
CRUD APLICACION DE JDBC A PERSISTENCIA

INDICE

1. INTRODUCCIÓN
2. CRUD CON JDBC
3. EVOLUCION HACIA ENTITY
4. CRUD CON PERSISTENCIA (JPA-HIBERNATE)
5. HIBERNATE AUTOMATICO

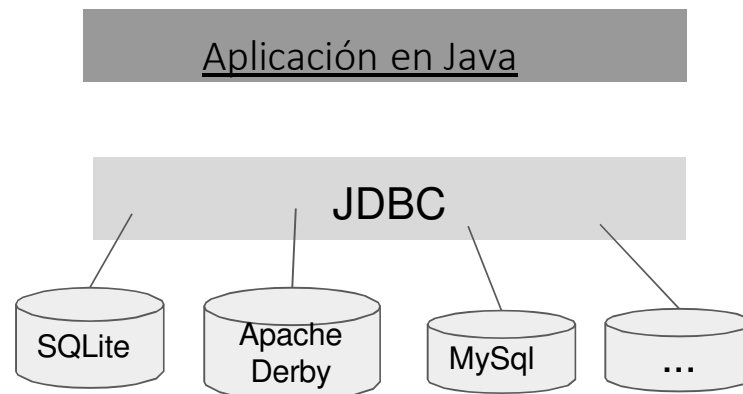
1. INTRODUCCION

- En tecnologías de base de datos podemos encontrarnos con dos normas de conexión a una base de datos:
 - **ODBC**: Open Database Connectivity. Define una API que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados. Las aplicaciones pueden usar esta API siempre y cuando el servidor de base de datos sea compatible con ODBC
 - **JDBC**. Java Database Connectivity. Define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales
 - **OLE-DB**. Object Linking and Embedding for Databases. De Microsoft. Es una API de C++ con objetivos parecidos a los de ODBC pero para orígenes de datos que no son bases de datos. En ODBC los comandos siempre están en SQL, en OLE-DB pueden estar en cualquier lenguaje soportado por el origen de datos.

1.1. ACCESO A DATOS MEDIANTE JDBC

- JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL.
- No solo provee una interfaz sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos.
- JDBC dispone de una interfaz distinta para cada base de datos, es lo que llamamos **driver** (controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.

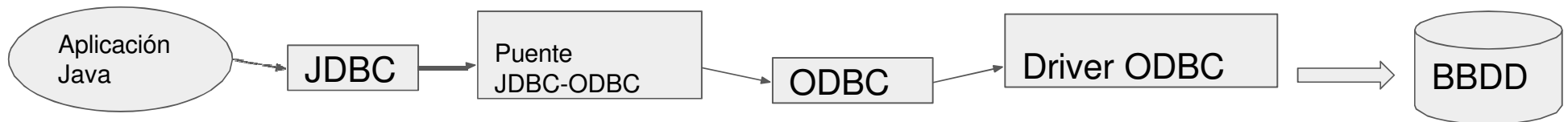
1.1. ACCESO A DATOS MEDIANTE JDBC



- JDBC consta de un conjunto de clases e interfaces que nos permiten escribir aplicaciones Java para gestionar las siguientes tareas con una bbdd relacional:
 - Conectarse a la base de datos
 - Enviar consultas e instrucciones DML a la bbdd
 - Recuperar y procesar los resultados recibidos

1.2. TIPOS DE DRIVERS JDBC

- **Tipo 1 JDBC-ODBC Bridge:** permite el acceso a bases de datos JDBC mediante un driver ODBC. Convierte las llamadas al API de JDBC en ODBC. Exige la instalación y configuración de ODBC en la máquina cliente

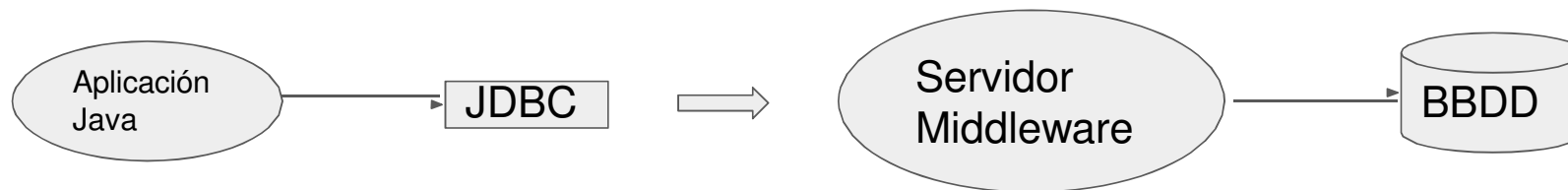


- **Tipo 2 Native:** controlador escrito parcialmente en Java y en código nativo de la base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del motor de base de datos. Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.



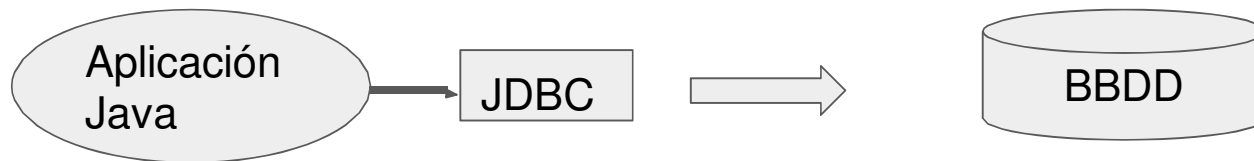
1.2. TIPOS DE DRIVERS JDBC

- **Tipo 3 Network:** controlador de Java puro que utiliza un protocolo de red (por ejemplo HTTP) para comunicarse con el servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red y a continuación son traducidas por un software intermedio (Middleware) al protocolo usado por la bbdd. No exige instalación en cliente.



1.2. TIPOS DE DRIVERS JDBC

- **Tipo 4 Thin:** controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de base de datos. No exige instalación en cliente.



Los tipos 3 y 4 son la mejor forma de acceder. Los tipos 1 y 2 se usan normalmente cuando no queda otro remedio. En la mayoría de los casos la opción más adecuada será el tipo 4.

1.3. COMO FUNCIONA JDBC

- JDBC define varias interfaces que permiten realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes.
- Estas clases están definidas en el paquete [java.sql](#)
- El funcionamiento de un programa con JDBC requiere los siguientes pasos:
 - Importar las clases necesarias
 - Cargar el **driver** JDBC
 - Identificar el origen de datos
 - Crear un objeto **Connection**
 - Crear un objeto **Statement**
 - Ejecutar una consulta con el objeto **Statement**
 - Recuperar los datos del objeto **ResultSet**
 - Liberar sucesivamente **ResultSet**, **Statement**, **Connection**

1.3. COMO FUNCIONA JDBC

```
try {
    //Para el Driver mysql-connector-java-5.1.47.jar
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();
    String conex = "jdbc:mysql://localhost:3306/futbol_mysql";
    Connection conexion=DriverManager.getConnection(conex,"root","");

    Statement sentencia = conexion.createStatement();
    String sql = "select `CodEq`,`Equipo`,`Ciudad`,`Estadio` from equipos";
    ResultSet result = sentencia.executeQuery(sql);
    while(result.next()) {
        System.out.println(result.getString(1) + " : " + result.getString(2) + " : " +
            result.getString(3) + " : " + result.getString(4));
    }
    result.close();
    sentencia.close();
    conexion.close();
}
catch(Exception ex)
{
    System.out.print(ex.getMessage());
}
```

1.3. COMO FUNCIONA JDBC

Cargar driver

En primer lugar se carga el driver con el método `forName()` de la clase `Class` (`java.lang`). Para ello se le pasa un objeto `String` con el nombre de la clase del driver como argumento. En el ejemplo, como se accede a una base de datos `MySQL`, necesitamos cargar el driver `com.mysql.cj.jdbc.Driver`:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Establecer conexión

A continuación se establece la conexión con la base de datos. El servidor `MySQL` debe estar rodando. Usamos la clase `DriverManager` con el método `getConnection()` de la siguiente manera:

```
Connection conexion=DriverManager.getConnection  
("jdbc:mysql://localhost:3306/futbol_mysql","root","");
```

1.3. COMO FUNCIONA JDBC

Método getConnection()

public static Connection getConnection (String url, String user, String password) throws SQLException

El primer parámetro del método getConnection() representa la URL de conexión a la bbdd. Tiene el siguiente formato para conectarse a MySql:

jdbc:mysql://nombre_host:puerto/nombre_basedatos

- **jdbc:mysql** → indica que estamos utilizando un driver JDBC para MySql
- **nombre_host** → **indica** el nombre del servidor donde está la base de datos. Aquí puede ponerse una IP, el nombre de máquina, o localhost
- **puerto** → puerto donde está escuchando el servidor MySql. Por defecto es el 3306. Si no se indica se asume que es este valor.
- **nombre_basedatos** → nombre de la base de datos a la que queremos conectarnos. Debe existir previamente en MySql.

1.3. COMO FUNCIONA JDBC

Ejecutar sentencias SQL

- A continuación se realiza la consulta, para ello recurrimos a la interfaz **Statement** para crear una sentencia. Para obtener un objeto **Statement** se llama al método **createStatement()** de un objeto **Connection** válido. La sentencia obtenida (o objeto obtenido) tiene el método **executeQuery()** que sirve para realizar una consulta en la base de datos, se le pasa un String en que está la consulta SQL:
Statement sta =conn.createStatement();
String sql = "select `CodEq`,`Equipo`,`Ciudad`,`Estadio` from equipos";
ResultSet result = sentencia.executeQuery(sql);
- El resultado nos lo devuelve como un **ResultSet**, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de la tabla “departamentos”.

1.3. COMO FUNCIONA JDBC

Recorrer la lista mediante el método next()

- Los métodos **getInt()** y **getString()** nos van devolviendo los valores de los campos de cada registro. Entre paréntesis se pone la posición de la columna en la tabla. También se puede poner una cadena que indica el nombre de la columna:

```
while (result.next()){  
    System.out.printf("%d, %s, %s, %n", result.getInt(1),  
        result.getString(2), result.getString(3));  
}  
  
while (result.next()){  
    System.out.printf("%d, %s, %s, %n", result.getInt("codeq"),  
        result.getString("equipo"), result.getString("ciudad"));  
}
```

1.4. COMO FUNCIONA JDBC

Clase ResultSet

Dispone de varios métodos para mover el puntero que apunta a cada uno de los registros devueltos por la consulta:

- **boolean next()** → Mueve el puntero una fila hacia adelante a partir de la posición actual. Devuelve **true** si el puntero se posiciona correctamente y **false** si no hay registros en **ResultSet**.
- **boolean first()** → Mueve el puntero al primer registro de la lista.
- **boolean last()** → Mueve el puntero al último registro de la lista.
- **boolean previous()** → Mueve el puntero al registro anterior de la posición actual.
- **void beforeFirst()** → Mueve el puntero justo antes del primer registro.
- **int getRow()** → Devuelve el número de registro actual. Para el primer registro devuelve 1, para el segundo 2 y así sucesivamente.

Por último, se liberan todos los recursos y se cierra la conexión

- **result.close(); sentencia.close(); conexion.close();**

1.4. COMO FUNCIONA JDBC

Ejecución DML y DDL

Un objeto **Statement** crea un espacio de trabajo para realizar consultas SQL, ejecutarlas y recibir los resultados. Se pueden usar los siguientes métodos:

- **ResultSet executeQuery(String)** → para sentencias SELECT
- **int executeUpdate(String)** → se utiliza para sentencias que no devuelvan un **ResultSet** como son las sentencias DML: INSERT, UPDATE Y DELETE; y las sentencias DDL: CREATE, DROP y ALTER. El método devuelve un entero indicando el número de filas que se vieron afectadas y, en caso de las sentencias DDL, devuelve 0.
- **boolean execute(String)** → se puede utilizar para ejecutar cualquier sentencia SQL. Tanto las que devuelven un **ResultSet** (SELECT), como para las que devuelven el número de filas afectadas (INSERT, UPDATE, DELETE) y para las de definición de datos (CREATE). El método devuelve *true* si devuelve un **ResultSet** (para recuperar las filas será necesario llamar al método **getResultSet()**) y *false* si se trata de actualizaciones o no hay resultados; en este caso se usará el método **getUpdateCount()** para recuperar el valor devuelto de filas afectadas.

1.4. COMO FUNCIONA JDBC

```
try {
    //Para el Driver mysql-connector-java-5.1.47.jar
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();
    String conex = "jdbc:mysql://localhost:3306/futbol_mysql";
    Connection conexion=DriverManager.getConnection(conex,"root","");

    String sql = "INSERT INTO EQUIPOS (`codeq`, `equipo`, `ciudad`, `estadio`) VALUES (?, ?, ?, ?);";
    PreparedStatement sentencia= conexion.prepareStatement(sql);
    sentencia.setInt(1,70 );
    sentencia.setString(2, "Manresa FC");
    sentencia.setString(3, "Igualada");
    sentencia.setString(4, "Congost");
    System.out.print(sentencia);

    int filas = sentencia.executeUpdate(sql);
    System.out.print("Filas afectadas: "+ filas);
    sentencia.close();
    conexion.close();
}
catch(Exception ex)
{
    System.out.print(ex.getMessage());
}
```

2. CRUD CON JDBC-MYSQL

Realizaremos una aplicación en modo terminal que realice un mantenimiento CRUD sobre una tabla “empleados” situada en la base de datos “empresa” de mysql.

```
create table empleados(  
    cod_emp int primary key auto_increment,  
    nombre varchar(20),  
    apellidos varchar(20),  
    salario double  
);
```

