Índice

Introducción

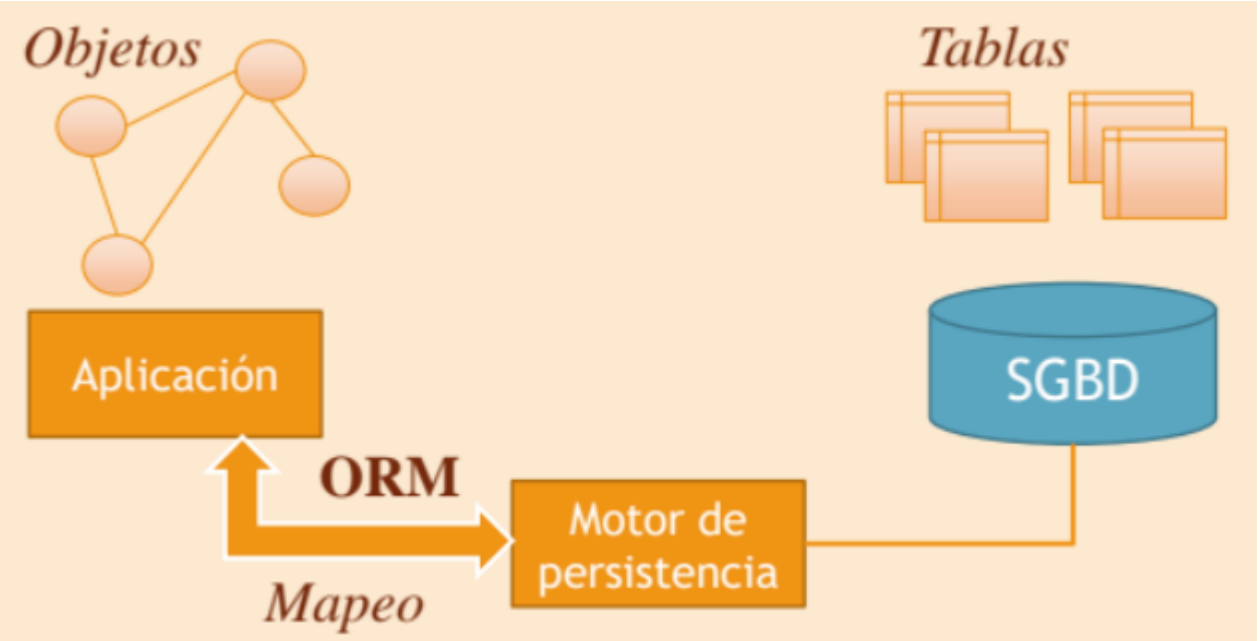
Veremos como acceder a una base de datos relacional utilizando el lenguaje orientado a objetos Java. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesario una interfaz que traduzca la lógica de los objetos a la lógica relacional.

Esta interfaz se llama **ORM (Object Relactional Mapping)** y es la herramienta que nos sirve para **transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales a representaciones de objetos**. Es decir:

- Las tablas de nuestra base de datos, pasan a ser las clases.
- Las filas de la tabla (o Registros) serán los objetos que podremos manejar.

> **Object-Relational Mapping (O/RM - ORM - O/R mapping)**
> *Wikipedia: "Es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilziando un motor de persistencia*

En la práctica, esto crea una base de datos orientada a objetos virtual, que **posibilita el uso de las características de la orientación a objetos** (Herencia y polimorfismo)



Herramientas ORM. Características

Las herramientas ORM nos permitirán **crear una capa de acceso a datos**. Una forma sencilla de hacerlo, es crear una clase por cada una de las tablas de la base de datos y mapearlas una a una.

Estas herramientas también nos aportan un **lenguaje de consultas orientado a objetos propio** y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, simplemente cambiando algunas líneas en los ficheros de configuración.

Ventajas:

- Ayudan a reducir el tiempo de desarrollo de software.
- Abstracción de la base de datos.
- Reutilización.
- Permiten la producción de mejor código.
- Son independientes de la base de datos. (Funcionan en cualquier BD).
- Lenguaje propio para realización de consultas.
- Incentivan la portabilidad y escalabilidad de los programas de software.

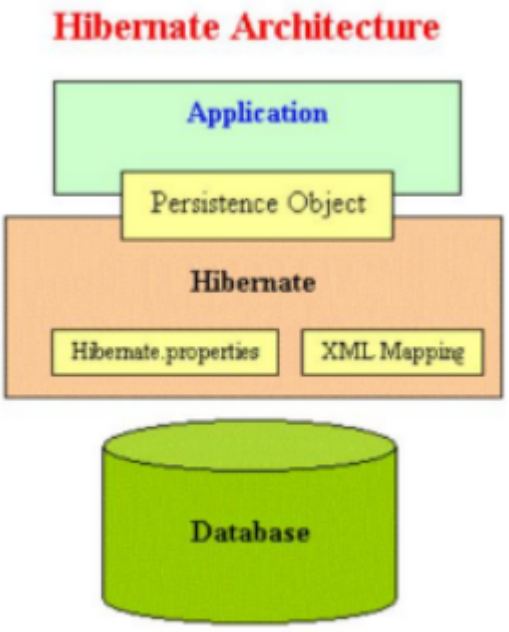
Inconvenientes:

- El principal inconveniente es que las aplicaciones **son algo más lentas** debido a que todas las consultas que se hagan sobre la base de datos, deben **transformarse al lenguaje propio de la herramienta, leer los registros y crear los objetos con los que trabajaremos**.

Herramientas ORM. Hibernate

Existen muchas herramientas ORM para los distintos lenguajes de programación (Doctrine, Propel, ADOdb, Active Record, ...). **Para la tecnología Java, se ha desarrollado Hibernate, que es software libre bajo licencia GNU LGPL**, aunque existen otros como QuickDB, iPersist, Java Data Objects, Oracle Toplink, etc.

> Hibernate, es uno de los ORM más populares y permite definir el mapeo de atributos entre una base de datos relacional y el modelo orientado a objetos, **mediante ficheros declarativos (XML)** que establecen las relaciones.

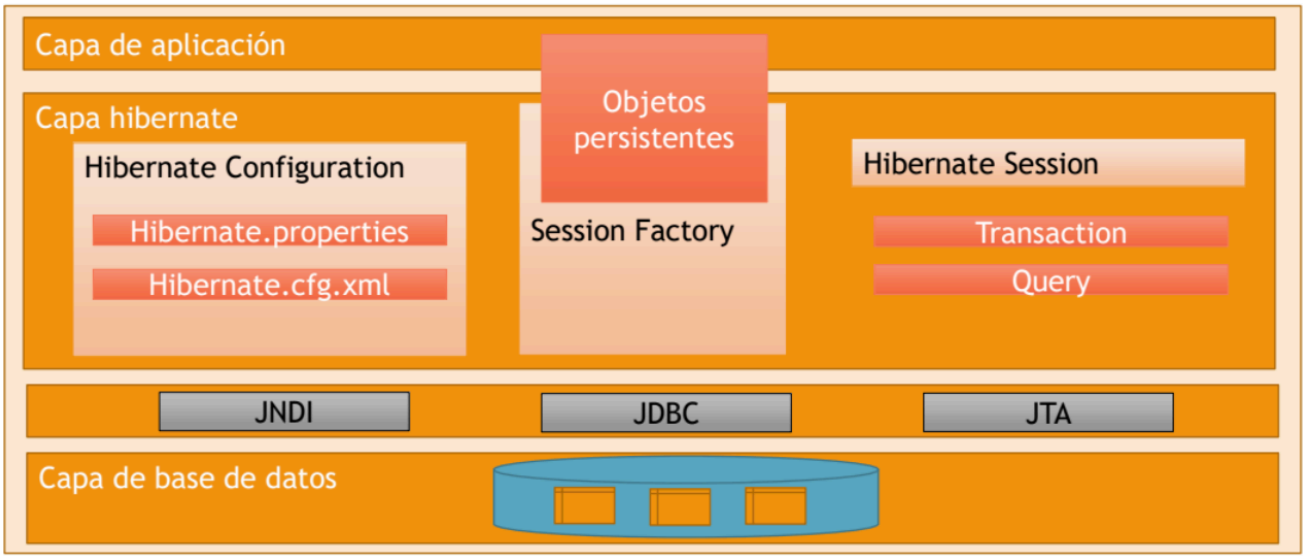


Con Hibernate no emplearemos habitualmente el lenguaje SQL para acceder a los datos, sino que el propio motor de Hibernate, **mediante el uso de factorías** (Patrón de diseño Factory) y otros elementos de programación, construirá esas consultas para nosotros.

Hibernate pone a disposición del desarrollador un lenguaje llamado **HQL (Hibernate Query Language)** que permite acceder a los datos mediante **Programación Orientada a Objetos**.

Arquitectura Hibernate

Hibernate parte de una filosofía de mapear objetos Java normales o más conocidos como **POJO's (Plain Old Java Objects)**. Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial **sesión (clase Session)**. Esta sesión se equipararía al concepto de conexión JDBC, por lo que **debemos crear y cerrar la sesión igualmente**.



La clase Session (org.hibernate.Session)

Esta clase ofrece métodos para interactuar con la base de datos como **save (Object objeto)**, **createQuery (String consulta)**, **beginTransaction()**, **close()**, etc.

Una instancia de `Session` no consume mucha memoria y su creación y destrucción es muy "barata". Esto es importante, ya que nuestras aplicaciones **van a necesitar crear y destruir sesiones casi en cada petición**. Puede ser útil pensar en una sesión como en una **caché o colección de objetos cargados**, además Hibernate **puede detectar cambios en los objetos** pertenecientes a una unidad de trabajo.

Interfaz	Descripción
SessionFactory (`org.hibernate.SessionFactory`)	Permite crear instancias **Session**. Esta interfaz se comparte entre muchos hilos de ejecución, ya que normalmente **hay una única SessionFactory para toda la aplicación**, creándola al inicio de la misma. Si la aplicación accede a distintas bases de datos, será necesario una SessionFactory para cada base de datos.
Configuration (`org.hibernate.cfg.Configuration`)	Se utiliza para **configurar Hibernate**. Utiliza la instancia de `Configuration` para especificar la ubicación de los documentos que indican el mapeado de los objetos y las propiedades

Interfaz	Descripción
	específicas de Hibernate. A continuación, se creará la `SessionFactory`.
Query (`org.hibernate.Query`)	Permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escribirán en **HQL** o en el dialecto **SQL nativo** de la base de datos que estemos utilizando. Una instancia de `Query` se utilizará para **enlazar los parámetros de consulta, limitar el número de resultados y para la ejecución de la consulta**.
Transaction (`org.hibernate.Transaction`)	Permite controlar que cualquier error entre el inicio y el final de la transacción **produzca el fallo de la misma**.

Instalación y configuración de Hibernate

Durante el siguiente apartado, veremos cómo instalar y configurar Hibernate en el IDE Eclipse, utilizando un SGBD como MySQL. Partiremos por tanto de la base de datos ejemplo.

> **Nota:** Debemos usar los parámetros de usuario y contraseña definidos para las conexiones anteriores. (root/12345678)

La definición de nuestra base de datos era:

```
CREATE DATABASE ejemplo;

CREATE TABLE ejemplo.departamentos (
    dept_no    TINYINT NOT NULL PRIMARY KEY,
    dnombre    VARCHAR (15),
    localidad  VARCHAR (15)
)ENGINE=InnoDB;

CREATE TABLE ejemplo.empleados (
    emp_no      SMALLINT NOT NULL PRIMARY KEY,
    apellido    VARCHAR (10),
    puesto      VARCHAR (10),
    fecha_alta  DATE,
    salario     FLOAT,
    variable    FLOAT,
    dept_no     TINYINT NOT NULL,
    CONSTRAINT FK_DEP FOREIGN KEY (dept_no) REFERENCES departamentos
(dept_no)
)ENGINE=InnoDB;
```

Instalación y configuración de Hibernate.

Pasos previos

Antes de empezar:

- Es recomendable actualizar el software de Eclipse de vez en cuando (si no tenemos proyectos a los que pueda afectar).
- Verificaremos que tenemos instalada una versión JDK o instalaremos una.

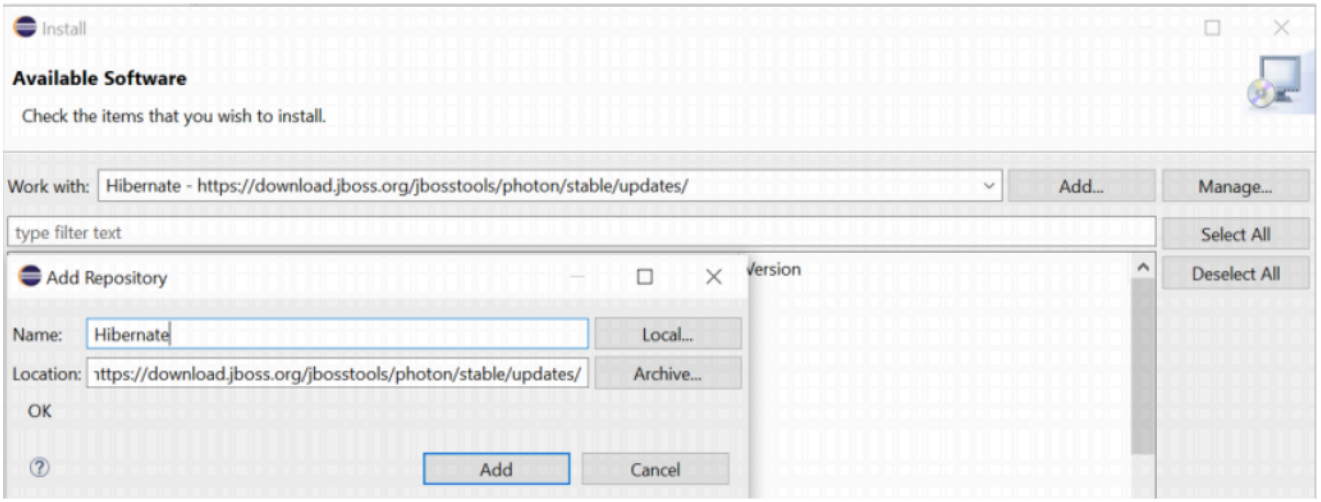
Para instalar una versión de JDK:

1. Descargamos la última versión de https://www.oracle.com/java/technologies/downloads/ e instalamos (en nuestro caso la 17).
2. Vamos a Eclipse y entramos en el menú **Windows ➔ Preferences ➔ Java** y seleccionamos la opción **Installed JREs**.
   Pulsamos **Add** y en el cuadro de diálogo elegimos **Standard VM**.
3. En la opción **JRE home** seleccionaremos la carpeta donde hemos instalado el JDK y pulsaremos **Finish**.
4. Marcaremos el check de nuestro **JDK** y pulsamos en **Apply**.
5. En la opción **Compiler** elegiremos en **Compiler compliance level**, la versión de nuestro JDK (en nuestro caso la 17) y pulsamos en **Apply and Close**.

