivo pom.xml los siguientes bloques, para que funcione correctamente el acceso HTTPS a los repositorios de Maven:

```
<repositories>
  <repository>
    <id>central</id>
    <name>Central Repository</name>
    <url>https://repo.maven.apache.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>central</id>
    <name>Central Repository</name>
    <url>https://repo.maven.apache.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
    <releases>
      <updatePolicy>never</updatePolicy>
    </releases>
  </pluginRepository>
</pluginRepositories>
```

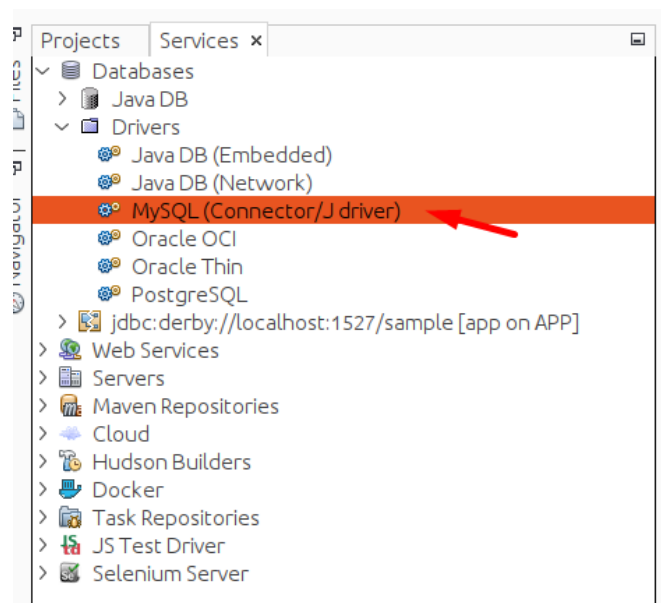
- Una vez modificado el archivo pom.xml, construimos el proyecto. Esto puede hacerse de varias formas:
  - a) pinchando en el icono 'Clean and Build' de la barra de herramientas (brocha y martillo)
  - b) en el menú 'Run' -> Clean and Build Project (Mayús + F11)
  - c) Click botón derecho sobre el nombre del proyecto -> Clean and Build
 Si todo funciona correctamente debemos ver el mensaje "BUILD SUCCESS":

```
--- maven-install-plugin:2.3.1:install (default-install) @
Installing /home/alumno/NetBeansProjects/UT3_24_25/UT3_24_
Installing /home/alumno/NetBeansProjects/UT3_24_25/UT3_24_
-----
BUILD SUCCESS
-----
Total time: 3.905s
```

## 1.4. hibernate.cfg.xml

Creamos a continuación el fichero XML hibernate.cfg.xml, que contendrá los datos de conexión con la base de datos MySQL creada en el primer apartado. Utilizaremos la herramienta "Hibernate Configuration Wizard".

1. En primer lugar, vamos a la pestaña Services y nos aseguramos de que no hay ninguna conexión a bases de datos MySQL, ni ningún Driver MySQL JDBC bajo el desplegable 'Databases':

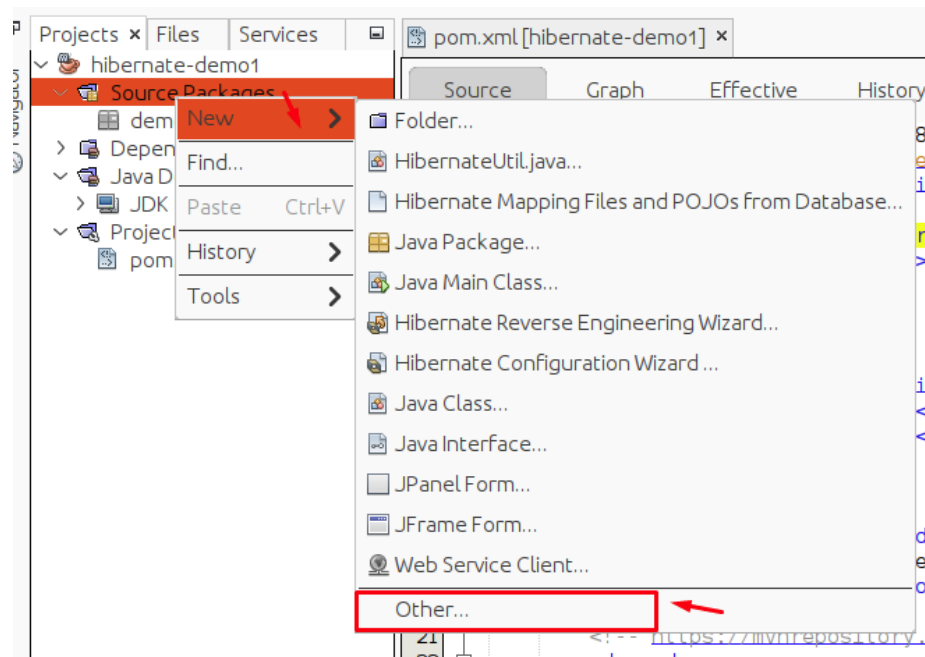


Si los hay, simplemente los borramos (botón derecho Delete).

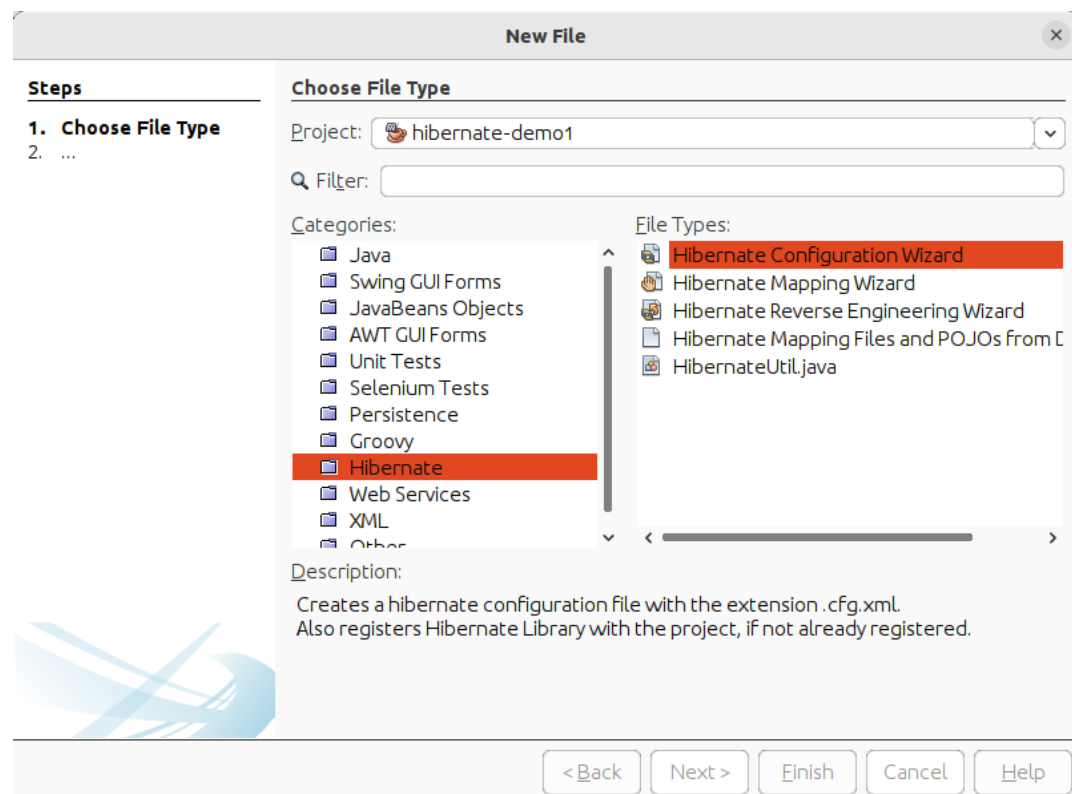
Cerramos NetBeans y lo volvemos a abrir.