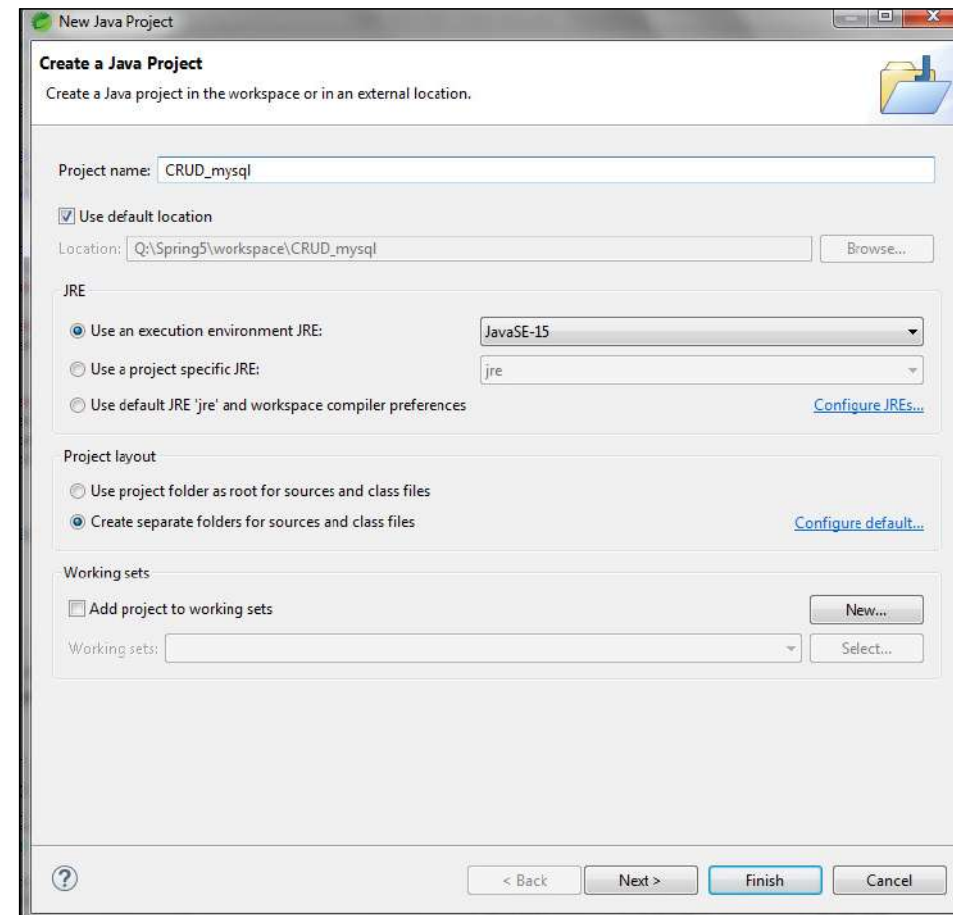
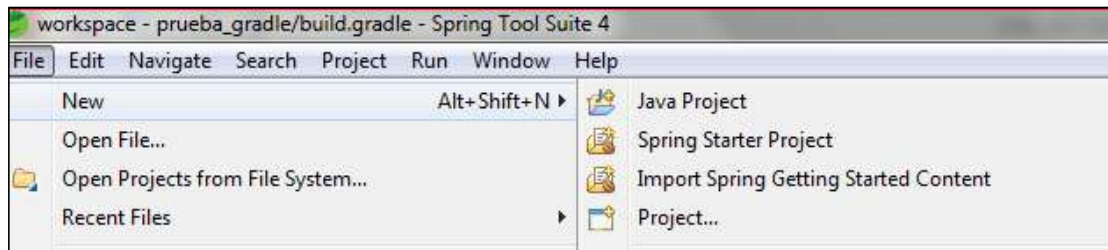
Debe ofrecer un menú con las siguientes opciones:

- a) Visualizar la lista de empleados
- b) Actualizar el salario de un empleado
- c) Insertar un nuevo empleado.
- d) Borrar un empleado.
- e) Salir



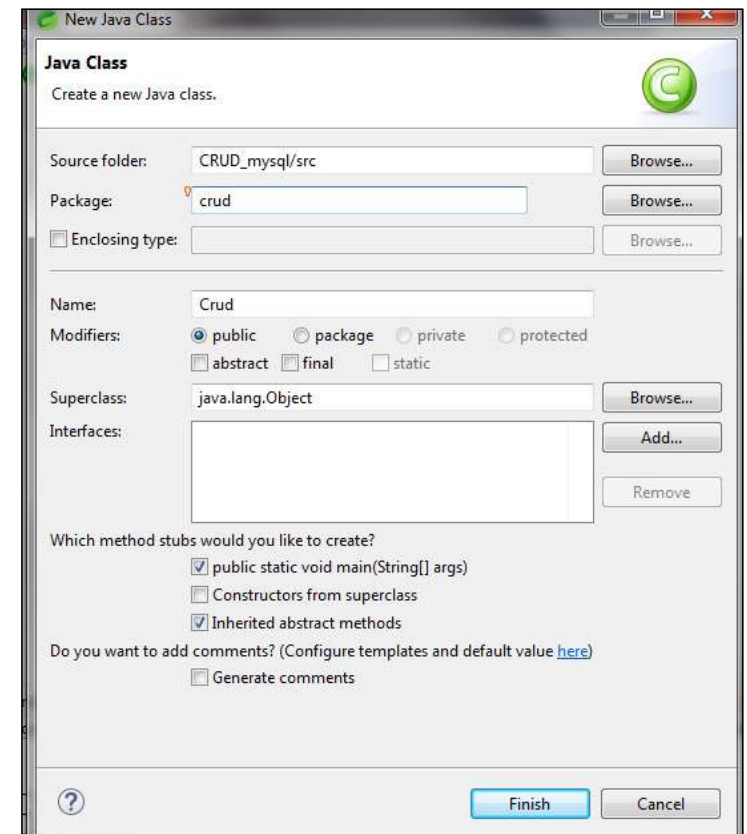
2. CRUD CON JDBC-MYSQL

Paso 1. Primero de todo creamos un proyecto Java, de nombre por ejemplo CRUD_mysql:



2. CRUD CON JDBC-MYSQL

Paso 2. Nos situamos sobre la carpeta src, y hacemos click botón derecho y seleccionamos New/Class. Pondremos nombre Crud, por ejemplo. Debemos activar la casilla de public static void main:



2. CRUD CON JDBC-MYSQL

Paso 3. Escribimos el siguiente código en el main del programa, en 3 fases diferenciadas:

- Carga del driver Mysql Java
- Instanciación e inicialización de la variable Connection de conexión con la base de datos
- Creación de un CRUD mediante un simple while con las diferentes acciones que queremos implementar en la base de datos

```
public class Crud {  
  
    static Scanner reader;  
    static Connection conexion;  
  
    public static void main(String[] args) throws Exception {  
  
        //CARGA DEL DRIVER MYSQL JAVA  
        Class.forName("com.mysql.cj.jdbc.Driver");  
        String conex="jdbc:mysql://localhost:3306/empresa";  
  
        //INICIALIZAMOS LA CONEXIÓN A LA BASE DE DATOS  
        conexion = DriverManager.getConnection (conex,"root","");  
  
        reader = new Scanner(System.in);  
        //CRUD--Create, Read, Update, Delete  
        while (true) {  
            System.out.println("1. Visualizar la lista de empleados");  
            System.out.println("2. Incrementar salario de empleado");  
            System.out.println("3. Insertar un nuevo empleado");  
            System.out.println("4. Borrar un nuevo empleado");  
            System.out.println("5. Ejecuta Procedimiento usuarios");  
            System.out.println("6. Ejecuta Function inversa");  
            System.out.println("7. Salir");  
            System.out.print("Introduce que opcion quieres?");  
            int opcion = reader.nextInt();  
            switch(opcion) {  
                case 1:  
                    listar();  
                    break;  
                case 2:  
                    listar();  
            }  
        }  
    }  
}
```

2. CRUD CON JDBC-MYSQL

Paso 3 (Opcional). El nuevo driver versión 8 en determinadas circunstancias podría necesitar de algunas modificaciones en el string de conexión de la llamada desde Java:

```
public static void main(String[] args) {
    try {
        //Para el Driver mysql-connector-java-8.0.15.jar
        Class.forName("com.mysql.cj.jdbc.Driver").newInstance();
        String conex = "jdbc:mysql://localhost/futbol_mysql?"
            + "useUnicode=true&"
            + "useJDBCCompliantTimezoneShift=true&"
            + "useLegacyDatetimeCode=false&"
            + "serverTimezone=UTC";
        Connection conn=DriverManager.getConnection(conex,"root","");
        DatabaseMetaData pepe=conn.getMetaData();
        Statement s = conn.createStatement();
        String selTable = "select * from equipos";
        s.execute(selTable);
        ResultSet rs = s.getResultSet();
        while((rs!=null) && (rs.next()))
            System.out.println(rs.getString(1) + " : " + rs.getString(2) +
                " : " + rs.getString(3) + " : " + rs.getString(4));
    }
    catch(Exception ex)
    {
        System.out.print(ex.getMessage());
    }
}
```

2. CRUD CON JDBC-MYSQL

Paso 4. Implementamos la función listar. Básicamente lo que hace es un: "SELECT * FROM EMPLEADOS"

```
public static void listar() throws SQLException {
    System.out.println("-----");
    String sql= "select * from empleados;";
    Statement sta= conexion.createStatement();
    ResultSet rs = (ResultSet)sta.executeQuery(sql);
    while (rs.next()) {
        System.out.println(rs.getInt(1) + "-" + rs.getString(2) + "-" +
            rs.getString(3) + "-" +rs.getString(4));
    }
    System.out.println("-----");
}
```

2. CRUD CON JDBC-MYSQL

Paso 5. Implementamos la función actualizar. Básicamente hace un :
“UPDATE EMPLEADOS SET COLUMNA=VALOR WHERE ”

```
public static int actualizar() throws SQLException {
    String sql= "update `empleados` set `salario` = `salario` + 10000 "
        + " where `cod_emp`=?";
    System.out.print("Introduce id de empleado: ");

    int id = reader.nextInt();
    PreparedStatement sta = conexion.prepareStatement(sql);
    sta.setInt(1, id);
    return sta.executeUpdate();
}
```


2. CRUD CON JDBC-MYSQL

Paso 6. Implementamos la función insertar. Básicamente hace un :
“INSERT INTO EMPLEADOS VALUES () ”

```
public static int insertar() throws SQLException {
    System.out.print("Introduce nombre empleado: ");
    String nombre=reader.next();
    System.out.print("Introduce apellido empleado: ");
    String apellidos=reader.next();
    System.out.print("Introduce salario: ");
    int salario=reader.nextInt();
    String sql= "insert into empleados(nombre, apellidos, salario) "
               + " values (?, ?, ?)";
    PreparedStatement sta = conexion.prepareStatement(sql);
    sta.setString(1, nombre);
    sta.setString(2, apellidos);
    sta.setInt(3, salario);
    return sta.executeUpdate();
}
```

2. CRUD CON JDBC-MYSQL

Paso 7. Implementamos la función borrar. Básicamente hace un :
“DELETE FROM EMPLEADOS WHERE () ”

```
public static int borrar() throws SQLException {  
    String sql= "delete from empleados where cod_emp =?";  
    System.out.print("Introduce id de empleado: ");  
    int id=reader.nextInt();  
    PreparedStatement sta = conexion.prepareStatement(sql);  
    sta.setInt(1, id);  
    return sta.executeUpdate();  
}
```

2. CRUD CON JDBC-MYSQL

Paso 8. Crea un procedimiento almacenado en Mysql y realiza las llamadas desde Java:

```
CREATE PROCEDURE usuarios()  
    select * from mysql.user;
```

```
public static void executeProcedure() throws SQLException {  
    // String query = "{ call usuarios(?) }";  
    String query = "{ call usuarios() }";  
    CallableStatement stmt = conexion.prepareCall(query);  
    //stmt.setInt(1, candidateId);  
  
    ResultSet rs = stmt.executeQuery();  
    while (rs.next()) {  
        System.out.println(rs.getString(1) + "-" + rs.getString(2) + "-" +  
            rs.getString(3) + "-" + rs.getString(4));  
    }  
    System.out.println();  
}
```

2. CRUD CON JDBC-MYSQL

Paso 9. Crea una función en Mysql y realiza la llamada desde Java:

```
CREATE FUNCTION inversa (PALABRA VARCHAR(20))  
    RETURNS VARCHAR(20)  
    RETURN REVERSE(PALABRA);
```

```
public static void executeFunction() throws SQLException {  
    //Preparing a CallableStatement to call a function  
    CallableStatement cstmt = conexion.prepareCall("{? = call inversa(?)}");  
    //Registering the out parameter of the function (return type)  
    cstmt.registerOutParameter(1, Types.VARCHAR);  
    //Setting the input parameters of the function  
    cstmt.setString(2, "Pedro");  
    //Executing the statement  
    cstmt.execute();  
    System.out.println("Pedro invertida es : "+cstmt.getString(1));  
}
```

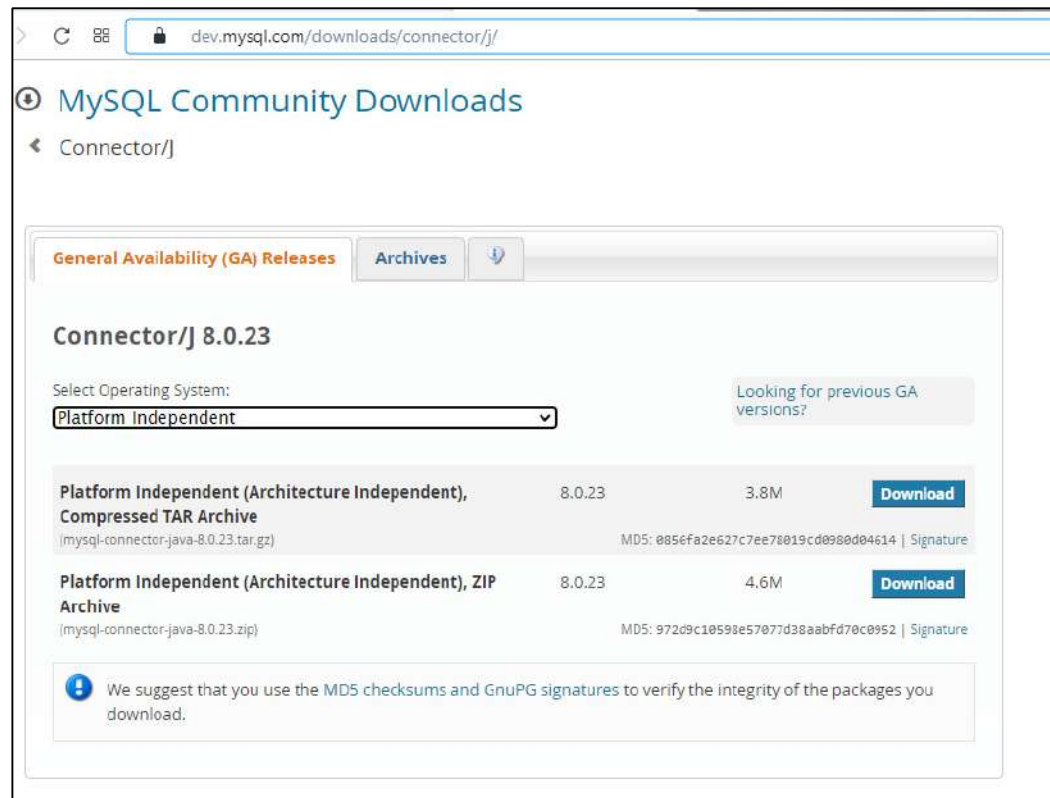
2. CRUD CON JDBC-MYSQL

Paso 10. Tenemos 3 opciones para agregar el driver de conexión Mysql Connector para java en nuestro proyecto:

- Manualmente, descargando el driver jar de la pagina web Mysql Connector
- Mediante el repositorio Maven
- Mediante el repositorio Gradle

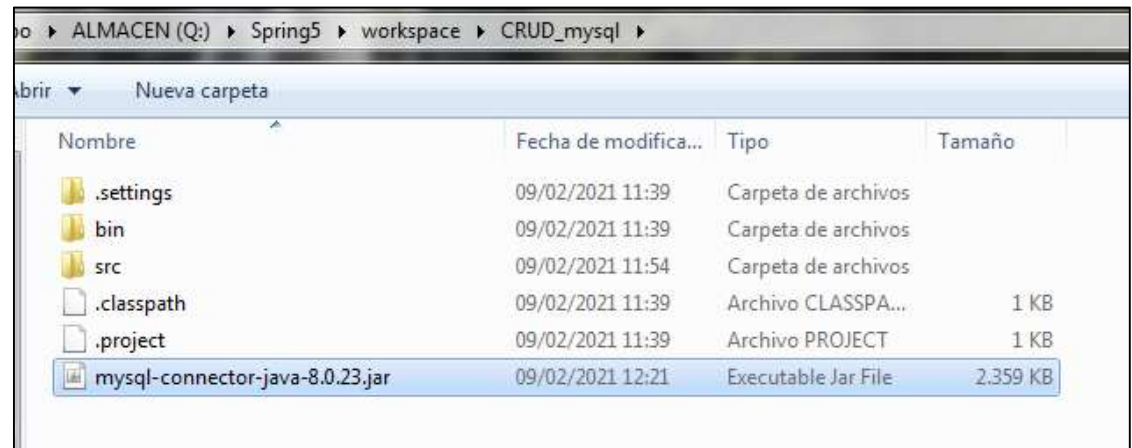
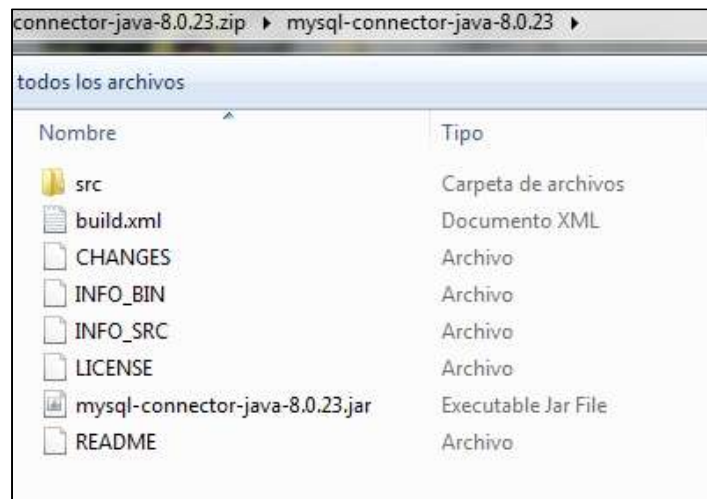
2. CRUD CON JDBC-MYSQL

Paso 11 (Manual). Si lo hacemos manualmente, debemos ir a la pagina Mysql Connector <https://dev.mysql.com/downloads/connector/j/> y descargar la ultima versión del driver:



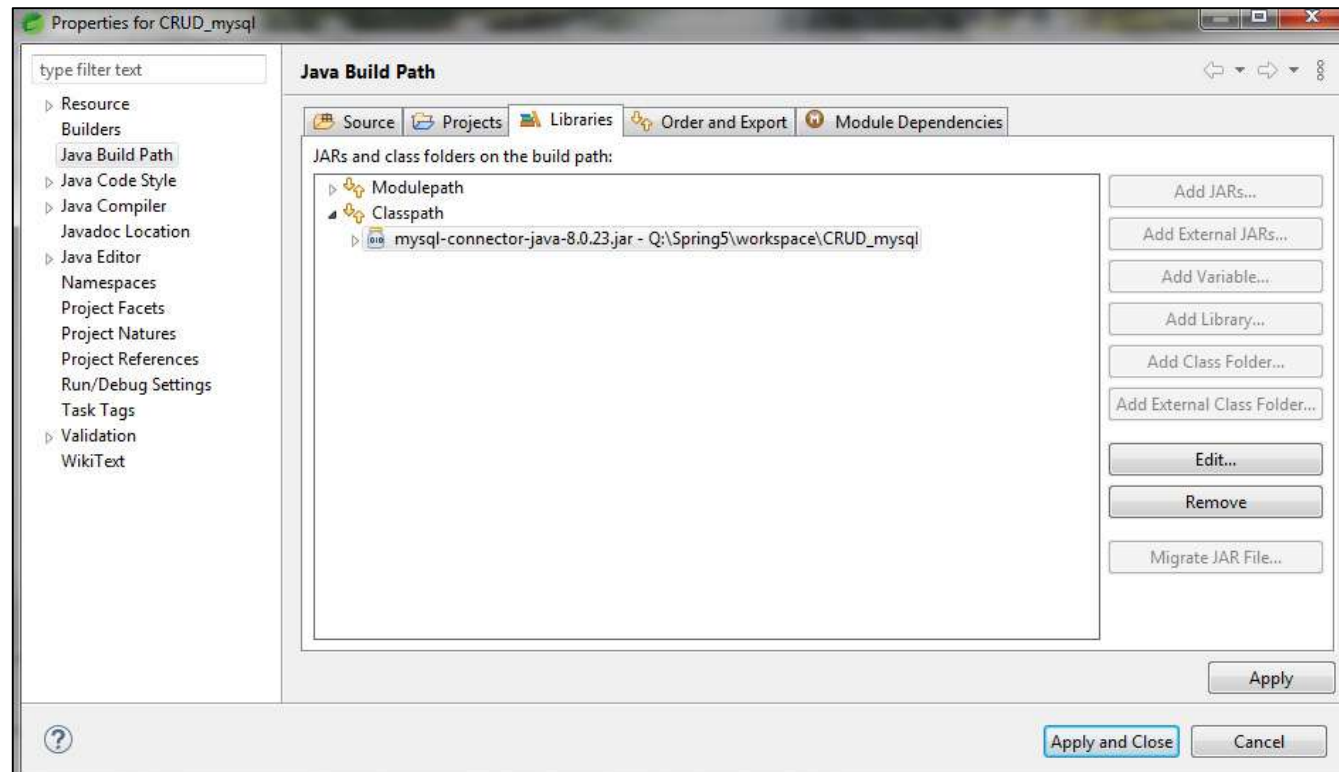
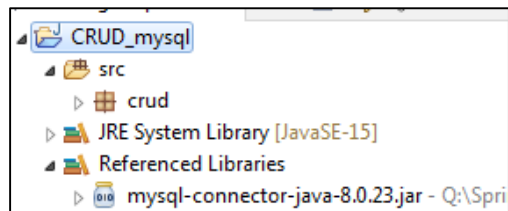
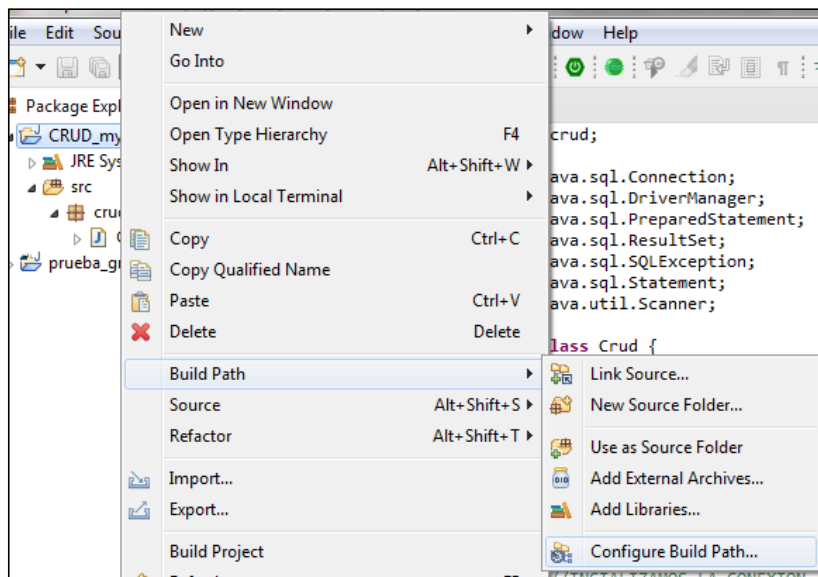
2. CRUD CON JDBC-MYSQL

Paso 12 (Manual). Descomprimos el fichero zip, extraemos el driver jar, y lo agregamos a nuestro proyecto. Dejamos el jar dentro de nuestro proyecto, por si tenemos que hacer una copia, el jar venga ya incluido



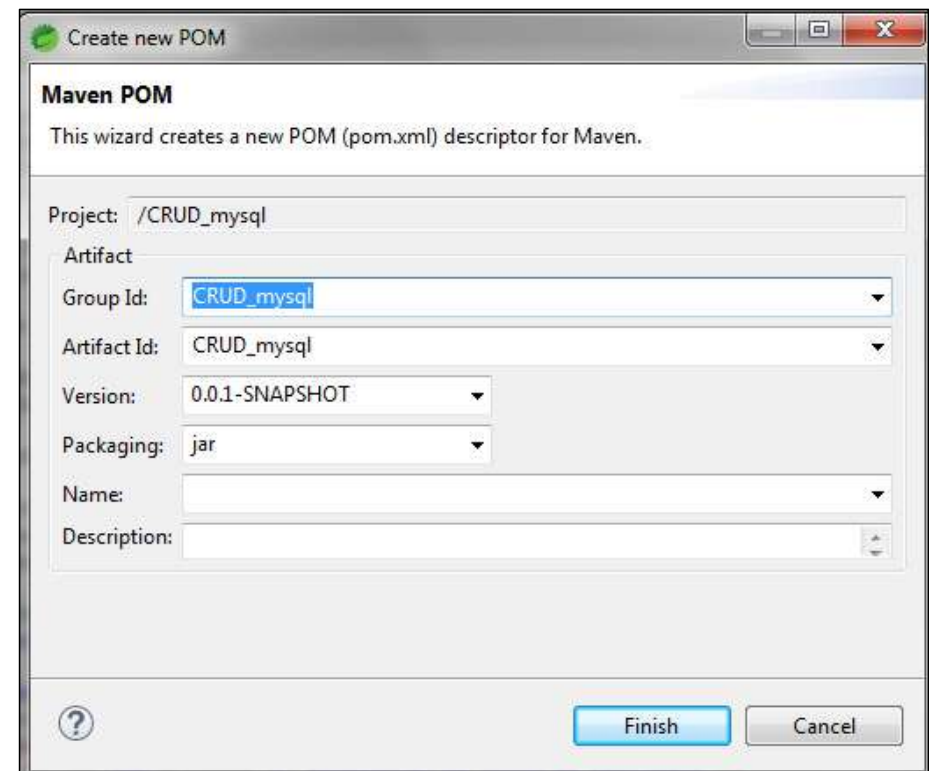
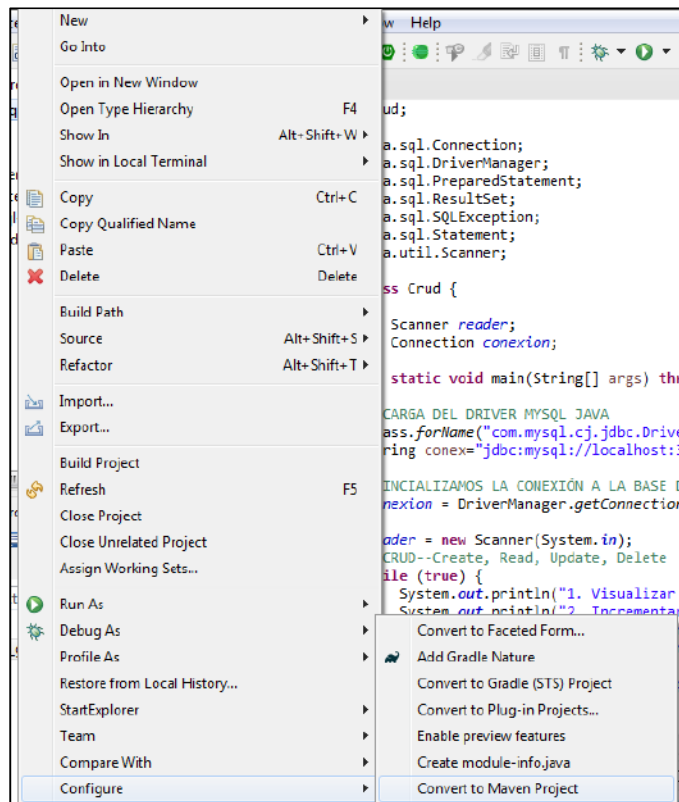
2. CRUD CON JDBC-MYSQL

Paso 13 (Manual). Vamos a la opción Java Build Path y agregamos esta librería a nuestro proyecto:



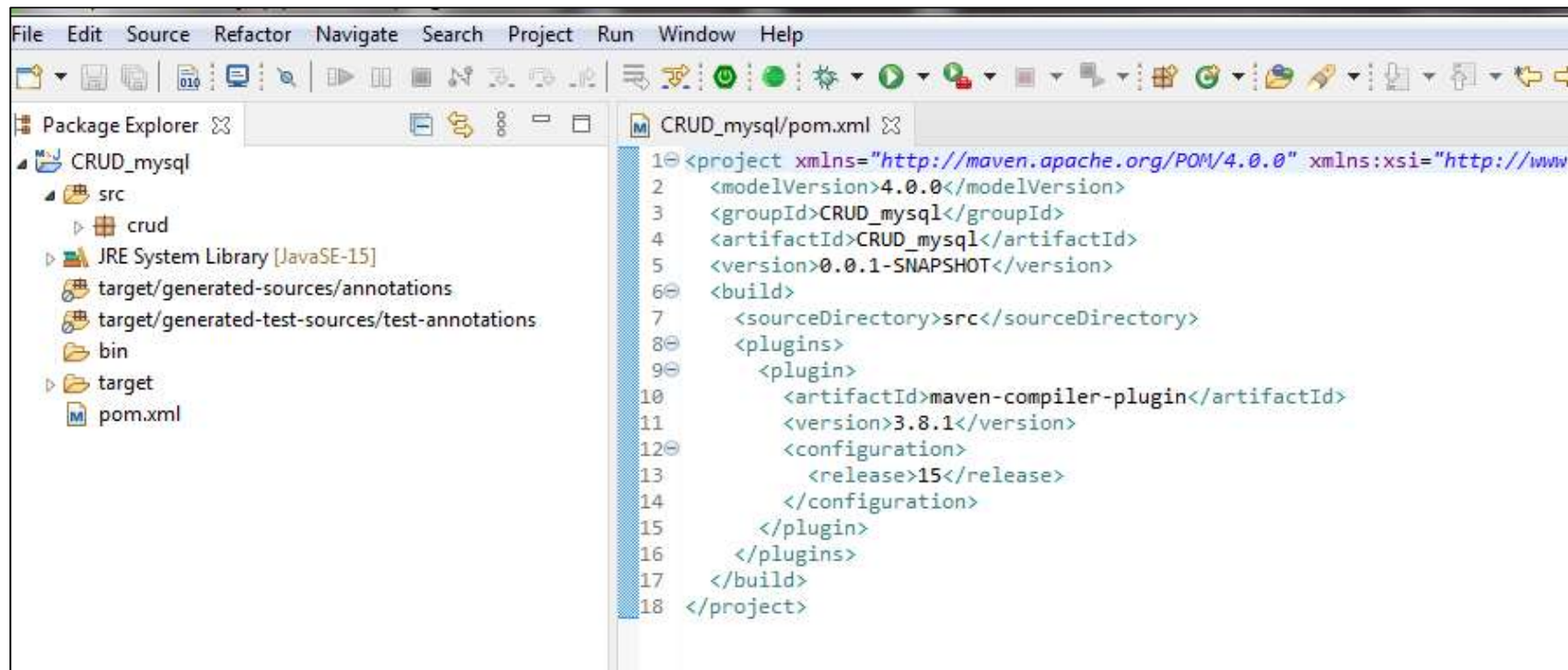
2. CRUD CON JDBC-MYSQL

Paso 14 (Maven). Para poder importar la librería mysql connector a nuestro proyecto via Maven, primero debe de convertirse nuestro proyecto en un proyecto Maven:



2. CRUD CON JDBC-MYSQL

Paso 15 (Maven). Una vez convertido nuestro proyecto en proyecto maven, podemos observar el fichero pom.xml, lugar donde se guardan las dependencias que debemos agregar para obtener las librerías del repositorio:

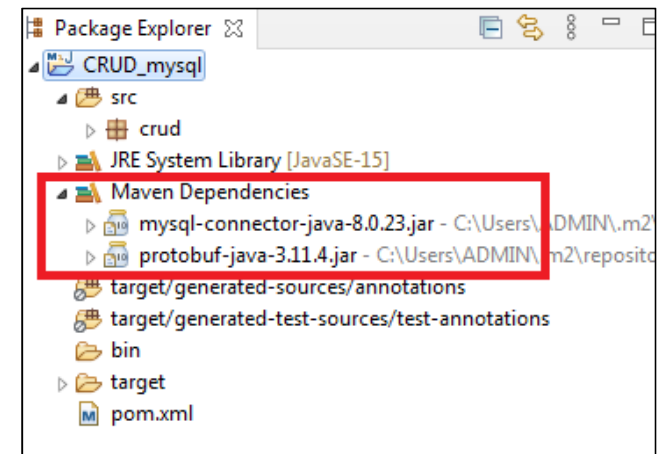


2. CRUD CON JDBC-MYSQL

Paso 16 (Maven). Agregamos la dependencia del mysql connector en el fichero pom.xml

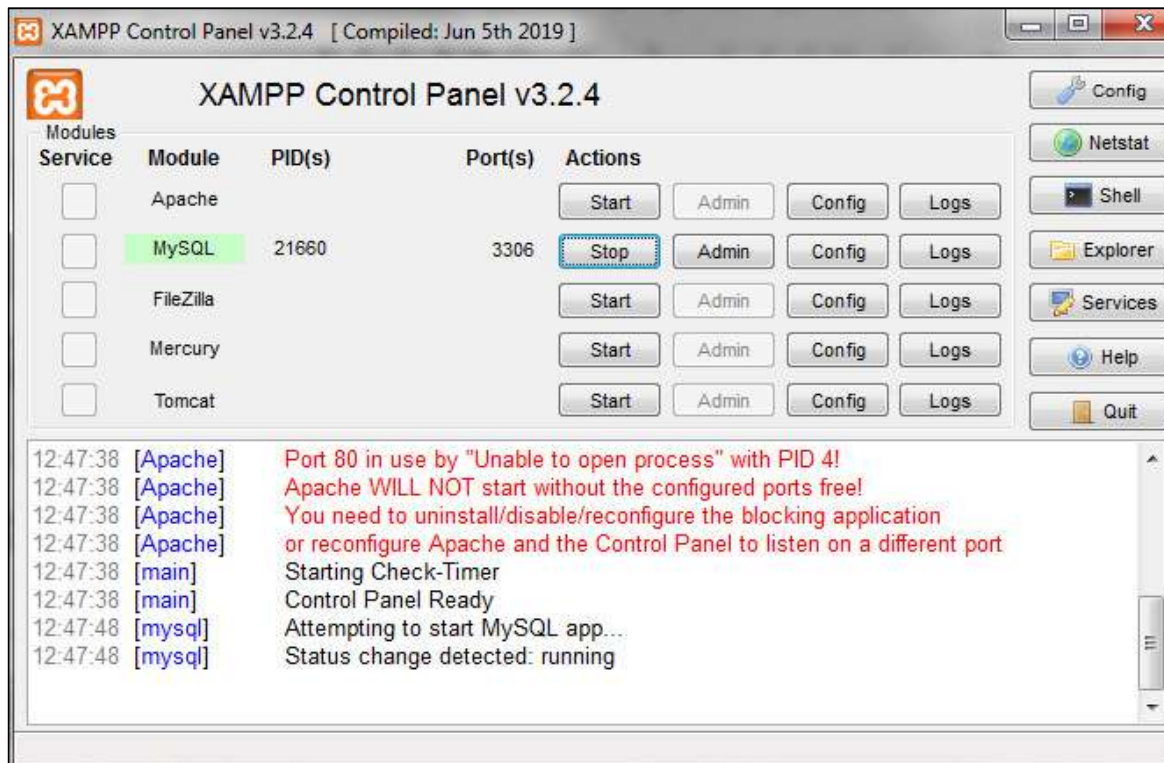


```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
2 <modelVersion>4.0.0</modelVersion>
3 <groupId>CRUD_mysql</groupId>
4 <artifactId>CRUD_mysql</artifactId>
5 <version>0.0.1-SNAPSHOT</version>
6 <build>
7 <sourceDirectory>src</sourceDirectory>
8 <plugins>
9 <plugin>
10 <artifactId>maven-compiler-plugin</artifactId>
11 <version>3.8.1</version>
12 <configuration>
13 <release>15</release>
14 </configuration>
15 </plugin>
16 </plugins>
17 </build>
18
19 <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
20 <dependencies>
21 <dependency>
22 <groupId>mysql</groupId>
23 <artifactId>mysql-connector-java</artifactId>
24 <version>8.0.23</version>
25 </dependency>
26 </dependencies>
27
28 </project>
29
```



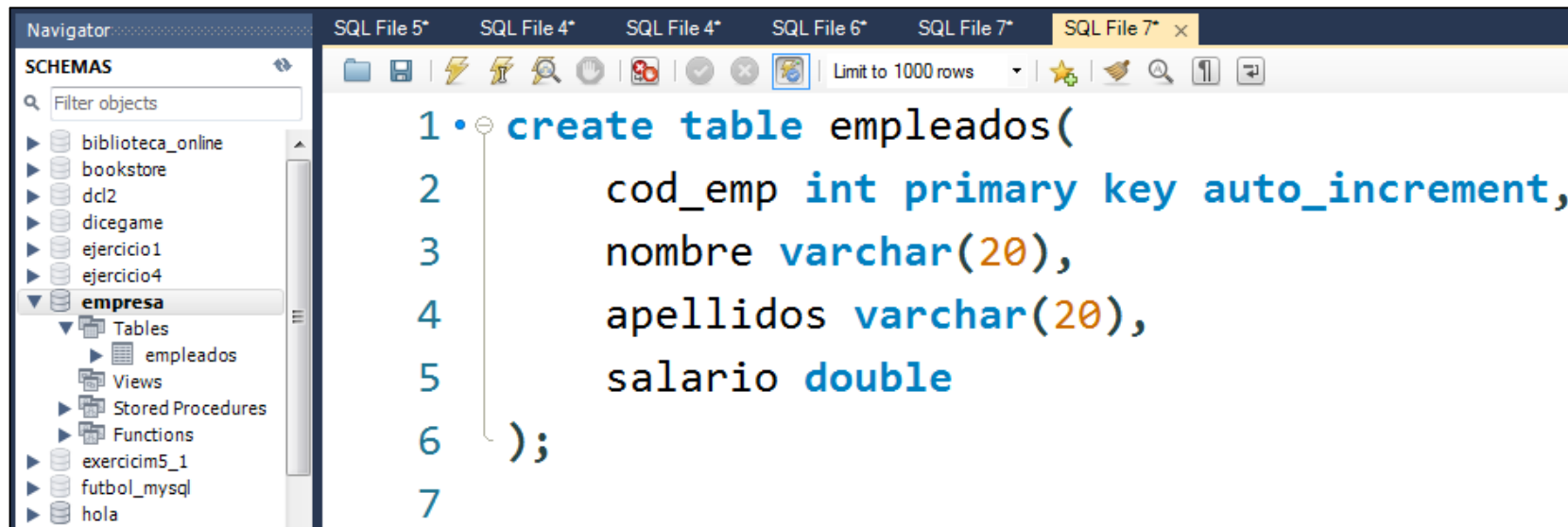
2. CRUD CON JDBC-MYSQL

Paso 17. En el caso de la base de datos mysql, debemos activar previamente el servicio desde el control panel de xampp. Se levanta en el puerto 3306.



2. CRUD CON JDBC-MYSQL

Paso 18. Desde Phpmyadmin o MySql Workbench podemos crear la base de datos empresa y la tabla empleados si no las tenemos creadas:



2. CRUD CON JDBC-MYSQL

Paso 19. Resultado de la ejecución:

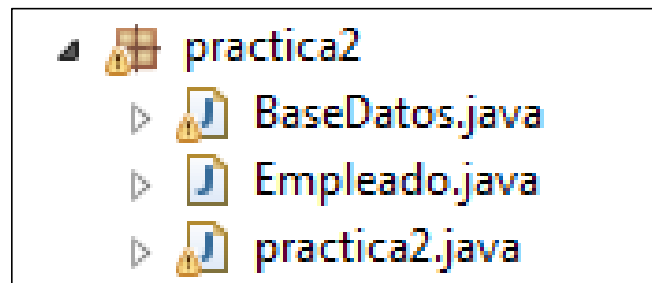
```
Problems @ Javadoc Declaration Console
Crud [Java Application] Q:\Spring5\sts-4.9.0.RELEASE\pl
1. Visualizar la lista de empleados
2. Incrementar salario de empleado
3. Insertar un nuevo empleado
4. Borrar un nuevo empleado
5. Ejecuta Procedimiento usuarios
5. Ejecuta Function inversa
6. Salir
Introduce que opcion quieres?5
localhost-root--Y
%-Maria--Y
127.0.0.1-root--Y
::1-root--Y
localhost-pma--N

Pedro invertida es : ordeP
1. Visualizar la lista de empleados
2. Incrementar salario de empleado
3. Insertar un nuevo empleado
4. Borrar un nuevo empleado
5. Ejecuta Procedimiento usuarios
5. Ejecuta Function inversa
6. Salir
Introduce que opcion quieres?6
Pedro invertida es : ordeP
1. Visualizar la lista de empleados
2. Incrementar salario de empleado
3. Insertar un nuevo empleado
4. Borrar un nuevo empleado
5. Ejecuta Procedimiento usuarios
5. Ejecuta Function inversa
6. Salir
Introduce que opcion quieres?
```

3. EVOLUCION HACIA JPA-HIBERNATE

Dividiremos el único fichero anterior crud.java en otros 3 ficheros:

- BaseDatos.java: Agrupa las funcionalidades del acceso a base de datos en la una clase java. Debe tener un atributo Connection y establecer la conexión en el constructor. Debe contener el resto de funciones de base de datos: insertar, listar, etc
- Empleados.java: La clase pojo Entity que utilizaran las funciones insertar, borrar y actualizar como parámetro de entrada.
- Main.java : Contendrá el programa principal con el bucle y las opciones del menu



3. EVOLUCION HACIA JPA-HIBERNATE

BaseDatos.java

```
11 public class BaseDatos {
12     private Connection conexion;
13
14     public BaseDatos() {
15         try {
16             Class.forName("com.mysql.cj.jdbc.Driver");
17             String conex="jdbc:mysql://localhost:3306/empresa";
18             this.conexion = DriverManager.getConnection (conex,"root","");
19
20         } catch (Exception e) {
21             e.printStackTrace();
22         }
23     }
24
25     public void listar() { }
40     public int actualizar(Empleado emp) {}
53     public int insertar(Empleado emp) {}
68     public int borrar(Empleado emp) {}
80 }
91
```


3. EVOLUCION HACIA JPA-HIBERNATE

Empleado.java

```
public class Empleado {
    private int id;
    private String nombre;
    private String apellidos;
    private int salario;

    public Empleado() {
    }

    public Empleado(int id, String nombre, String apellidos, int salario) {
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.salario = salario;
    }

    public int getId() {
        return id;
    }

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

3. EVOLUCION HACIA JPA-HIBERNATE

Main.java