Instalación del plugin (I)

Para el proceso de instalación del plugin, vamos a necesitar tener conexión a internet. Haremos esta instalación desde el propio Eclipse:

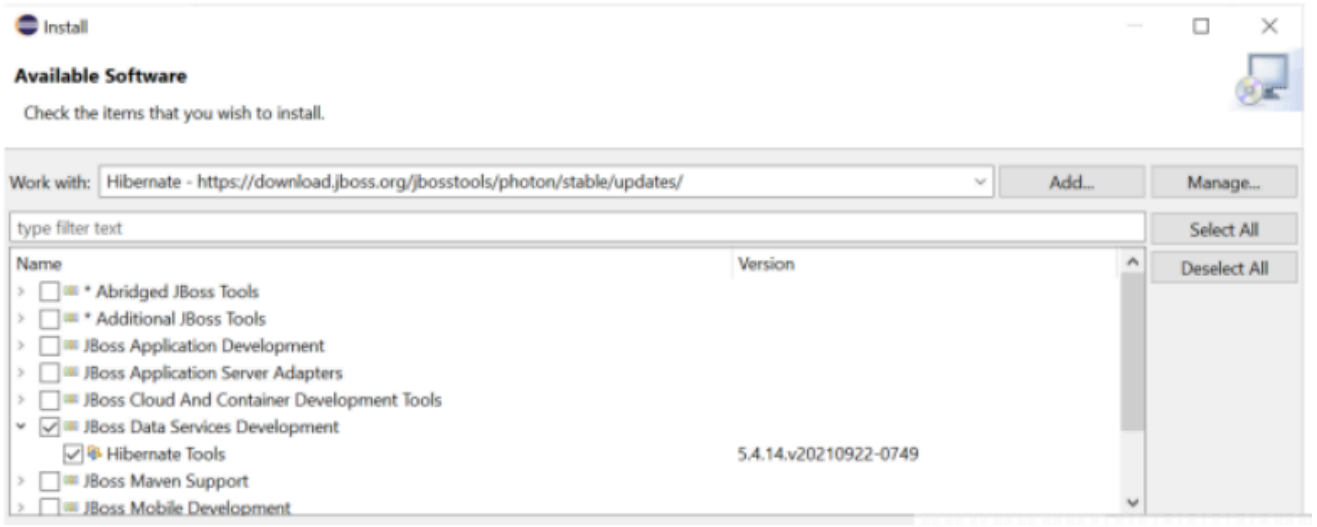
- Menú **Help ➔ Install New Software**.
- Rellenaremos el campo **Work With** con la URL:
  https://download.jboss.org/jbosstools/photon/stable/updates/
  y pulsamos el botón **Add**.
- Pondremos el nombre, por ejemplo, **Hibernate**, y pulsamos **OK**.



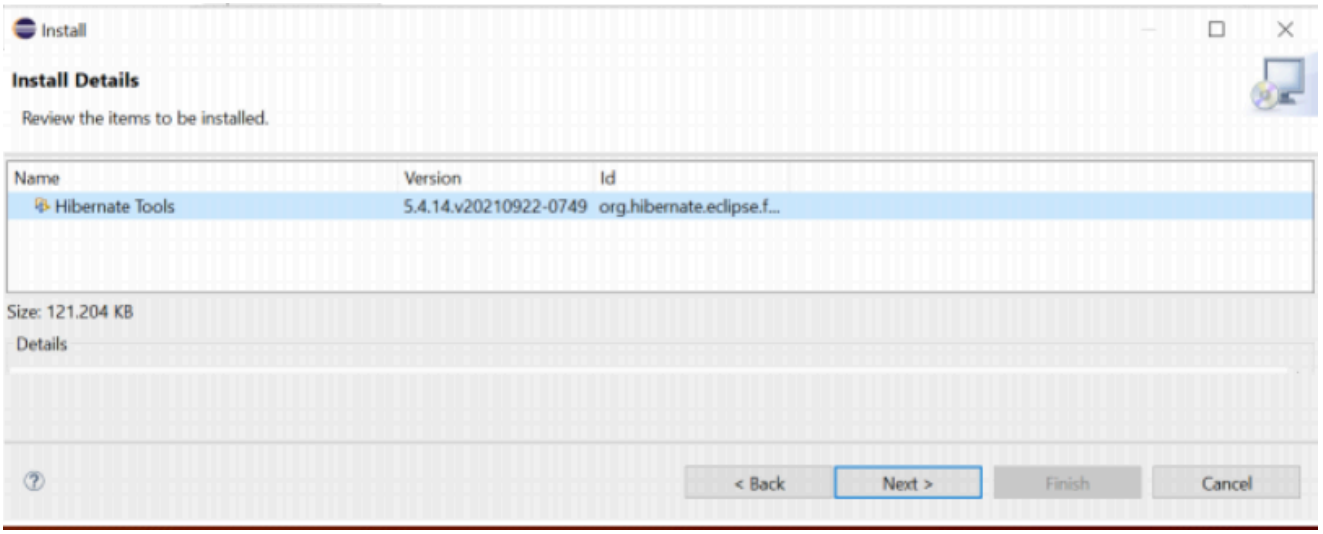
Nota: La URL de descarga dependerá de la versión de Eclipse que tengamos instalada.
Debemos descargar la última versión estable de JBoss para nuestro IDE de Eclipse.

Instalación del plugin (II)

1. Elegiremos en la sección **JBoss Data Services Development**, la opción **Hibernate Tools** y pulsamos **Next**.



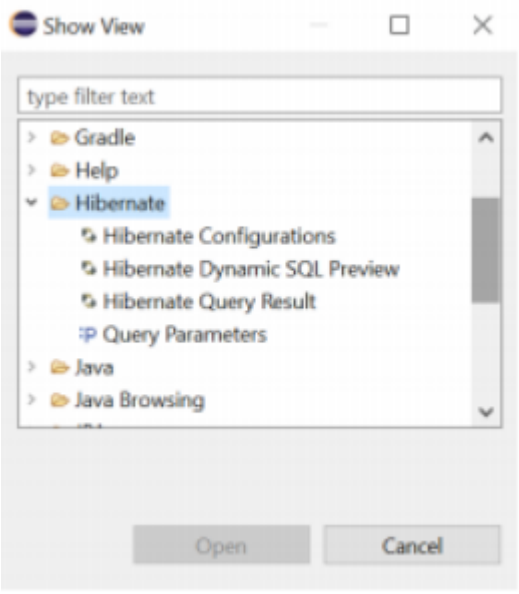
2. Una vez descargado, se muestra una ventana con los detalles del elemento a instalar y pulsamos **Next** de nuevo, aceptando el acuerdo de licencia.



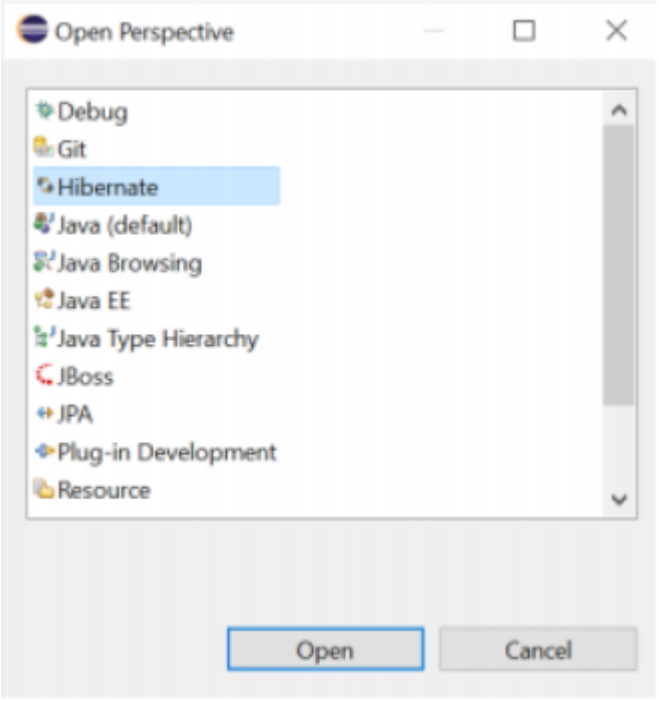
3. Durante el proceso de instalación, nos pedirá confirmación para continuar ya que el plugin contiene software sin firmar. Aceptaremos y, una vez instalado, procederemos a reiniciar Eclipse, tal como nos solicita.

Instalación del plugin (III)

Para verificar la correcta instalación del plugin, una vez reiniciado Eclipse, podemos ir al menú **Windows ➔ Show View ➔ Other**, donde deben aparecer las opciones de Hibernate.



También lo podemos comprobar desde el menú **Windows ➔ Perspective ➔ Open Perspective ➔ Other ➔ Hibernate**, donde debe aparecer la perspectiva de Hibernate.

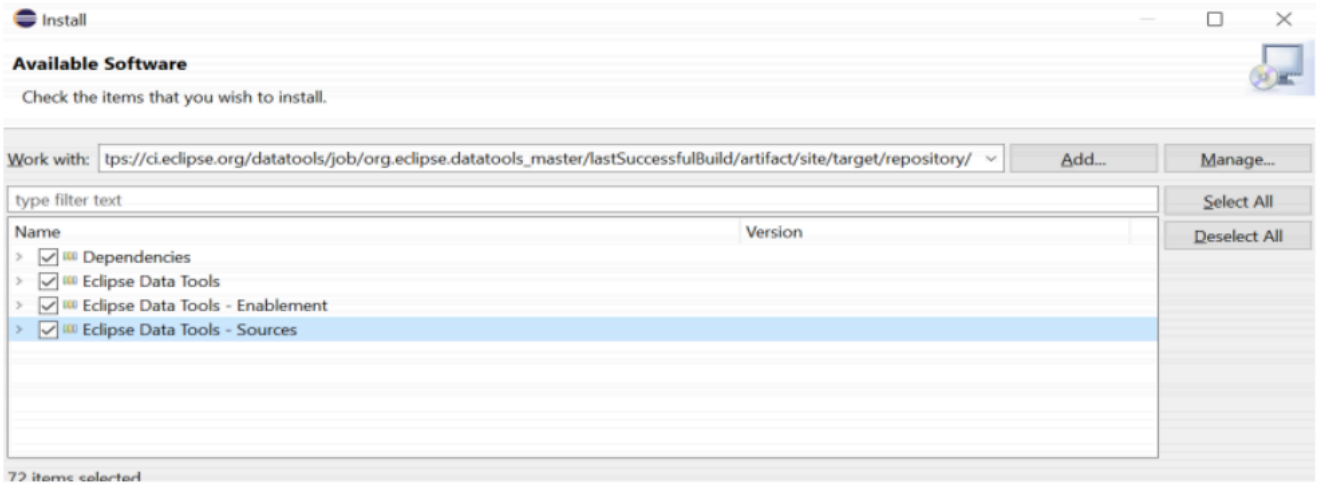


Por el momento no necesitamos abrir ninguna vista o perspectiva, simplemente estamos verificando la instalación.