2. Hacemos clic derecho en la vista Projects de NetBeans sobre Source Packages, New, Other.

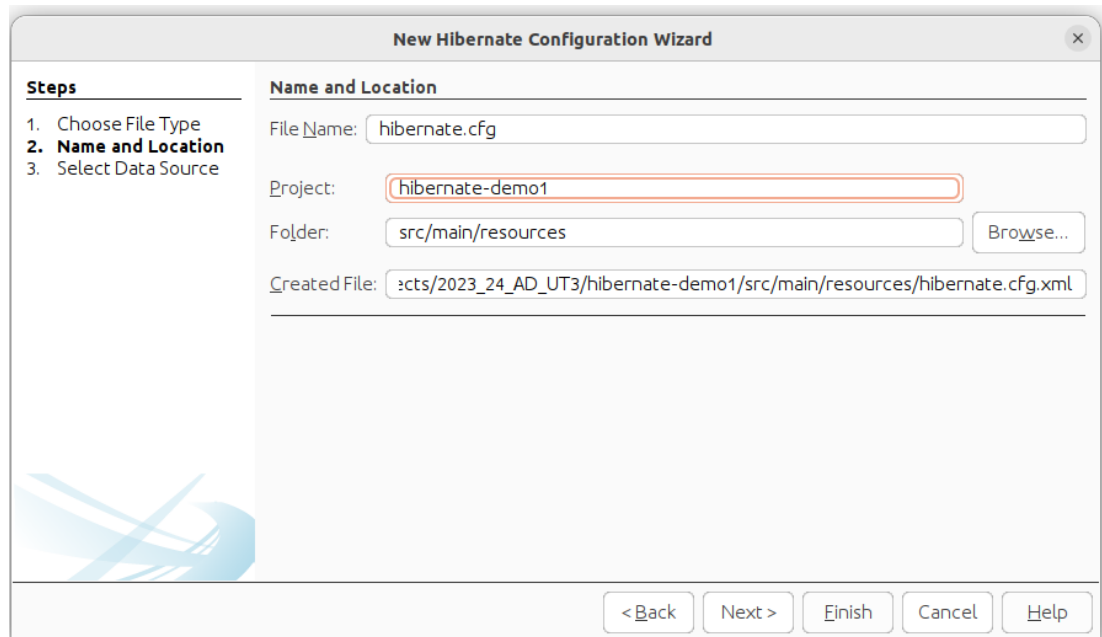




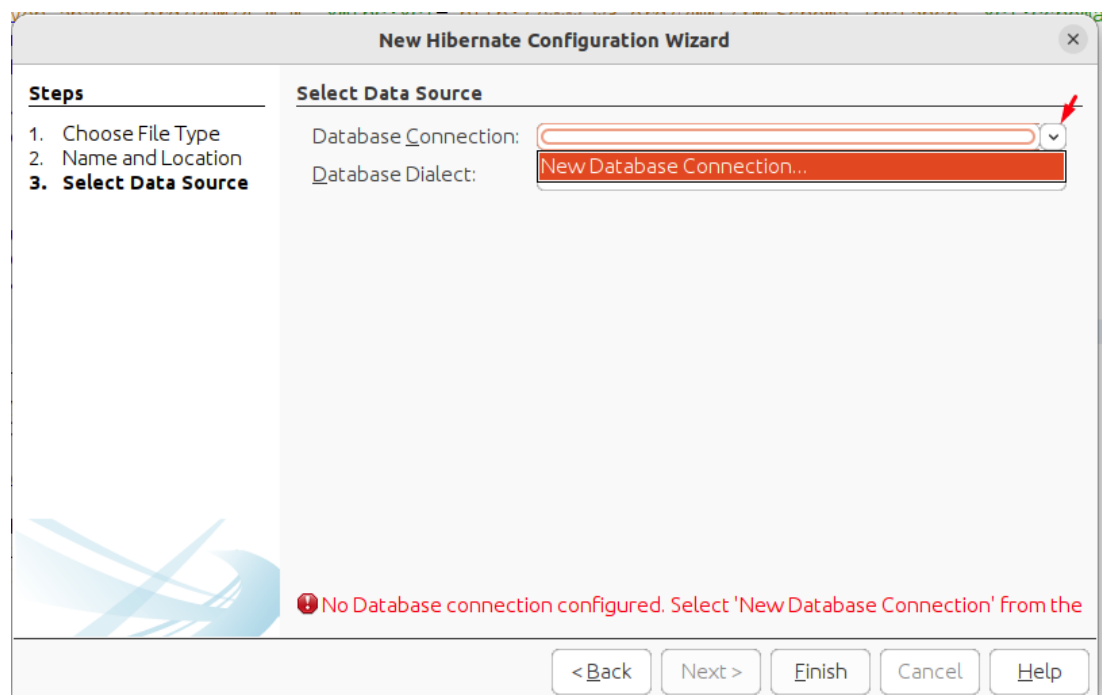
3. Elegimos en Categories la carpeta Hibernate y a la derecha **Hibernate Configuration Wizard**:



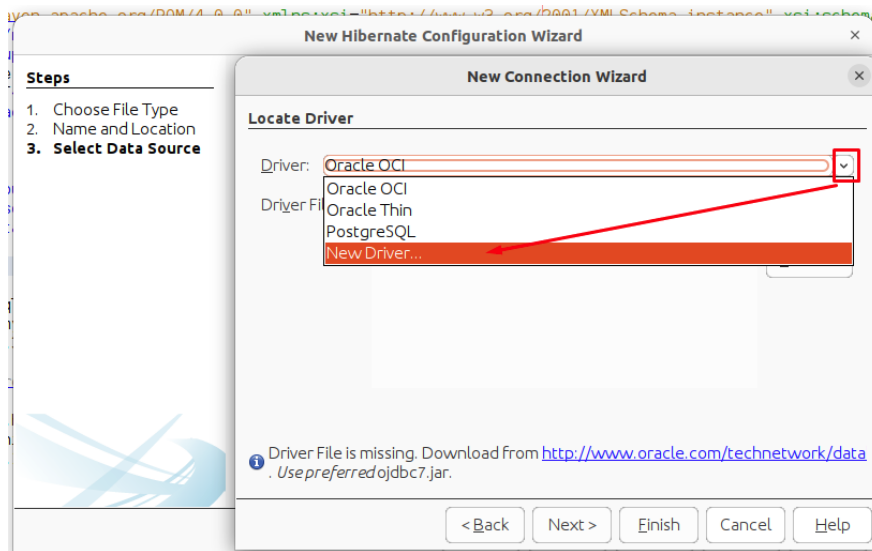
4. Pulsamos **Next**.
5. El nombre por defecto es *hibernate.cfg*. Sin cambiar nada se pulsa de nuevo en **Next**.



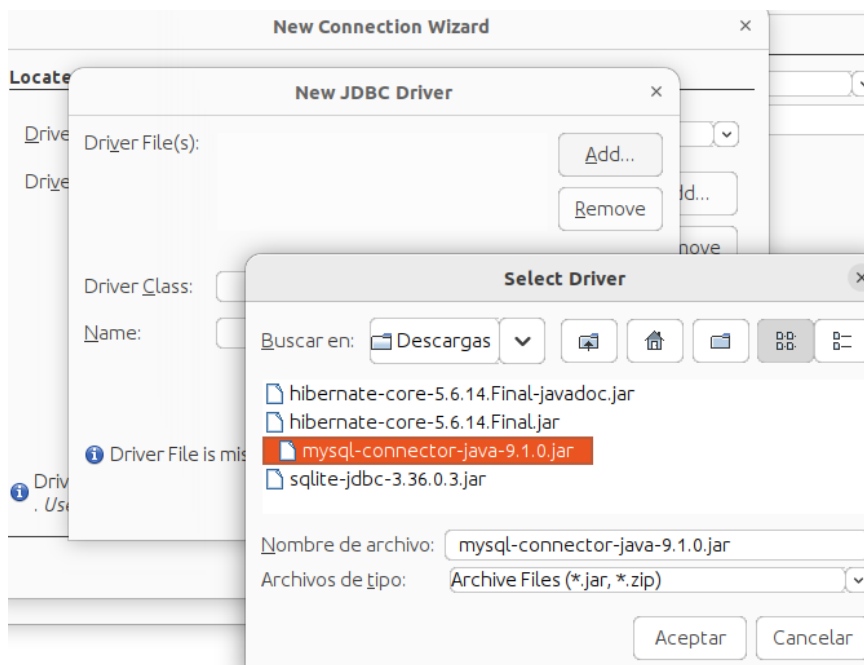
6. Pinchamos en la lista desplegable 'Database Connection' y seleccionamos New Database Connection....



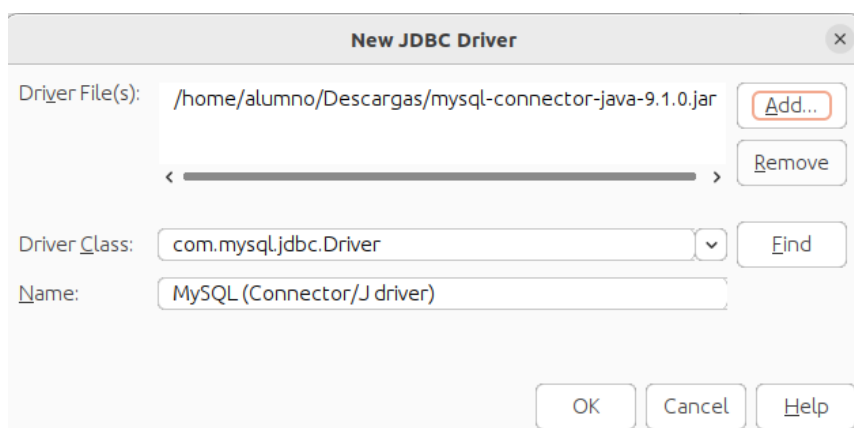
7. En la ventana 'Locate Driver' vamos a añadir el Driver JDBC de MySQL. Pinchamos en el desplegable Driver y elegimos 'New Driver...':



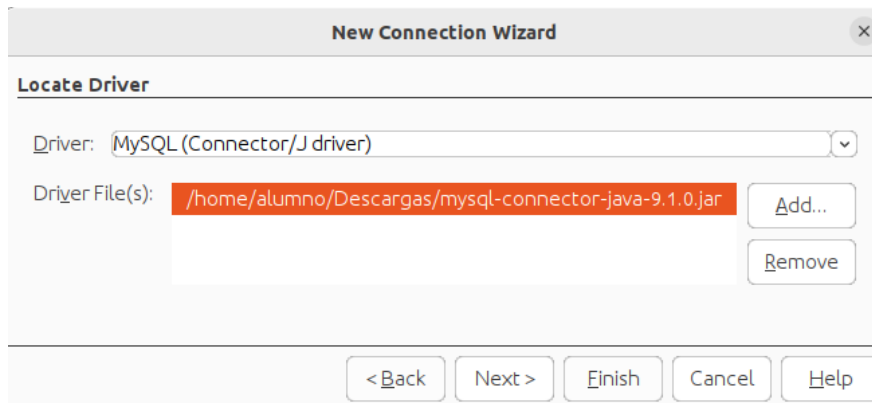
8. En la siguiente ventana, pulsamos Add... y elegimos el conector java MySQL (archivo .jar) descargado en la Unidad anterior:



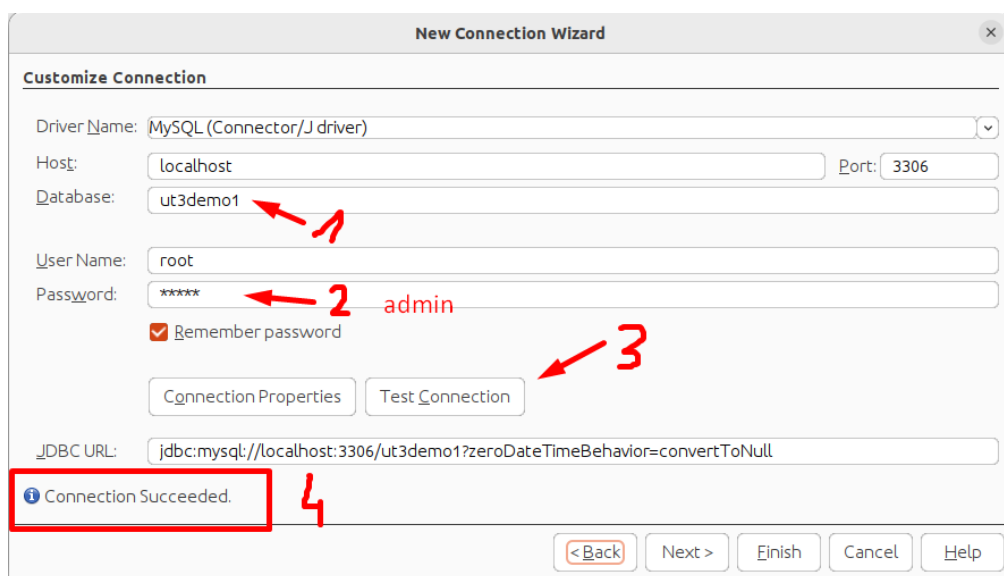
9. Pulsamos OK:



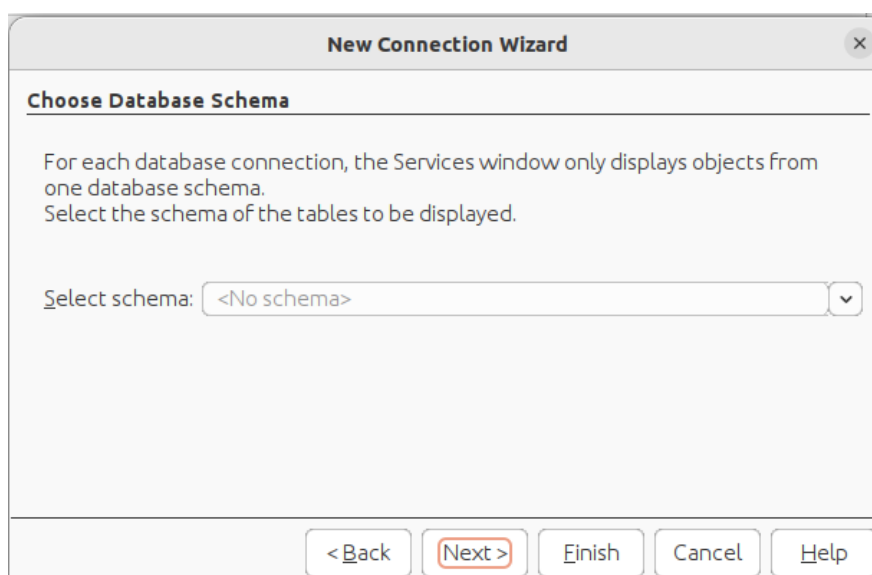
10. Seleccionado ya el Driver, pulsamos Next:



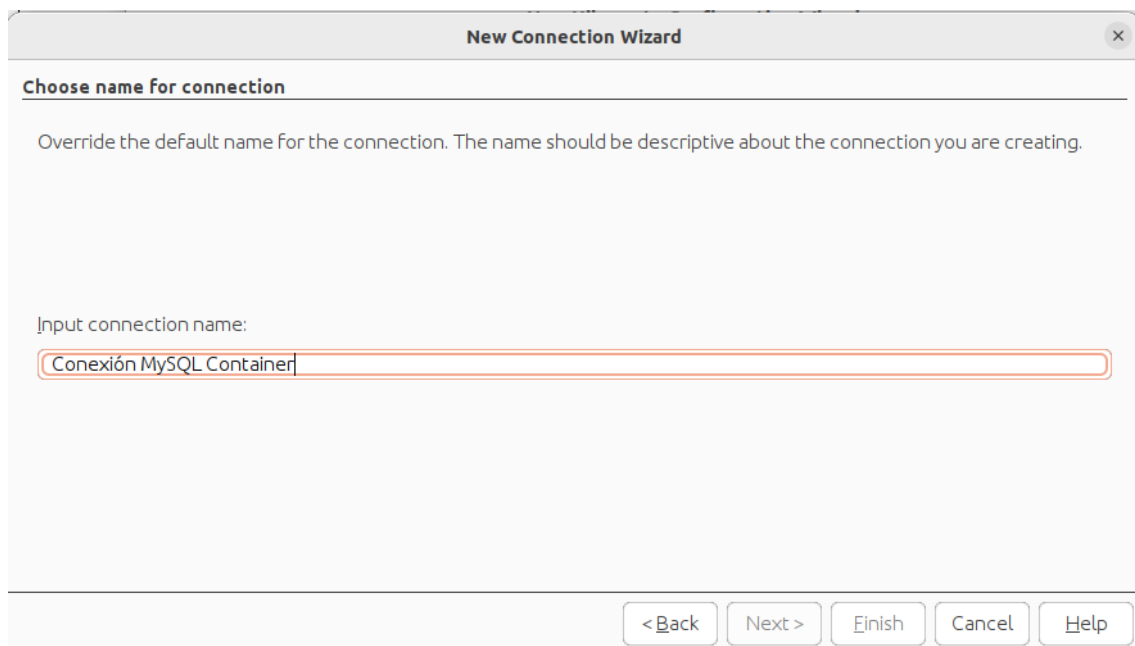
11. Aparece la ventana 'Customize Connection', con el driver creado y los datos de conexión a nuestra base de datos. Rellenamos los campos, pulsamos **Test Connection** y si todo ha ido OK pulsamos Next:



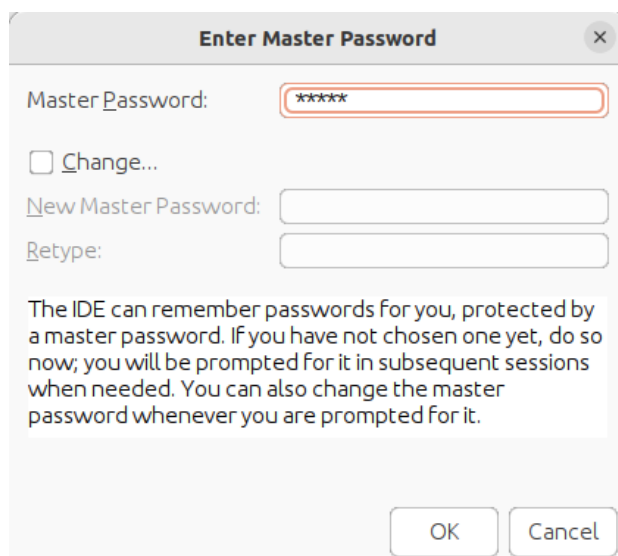
12. En la siguiente ventana (Choose Database Schema) no hace falta tocar nada. Simplemente pulsamos Next:



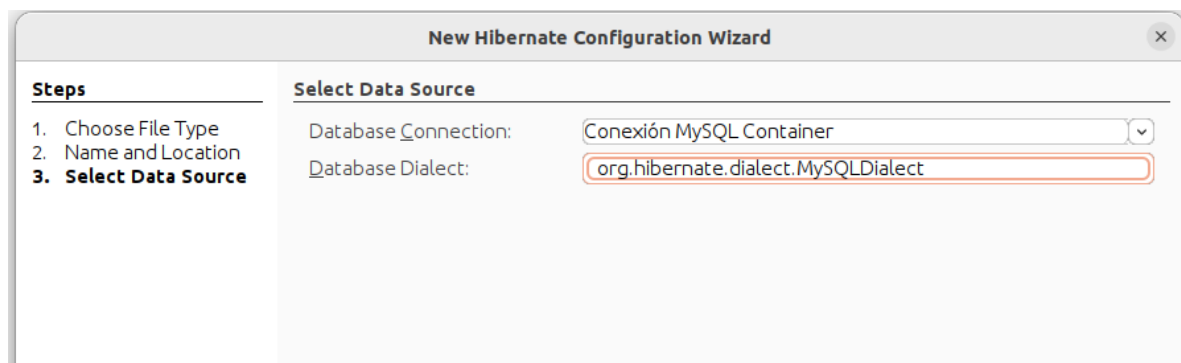
13. En la siguiente ventana cambiamos el 'Input connection name' por algo más intuitivo (por ejemplo, Conexión MySQL Container) y pulsamos Finish.