```
public static void main(String[] args) {

    Scanner reader = new Scanner(System.in);
    BaseDatos db = new BaseDatos();
    //CRUD Create - Read - Update - Delete
    while (true) {
        System.out.println("1. Visualizar la lista de empleados");
        System.out.println("2. Incrementar salario de empleado");
        System.out.println("3. Insertar un nuevo empleado");
        System.out.println("4. Borrar un nuevo empleado");
        System.out.println("5. Salir");
        System.out.print("Introduce que opcion quieres?");
        int opcion = reader.nextInt();
        switch(opcion) {
            case 1:
                db.listar();
                break;
            case 2:
                db.listar();
                System.out.print("Introduce id de empleado: ");
                int id = reader.nextInt();
                db.actualizar(new Empleado(id, "", "", 0));
                db.listar();
                break;
        }
    }
}
```

4. HIBERNATE DE FORMA MANUAL

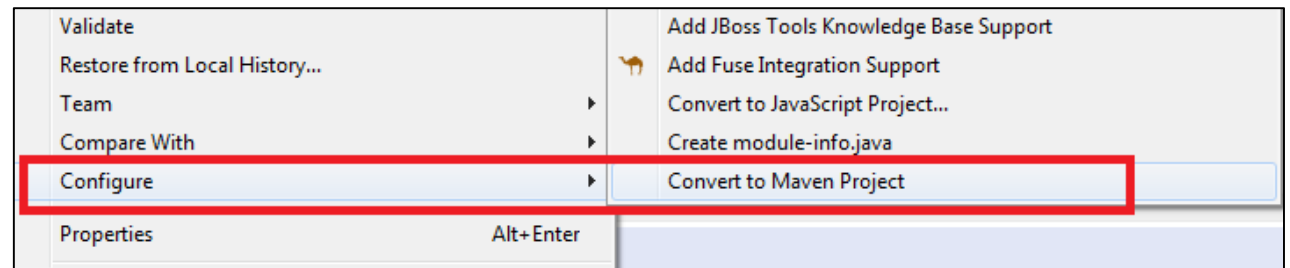
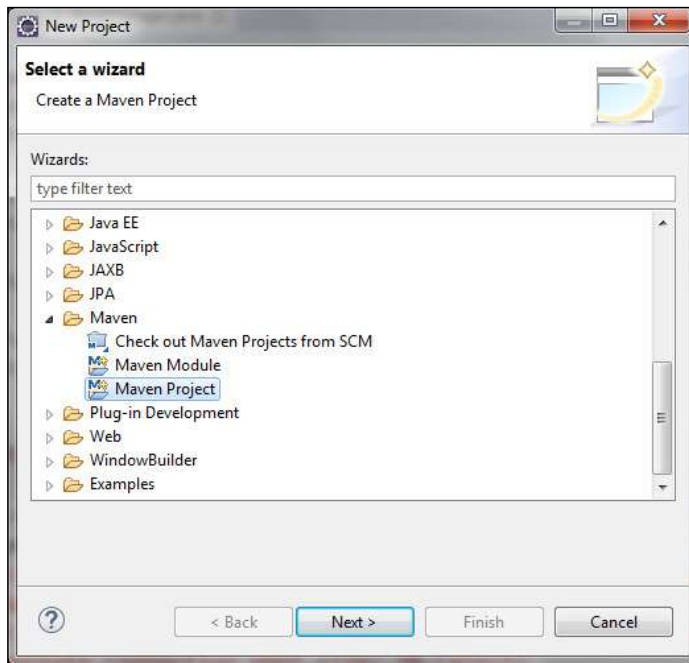
Apache Maven

- Herramienta de gestión de proyectos de software. Basado en el concepto de un modelo de objetos de proyecto (POM),
- Maven puede administrar la construcción, informes y documentación de un proyecto a partir de una pieza central de información: el fichero POM.xml.
- Maven es un repositorio de Internet que permite añadir dependencias a nuestro proyecto sin necesidad de descargar las librerías físicamente.
- **Para ello utiliza el fichero de configuración del Proyecto Maven (pom.xml). Basta con escribir las dependencias necesarias en el fichero pom.xml y Maven se encarga de descargar automáticamente todas esas librerías.**
- **MAVEN no es necesario para HIBERNATE, pero ayuda a la obtención de las librerías**

4. HIBERNATE DE FORMA MANUAL

Paso 1 (opcional con Maven). Tenemos dos opciones:

- a) Crear un nuevo proyecto Maven yendo a **File/New/Project/Maven Project**
- b) Convertir nuestro proyecto a Maven, en **Configure/Convert to Maven Project**. Esta será la opción elegida.



4. HIBERNATE DE FORMA MANUAL

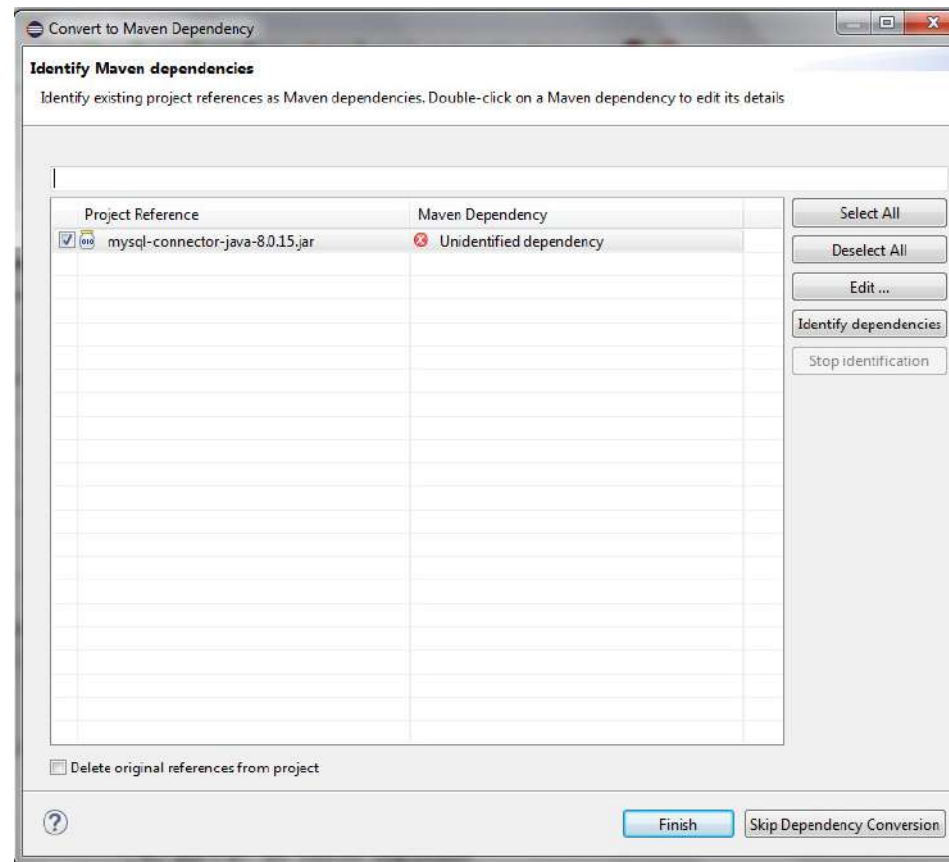
Paso 2 (opcional con Maven). Entra la información para el artifact y haz click en **Finish**:

- Group Id: practica1
- Artifact Id: practica1
- Name: practica Hibernate

The screenshot shows a 'Create new POM' dialog box. The title bar says 'Create new POM'. The main title is 'Maven POM'. Below the title, it says 'This wizard creates a new POM (pom.xml) descriptor for Maven.' The dialog contains several input fields: 'Project' with the value '/practica', 'Artifact' section with 'Group Id' and 'Artifact Id' both set to 'practica1', 'Version' set to '0.0.1-SNAPSHOT', 'Packaging' set to 'jar', 'Name' set to 'practica hibernate', and an empty 'Description' field. At the bottom right, there are 'Finish' and 'Cancel' buttons. A help icon (?) is at the bottom left.

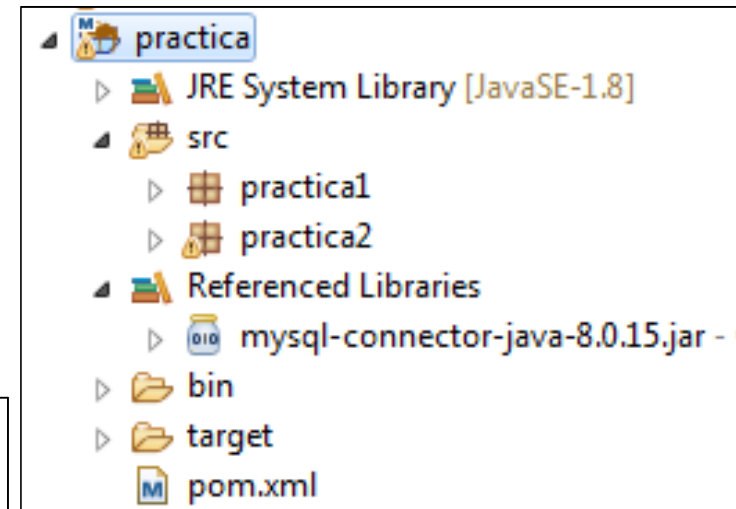
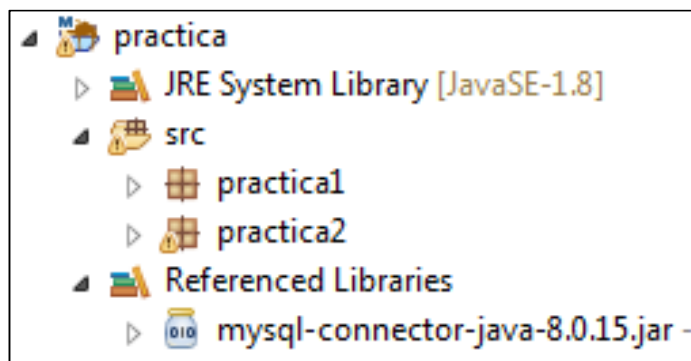
4. HIBERNATE DE FORMA MANUAL

Paso 3 (opcional con Maven). Detecta una dependencia en el proyecto pero no la reconoce. Hacemos click en Finish.

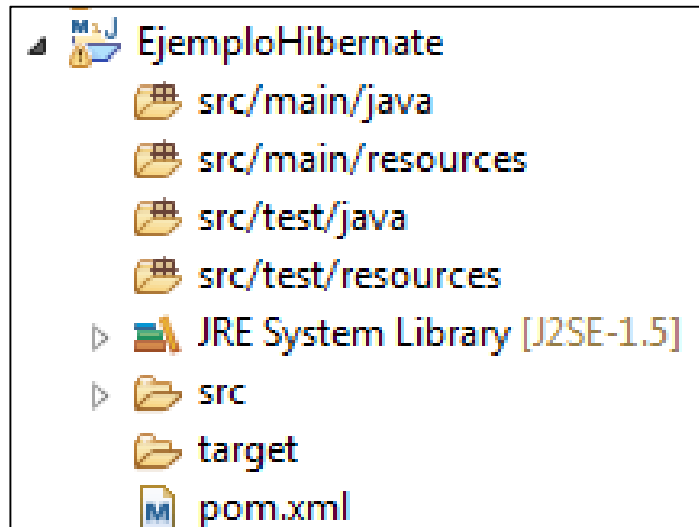


4. HIBERNATE DE FORMA MANUAL

Paso 4 (opcional con Maven). Eclipse generará la estructura para el proyecto:



Si hubiésemos creado un Nuevo Proyecto Maven, se hubiera generado la siguiente estructura del proyecto



4. HIBERNATE DE FORMA MANUAL

Paso 5 (opcional con Maven). Accede al fichero **pom.xml** para sumar las dependencias para las librerías Hibernate y MySQL Connector Java. Agrega el siguiente código XML entre el elemento `<project></project>`:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.6.Final</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.40</version>
  </dependency>
</dependencies>
```

```
</project>
```


4. HIBERNATE DE FORMA MANUAL

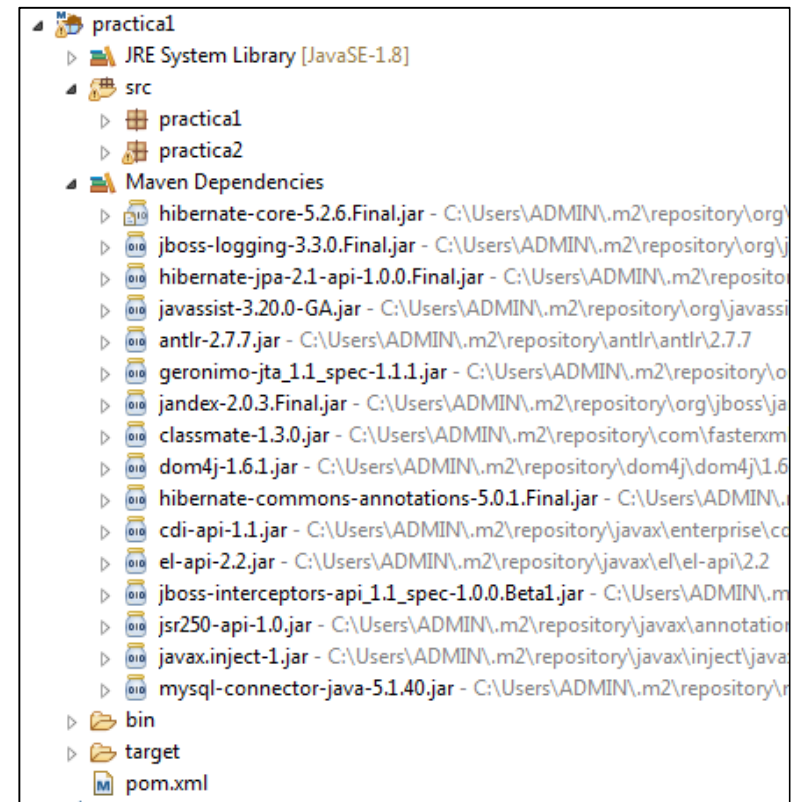
Paso 6 (opcional con Maven). Si se tiene instalada la versión del jdk de java 9 o superior, se debe de agregar en el fichero **pom.xml** las siguientes dependencias:

```
<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.2</version>
</dependency>
</dependencies>
```

Importante

4. HIBERNATE DE FORMA MANUAL

Paso 7 (opcional con Maven). Haz click en **Save (Ctrl + S)** y Maven automáticamente descargará los ficheros JAR de dependencias: Hibernate core y el driver MySQL Connector Java. Se pueden ver los ficheros JAR agregados bajo la entrada del Proyecto **Maven Dependencies**.