Instalación del DTP

Para el proceso de instalación del Data Tools Platform, seguiremos los pasos de instalación anteriores:

- Menú **Help ➔ Install New Software**.
- Rellenaremos el campo **Work With** con la URL: https://ci.eclipse.org/datatools/job/build/job/master/lastSuccessfulBuild/artifact/site/target/repository/ y pulsamos el botón **Add**.
- Pondremos el nombre, por ejemplo **DTP**, y pulsamos **OK**. En este caso, instalaremos **todas las opciones disponibles**:



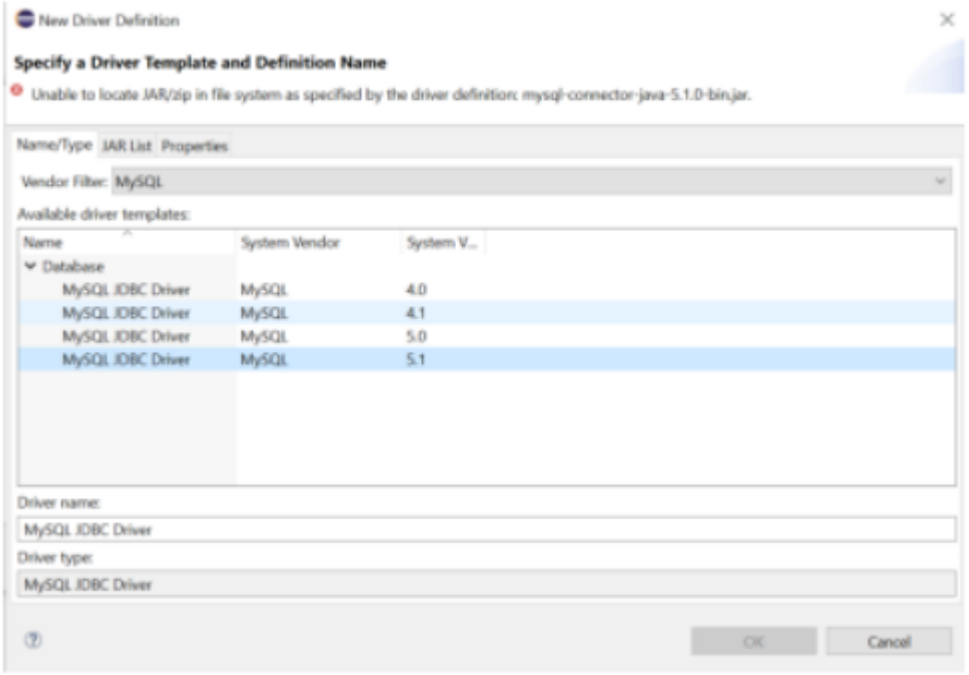
> **Nota:** La URL de descarga dependerá de la versión de Eclipse que tengamos instalada. Debemos descargar **la última versión estable de DTP para nuestro IDE de Eclipse**.

Configuración del driver MySQL (I)

Una vez instalado Hibernate, el siguiente paso es configurarlo para que se comunique con MySQL. En primer lugar, debemos añadir el driver MySQL que ya utilizamos en la conexión JDBC.

Para ello, iremos al menú **Window ➔ Preferences ➔ Data Management ➔ Connectivity ➔ Driver Definitions** y pulsamos el botón **Add**.

Seleccionamos en el **Vendor Filter "MySQL"** y elegimos el último driver disponible de la lista. Después pinchamos sobre la pestaña **JAR List**, donde añadiremos el `.JAR` de nuestro MySQL.



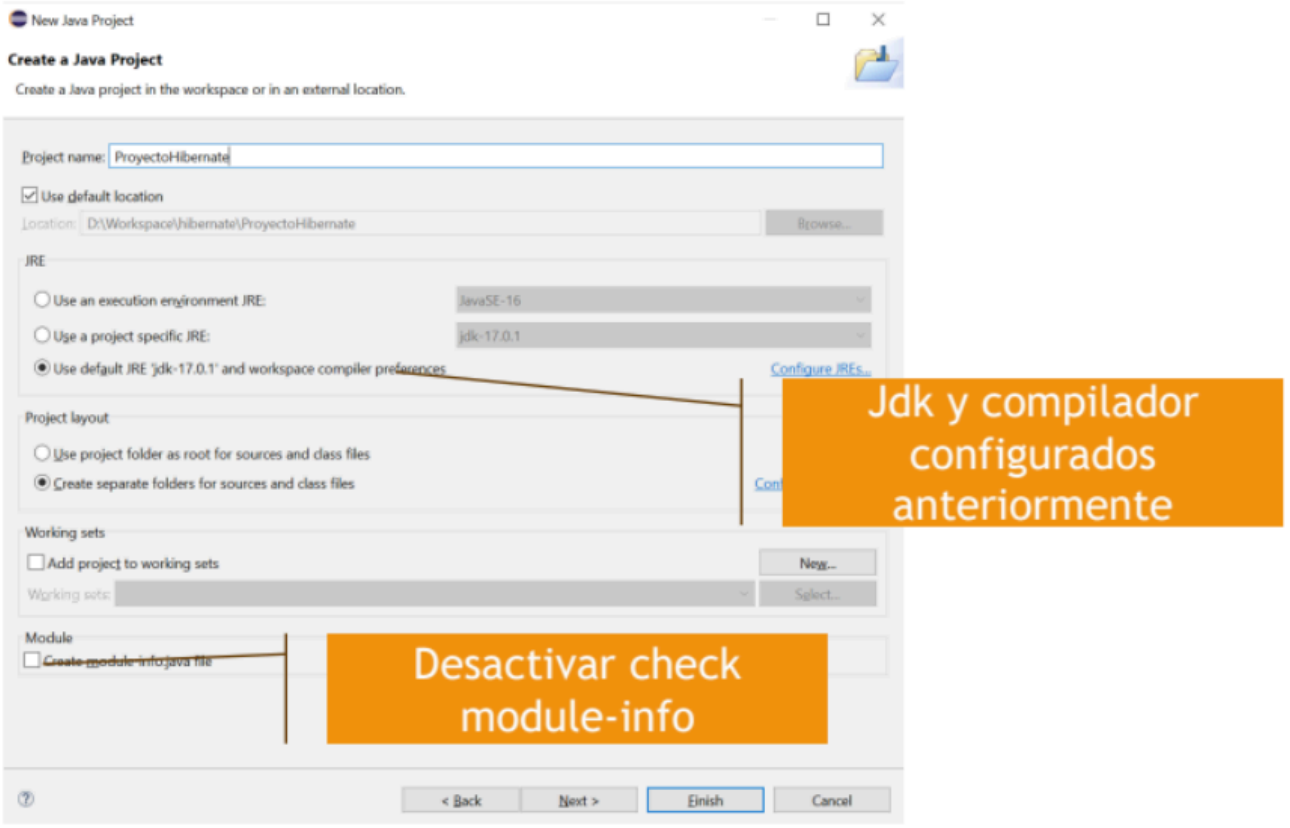
> **Nota:** Eliminaremos las versiones anteriores con el botón **Clear All** antes de añadir nuestro `.jar`.

Configuración del driver MySQL (II)

El siguiente paso será crear un proyecto Eclipse, por ejemplo, **"ProyectoHibernate"**, desde el que nos comunicaremos con MySQL.

- Menú **File ➔ New ➔ Project ➔ Java Project**.

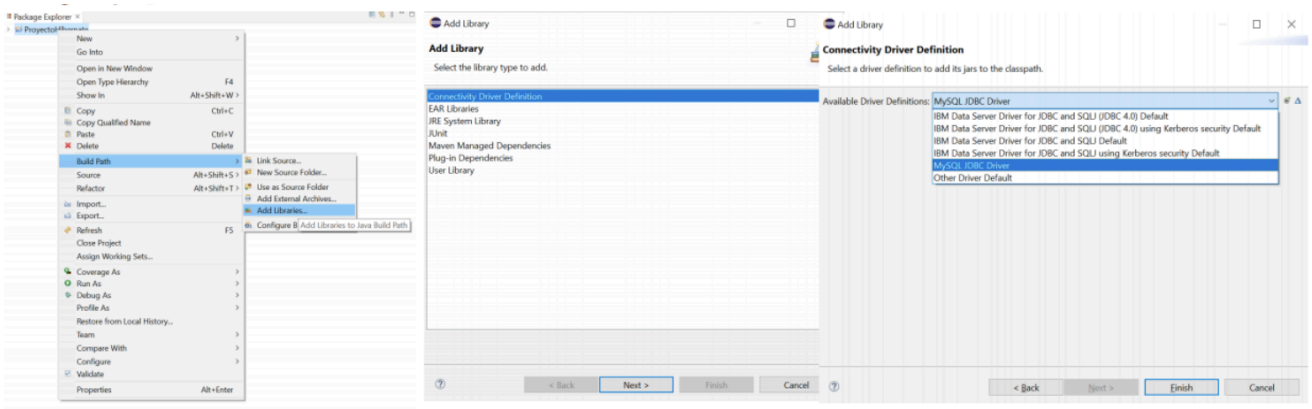
> **Nota:** Elegiremos **usar el JDK y la configuración del compilador por defecto** (establecidos en los pasos previos).



Configuración del driver MySQL (III)

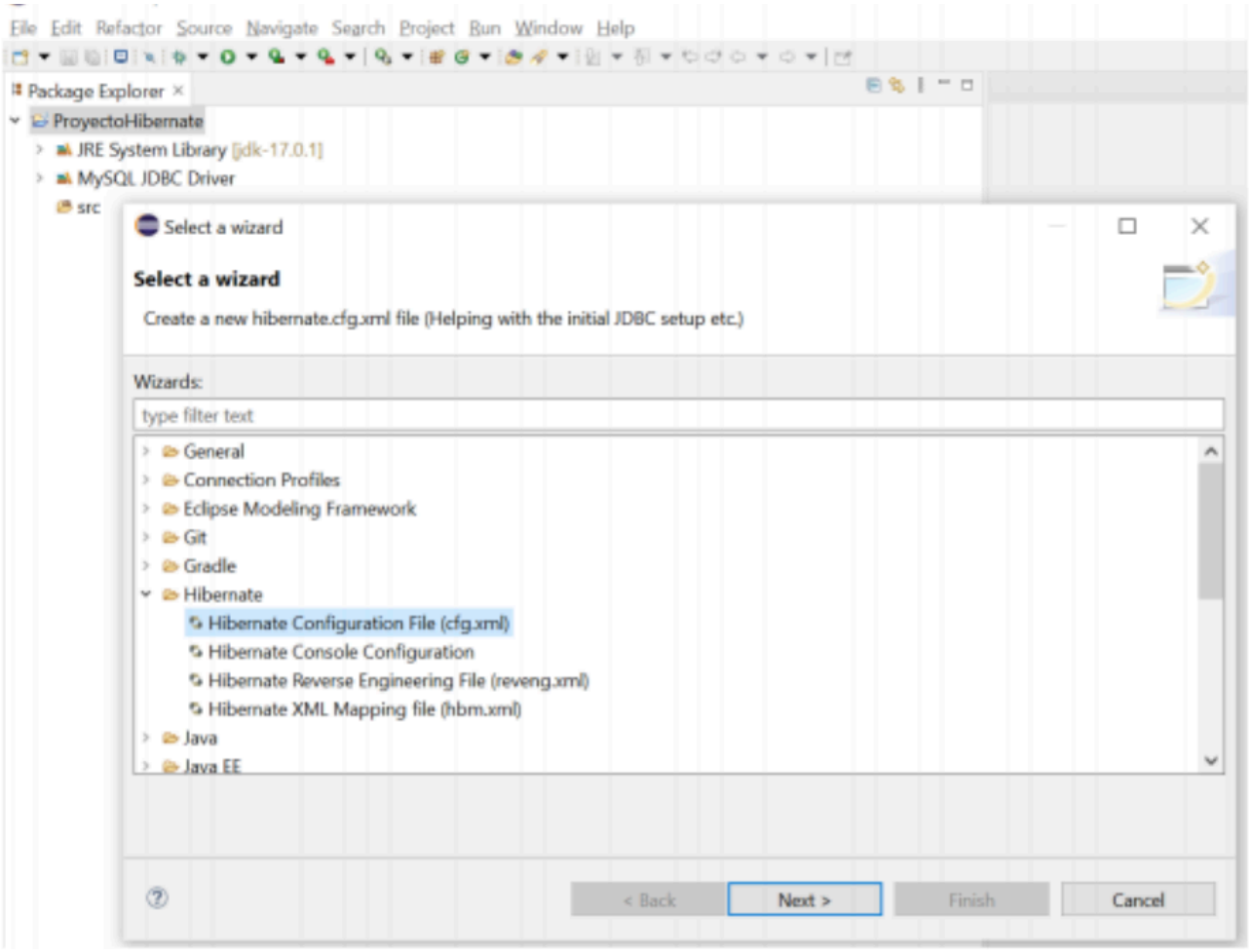
Una vez creado el nuevo proyecto, debemos agregar el driver MySQL, seleccionando en nuestro proyecto con el botón derecho **Build Path ➔ Add Libraries**.

En la ventana que aparece, seleccionaremos **Connectivity Driver Definition** y pulsaremos **Next**. En la ventana se mostrará una lista de opciones disponibles para conectarnos a una fuente de datos, de la que debemos elegir **MySQL JDBC Driver**.