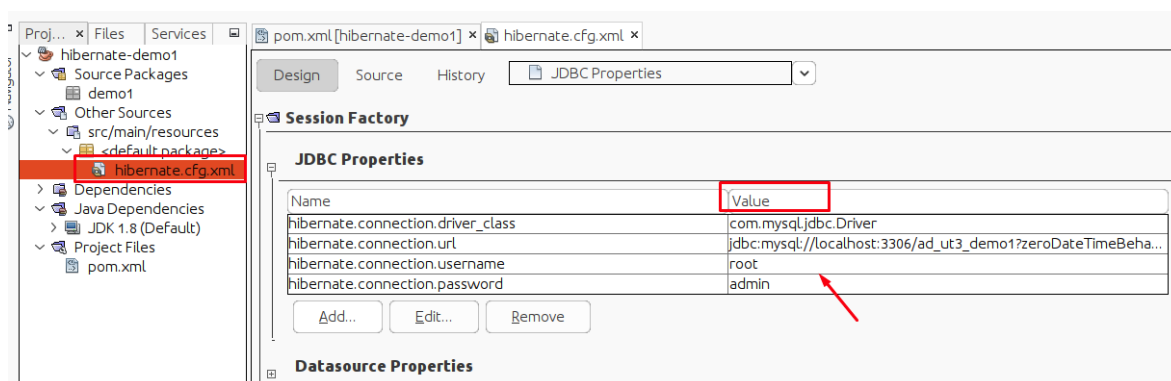
14. Si nos pide una Master Password, elegimos *admin* por ejemplo para que sea fácil de recordar y pulsamos ok:



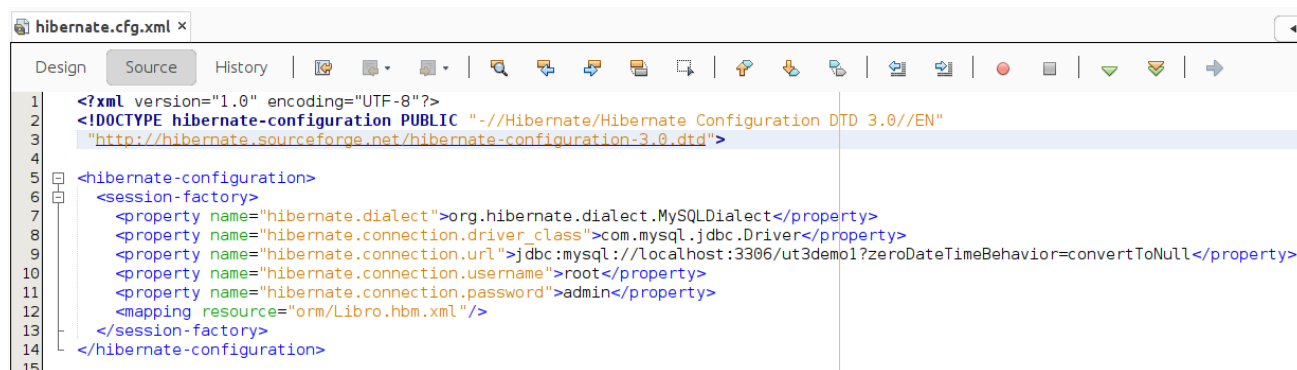
15. Aparecerá de nuevo la ventana 'Select Data Source' con la nueva conexión creada (Conexión MySQL Container) seleccionada y el dialecto con el que nos comunicaremos con la base de datos:



16. Pulsamos Finish. Se creará en el paquete por defecto del proyecto (<default package>) un fichero **hibernate.cfg.xml** con la configuración para la conexión con la base de datos (url, usuario y clave para acceder). Si aparecen los campos 'Value' rellenos es que todo ha ido bien:



17. Seleccionamos la pestaña **Source** del fichero recién creado. Veremos las etiquetas y subetiquetas que ha creado NetBeans por nosotros:

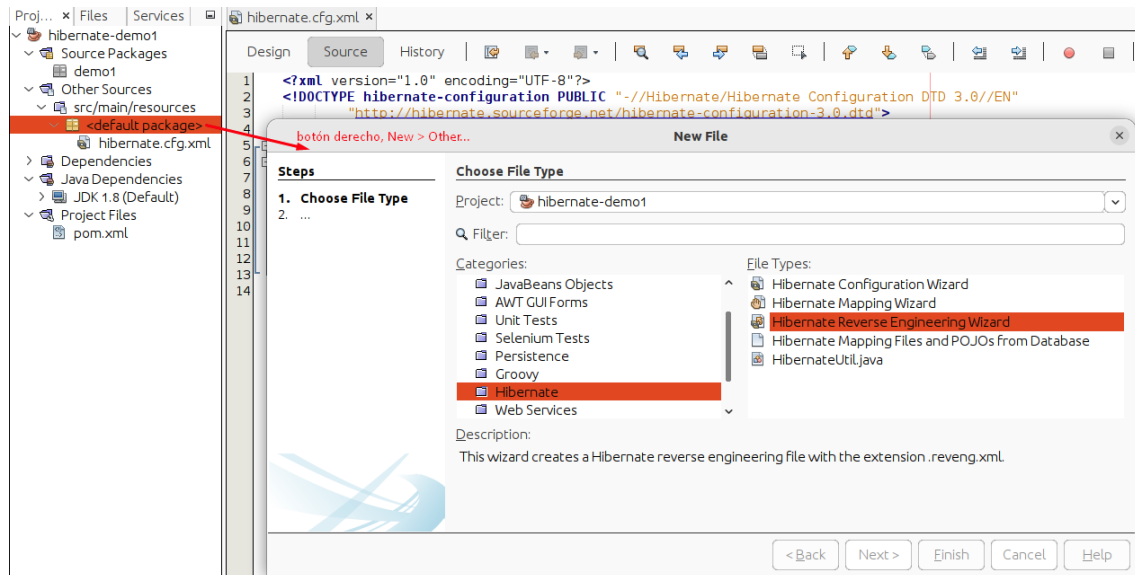


## 1.5. Fichero hibernate.reveng.xml (Reverse Engineering)

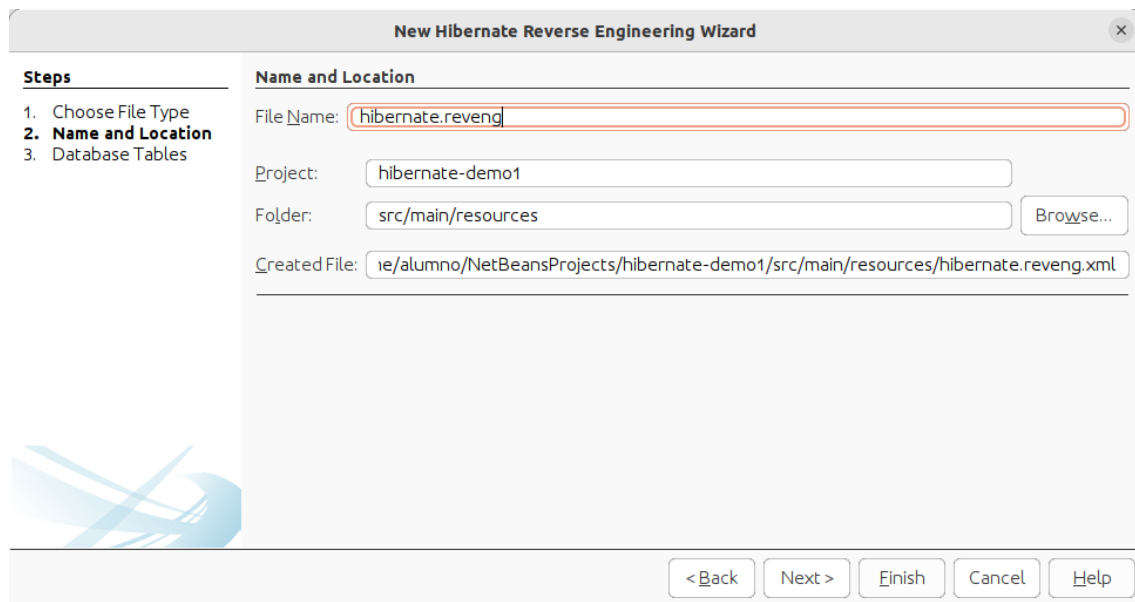
Este archivo es el encargado de crear las clases asociadas a tablas de la base de datos. Indica el esquema (en nuestro caso, ROOT) en el que se encontrarán las tablas a mapear y el nombre de las tablas (en nuestro caso sólo tenemos la tabla LIBRO). Lo crearemos con la herramienta "Hibernate Reverse Engineering Wizard".