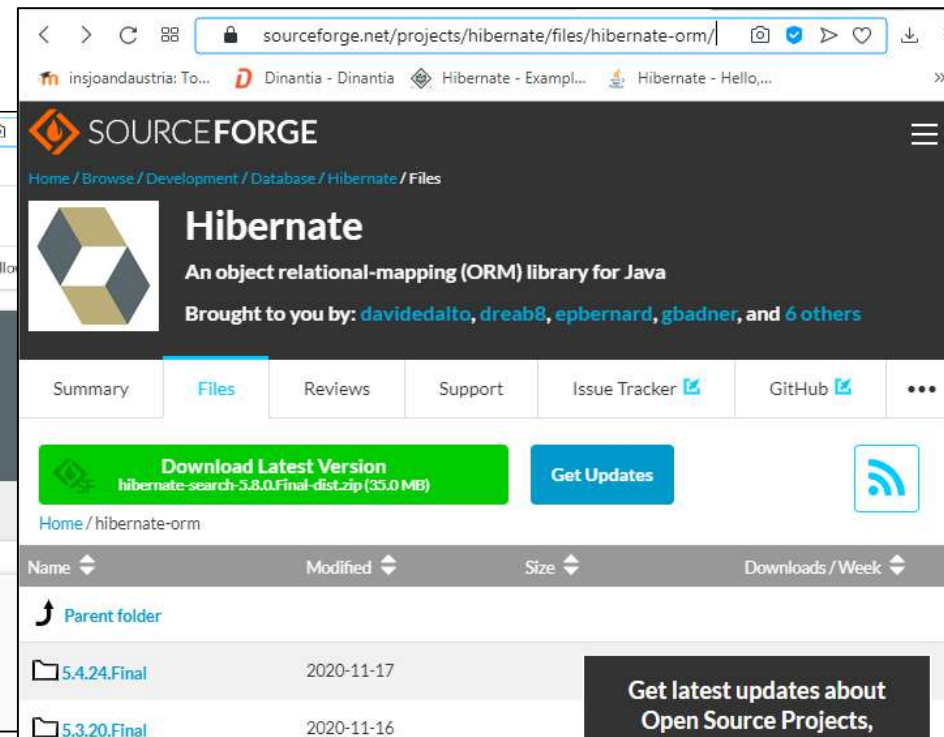
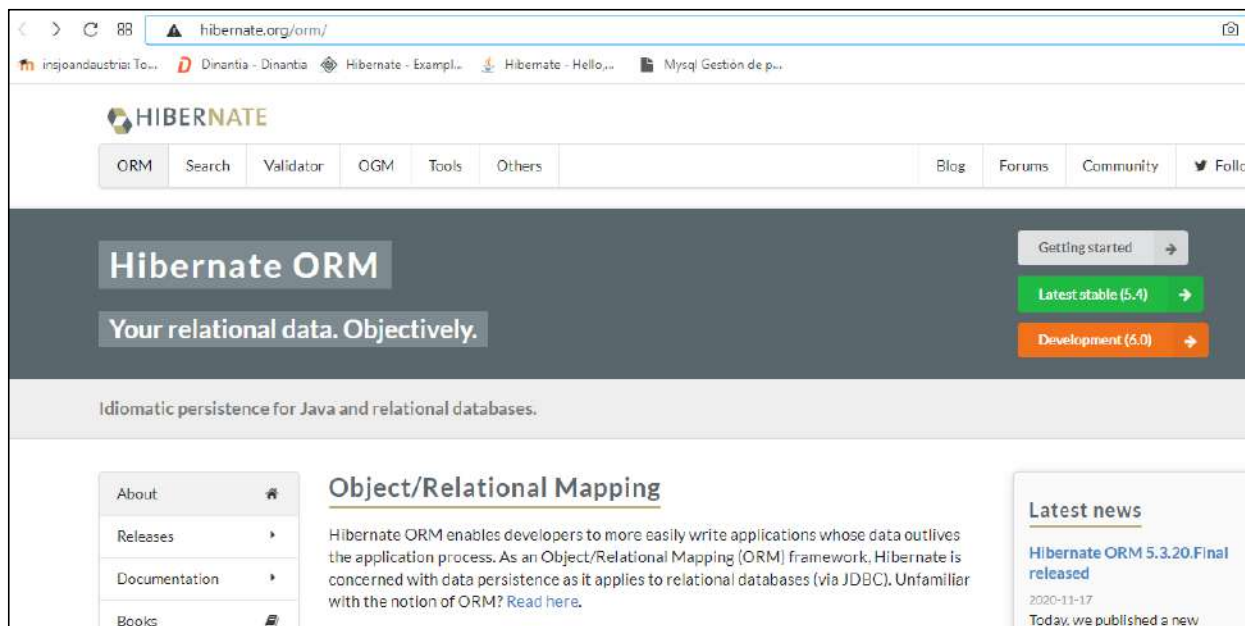


4. HIBERNATE DE FORMA MANUAL

Paso 1 (sin Maven). Descarga las librerías de Hibernate de la siguiente URL:

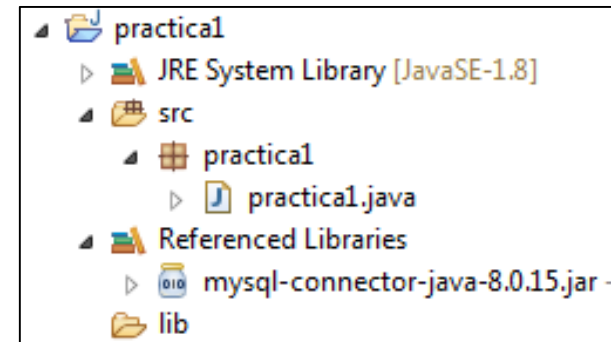
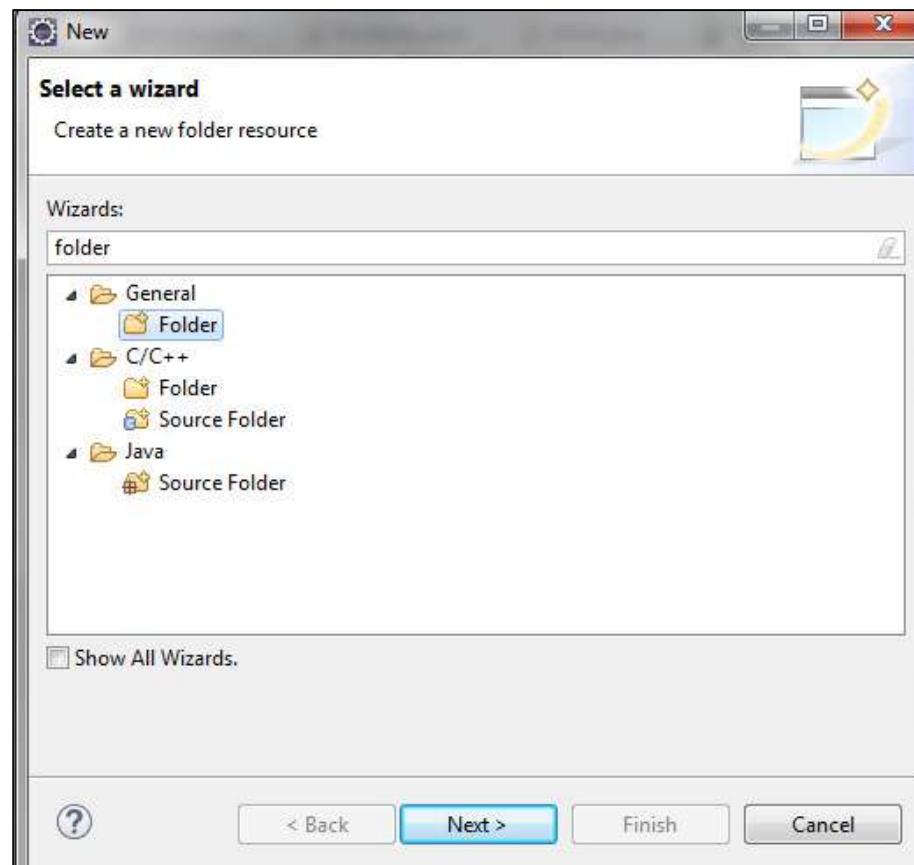
<http://hibernate.org/orm/>

<https://sourceforge.net/projects/hibernate/>



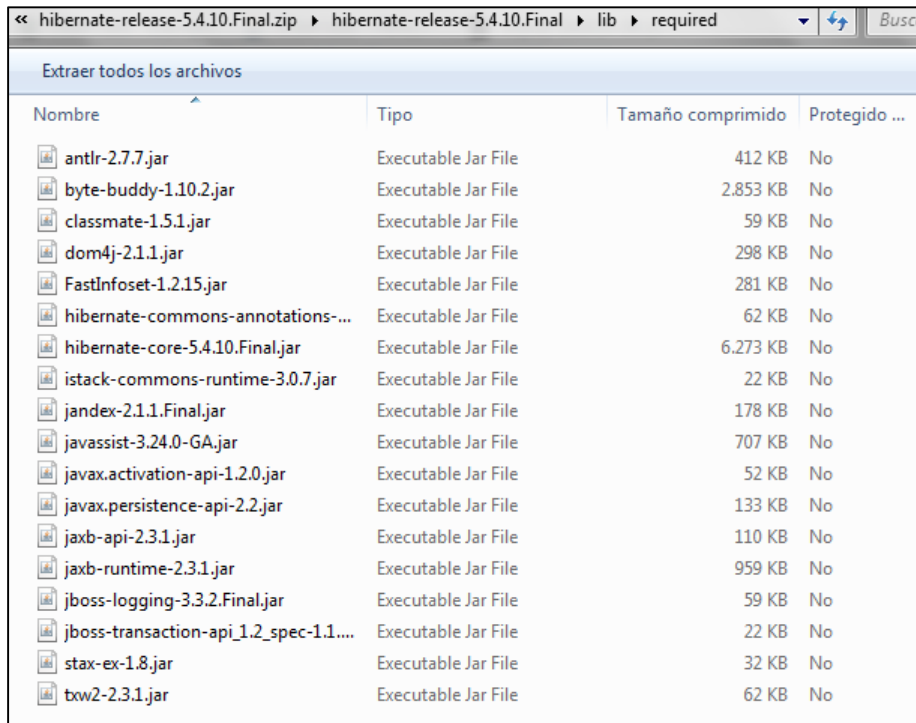
4. HIBERNATE DE FORMA MANUAL

Paso 2 (sin Maven). Crea un nueva carpeta “lib” en el proyecto.

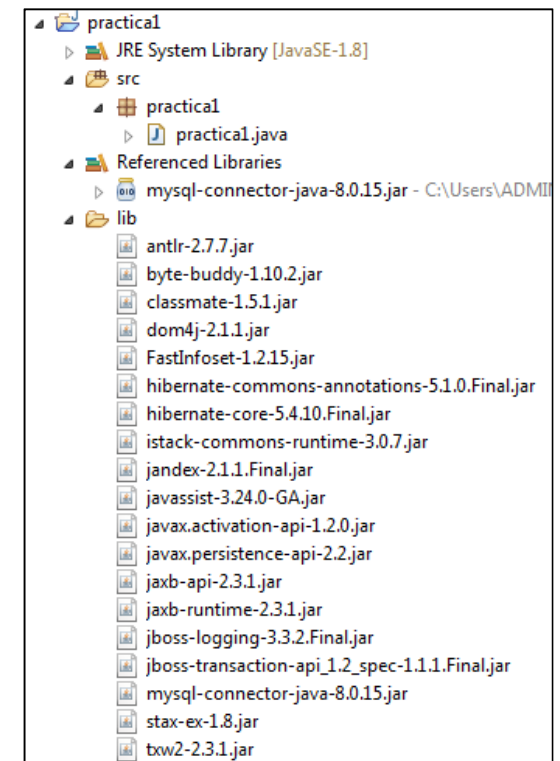


4. HIBERNATE DE FORMA MANUAL

Paso 3 (sin Maven). Copia todos los ficheros **.jar* files de la carpeta “*hibernate-release-xxx/lib/required*” y pégalos en la carpeta “*lib*”. Copia también el *mysql-connector-java-xxx-bin.jar* a la carpeta “*lib*” del proyecto.

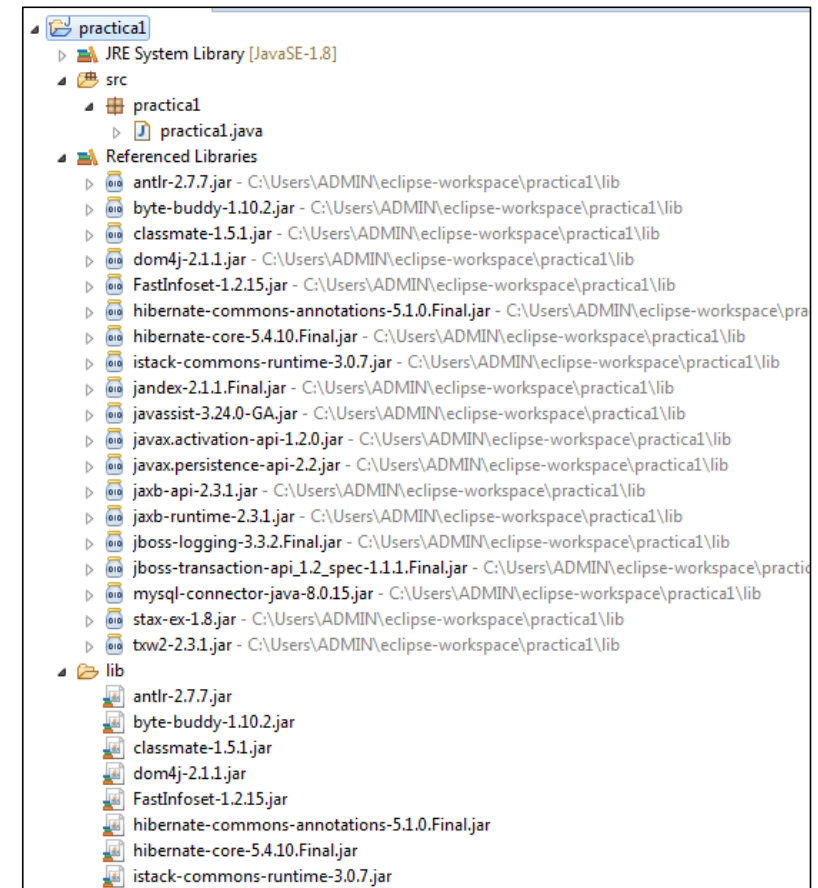
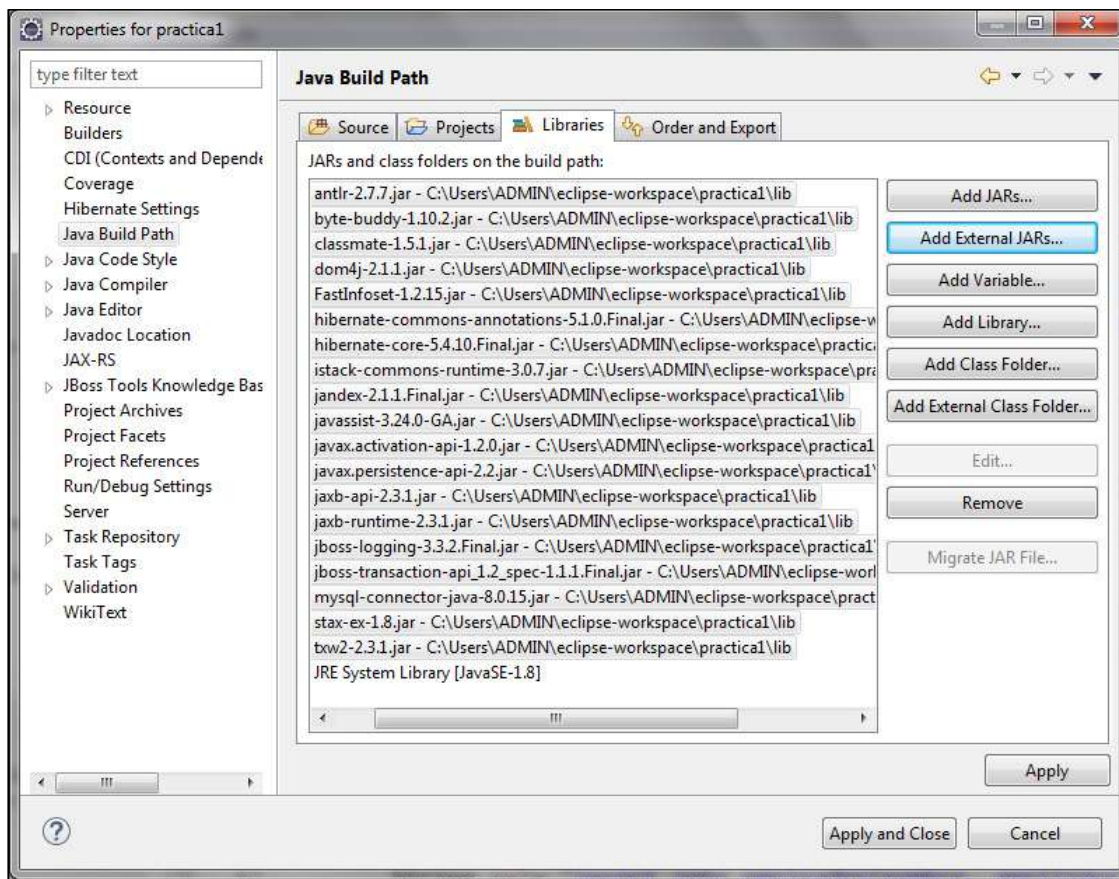