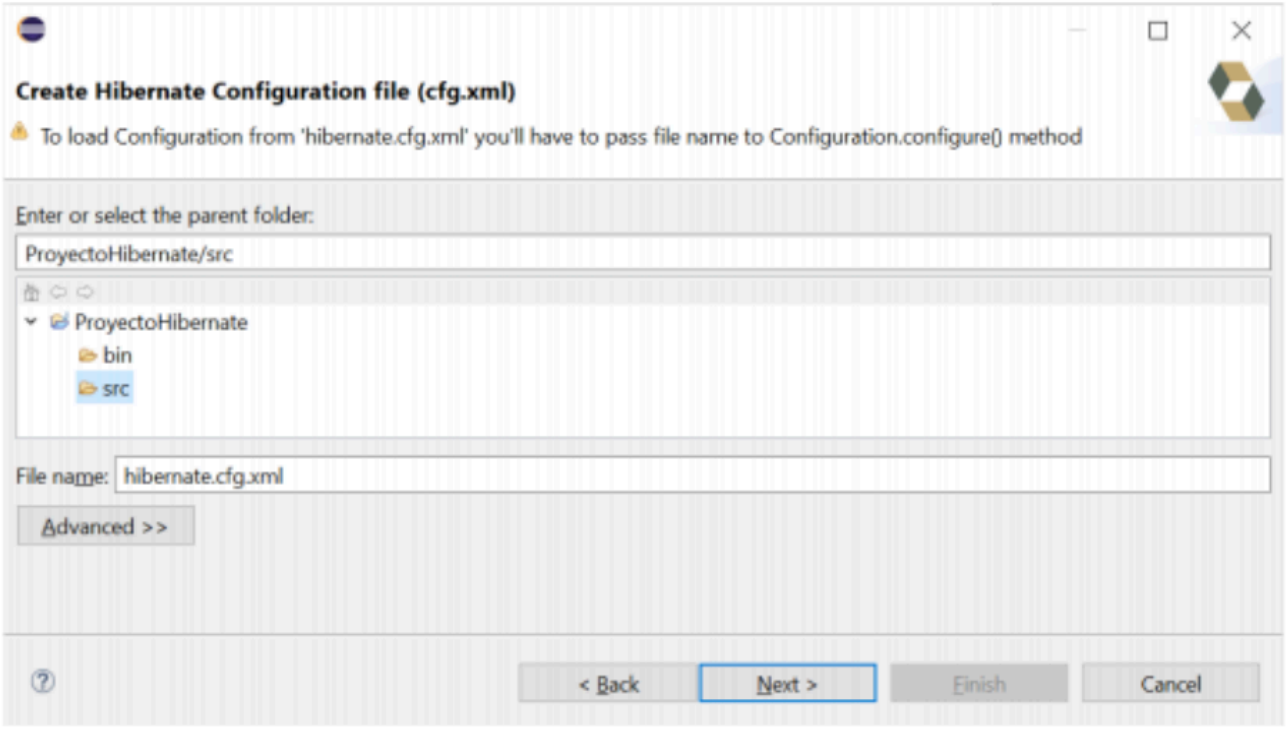


Configuración de Hibernate - *Hibernate Configuration File (hibernate.cfg.xml)*

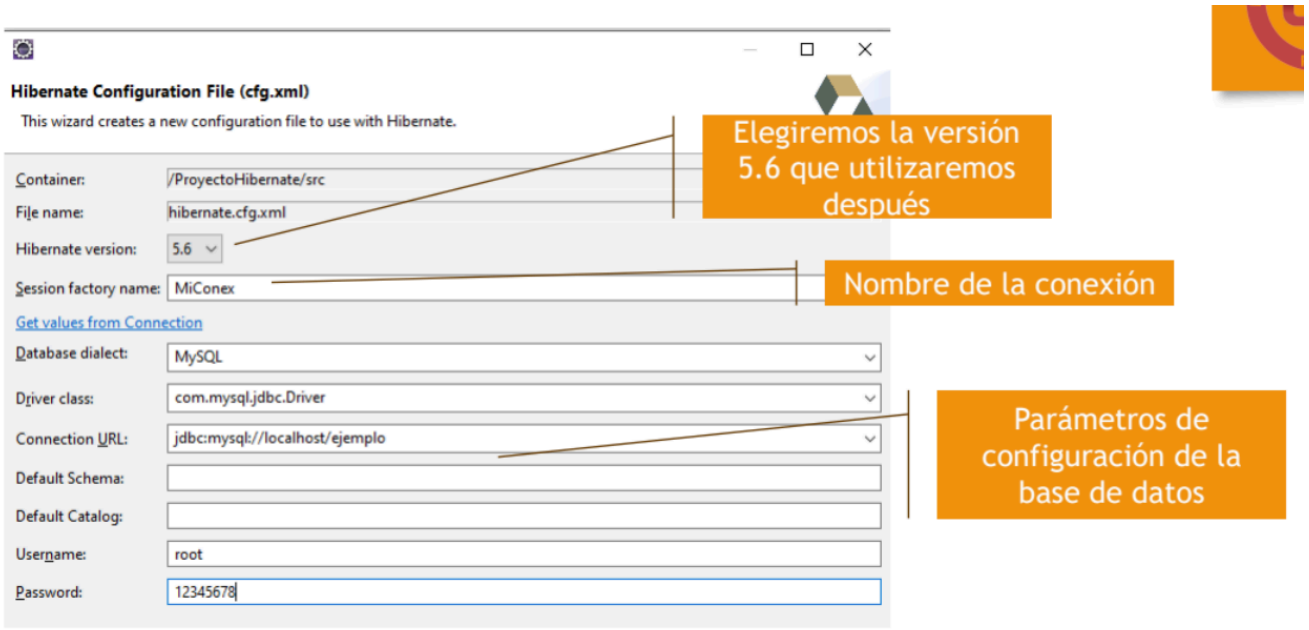
Una vex tenemos la librería (jdbc driver) en nuestro proyecto, hemos de crear el fichero de configuración llamado **hibernate.cfg.xml**. Para ello, seleccionamos con el botón derecho nuestro proyecyo y entramos en el menú **New -> Other -> Hibernate -> Hibernate Configuration File (cfg.xml)**. Este fichero es un XML que contiene todo lo necesario para la conexión con la base de datos.



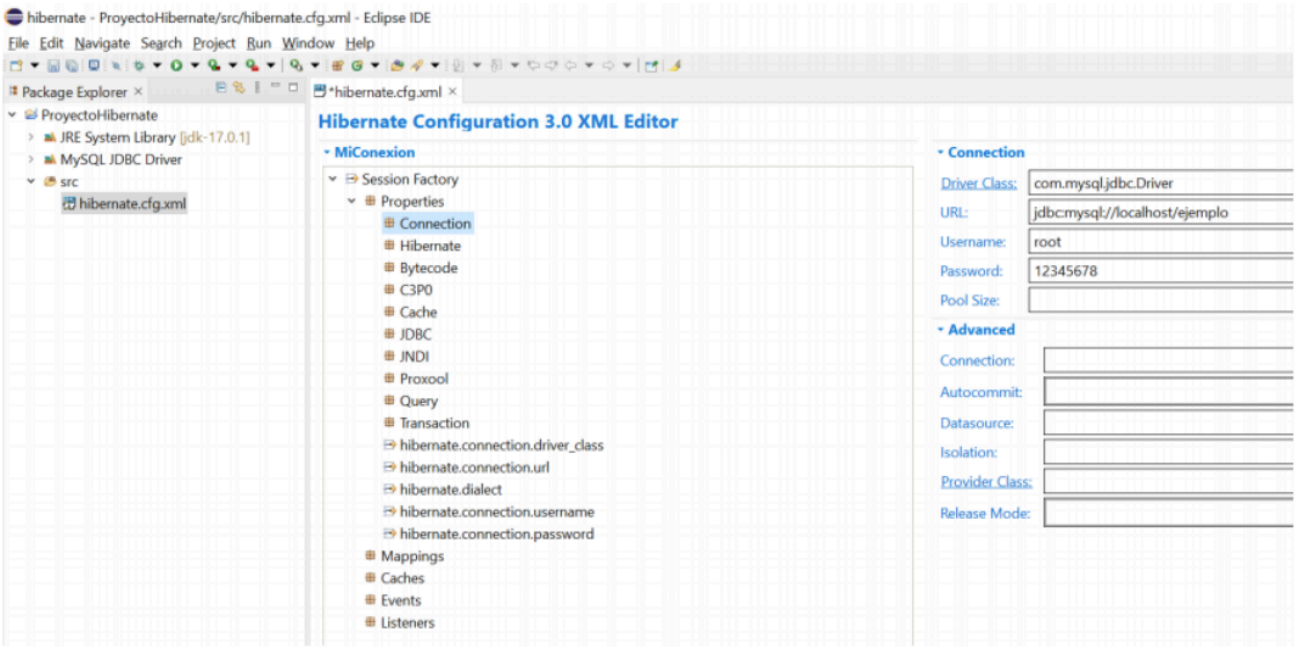
A continuación nos pedirá donde guardar el fichero y, en nuestro caso, lo guardaremos en la carpeta **src**.



En este paso, configuraremos los datos de conexión a nuestra base de datos:



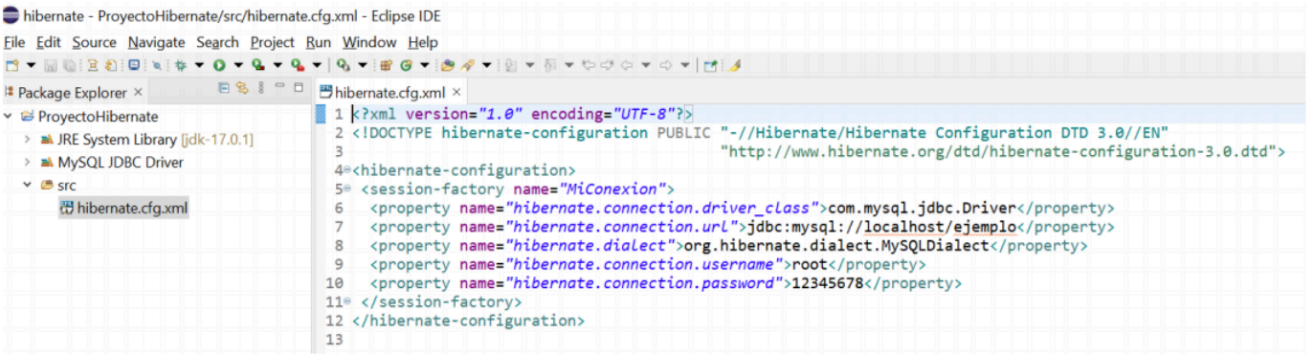
Dependiendo de la versión y el procedimiento que sigamos para completar la conexión en el fichero de configuración, nos pedirá el usuario y la contraseña. También podemos configurar este editando el fichero y seteando los campos en el parámetro **Connection**.



En resumen, los campos que debemos rellenar **para configurar la conexión** en el fichero `hibernate.cfg.xml` son:

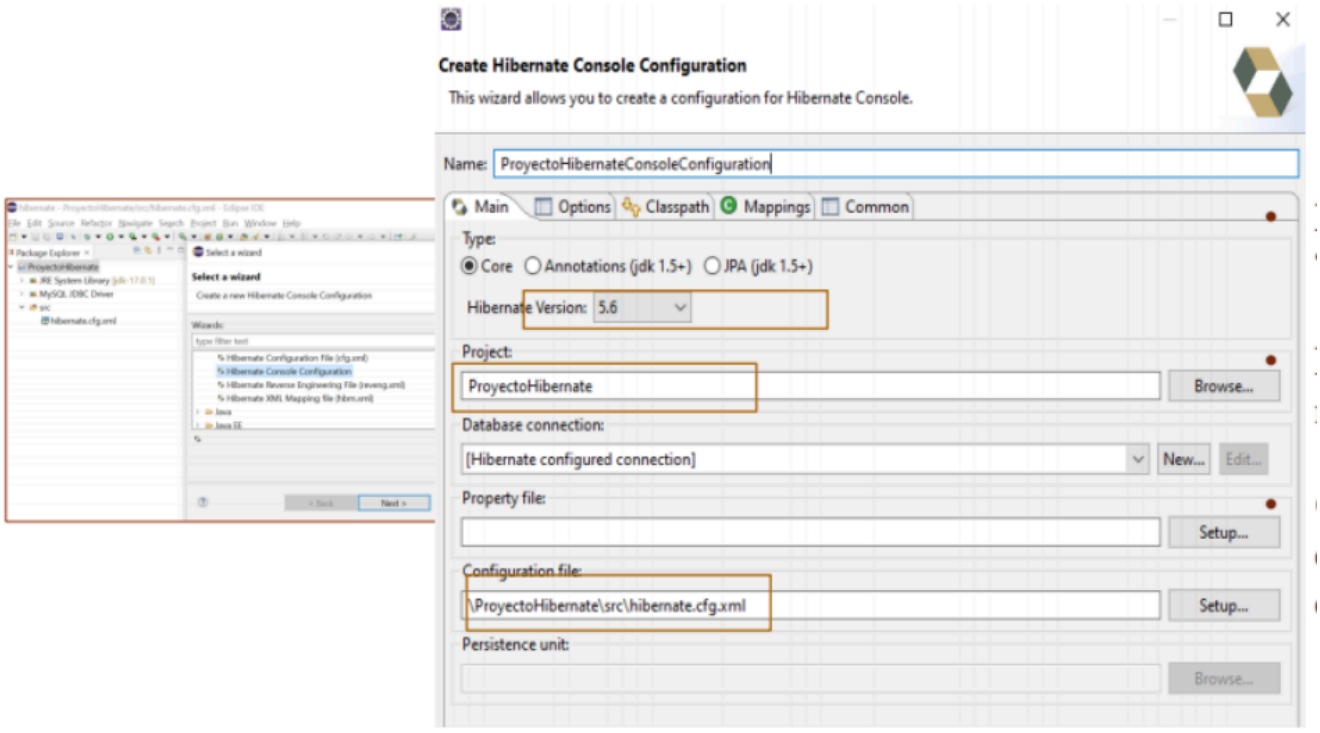
- **Session factory name:** Aquí se escribe el nombre con el que identificaremos nuestra conexión a MySQL, en este caso, hemos establecido **Miconexion**.
- **Database dialect:** Desde esta lista se elige cómo se comunica JDBC con la base de datos, en nuestro caso será **MySQL**.
- **Driver Class:** Se selecciona la clase de JDBC que se va a usar para la conexión, donde elegiremos **com.mysql.jdbc.Driver**.
- **Connection URL:** Se elige la ruta de conexión a nuestra base de datos. Podemos elegir patrones desde el desplegable para completar nuestra conexión. Nosotros utilizaremos: `jdbc:mysql://localhost/ejemplo`.
- **Username:** Usuario con el que conectaremos a la base de datos. En nuestro caso utilizaremos **root**.
- **Password:** Clave del usuario con el que queremos conectar a la base de datos. En este caso **12345678**.