Para crearlo se siguen los siguientes pasos:

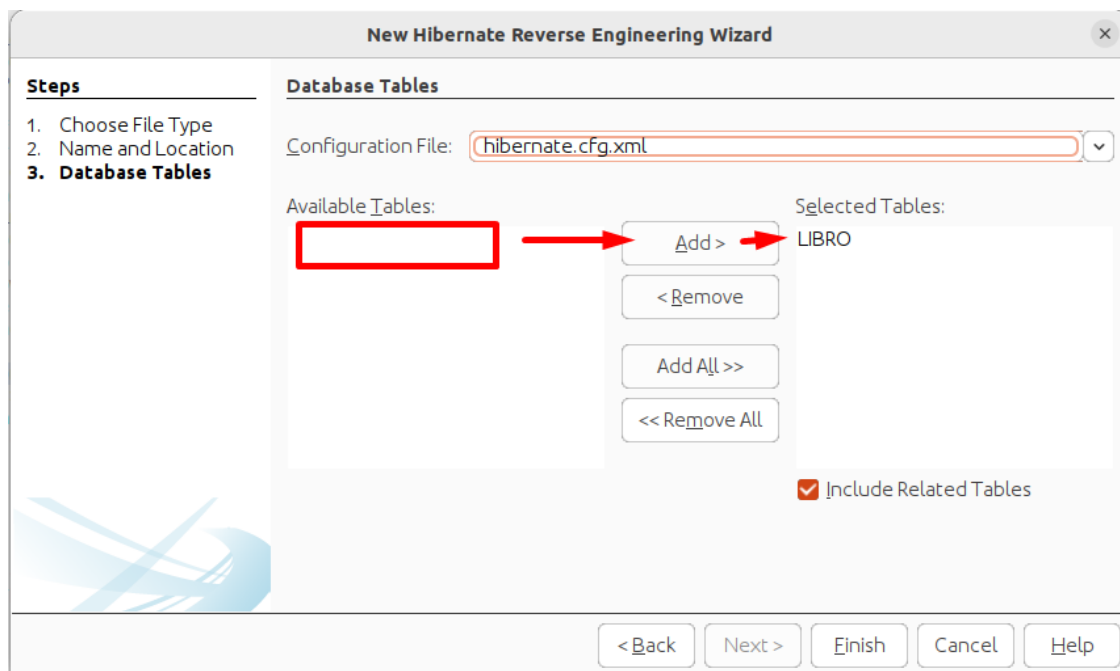
1. Seleccionando el paquete **default package** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y, se selecciona dentro de las posibilidades (otros...), un tipo de archivo **Hibernate Reverse Engineering Wizard** dentro de la carpeta Hibernate.



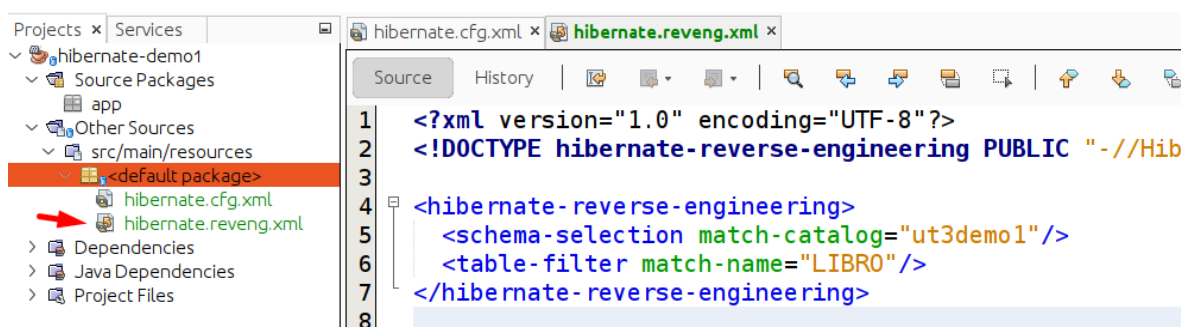
2. Se pulsa Next.



3. El nombre por defecto del fichero será **hibernate.reveng**. Sin cambiar nada se pulsa en Next.
4. En la ventana 'Database Tables' aparecen todas las tablas de la base de datos, para seleccionar aquellas que queremos que se mapeen a clases Java. En nuestro caso debe aparecer la tabla **libro** que creamos al inicio en MySQL Workbench. Seleccionamos la tabla y pinchamos en **Add >** para añadirla al campo 'Selected Tables':



5. Una vez seleccionada, pinchamos Finish. Con estos pasos se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero **hibernate.reveng.xml** con el nombre del esquema/base de datos (ut3demo1) y de la tabla que se desea mapear (LIBRO):



## 1.6. Mapeo de tablas: generación de los POJOS y los ficheros de mapeo

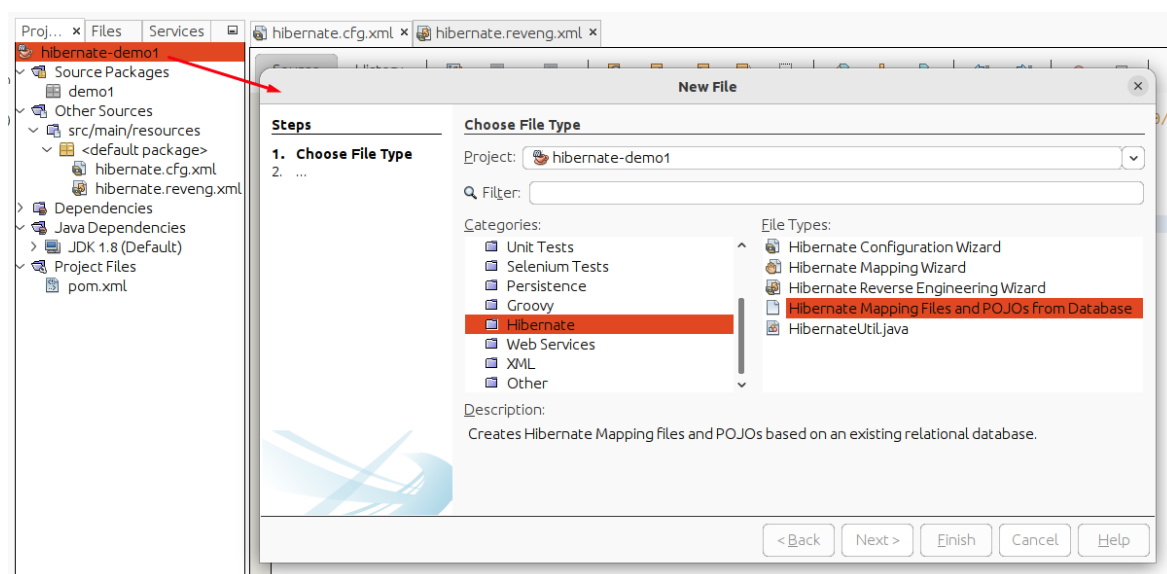
Una vez creados los ficheros hibernate.cfg.xml y hibernate.reveng.xml, procederemos a crear los ficheros necesarios para relacionar nuestros objetos Java con la tabla libro de la base de datos. Para ello utilizamos la herramienta "Hibernate mapping files and POJO from Database", que generará, por cada tabla de la base de datos que queramos mapear, dos ficheros:

- el fichero `<nombre_clase>.java` que contiene la clase generada (POJO). A estas clases se les llama clases persistentes. Son las clases que implementan las entidades del problema, y 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. Tienen unos atributos y unos métodos set y get para acceder a los mismos.
- el fichero `<nombre_clase>.hbm.xml`, que contiene la información del mapeo de la clase a su respectiva tabla de la base de datos.

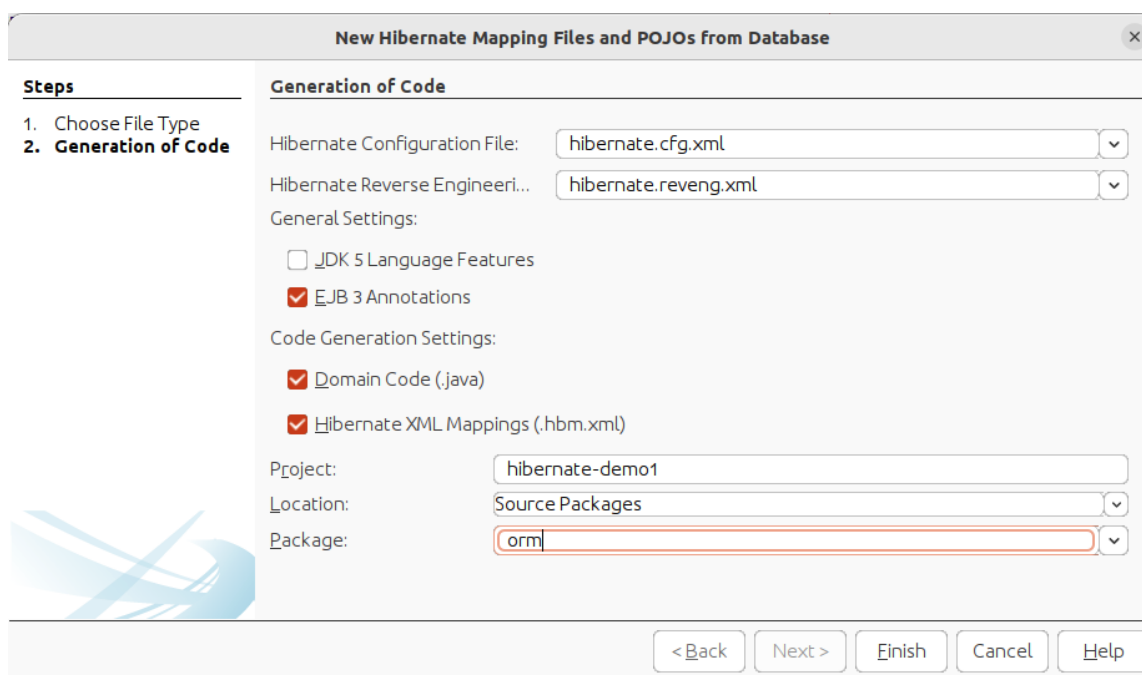
Pasos a seguir:



1. Seleccionando el nombre del proyecto en la vista Projects, pulsamos en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) > Others... > **Hibernate mapping files and POJO from Database** dentro de la carpeta Hibernate.