Nombre	Tipo	Tamaño comprimido	Protegido ...
antlr-2.7.7.jar	Executable Jar File	412 KB	No
byte-buddy-1.10.2.jar	Executable Jar File	2.853 KB	No
classmate-1.5.1.jar	Executable Jar File	59 KB	No
dom4j-2.1.1.jar	Executable Jar File	298 KB	No
FastInfoset-1.2.15.jar	Executable Jar File	281 KB	No
hibernate-commons-annotations-...	Executable Jar File	62 KB	No
hibernate-core-5.4.10.Final.jar	Executable Jar File	6.273 KB	No
istack-commons-runtime-3.0.7.jar	Executable Jar File	22 KB	No
jandex-2.1.1.Final.jar	Executable Jar File	178 KB	No
javassist-3.24.0-GA.jar	Executable Jar File	707 KB	No
javax.activation-api-1.2.0.jar	Executable Jar File	52 KB	No
javax.persistence-api-2.2.jar	Executable Jar File	133 KB	No
jaxb-api-2.3.1.jar	Executable Jar File	110 KB	No
jaxb-runtime-2.3.1.jar	Executable Jar File	959 KB	No
jboss-logging-3.3.2.Final.jar	Executable Jar File	59 KB	No
jboss-transaction-api_1.2_spec-1.1....	Executable Jar File	22 KB	No
stax-ex-1.8.jar	Executable Jar File	32 KB	No
txw2-2.3.1.jar	Executable Jar File	62 KB	No



4. HIBERNATE DE FORMA MANUAL

Paso 4 (sin Maven). Agrega todas las librerías, yendo a Properties/Java Build Path/Libraries



4. HIBERNATE DE FORMA MANUAL

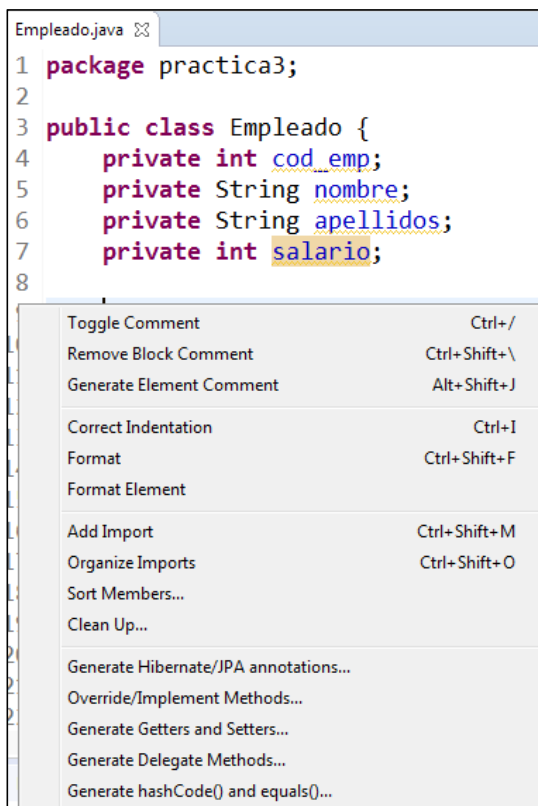
Paso 5. Crea la clase Empleado bajo el package practica3 (o el que corresponda), que tenga 4 campos igual que las columnas en la tabla empleados.

Creamos la clase entity class que mapeará la tabla empleados de la base de datos, usando anotaciones.

```
Empleado.java ✕  
1 package practica3;  
2  
3 public class Empleado {  
4     private int cod_emp;  
5     private String nombre;  
6     private String apellidos;  
7     private int salario;  
8 }  
9
```

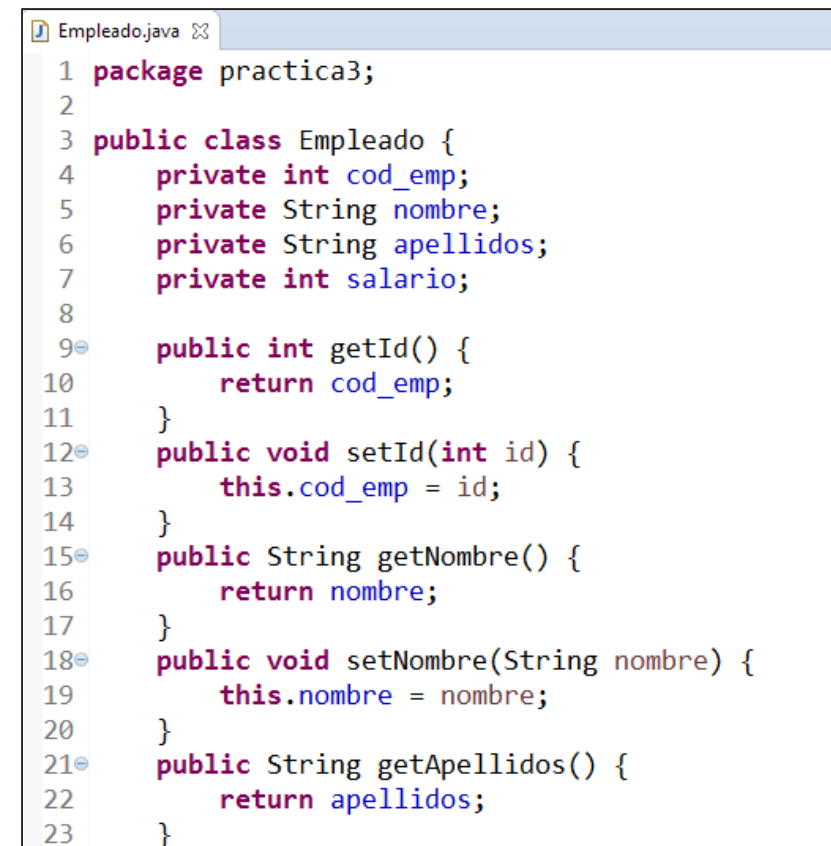
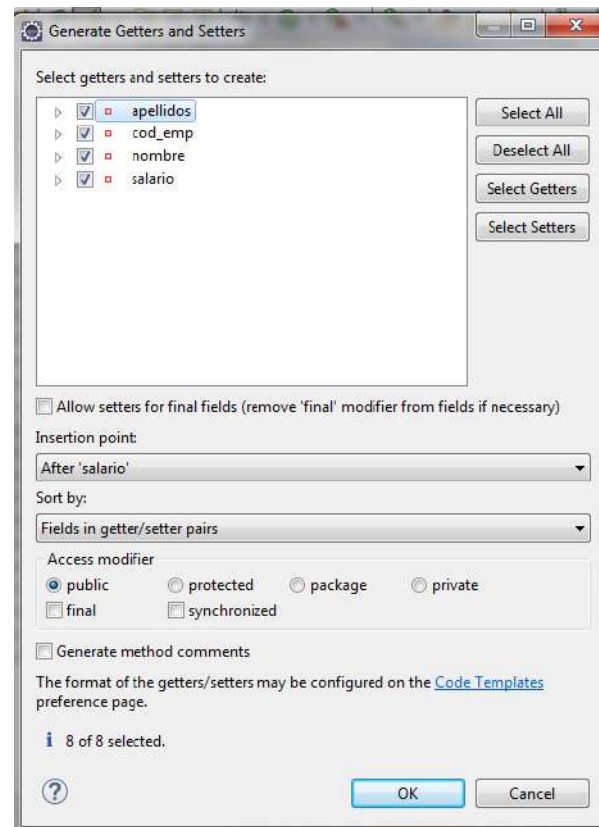
4. HIBERNATE DE FORMA MANUAL

Paso 6. Genera automáticamente los getters y los setters de estos campos (Shortcut: **Alt + Shift + S**). Genera un constructor vacío para la clase Empleado.



```
1 package practica3;
2
3 public class Empleado {
4     private int cod_emp;
5     private String nombre;
6     private String apellidos;
7     private int salario;
8 }
```

- Toggle Comment (Ctrl+ /)
- Remove Block Comment (Ctrl+Shift+ \)
- Generate Element Comment (Alt+Shift+J)
- Correct Indentation (Ctrl+I)
- Format (Ctrl+Shift+F)
- Format Element
- Add Import (Ctrl+Shift+M)
- Organize Imports (Ctrl+Shift+O)
- Sort Members...
- Clean Up...
- Generate Hibernate/JPA annotations...
- Override/Implement Methods...
- Generate Getters and Setters...
- Generate Delegate Methods...
- Generate hashCode() and equals()...



```
1 package practica3;
2
3 public class Empleado {
4     private int cod_emp;
5     private String nombre;
6     private String apellidos;
7     private int salario;
8
9     public int getId() {
10         return cod_emp;
11     }
12     public void setId(int id) {
13         this.cod_emp = id;
14     }
15     public String getNombre() {
16         return nombre;
17     }
18     public void setNombre(String nombre) {
19         this.nombre = nombre;
20     }
21     public String getApellidos() {
22         return apellidos;
23     }
24 }
```


4. HIBERNATE DE FORMA MANUAL

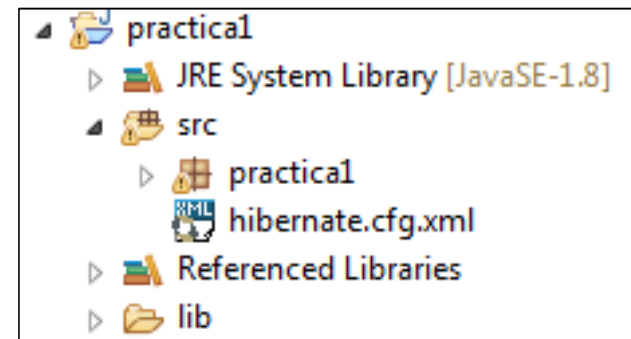
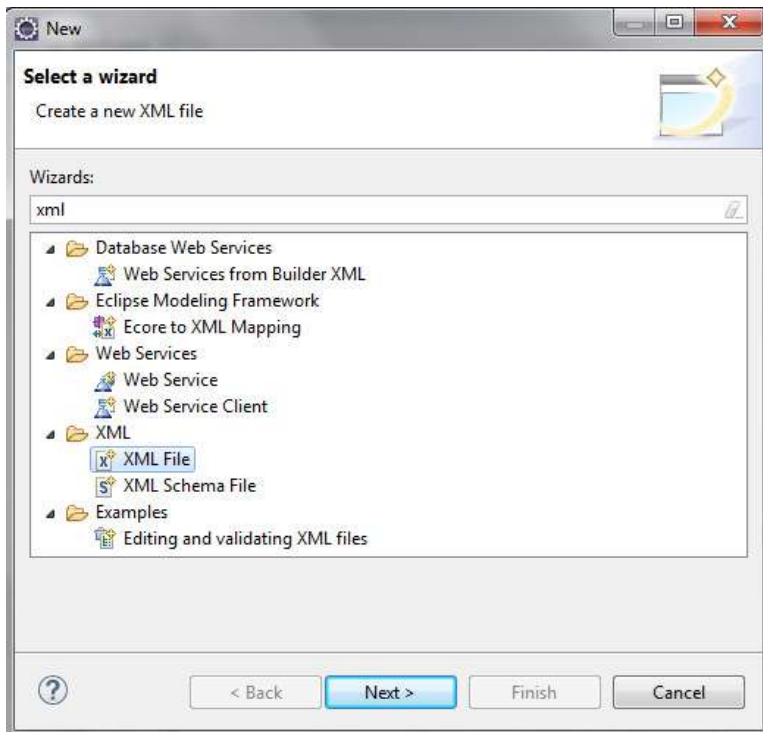
Paso 7. Mapea esta clase a la tabla Empleados de la base de datos, usando anotaciones de JPA:

```
Empleado.java
1 package practica3;
2
3 import javax.persistence.*;
4
5 @Entity
6 @Table(name="empleados")
7 public class Empleado {
8     private int cod_emp;
9     private String nombre;
10    private String apellidos;
11    private int salario;
12
13    @Id
14    @Column(name = "cod_emp")
15    @GeneratedValue(strategy = GenerationType.IDENTITY)
16    public int getId() {
17        return cod_emp;
18    }
19    public void setId(int id) {
20        this.cod_emp = id;
21    }
}
```

- Se utiliza la anotación `@Entity` y `@Table` antes de la clase, para mapearla a la table.
- La anotación `@Id` indica a Hibernate cual es la columna ID de la tabla.
- La anotación `@Column` mapea el campo a una columna de la table de la base de datos.
- La anotación `@GeneratedValue` indica a Hibernate cual que esta columna ID es auto-increment.
- Hibernate es inteligente, y puede mapear automaticamente campos de la clase a campos de una tabla si los campos tienen el mismo nombre y automaticamente mapea tipos Java a tipos SQL.
- No se tiene que mapear explicitamente el resto de campos: nombre, apellidos y salario.