Desde la pestaña "Source", podemos ver como quedaría el fichero de configuración XML



Configuración de Hibernate - *Hibernate Console Configuration*

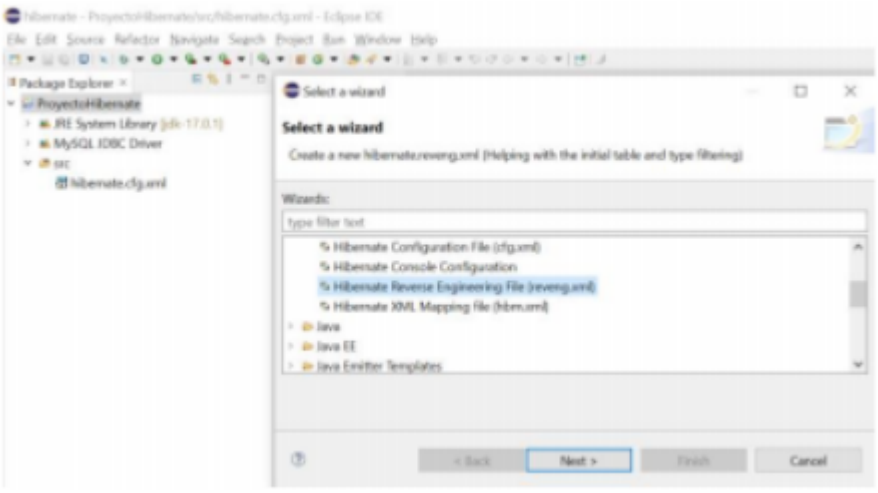
Una vez creado el fichero `hibernate.cfg.xml`, hemos de crear el fichero **XML Hibernate Console Configuration**.
Seleccionamos nuestro proyecto con el botón derecho y elegimos:
New ➔ Other ➔ Hibernate ➔ Hibernate Console Configuration.



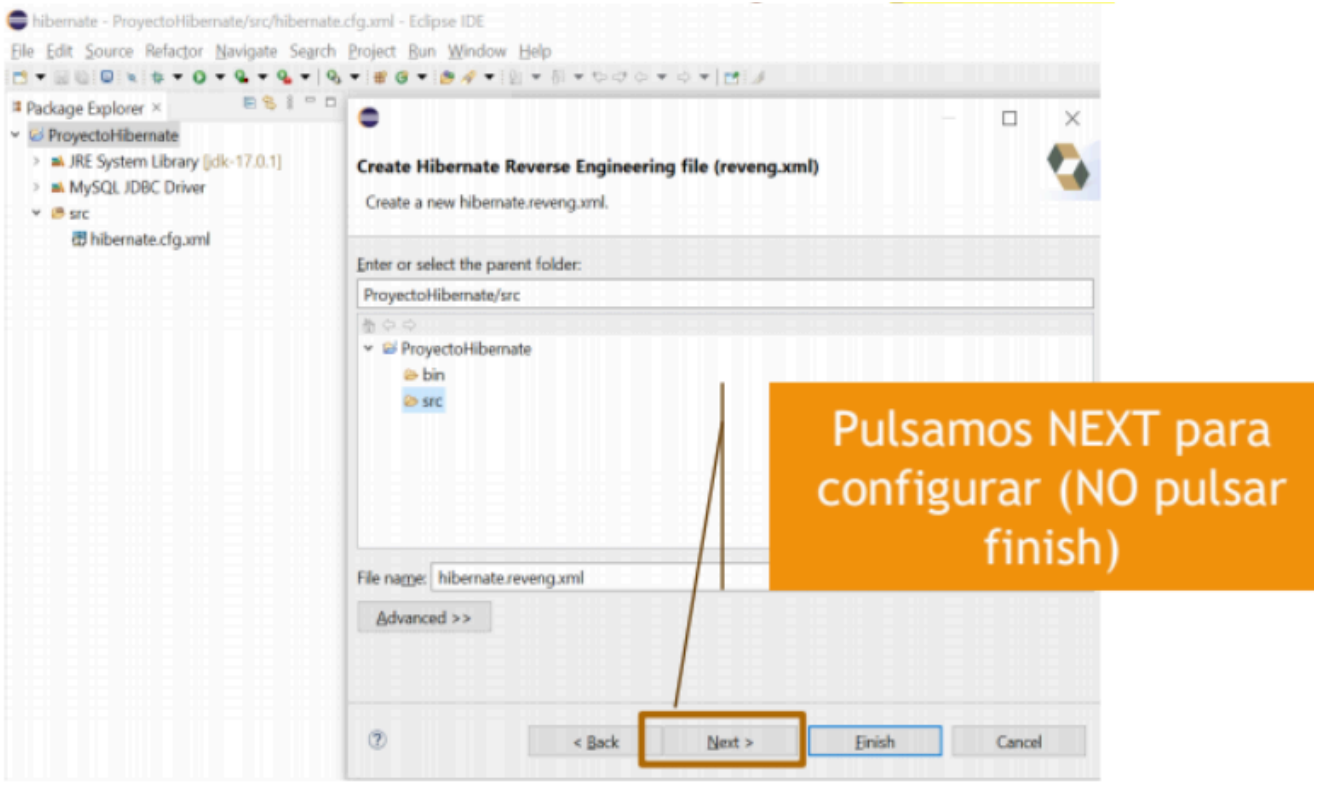
- **Name:** Por ejemplo, **ProyectoHibernateConsoleConfiguration**.
- **Project:** Verificamos que corresponde con nuestro proyecto **"ProyectoHibernate"**.
- **Configuration File:** Verificamos que corresponde con el fichero que acabamos de crear en el paso anterior.

Configuración de Hibernate - *Hibernate Reverse Engineering (hibernate.reveng.xml)*

Por último, debemos crear el fichero **XML Hibernate Reverse Engineering (reveng.xml)**. Este fichero es el encargado de crear las clases de nuestra tablas MySQL.
Seleccionamos el menú: **New ➔ Other ➔ Hibernate ➔ Hibernate Reverse Engineering (reveng.xml)**.



Nota: Este fichero debemos guardarlo en **el mismo lugar que el fichero de configuración**, es decir, en nuestro caso **"src"**. Una vez configurado, pulsaremos el botón **Next**.

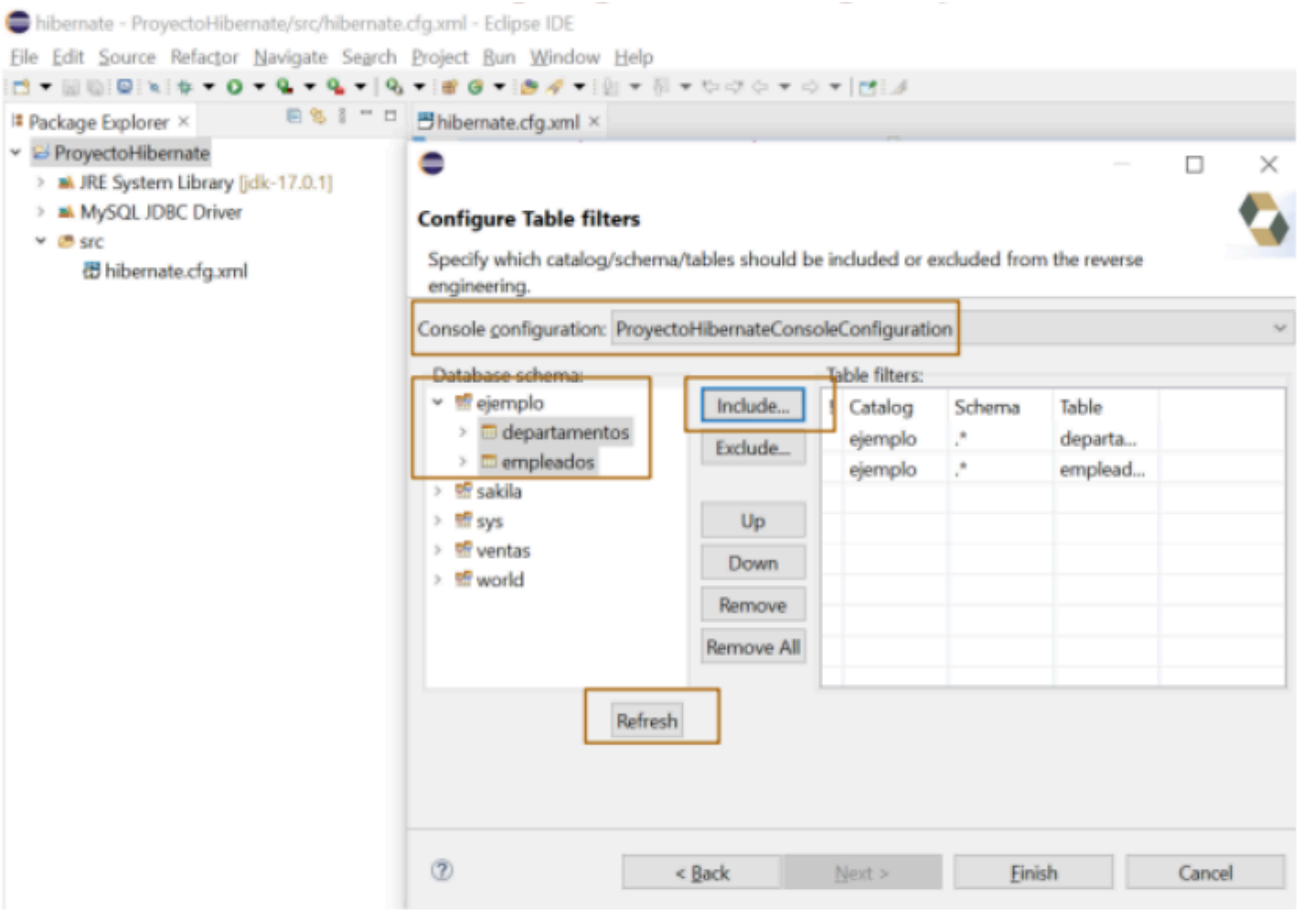


Importante: Pulsamos **Next** para configurar (NO pulsar **Finish** todavía).

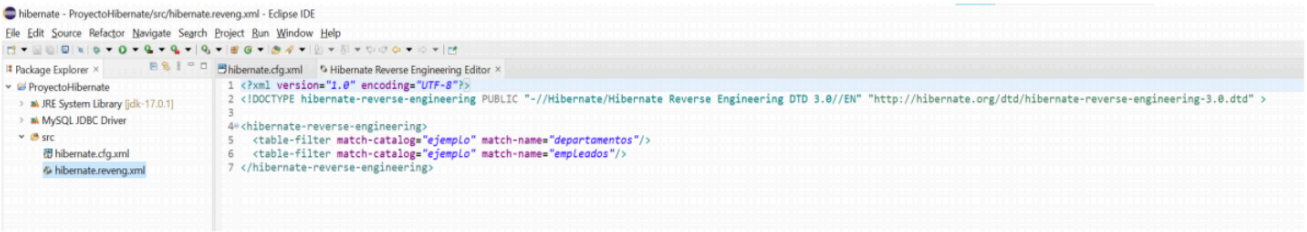
Configuración de Hibernate - *Hibernate Reverse Engineering (hibernate.reveng.xml)* - Mapear tablas

Desde la ventana **Configure Table Filters** que aparece, podemos configurar las tablas que queremos mapear. Para ello, seleccionaremos en el desplegable el **console configuration** creado anteriormente y pulsaremos **Refresh**.

Mediante el botón **Include**, añadiremos las tablas que queremos mapear y, una vez añadidas todas las tablas, pulsaremos **Finish**.

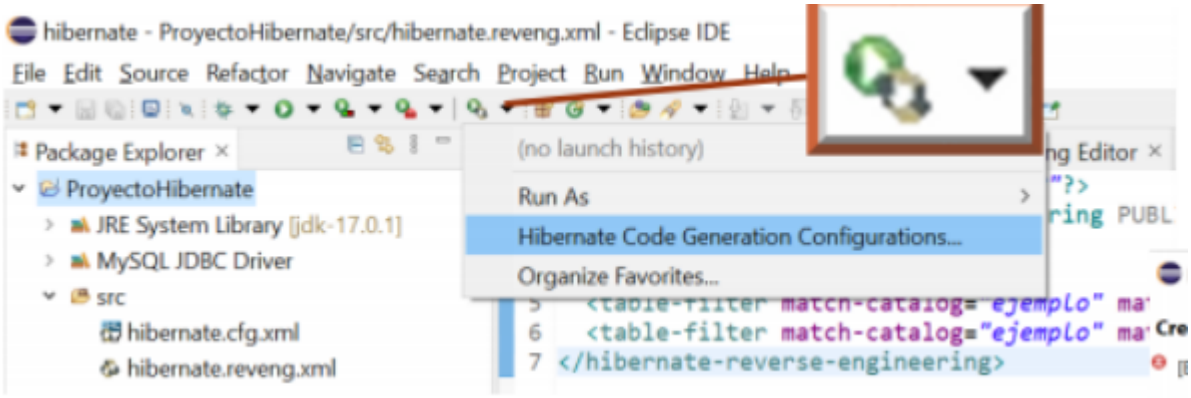


Desde la pestaña "Source", podemos ver como quedaría el fichero de configuración XML