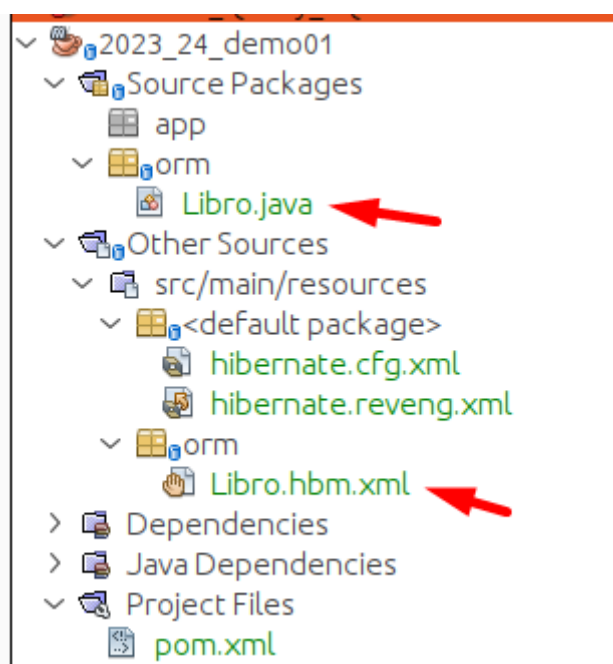


2. Pulsamos Next.
3. En la ventana que aparece se puede observar como ya salen seleccionados los ficheros de configuración creados en los pasos anteriores, que son los que necesita Hibernate para crear la clase Java (POJO, *Plain Old Java Object*) a partir de la tabla **libro** y el fichero de mapeo entre esa clase y la tabla. Para terminar de configurar esta parte solo es necesario indicar el nombre de un paquete en el que almacenar los nuevos ficheros que se van a crear. Para nosotros **orm** (de *Object Relational Mapping*) y marcar (opcional) el checkbox 'EJB 3 Annotations'. Con esta opción los POJOs generados incluirán anotaciones compatibles con el estándar EJB 3 (Enterprise JavaBeans 3) y, por extensión, con JPA (Java Persistence API, para mapear las clases a tablas de la base de datos, en lugar de utilizar exclusivamente archivos XML (como \*.hbm.xml) para realizar el mapeo.



4. Pulsamos Finish.

- Con estos pasos se habrán creado en el paquete **orm** dos ficheros, uno es la clase Java (POJO) obtenida a partir de la tabla libro (Libro.java) y el otro es el fichero de mapeo entre la tabla y la clase (Libro.hbm.xml).



A continuación, puedes ver el código de los 2 ficheros creados por NetBeans.

### **Libro.java**

Aquí se definen los atributos y los métodos getter y setter de la clase Libro.

**Nota:** las líneas que comienzan con '@' son las anotaciones generadas al marcar el checkbox 'EJB 3 Annotations'.

```
package orm;
// Generated 01-ene-2025 11:55:21 by Hibernate Tools 4.3.1

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * Libro generated by hbm2java
 */
@Entity
@Table(name="LIBRO"
        ,catalog="ut3demo1"
)
public class Libro implements java.io.Serializable {

    private int isbn;
    private String titulo;
    private String autor;

    public Libro() {
    }
}
```

```

public Libro(int isbn) {
    this.isbn = isbn;
}
public Libro(int isbn, String titulo, String autor) {
    this.isbn = isbn;
    this.titulo = titulo;
    this.autor = autor;
}

@Id

@Column(name="isbn", unique=true, nullable=false)
public int getIsbn() {
    return this.isbn;
}

public void setIsbn(int isbn) {
    this.isbn = isbn;
}

@Column(name="titulo", length=100)
public String getTitulo() {
    return this.titulo;
}

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

@Column(name="autor", length=100)
public String getAutor() {
    return this.autor;
}

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

```

### Libro.hbm.xml

---

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 01-ene-2025 11:55:22 by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
    <class name="orm.Libro" table="LIBRO" catalog="ut3demo1" optimistic-
lock="version">
        <id name="isbn" type="int">
            <column name="isbn" />
            <generator class="assigned" />
        </id>
        <property name="titulo" type="string">
            <column name="titulo" length="100" />
        </property>
        <property name="autor" type="string">
            <column name="autor" length="100" />
        </property>
    </class>
</hibernate-mapping>

```

Veamos el significado del contenido del fichero Libro.hbm.xml:

- **Hibernate-mapping**: todos los ficheros de mapeo comienzan y acaban con esta etiqueta.
- **class**: esta etiqueta engloba la clase con sus atributos, indicando siempre el mapeo a la tabla de la base de datos. En *name* se indica el nombre de la clase (orm.libro), en *table* el nombre de la tabla a la que representa este objeto (LIBRO), y en *catalog* se indica el nombre de la base de datos (ut3demo1).
- Etiqueta **id** (dentro de class): indica en *name* el campo que representa el atributo clave en la clase (isbn), en *column* su nombre en la tabla (isbn) y en *type* el tipo de datos (int). En id tenemos la propiedad *generator* que indica la naturaleza del campo clave. En este caso es *"assigned"* porque es el usuario el que se encarga de asignar la clave. Si fuese *"increment"* indicaría que es un identificador autogenerado por la base de datos. Este atributo se corresponde con la columna isbn (isbn INTEGER PRIMARY KEY) de la tabla LIBRO.
- El resto de atributos se indican en las etiquetas *property* asociando el nombre del campo de la clase con el nombre de la columna de la tabla y el tipo de datos. Por ejemplo, la columna titulo (titulo VARCHAR(100)) se define así:

```
<property name="titulo" type="string">
  <column name="titulo" length="100" />
</property>
```

Los tipos que aparecen en los ficheros de mapeo no son tipos de datos Java y tampoco son tipos de la base de datos SQL. Estos tipos se llaman tipos de mapeo Hibernate, convertidores que pueden traducir de tipos de datos Java a SQL y viceversa. Hibernate tratará de determinar el tipo correcto de conversión y de mapeo por sí mismo.

### Fichero hibernate.cfg.xml modificado

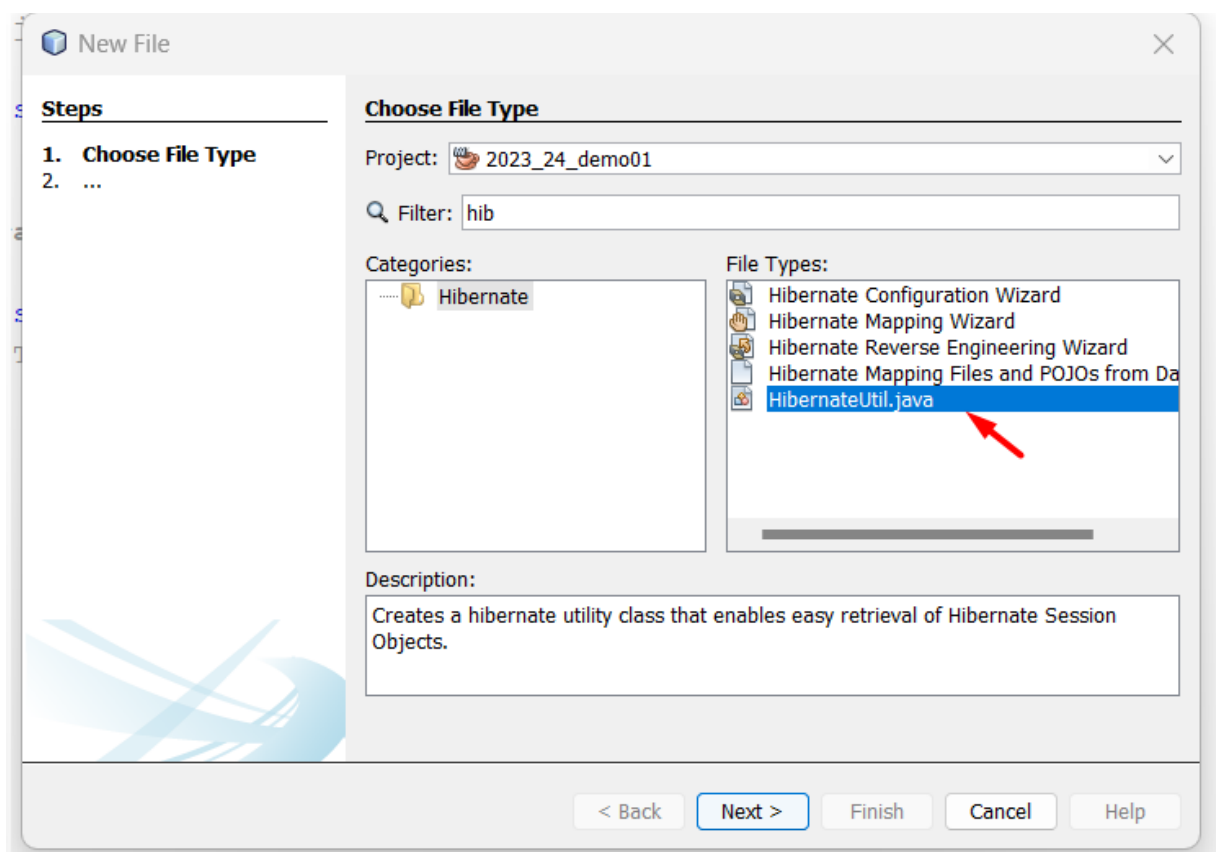
Cabe destacar que en el fichero **hibernate.cfg.xml** se ha añadido automáticamente una nueva línea, con una etiqueta **'mapping'** y un atributo **resource** que apunta al fichero **Libro.hbm.xml** generado:

```
<?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>
    <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/ut3demo1?zeroDateTime
Behavior=convertToNull</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">admin</property>
    <mapping resource="orm/Libro.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

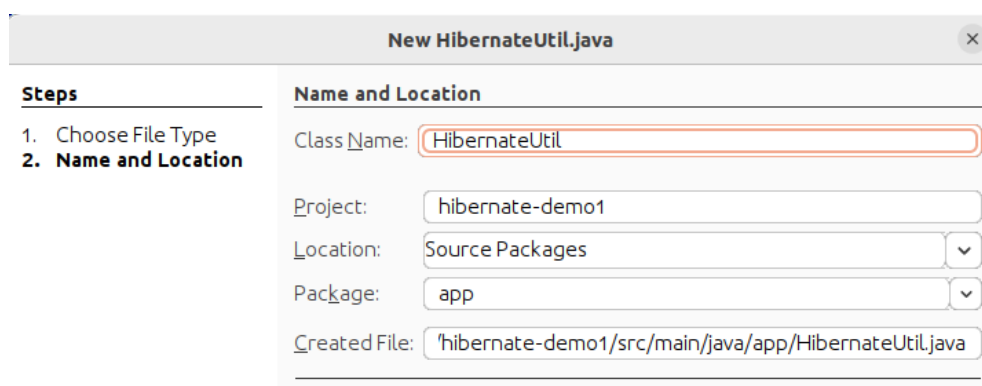
## 1.7. Clase HibernateUtil.java

Este fichero se utiliza para gestionar las conexiones (sesiones en Hibernate) que se hacen a las bases de datos y que permitirán mapear los objetos en las tablas correspondientes. Para crear este fichero se pueden seguir los siguientes pasos.

1. Seleccionando el paquete **app** creado al inicio, se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y, se selecciona, dentro de las posibilidades (otros...), el tipo de archivo HibernateUtil.java.

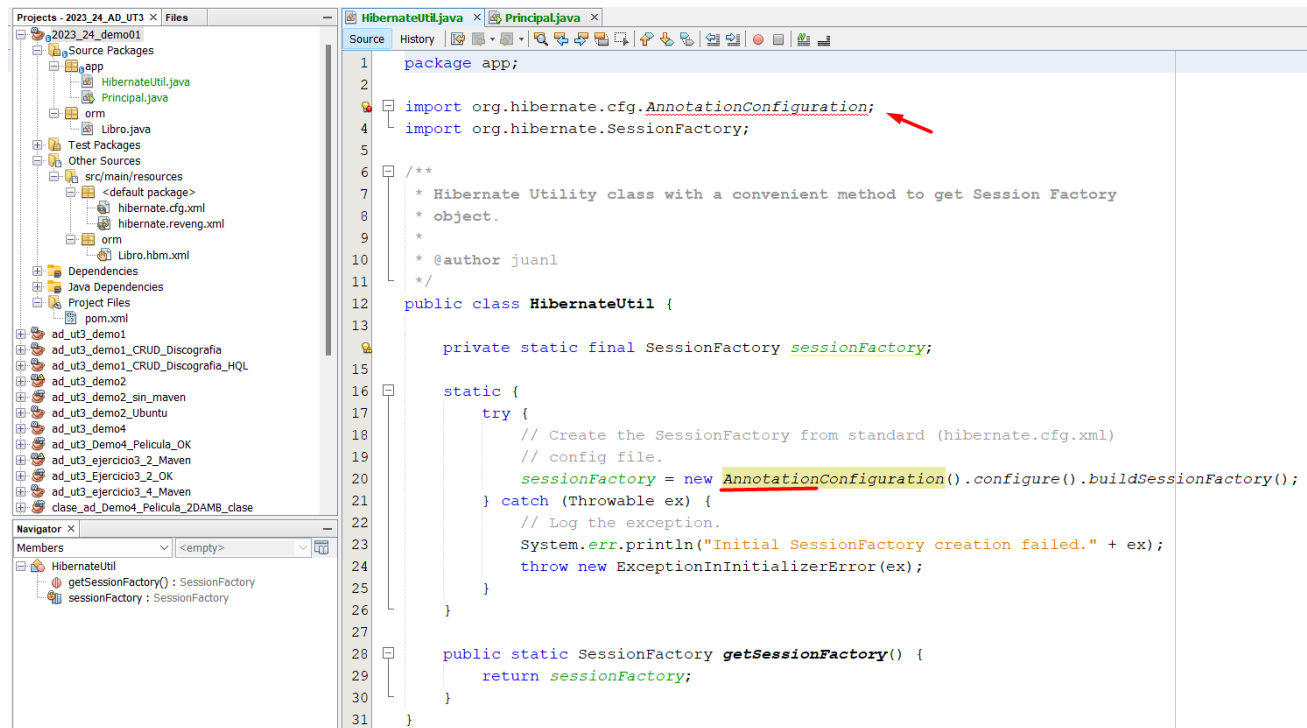


2. Se pulsa en Next.
3. Opcionalmente, cambiamos el nombre por defecto (NewHibernateUtil) por HibernateUtil:



4. Una vez hecho, pinchamos en Terminar.

Con estos pasos se habrá creado un fichero HibernateUtil.java dentro del paquete **app** con el siguiente código:



Es necesario realizar un paso adicional. Dado que la clase `AnnotationConfiguration` quedó obsoleta (*deprecated*), tendremos que sustituirla donde aparezca por `Configuration`. El fichero `HibernateUtil.java` quedaría de la siguiente manera:

```

package app;

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

/**
 * Hibernate Utility class with a convenient method to get Session Factory object.
 *
 * @author alumno
 */
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

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

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

```

Esta clase HibernateUtil sigue el **patrón de diseño Singleton**, donde:

- Se garantiza que solo haya **una instancia de SessionFactory** durante toda la ejecución de la aplicación.
- Se centraliza el proceso de creación, configuración y acceso a la SessionFactory.
- Simplifica el uso de Hibernate en el resto del código, ya que no se requiere inicializar la SessionFactory repetidamente.

### Estructura de la Clase

#### 1. Variable Estática sessionFactory:

- Declarada como `private static final`, lo que significa que será única para toda la aplicación (estática) y no puede cambiarse una vez inicializada (final).
- Es la instancia de SessionFactory que manejará la creación de sesiones (objetos Session) para las transacciones de la base de datos.

#### 2. Bloque static:

- Es un bloque estático que se ejecuta solo una vez cuando se carga la clase en memoria.
- Encierra la lógica de inicialización para la SessionFactory.

#### 3. Inicialización de sessionFactory:

- Utiliza una instancia de Configuration para cargar el archivo hibernate.cfg.xml y construir la SessionFactory.
- Maneja excepciones (Throwable) durante la inicialización y lanza un `ExceptionInInitializerError` en caso de fallos.

#### 4. Método getSessionFactory():

- Método público y estático que devuelve la instancia de SessionFactory.
- Sirve como punto de acceso central para obtener la SessionFactory en otras partes de la aplicación.

El objeto sessionFactory se crea con la instrucción:

```
sessionFactory = new Configuration().configure().buildSessionFactory();
```

donde:

#### 1. new Configuration():

- Crea una nueva instancia de la clase Configuration de Hibernate.
- Esta clase es responsable de leer la configuración de Hibernate (habitualmente desde un archivo hibernate.cfg.xml, como en el caso de nuestro proyecto, o similar) que contiene información como la conexión a la base de datos y las propiedades del mapeo objeto-relacional.

## 2. `configure()`:

- Llama al método `configure()` sobre el objeto `Configuration`.
- Indica a Hibernate que debe cargar las configuraciones predeterminadas. Si no se pasa un archivo de configuración específico como argumento (por ejemplo, `configure("miArchivo.cfg.xml")`), se usará el archivo `hibernate.cfg.xml` ubicado en el classpath del proyecto (como es nuestro caso).
- Retorna la instancia del objeto `Configuration` actual, lo que permite continuar invocando otros métodos de esta clase.

## 3. `buildSessionFactory()`:

- Este método construye una instancia de `SessionFactory`, que es el punto central para manejar sesiones de Hibernate.
- La `SessionFactory` es responsable de crear objetos `Session`, que se utilizan para realizar operaciones de base de datos como consultas y transacciones.

# 1.8. Programa principal

Una vez creados todos los elementos necesarios, vamos a crear el código para conectarnos a la base de datos y añadir un nuevo objeto `Libro` a la base de datos. Pinchamos botón derecho en el paquete "app" y creamos una Java Main Class denominada por ejemplo `Principal`. En esta clase incluimos el método `main`, en el que se van a realizar las siguientes operaciones:

- Crear un objeto `SessionFactory` utilizando el método `getSessionFactory()` de la clase `HibernateUtil`.
- Crear un objeto `Session` con el método `openSession()` de `SessionFactory`
- Crear un objeto `Transaction` con el método `beginTransaction()` de `Session`
- Realizar operaciones con la base de datos. En este caso, añadir un registro a la tabla `Libro`, creando un objeto `Libro` y utilizando el método `save()` de `Session`.
- Hacer el "commit" de la transacción mediante el método `commit()` de `Transaction`.
- Cerrar la sesión con el método `close()` de `Session`.