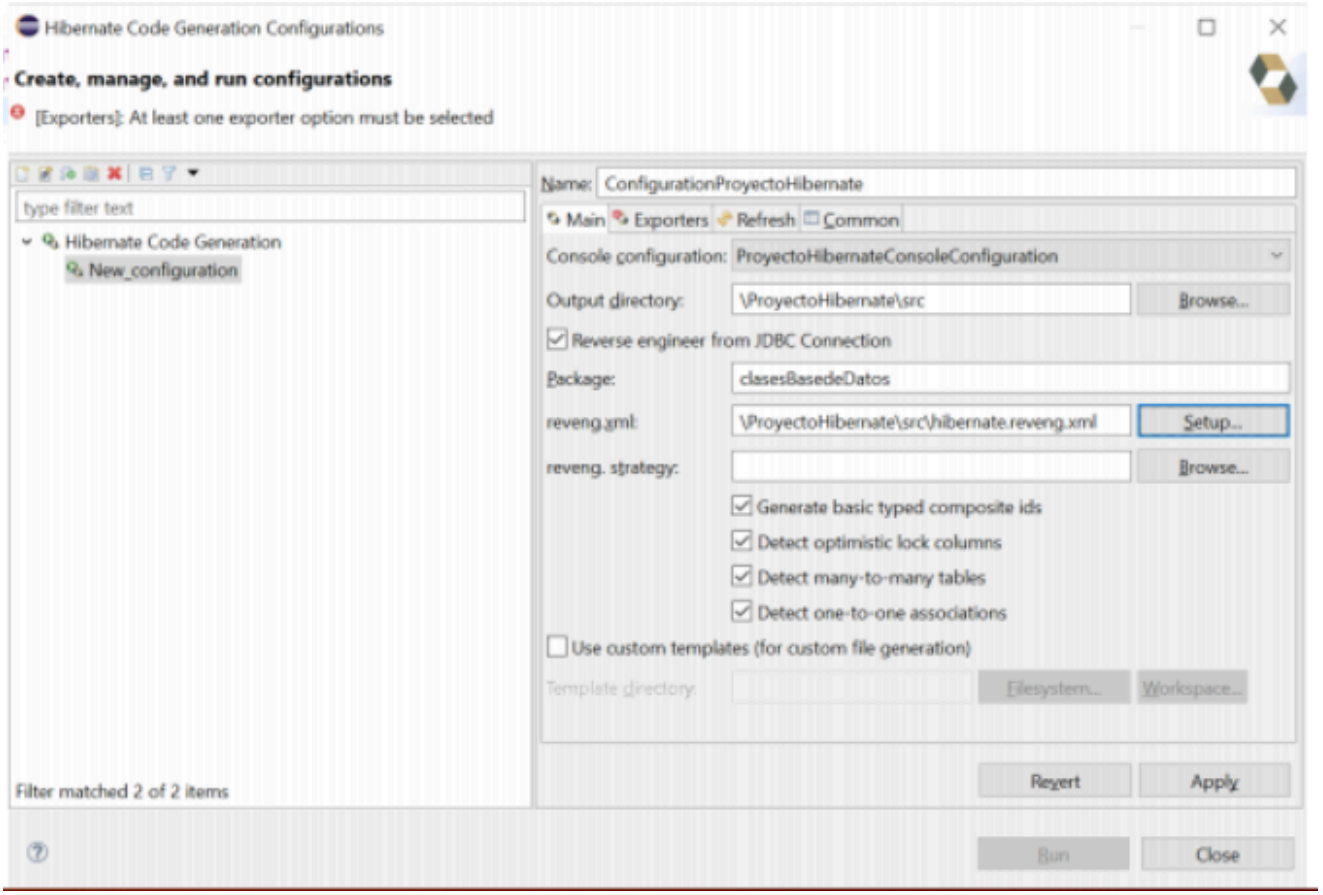
Generación de clases de la base de datos - *Run as (HIBERNATE)* - Pestaña Main

Nuestro siguiente paso será **generar las clases** de nuestra base de datos de ejemplo.
Para ello, pulsamos en la flecha situada a la derecha del botón **Run as (HIBERNATE)** y seleccionamos **Hibernate Code Generation Configuration**:



En la ventana que aparece, haremos **doble click** en la opción **Hibernate Code Generation** y procederemos a configurar la pestaña **Main**:

- **Name:** Por ejemplo, **ConfigurationProyectoHibernate**.
- **Console Configuration:** Elegimos el creado anteriormente.
- **Output directory:** Será **"src"** donde tenemos los ficheros.
- **Package:** Donde se crearán las clases.
- **Reveng.xml:** Configuraremos el creado anteriormente (**use existing...**).

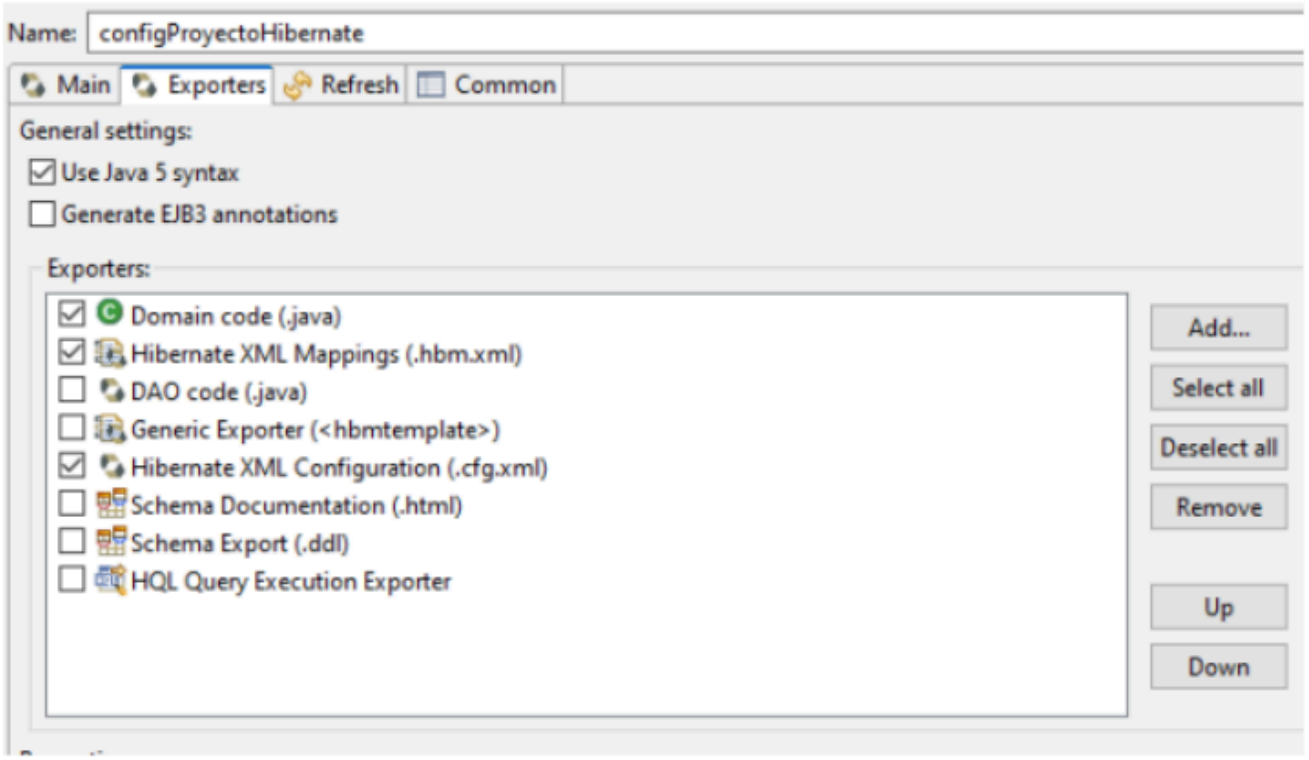


Generación de clases de la base de datos - *Run as (HIBERNATE)* - Pestaña Exporters

Tras la pestaña **Main**, configuraremos la pestaña **Exporters**, donde se indican los ficheros que queremos generar.
Se marcarán las opciones:

- **Use Java 5 syntax**
- **Domain code**
- **Hibernate XML Mappings**
- **Hibernate XML Configuration**

Una vez seleccionados, pulsaremos **Apply** y después **Run**.



Tras finalizar la ejecución, se habrá generado un paquete con:

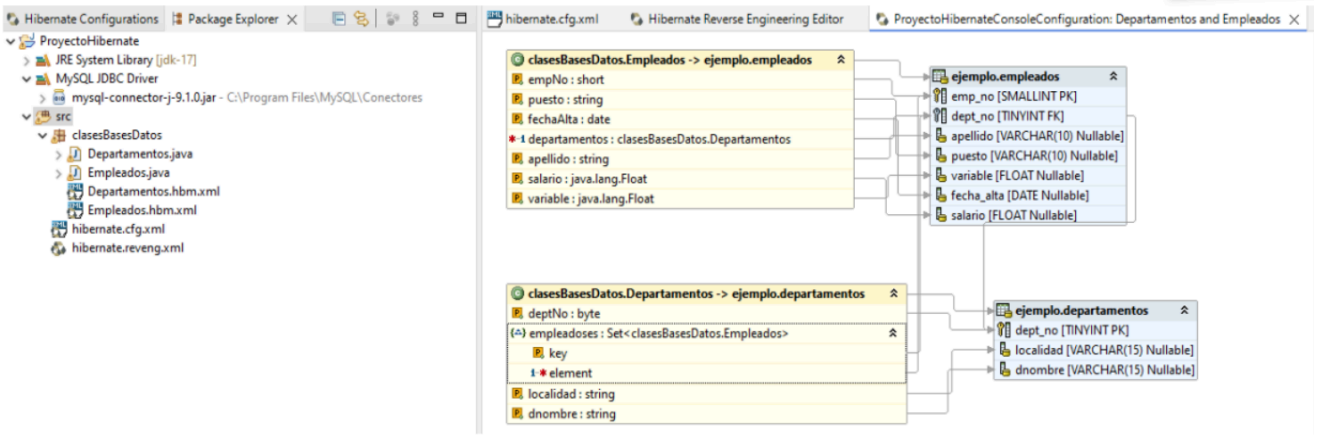
- **Las clases .Java** que contienen los métodos getter y setter de cada uno de los campos de la tabla.
- **Los ficheros .xml** que contienen la información del mapeo de su respectiva tabla.

> **Nota:** El atributo `empleados` define la relación de clave ajena entre las tablas.
> **Nota:** El atributo que relaciona el departamento de un empleado será un objeto `Departamentos` en lugar del código de departamento.

Generación de clases de la base de datos - Diagrama del mapeo

Podemos ver el diagrama de mapeo si abrimos la perspectiva de Hibernate desde la opción de menú:
Windows ➔ Perspective ➔ Open Perspective ➔ Other ➔ Hibernate.

Desde la pestaña **Hibernate Configuration**, elegiremos la configuración de **nuestro proyecto** con el botón derecho y seleccionaremos **Mapping Diagram**.



Estructura de los ficheros de Mapeo

Hibernate utiliza ficheros de mapeo para **relacionar las tablas de la base de datos con los objetos Java**. Estos ficheros están en formato XML y tienen la extensión `.hbm.xml`. En nuestro ejemplo, corresponden con los ficheros `Empleados.hbm.xml` y `Departamentos.hbm.xml` asociados a sus respectivas tablas de nuestra base de datos de ejemplo.

Estos ficheros se crean y almacenan **en la misma ruta y dentro del mismo paquete** que las clases Java `Empleados.Java` y `Departamentos.Java` y representan un objeto empleado y departamento respectivamente.

Pasted image

Departamentos.hbm.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Generated 3 nov 2021 19:50:48 by Hibernate Tools 5.5.7.Final -->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping auto-import="true" default-cascade="none" default-lazy="true">
```

```
6       <class catalog="ejemplo" dynamic-insert="false" dynamic-
    update="false" mutable="true" name="clasesBasesDatos.Departamentos"
    optimistic-lock="none" polymorphism="implicit" select-before-
    update="false" table="departamentos">
7           <id name="deptNo" type="byte">
8               <column name="dept_no"/>
9               <generator class="assigned"/>
10          </id>
11          <property generated="never" lazy="false" name="dnombre"
    optimistic-lock="true" type="string" unique="false">
12              <column length="15" name="dnombre"/>
13          </property>
14          <property generated="never" lazy="false" name="localidad"
    optimistic-lock="true" type="string" unique="false">
15              <column length="15" name="localidad"/>
16          </property>
17          <set embed-xml="true" fetch="select" inverse="true" lazy="true"
    mutable="true" name="empleadoses" optimistic-lock="true" sort="unsorted"
    table="empleados">
18              <key on-delete="noaction">
19                  <column name="dept_no" not-null="true"/>
20              </key>
21              <one-to-many class="clasesBasesDatos.Empleados" embed-
    xml="true" not-found="exception"/>
22          </set>
23      </class>
24  </hibernate-mapping>
```

El significado de este fichero XML será:

- **hibernate-mapping**: Esta etiqueta indica el comienzo y final del fichero de mapeo.
- **class**: Engloba a la clase y a sus atributos, haciendo referencia al mapeo de la tabla de base de datos.
  - **name**: Nombre de la clase.
  - **table**: Nombre de la tabla.
  - **catalog**: Nombre de la base de datos.
- **id**: Identificador del campo **clave** dentro de `class`:
  - **name**: Campo que representa el atributo de la clase.
  - **column**: Su nombre sobre la tabla.
  - **type**: Tipo de datos.
  - **generator**: Indica la naturaleza del campo.
    - **assigned**: Si es el usuario el que se encarga de asignar la clave.
    - **increment**: Si es un identificador autogenerado por la base de

datos.

- **property**: Resto de atributos dentro de `class`.
  - **name**: Campo que representa el atributo de la clase.
  - **column**: Su nombre sobre la tabla.
  - **type**: Tipo de datos.
- **set**: Se utiliza para mapear colecciones (*).

Empleados.hbm.xml

```
<!-- Empleados.hbm.xml -->
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping auto-import="true" default-access="property" default-cascade="none" default-lazy="true">
    <class catalog="ejemplo" dynamic-insert="false" dynamic-update="false" mutable="true" name="clasesBasesDatos.Empleados" optimistic-lock="none" polymorphism="implicit" select-before-update="false" table="empleados">
        <id name="empNo" type="short">
            <column name="emp_no"/>
            <generator class="assigned"/>
        </id>
        <many-to-one class="clasesBasesDatos.Departamentos" embed-xml="true" fetch="select" insert="true" name="departamentos" not-found="exception" optimistic-lock="true" unique="false" update="true">
            <column name="dept_no" not-null="true"/>
        </many-to-one>
        <property generated="never" lazy="false" name="apellido" optimistic-lock="true" type="string" unique="false">
            <column length="25" name="apellido" not-null="true"/>
        </property>
        <property generated="never" lazy="false" name="puesto" optimistic-lock="true" type="string" unique="false">
            <column length="10" name="puesto"/>
        </property>
        <property generated="never" lazy="false" name="fechaAlta" optimistic-lock="true" type="date" unique="false">
            <column name="fecha_alta"/>
        </property>
        <property generated="never" lazy="false" name="salario" optimistic-lock="true" type="java.lang.Float" unique="false">
            <column name="salario" precision="12" scale="0"/>
        </property>
        <property generated="never" lazy="false" name="variable" optimistic-lock="true" type="java.lang.Float" unique="false">
            <column name="variable" precision="12" scale="0"/>
```

```
        </property>
    </class>
</hibernate-mapping>
```

El significado de este fichero XML será:

- **set:** (*) Del fichero `Departamentos.hbm.xml`:
  - **name:** Nombre del atributo generado `[empleadoses]` (nombre de la tabla donde se tomará la colección indicada en el atributo `table` `[empleados]`).
  - **key:** Nombre de la columna identificadora en la asociación `[dep_no - en departamentos]`.
  - **one-to-many:** Relación 1 a N.
- **many-to-one:** (*) Del fichero `Empleados.hbm.xml`:
  - **name:** Atributo de la clase Java de referencia `[departamentos]`.
  - **class:** Indica la clase de referencia `[clasesBasesDatos.Departamentos]`.
  - **column name:** Indica el nombre de la columna de referencia de la tabla empleados `[dept_no - en empleados]`.
  - **fetch:** Escoge la recuperación por unión (`outer-join`) o por selección natural (por defecto es `select`).

Clases persistentes

Si nos fijamos entre el cierre y la apertura de las etiquetas `<hibernate-mapping>` del fichero XML, vemos que se incluye un elemento `class` que hace **referencia a una clase**:

```
<class catalog="ejemplo" dynamic-insert="false" dynamic-update="false" mutable="true"
       name="clasesBasesDatos.Departamentos" optimistic-lock="none"
       polymorphism="implicit" select-before-update="false" table="departamentos">
<class catalog="ejemplo" dynamic-insert="false" dynamic-update="false" mutable="true"
       name="clasesBasesDatos.Empleados" optimistic-lock="none"
       polymorphism="implicit" select-before-update="false" table="empleados">
```

Estas son clases que se han generado en nuestro proyecto, a las cuales se les llama **clases persistentes** y que deben implementar la interfaz `Serializable`. Equivalen a una tabla de la bases de datos y un **registro o fila** sería un **objeto persistente** de esa clase. Definen unos **atributos** y unos **métodos get y set** para el acceso a los mismos.

La definición usa convenciones de nombrado estándares de **JavaBean** para los métodos, así como visibilidad privada para los campos. Al ser atributos de objetos privados, se crean métodos públicos para retornar un valor de un atributo o cargar un valor a un atributo.

A estas reglas se les llama modelo de programación POJO (Plain Old Java Object).

```
1  package clasesBasesDatos;
2  // Generated 3 nov 2021 19:50:48 by Hibernate Tools 5.5.7.Final
3
4  import java.util.HashSet;
5  import java.util.Set;
6
7  /**
8   * Departamentos generated by hbm2java
9   */
10 public class Departamentos implements java.io.Serializable {
11
12     private byte deptNo;
13     private String dnombre;
14     private String localidad;
15     private Set empleadoses = new HashSet();
16
17     public Departamentos() {
18     }
19
20     public Departamentos(byte deptNo) {
21         this.deptNo = deptNo;
22     }
23
24     public Departamentos(byte deptNo, String dnombre, String localidad,
   Set empleadoses) {
25         this.deptNo = deptNo;
26         this.dnombre = dnombre;
27         this.localidad = localidad;
28         this.empleadoses = empleadoses;
29     }
30
31     public byte getDeptNo() {
32         return this.deptNo;
33     }
34
35     public void setDeptNo(byte deptNo) {
36         this.deptNo = deptNo;
37     }
38
39     public String getDnombre() {
```

```
40          return this.dnombre;
41      }
42
43      public void setDnombre(String dnombre) {
44          this.dnombre = dnombre;
45      }
46
47      public String getLocalidad() {
48          return this.localidad;
49      }
50
51      public void setLocalidad(String localidad) {
52          this.localidad = localidad;
53      }
54
55      public Set getEmpleadoses() {
56          return this.empleadoses;
57      }
58
59      public void setEmpleadoses(Set empleadoses) {
60          this.empleadoses = empleadoses;
61      }
62  }
```

Sesiones y objetos Hibernate

Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y **obtener un objeto sesión a partir de la clase SessionFactory**, como hemos visto en nuestros ejemplos.

- La llamada `Configuration().Configure()` carga el fichero de configuración de `hibernate.cfg.xml` e inicializa nuestro entorno.
- Se necesita crear un objeto del tipo `StandardServiceRegistry` que contiene la lista de servicios que utiliza Hibernate **para crear SessionFactory**.

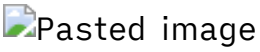
Transacciones

Un objeto `Session` representa **una única unidad de trabajo** para el almacén de datos. Seguidamente **se crea la transacción para esta sesión** y siempre **debemos cerrar la sesión cuando finalicemos este trabajo**.

El método `beginTransaction()` marca el comienzo de la transacción:

- **`Commit()`** valida una transacción.
- **`Rollback()`** deshace la transacción.

Estados de los objetos Hibernate



Transitorio (Transient)

- Si ha sido recién instanciado mediante `new` y no está asociado a una `Session` de Hibernate.
- No tiene representación persistente en la base de datos y no se le ha asignado un valor de identificación.
- Serán destruidas por el recolector de basura si la aplicación no mantiene más de una referencia.
- Utiliza la `Session` de Hibernate para hacer un objeto persistente y deja que Hibernate se ocupe de las declaraciones SQL.
- Las instancias de la clase persistente se consideran transitorias.

Persistente (Persistent)

- Un objeto estará en este estado cuando ya está almacenado en la base de datos. Puede haber sido guardado o cargado, sin embargo, por definición se encuentra en el ámbito de una `Session`.
- Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo.
- Los objetos transitorios solo existen en memoria y no en un almacén de datos. Los persistentes se caracterizan por haber sido creados y almacenados en una sesión o bien devueltos en una consulta.

Separado (Detached)

- Un objeto está en este estado cuando cerramos la sesión mediante el método `Close()` de `Session`.
- Una instancia separada es un objeto que se ha hecho persistente pero su sesión ha sido cerrada.
- La referencia a este objeto todavía es válida y la instancia separada podría incluso ser modificada.
- Una instancia separada puede ser asociada a una nueva `Session` más tarde, haciéndola persistente de nuevo (incluyendo todas las modificaciones).

Para saber más: Estados de los objetos Hibernate

Carga de objetos

En la carga de objetos vamos a utilizar los siguientes métodos de `Session`:

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

En el siguiente ejemplo, utilizamos el método `load()` para obtener los datos del **departamento 10**:

- En el primer parámetro se indica la clase **Departamentos**.
- En el segundo parámetro se indica el número de departamento a recuperar.
  - Se debe hacer un cast para **convertirlo al tipo de dato definido en el atributo** de la clase.
  - En este caso, `deptNo`, que es de tipo **byte**.
- Si la fila no existe, nos devolverá una excepción **ObjectNotFoundException**:

```
1  import clasesBasesDatos.*;
2  import org.hibernate.ObjectNotFoundException;
3  import org.hibernate.Session;
4  import org.hibernate.SessionFactory;
5
6  public class ConsultaDepartamento {
7      public static void main(String[] args) {
8          // Lo primero será obtener la sesión actual
9          SessionFactory sesion = HibernateUtil.getSessionFactory();
10         // Crear la sesión
11         Session session = sesion.openSession();
12
13         // Visualizar los datos del departamento X
14         Departamentos dep = new Departamentos();
15         try {
16             dep = (Departamentos) session.load(Departamentos.class,
   (byte)10);
17             System.out.printf("Nombre Dep: %s%n", dep.getDnombre());
```

```
18              System.out.printf("Localidad: %s%n", dep.getLocalidad());
19          } catch (ObjectNotFoundException o) {
20              System.out.println("NO EXISTE EL DEPARTAMENTO");
21          }
22
23          // Cierro la sesión
24          session.close();
25
26          System.exit(0);
27      }
28  }
```

Almacenamiento, modificación y borrado de objetos

En el almacenamiento, modificación y borrado de objetos vamos a utilizar los siguientes métodos de **Session**:

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

Ejemplo 8.2

Escribe en Java las siguientes aplicaciones:

- **Apartado 1**: Como ya hemos visto un ejemplo de inserción en Hibernate, utiliza el código anterior para insertar un nuevo departamento en nuestra base de datos, por ejemplo:
  - Identificador del departamento: `80`
  - Nombre del departamento: `INFORMÁTICA`
  - Localidad: `TOLEDO`
- **Apartado 2**: Crea una aplicación que te permita modificar el salario de un empleado de tu base de datos aumentándolo en 1000 Euros. También debe modificar el departamento del empleado aplicándole el departamento nuevo.

- **Apartado 3**: Crea una aplicación que te permita intentar borrar un departamento que exista en tu base de datos. Se debe controlar que el departamento exista y que tenga empleados, mostrando un mensaje en caso de que no se pueda eliminar al tener empleados.

Ejemplo Apartado 2. Modificación

En el siguiente ejemplo, utilizamos el método `update()` para modificar el salario y departamento del empleado 101:

- En primer lugar, hemos de **cargar el objeto que queremos modificar**.
- Una vez tenemos el objeto, utilizaremos el método `update()` para modificarlo.
- Si la fila no existe, nos devolverá una excepción `ObjectNotFoundException` y si no existe el departamento, `ConstraintViolationException`:

```
1  import clasesBasesDatos.*;
2
3  import org.hibernate.ObjectNotFoundException;
4  import org.hibernate.Session;
5  import org.hibernate.SessionFactory;
6  import org.hibernate.Transaction;
7  import org.hibernate.exception.ConstraintViolationException;
8
9  public class modificarEmpleado {
10
11     public static void main(String[] args) {
12         // Lo primero será obtener la sesión actual
13         SessionFactory sesion = HibernateUtil.getSessionFactory();
14         // Crear la sesión
15         Session session = sesion.openSession();
16         // Crear la transacción de la sesión
17         Transaction tx = session.beginTransaction();
18
19         Empleados empleado = new Empleados();
20         try {
21             empleado = (Empleados) session.load(Empleados.class, (short)
   101);
22             System.out.printf("Modificación del empleado número: %d%n",
   empleado.getEmpNo());
23             System.out.printf("Salario antiguo: %.2f%n",
   empleado.getSalario());
24             System.out.printf("Departamento Antiguo: %s%n",
   empleado.getDepartamentos().getDnombre());
25
26         //nuevo salario
```

```
27          float nuevoSalario = empleado.getSalario() + 1000;
28          empleado.setSalario(nuevoSalario);
29
30          //nuevo departamento
31          Departamentos dep = (Departamentos)
    session.get(Departamentos.class, (byte) 80);
32
33          if (dep == null) {
34              System.out.println("EL departamento NO existe");
35          } else {
36              empleado.setDepartamentos(dep);
37              //modificamos el objeto
38              session.update(empleado);
39              tx.commit();
40              System.out.printf("salario nuevo: %.2f%n",
    empleado.getSalario());
41              System.out.printf("Departamento nuevo: %s%n",
    empleado.getDepartamentos().getDnombre());
42          }
43          } catch (ObjectNotFoundException o) {
44              System.out.println("NO EXISTE EL EMPLEADO");
45          } catch (ConstraintViolationException c) {
46              System.out.println("NO SE PUEDE ASIGNAR UN DEPARTAMENTO QUE
    NO EXISTE");
47          } catch (Exception e) {
48              System.out.println("ERROR NO CONTROLADO");
49              e.printStackTrace();
50          }
51          session.close();
52          System.exit(0);
53      }
54  }
```

Ejemplo Apartado 3. Borrado

```
1   import clasesBasesDatos.*;
2   import org.hibernate.ObjectNotFoundException;
3   import org.hibernate.Session;
4   import org.hibernate.SessionFactory;
5   import org.hibernate.Transaction;
6   import org.hibernate.exception.ConstraintViolationException;
7
8   public class BorrarDepartamento {
9       public static void main(String[] args) {
10          // Lo primero será obtener la sesión actual
11          SessionFactory sesion = HibernateUtil.getSessionFactory();
12          // Crear la sesión
13          Session session = sesion.openSession();
```