## Principal.java

---

```
package app;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import orm.Libro;

public class Principal {

    public static void main(String[] args) {

        // Creamos la factoría de sesiones
        SessionFactory sf = HibernateUtil.getSessionFactory();

        // Creamos/Abrimos la sesión
        Session s = sf.openSession();

        // Creamos una transacción
        Transaction t = s.beginTransaction();

        // Hacemos lo que tengamos que hacer con la BDD. En este caso
        // añadir un objeto Libro
        Libro d = new Libro(978013187, "Thinking in Java", "Bruce Eckel");



        // Guardamos el objeto en la base de datos
        s.save(d);

        // Hacemos commit de la transacción
        t.commit();

        // Cerramos la sesión
        s.close();
    }
}
```

Ejecutamos y veremos que en nuestra base de datos se ha creado un nuevo registro en la tabla libro:

1 • `SELECT * FROM ut3demo1.libro;`

Result Grid   Filter Rows:

	isbn	titulo	autor
▶	978013187	Thinking in Java	Bruce Eckel
★	NULL	NULL	NULL

## 2. DEMO 3.1 PARTE 2. OPERACIONES CRUD CON HIBERNATE

Continuamos a partir del apartado anterior (base de datos ut3demo1 y tabla LIBRO). Puedes hacer una copia del proyecto anterior y llamar al nuevo proyecto por ejemplo "hibernate-demo-01\_p2"

### 2.1. Métodos cerrarSessionFactory y cerrarSesion (HibernateUtil.java)

Vamos a añadir 2 métodos estáticos a la clase HibernateUtil.java. El primero lo utilizaremos para cerrar la factoría (fábrica) de sesiones y el segundo para cerrar una sesión.

```
//Método cerrarSessionFactory
public static void cerrarSessionFactory() {
    try {
        if (sessionFactory != null && !sessionFactory.isClosed()) {
            sessionFactory.close();
        }
    } catch (HibernateException he) {
        System.err.println("Error al cerrar SessionFactory: " + he.getMessage());
    }
} //Fin método cerrarSessionFactory

//Método cerrarSesion
public static void cerrarSesion(Session session){
    try {
        if (session != null && session.isOpen() ) {
            session.close();
        }
    } catch (HibernateException he) {
        System.err.println("Error al cerrar Session: " + he.getMessage());
    }
} //Fin método cerrarSesion
```

### 2.2. Nueva clase Operaciones.java (CRUD)

Creamos una nueva clase **Operaciones** en la que incluiremos todo lo que tenga que ver con tareas a realizar con la base de datos.

Primero creamos la clase, sus atributos y un constructor:

```
public class Operaciones {
    //Atributos
    SessionFactory sf;
    Session s;
    Transaction t;

    // Constructor
    public Operaciones() {
        // Creamos la factoría de sesiones
        sf = HibernateUtil.getSessionFactory();
    }
}
```

Como podemos observar, en el constructor de la clase inicializamos/creamos la fábrica de sesiones (SessionFactory) (línea `sf = HibernateUtil.getSessionFactory();`).

Vamos a añadir a continuación a la clase **Operaciones** los métodos que nos van a permitir realizar las operaciones básicas con la base de datos.

### **Método para inicializar los valores de Session y Transaction**

```
//Método para inicializar los valores de Session y Transaction
private void abrirSesionYTransaccion() {
    try {
        // Creamos/Abrimos la sesión
        s = sf.openSession();

        // Creamos una transacción
        t = s.beginTransaction();

    } catch (HibernateException he) {
        System.out.println("Excepción generada en abrir sesión "
            + he.getMessage());
    }
}
```

### **Método para insertar (crear) un Libro**

```
//Método para insertar (crear) un Libro
public void crearLibro(int isbn, String titulo, String autor) {
    abrirSesionYTransaccion();
    try {
        // Hacemos lo que tengamos que hacer con la BDD. En este caso
        // añadir un Libro (Create)
        Libro libro = new Libro(isbn, titulo, autor);
        s.save(libro);

        // Hacemos commit de la transacción
        t.commit();

    } catch (HibernateException he) {
        // Si hay algún error hacemos un rollback
        if (t != null) {
            t.rollback();
        }

        System.out.println("Excepción generada en crear Libro "
            + he.getMessage());
    } finally {
        // Cerramos la sesión
        HibernateUtil.cerrarSesion(s);
    }
}
```

**Método para leer un Libro (facilitando el ISBN)**

```
//Método para leer un Libro (facilitando el ISBN)
public void leerLibro(int isbn) {
    abrirSesionYTransaccion();
    try {

        // Leemos una fila de nuestra tabla
        // Con el metodo get de la clase Session lo guardamos pasándole la clase
        // Libro y el isbn
        // Sería como un SELECT * FROM libro WHERE isbn=?.
        Libro libro = s.get(Libro.class, isbn);

        System.out.println(libro); // Muestra el objeto (=fila de la tabla)
        System.out.println("Autor: " + libro.getAutor()); // Muestra sólo el autor

    } catch (HibernateException he) {

        System.out.println("Excepción generada en leer Libro "
            + he.getMessage());
    } finally {

        // Cerramos la sesión
        HibernateUtil.cerrarSesion(s);
    }
}
```

**Método para modificar un Libro (facilitando el ISBN)**

```
//Método para modificar un Libro (facilitando el ISBN)
public void modificarLibro(int isbn, String nuevoTitulo, String nuevoAutor) {
    abrirSesionYTransaccion();
    try {
        // Leemos el Libro a modificar y lo almacenamos en l
        Libro l = s.get(Libro.class, isbn);

        // Una vez leído modificamos el título y el autor
        l.setTitulo(nuevoTitulo);
        l.setAutor(nuevoAutor);

        //Otra manera de hacerlo es creando un objeto con los datos nuevos.
        //Esta forma solo funciona si el nuevo objeto tiene el mismo isbn,
        //ya que este actúa como clave para Hibernate.
        // Libro l = new Libro(isbn, nuevoTitulo, nuevoAutor);

        //Actualizamos la modificación.
        //Hibernate registra la actualización del objeto y lo sincroniza con la base de datos.
        s.update(l);

        //Hacemos commit de la transacción
        t.commit();

    } catch (HibernateException he) {
        System.out.println("Excepción generada en modificar Libro "
            + he.getMessage());
        if (t != null) {
            t.rollback();
        }
    } finally {
        //Cerramos la sesión
        HibernateUtil.cerrarSesion(s);
    }
}

} //Fin metodo modificarLibro
```

## Notas:

## 1. Sincronización con la Base de Datos.

En Hibernate, un objeto recuperado de la base de datos (como Libro l) está asociado a la sesión. Los cambios realizados en él se reflejan automáticamente en la base de datos al hacer commit, incluso sin llamar explícitamente a `s.update(l)`.

## 2. Manejo de Nulos

Si no se encuentra un libro con el isbn proporcionado, `s.get` devolverá null. Este caso debería manejarse en el código explícitamente, por ejemplo:

```
if (l == null) {  
    throw new IllegalArgumentException("No existe un libro con el ISBN: " + isbn);  
}
```

**Método para borrar un Libro (facilitando el ISBN)**

```
//Método para borrar un Libro (facilitando el ISBN)  
public void borrarLibro(int isbn) {  
    abrirSesionYTransaccion();  
    try {  
        // Almacenamos el objeto que queremos borrar (isbn recibido como parámetro)  
        Libro d = s.get(Libro.class, isbn);  
  
        // Borramos el Libro con el metodo delete de Session  
        s.delete(d);  
  
        // Hacemos commit de la transacción  
        t.commit();  
    } catch (HibernateException he) {  
        // Si hay algún error hacemos un rollback  
        if (t != null) {  
            t.rollback();  
        }  
        System.out.println("Excepción generada en borrar Libro "  
            + he.getMessage());  
    } finally {  
        // Cerramos la sesión  
        HibernateUtil.cerrarSesion(s);  
    }  
}
```

## 2.4. Programa principal ejemplo

```
package app;

import java.util.Scanner;

public class Principalv2 {
    public static void main(String[] args) {
        Operaciones operaciones = new Operaciones();

        //Probamos los métodos creados en la clase Operaciones.
        System.out.println("----- CREAMOS 3 LIBROS -----");
        operaciones.crearLibro(2, "Java, How to program", "Deitel");
        operaciones.crearLibro(3, "título 3", "autor 3");
        operaciones.crearLibro(4, "título 4", "autor 4");

        System.out.println("----- LEEMOS LIBRO POR ID -----");
        operaciones.leerLibro(1);

        System.out.println("----- MODIFICAR LIBRO -----");
        //Modificamos el libro, pidiendo al usuario el título y el autor.
        Scanner sc = new Scanner (System.in).useDelimiter("\n");

        System.out.print("Introduce el ISBN del libro que quieres modificar: ");
        int isbn = sc.nextInt();
        System.out.print("Introduce el nuevo título: ");
        String nuevoTitulo = sc.next();
        System.out.print("Introduce el nuevo autor: ");
        String nuevoAutor = sc.next();

        operaciones.modificarLibro(isbn, nuevoTitulo, nuevoAutor);

        operaciones.leerLibro(isbn); // Comprobamos la modificación

        System.out.println("----- BORRAR LIBRO -----");
        //Borramos un libro pidiendo al usuario el isbn del libro que se quiere borrar.
        System.out.print("Introduce el ID del libro que quieres borrar: ");
        isbn = sc.nextInt();

        operaciones.borrarLibro(isbn);

        // NO OLVIDAR. Cerramos la factoría de sesiones
        HibernateUtil.cerrarSessionFactory();
    }
}
```