```
14          // Crear la transacción de la sesión
15          Transaction tx = session.beginTransaction();
16
17          // Departamento a eliminar
18          Departamentos dep = (Departamentos)
    session.load(Departamentos.class, (byte) 80);
19
20          try {
21              // Eliminación del objeto
22              session.delete(dep);
23              tx.commit();
24              System.out.println("Departamento eliminado");
25          } catch (ObjectNotFoundException o) {
26              System.out.println("NO EXISTE EL DEPARTAMENTO");
27          } catch (ConstraintViolationException c) {
28              System.out.println("NO SE PUEDE ELIMINAR, TIENE EMPLEADOS");
29          } catch (Exception e) {
30              System.out.println("ERROR NO CONTROLADO");
31              e.printStackTrace();
32          }
33
34          session.close();
35          System.exit(0);
36      }
37  }
```

Consultas en Hibernate con HQL

Hibernate soporta un **lenguaje de consultas orientado a objetos** denominado **HQL (Hibernate Query Language)**. Este lenguaje es fácil de usar sin perder potencia, y es una extensión orientada a objetos de SQL.

Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.Query` o `org.hibernate.query.Query`, dependiendo de la versión de Hibernate. Esta interfaz ofrece métodos para ligar parámetros, manejar el conjunto de resultados y para la ejecución de la consulta real. Siempre se obtiene una **Query** utilizando el objeto **Session** actual.

Métodos más importantes:

Método	Descripción
`Iterator iterate()`	Devuelve un objeto `Iterator` con el resultado de una consulta
`List list()`	Devuelve el resultado de una consulta en un `List`
`Query setFetchSize(int size)`	Fija el número de resultados a recuperar en cada acceso a la base de datos según el valor

Método	Descripción
	indicado.
`String getQueryString()`	Devuelve la consulta en un `String`.
`Object uniqueResult()`	Devuelve un objeto (cuando sabemos que la consulta devuelve un único resultado) o `null`.
`Query setCharacter(int posición, char valor)`	Asigna el valor indicado en el método a un parámetro de tipo CHAR.
`Query setCharacter(String nombre, char valor)`	Nombre y valor del parámetro dentro de la consulta.

Método	Descripción
`Query setDate(int posición, Date fecha)`	Asigna la fecha a un parámetro de tipo `Date`.
`Query setDate(String nombre, Date fecha)`	Similar al anterior pero con un nombre específico.
`Query setDouble(int posición, double valor)`	Asigna un valor a un parámetro de tipo decimal (en MySQL tipo FLOAT).
`Query setDouble(String nombre, double valor)`	Igual que el anterior pero especificando el nombre.
`Query setInteger(int posición, int valor)`	Asigna un valor a un parámetro de tipo entero.
`Query setInteger(String nombre, int valor)`	Igual que el anterior pero especificando el nombre.
`Query setString(int posición, String valor)`	Asigna un valor a un parámetro de tipo VARCHAR.
`Query setString(String nombre, String valor)`	Igual que el anterior pero especificando el nombre.
`Query setParameterList(String nombre, Collection valores)`	Asigna una colección de valores al parámetro cuyo nombre se indica.
`Query setParameter(int posición, Object valor)`	Asigna el valor al parámetro indicado en posición.
`Query setParameter(String nombre, Object valor)`	Asigna el valor al parámetro indicado en nombre.
`Int executeUpdate()`	Ejecuta una sentencia de actualización o delete. Devuelve el número de entidades afectadas.

Consulta la API de Hibernate: Documentación oficial

Realizar una consulta con `createQuery()`

Para realizar una consulta usamos el método `createQuery()` de la interfaz `SharedSessionContract`. Se le pasará un **String** con la consulta HQL.

Ejemplo:

```
Query q = session.createQuery("FROM Departamentos");
```

Recuperar los datos de la consulta:

- `list()`: Devuelve en una colección todos los resultados de la columna. Se encuentran instanciadas todas las entidades correspondientes al resultado de la ejecución de la consulta.
  - Realiza una única comunicación con la base de datos donde se traen todos los resultados.
  - Requiere que haya memoria suficiente para almacenar todos los objetos resultantes de la consulta.
  - Si la cantidad de datos es extensa, el retraso del acceso a la base de datos será notorio.
- `iterate()`: Este método está **en desuso desde la versión 5.2**, al igual que la instancia `org.hibernate.Query`. Devuelve un iterador Java para recuperar los resultados de la consulta, obteniendo sólo los IDs de las entidades.

Ejemplo con `list()`

Vamos a ver un ejemplo para consultar todas las filas de la tabla **Departamentos**:

```
import clasesBasesDatos.*;
import org.hibernate.query.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import java.util.Iterator;
import java.util.List;

public class consultaDepartamentos {
    public static void main(String[] args) {
        // Lo primero será obtener la sesión actual
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        // Crear la sesión
        Session session = sesion.openSession();

        Query q = session.createQuery("from Departamentos");
        List<Departamentos> lista = q.list();

        // Obtenemos un iterador y recorremos la lista
```

```
        Iterator<Departamentos> iter = lista.iterator();
        System.out.printf("Numero de departamentos: %d%n", lista.size());

        while (iter.hasNext()) {
            // Extraer objeto
            Departamentos depar = iter.next();
            System.out.printf("%d, %s%n", depar.getDeptNo(),
depar.getDnombre());
        }

        session.close();
        System.exit(0);
    }
}
```

Actividad 9.1

Busca información y realiza una consulta HQL utilizando `createQuery()`, para obtener los datos del departamento 10 y visualizar también el apellido de sus empleados.

Solución: Ejemplo condiciones en HQL

Ahora vamos a ver cómo consultar el apellido y salario de los empleados del departamento 10:

```
import clasesBasesDatos.*;
import org.hibernate.query.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import java.util.Iterator;
import java.util.List;

public class consultaEmpleadosDep {
    public static void main(String[] args) {
        // Lo primero será obtener la sesión actual
        SessionFactory session = HibernateUtil.getSessionFactory();
        // Crear la sesión
        Session session = session.openSession();

        Query q = session.createQuery("from Empleados as e where
e.departamentos.deptNo = 10");
        List<Empleados> lista = q.list();

        // Obtenemos un iterador y recorremos la lista
        Iterator<Empleados> iter = lista.iterator();
```

```
20
21          while (iter.hasNext()) {
22              // Extraer objeto
23              Empleados emp = (Empleados) iter.next();
24              System.out.printf("%s, %.2f %n", emp.getApellido(),
    emp.getSalario());
25          }
26          session.close();
27          System.exit(0);
28      }
29  }
```

Ejemplo resultado único

Podemos utilizar el método `uniqueResult()`, si sabemos que el resultado nos devolverá un único objeto. Por ejemplo, vamos a consultar los datos del departamento 10:

```
1   import clasesBasesDatos.*;
2   import org.hibernate.query.Query;
3   import org.hibernate.Session;
4   import org.hibernate.SessionFactory;
5   import java.util.Iterator;
6   import java.util.List;
7
8   public class consultaEmpleadosDep {
9       public static void main(String[] args) {
10          // Lo primero será obtener la sesión actual
11          SessionFactory session = HibernateUtil.getSessionFactory();
12          // Crear la sesión
13          Session session = session.openSession();
14
15          Query q = session.createQuery("from Departamentos as dep where
    dep.deptNo = 10");
16          Departamentos dep = (Departamentos) q.uniqueResult();
17
18          System.out.printf("%d, %s, %s%n", dep.getDeptNo(),
    dep.getDnombre(), dep.getLocalidad());
19
20          session.close();
21          System.exit(0);
22      }
23  }
```

Parámetros en las consultas

Hibernate soporta **parámetros con nombre** en las consultas. Los parámetros con nombre son identificadores de la forma `:nombre` en la cadena de la consulta.

Estos parámetros se numeran en Hibernate desde 0, es decir, el primer parámetro estará en la posición 0 y el siguiente en la posición 1, ...

Para asignar valores a los parámetros se utilizan los métodos `setXXX` vistos en la tabla de las diapositivas iniciales:

- `setInteger`
- `setString`
- `setDate`
- ...

(Algunos de estos métodos están obsoletos a partir de la versión 5.2.)

La sintaxis más simple de utilizar es el método `setParameter()`.

```
1  import clasesBasesDatos.*;
2  import org.hibernate.query.Query;
3  import org.hibernate.Session;
4  import org.hibernate.SessionFactory;
5
6  public class consultaDepUnicoParam {
7      public static void main(String[] args) {
8          // Lo primero será obtener la sesión actual
9          SessionFactory session = HibernateUtil.getSessionFactory();
10         // Crear la sesión
11         Session session = session.openSession();
12
13         // Creamos nuestra consulta y hacemos la llamada al método
14         String hql = "from Departamentos where deptNo = :numdep";
15         Query q = session.createQuery(hql);
16
17         q.setParameter("numdep", (byte) 10);
18         Departamentos dep = (Departamentos) q.uniqueResult();
19         System.out.printf("%d, %s, %s %n", dep.getDeptNo(),
   dep.getDnombre(), dep.getLocalidad());
20
21         session.close();
22         System.exit(0);
23     }
24 }
```

Insert, Update y Delete con HQL

Con el lenguaje HQL también podemos realizar operaciones **INSERT**, **UPDATE** y **DELETE**. La sintaxis de estas operaciones será:

UPDATE/DELETE

```
(UPDATE | DELETE) [FROM] NombreEntidad [WHERE condición]
```

- La palabra clave `FROM` es opcional.
- La cláusula `WHERE` también es opcional.
- Sólo puede haber una entidad en la cláusula `FROM`.
- Si esta entidad usa alias, cualquier referencia se hará usando ese alias.
- Se pueden utilizar subconsultas en la cláusula `WHERE`.

> **Nota**:
>
> - Para ejecutar **UPDATE** o **DELETE** usaremos el método `executeUpdate()`, que devuelve el número de entidades afectadas.
> - También debemos usar `commit` para validar la transacción.

INSERT

```
INSERT INTO NombreEntidad (lista propiedades) sentencia select
```

- Sólo se soporta la forma `INSERT INTO ... SELECT ...` (no `VALUES ...`).
  - Sólo datos de otras tablas.
- La lista de propiedades es análoga a la lista de columnas en SQL.
- Se puede usar cualquier sentencia `SELECT` en HQL, pero hay que tener en cuenta que los tipos de la consulta coinciden con el `INSERT`.

Para el caso del `id`, hay 2 opciones:

1. Especificar en la lista de propiedades.
2. Omitir en la lista de propiedades (sólo será válido si se utiliza generadores automáticos de id).

> **Nota**:
> Debe existir configuración en nuestro archivo `xml` de configuración.

Ejemplo de Update con HQL

```
import clasesBasesDatos.*;
import org.hibernate.query.Query;
import org.hibernate.Session;
```

```java
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class modificarHQL {
    public static void main(String[] args) {
        // Lo primero será obtener la sesión actual
        SessionFactory session = HibernateUtil.getSessionFactory();
        // Crear la sesión
        Session session = session.openSession();
        // Crear la transacción de la sesión
        Transaction tx = session.beginTransaction();

        // Modificaciones
        String hqlModif = "update Empleados set salario = :nuevoSal where apellido = :ape";
        Query q1 = session.createQuery(hqlModif);
        q1.setParameter("nuevoSal", (float) 2500);
        q1.setParameter("ape", "GIL");

        int filasModif = q1.executeUpdate();
        System.out.printf("Filas modificadas: %d%n", filasModif);
        tx.commit();

        session.close();
        System.exit(0);
    }
}
```

Ejemplo de Delete con HQL

```java
import clasesBasesDatos.*;
import org.hibernate.query.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class BorrarHQL {
    public static void main(String[] args) {
        // Lo primero será obtener la sesión actual
        SessionFactory session = HibernateUtil.getSessionFactory();
        // Crear la sesión
        Session session = session.openSession();
        // Crear la transacción de la sesión
        Transaction tx = session.beginTransaction();

        // Borrado
        String hqlDel = "delete Empleados e where e.departamentos.deptNo = :dep";
```

```
        Query q2 = session.createQuery(hqlDel);
        q2.setParameter("dep", 20);

        int filasDel = q2.executeUpdate();
        System.out.printf("Filas eliminadas: %d%n", filasDel);

        tx.commit();

        session.close();
        System.exit(0);
    }
}
```

Ejemplo de Insert con HQL

```
import clasesBasesDatos.*;
import org.hibernate.query.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class insertHQL {
    public static void main(String[] args) {
        // Lo primero será obtener la sesión actual
        SessionFactory session = HibernateUtil.getSessionFactory();
        // Crear la sesión
        Session session = session.openSession();
        // Crear la transacción de la sesión
        Transaction tx = session.beginTransaction();

        // Inserción
        String hqlinsert = "insert into Departamentos (deptNo, dnombre,
localidad) "
                + "select n.deptNo, n.dnombre, n.localidad from Nuevos
n";
        Query cons = session.createQuery(hqlinsert);

        int filascreadas = cons.executeUpdate();
        System.out.printf("Filas insertadas: %d%n", filascreadas);

        tx.commit();

        session.close();
        System.exit(0);
    }
}
```

Nota: La tabla "Nuevos" debe existir y contener esos datos para poder extraerlos con HQL.

Consultas en Hibernate con SQL

Hibernate también nos permite ejecutar sentencias en lenguaje SQL Nativo. Para ello utilizaremos el método `createNativeQuery()`, donde podremos escribir las sentencias en lenguaje SQL que queramos ejecutar. Hay que tener en cuenta que, al no conocer la estructura, **debemos trabajar con objetos** que nos ofrecerán un **array result** con los datos. Veamos un ejemplo:

```
1  import clasesBasesDatos.*;
2
3  import org.hibernate.query.Query;
4  import org.hibernate.Session;
5  import org.hibernate.SessionFactory;
6
7  public class ejemploSQL {
8      public static void main(String[] args) {
9          // Lo primero será obtener la sesión actual
10         SessionFactory session = HibernateUtil.getSessionFactory();
11         // Crear la sesión
12         Session session = session.openSession();
13
14         // Creamos nuestra consulta y hacemos la llamada al método
15         Query nativeQuery = session.createNativeQuery("SELECT deptNo,
   dnombre, localidad FROM Departamentos WHERE deptNo = :numdep");
16         nativeQuery.setParameter("numdep", (byte)10);
17
18         Object[] result = (Object[]) nativeQuery.uniqueResult();
19         System.out.printf("%s, %s %n", result[0].toString(),
   result[1].toString());
20
21         session.close();
22         System.exit(0);
23     }
24 }
```