4. HIBERNATE DE FORMA MANUAL

Paso 8. Crea el fichero XML de configuración para Hibernate llamado *hibernate.cfg.xml*, en la carpeta src. Este fichero indica a Hibernate como conectar con la base de datos y que clases Java deberían ser mapeadas a que tablas de la base de datos.



4. HIBERNATE DE FORMA MANUAL

Paso 9. El fichero **hibernate.cfg.xml** especifica el connectstring de conexión a la base de datos (JDBC driver class, URL, username and password). El elemento **<mapping>** especifica la clase entidad Java necesaria para ser mapeada, en este caso practica1.Empleado.

```
hibernate.cfg.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5 <hibernate-configuration>
6   <session-factory>
7     <!-- Database connection settings -->
8     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
9     <property name="connection.url">jdbc:mysql://localhost:3306/empresa</property>
10    <property name="connection.username">root</property>
11    <property name="connection.password"></property>
12    <property name="show_sql">true</property>
13    <mapping class="practica1.Empleado" />
14
15  </session-factory>
16 </hibernate-configuration>
17
```

Cambia el username y el password de la base de datos de acuerdo a nuestra base de datos. La propiedad **show_sql** a true indica a Hibernate que imprima sentencias SQL para cada query que se haga

4. HIBERNATE DE FORMA MANUAL

Paso 10. Comenta todo lo anterior del fichero BaseDatos.java (o crea un nuevo fichero llamado GestorEmpleados), instancia una variable donde se cargará la **Session Factory** de Hibernate, e implementa todos los métodos de persistencia Hibernate:

```
BaseDatos.java
1 package practica3;
2
3 import org.hibernate.Session;
4
5
6
7
8
9 public class BaseDatos {
10     private SessionFactory sessionFactory;
11
12     public void iniciar() { //Iniciar hibernate.
13         //Carga del fichero hibernate.cfg en sessionFactory
14     }
15     public void salir() { //Salir Hibernate
16     }
17     public void insertar(Empleado emp) { //Codigo para insertar un empleado
18     }
19     public void listar() { //Codigo para listar o leer un empleado
20     }
21     public void actualizar(Empleado emp) { //Codigo para actualizar un empleado
22     }
23     public void borrar(Empleado emp) { //Codigo ara borra un empleado
24     }
25 }
```

Los nombres de los métodos son totalmente libres

4. HIBERNATE DE FORMA MANUAL

Paso 11. En Hibernate se realizan operaciones de base de datos a través del objeto **SessionFactory**. **SessionFactory** carga el fichero de configuración Hibernate, analiza el mapeo y crea la conexión con la base de datos. En el método **setup()** **SessionFactory** queda configurado según los parámetros del fichero **hibernate.cfg.xml**. En el método **exit()** se cierra la **SessionFactory**.

```
protected void setup() { // code to load Hibernate Session factory
    final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
        .configure() // configures settings from hibernate.cfg.xml
        .build();

    try {
        sessionFactory = new MetadataSources(registry).buildMetadata().buildSessionFactory();
    } catch (Exception ex) {
        System.out.println("The sessionFactory was not created. Could not connect to the database");
        StandardServiceRegistryBuilder.destroy(registry);
    }

    if (sessionFactory != null)
        System.out.println("Successfully connected to the database");
}

protected void exit() {
    // code to close Hibernate Session factory
    sessionFactory.close();
}
```

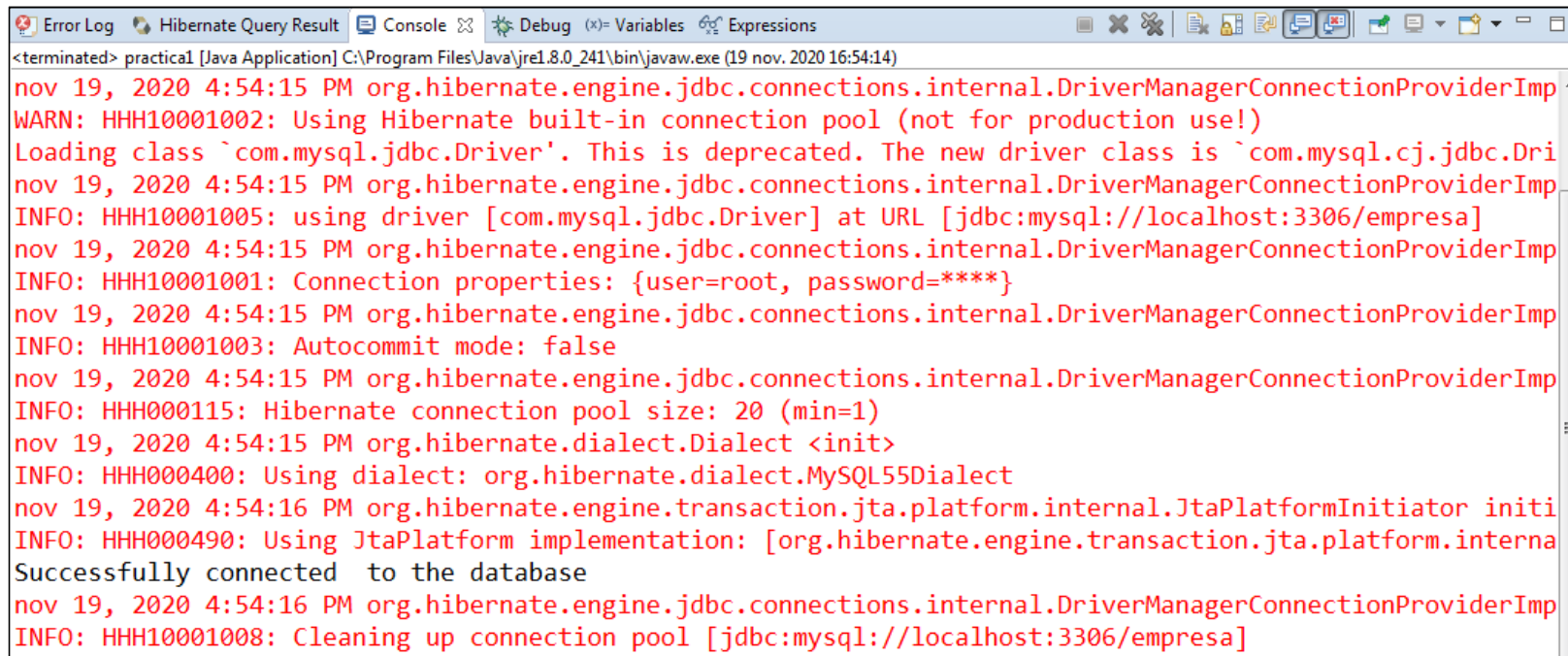
4. HIBERNATE DE FORMA MANUAL

Paso 12. En el método main de nuestro proyecto probamos el siguiente código:

```
practica3.java ✕
1 package practica3;
2
3 public class practica3 {
4
5     public static void main(String[] args) {
6
7         BaseDatos db = new BaseDatos();
8         db.iniciar();
9         db.salir();
10
11     }
12 }
```

4. HIBERNATE DE FORMA MANUAL

Paso 13. Ejecutamos el programa (shortcut: **Ctrl + F11**) para comprobar si la session factory se carga exitosamente. Si vemos algo como esto en la vista **Console**, significa que la instalación de Hibernate se ha realizado correctamente (se carga correctamente sesión Factory)



```
<terminated> practica1 [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (19 nov. 2020 16:54:14)
nov 19, 2020 4:54:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new driver class is `com.mysql.cj.jdbc.Dri
nov 19, 2020 4:54:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/empresa]
nov 19, 2020 4:54:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH10001001: Connection properties: {user=root, password=****}
nov 19, 2020 4:54:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH10001003: Autocommit mode: false
nov 19, 2020 4:54:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
nov 19, 2020 4:54:15 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL55Dialect
nov 19, 2020 4:54:16 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initi
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.interna
Successfully connected to the database
nov 19, 2020 4:54:16 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/empresa]
```

4. HIBERNATE DE FORMA MANUAL

Paso 14. Una vez SessionFactory de Hibernate está construido, se pueden construir los diferentes métodos para realizar las operaciones CRUD: Insertar, leer, actualizar y borrar. El denominador común de todos estos métodos es que todos:

- Abren una sessionFactory y empieza una transacción.
- Se hace la acción a realizar,
- Finalmente se hace el commit de la transaction y se cierra la sesión

```
protected void read() {  
    Session session = sessionFactory.openSession();  
    session.beginTransaction();  
  
    // code to get a book  
  
    session.getTransaction().commit();  
    session.close();  
}
```


4. HIBERNATE DE FORMA MANUAL

Paso 15. Transforma el fichero controlador de la BBDD según el estándar Hibernate.

- Cambia el atributo Connection por el SessionFactory
- Crea las nuevas funciones setup() y exit()
- Reescribe los métodos: actualizar, borrar

```
public class BaseDatos {  
  
    protected SessionFactory sessionFactory;  
  
    public BaseDatos() {  
        setup();  
    }  
    protected void setup() { // code to load Hibernate Session factory  
        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder().  
            .configure() // configures settings from hibernate.properties.  
            .build();  
    }  
}
```

```
protected void actualizar(Empleado emp) {  
    // code to modify a employee  
    Session session = sessionFactory.openSession();  
    session.beginTransaction();  
    session.update(emp);  
    session.getTransaction().commit();  
    session.close();  
}  
  
public void borrar(Empleado emp) {  
    // code to remove a employee  
    Session session = sessionFactory.openSession();  
    session.beginTransaction();  
    session.delete(emp);  
    session.getTransaction().commit();  
    session.close();  
}
```

5. HIBERNATE CON EL PLUGIN JBOSS

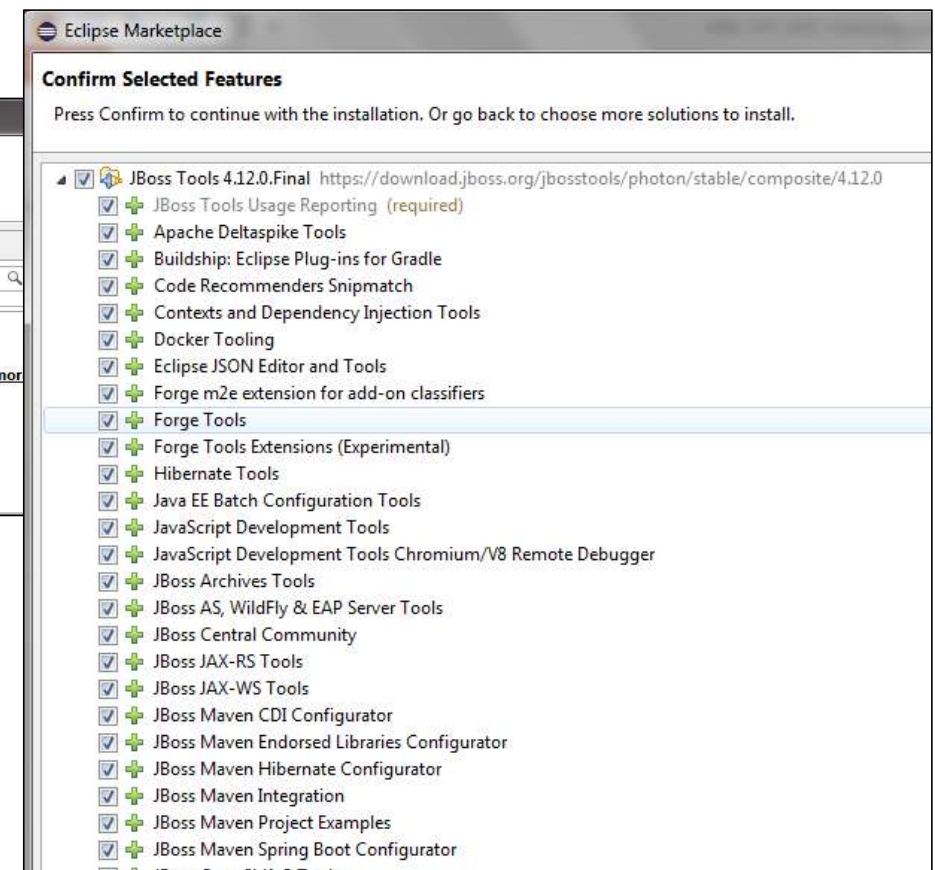
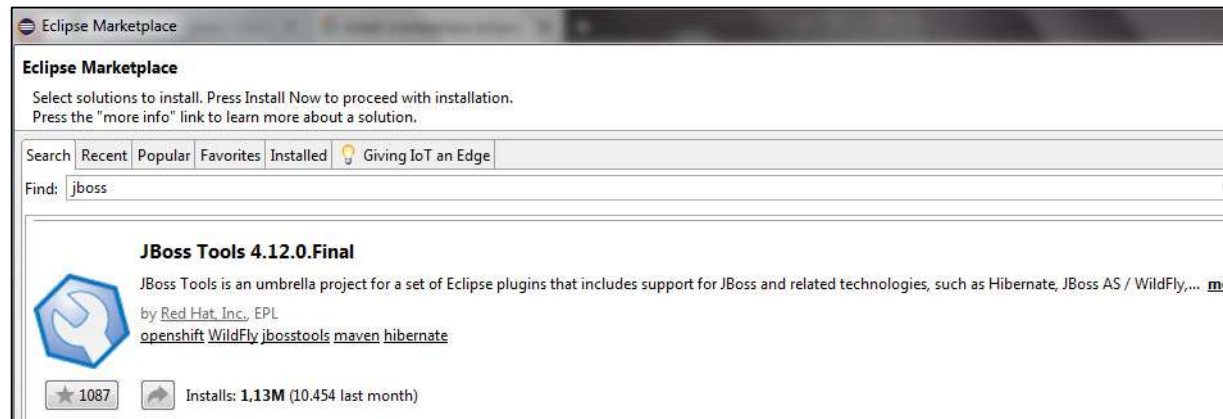
Paso 0. Script de la base de datos “empresa” en mysql

```
create table departamentos (  
  cod_dept INT PRIMARY KEY auto_increment,  
  nombre VARCHAR(20),  
  direccion VARCHAR(10),  
  objetivos int  
);
```

```
create table empleados(  
  id int primary key auto_increment,  
  nombre VARCHAR(20),  
  apellidos VARCHAR(25),  
  salario double,  
  cod_dept int  
);  
alter table empleados add foreign key (cod_dept)  
  references departamentos(cod_dept);
```

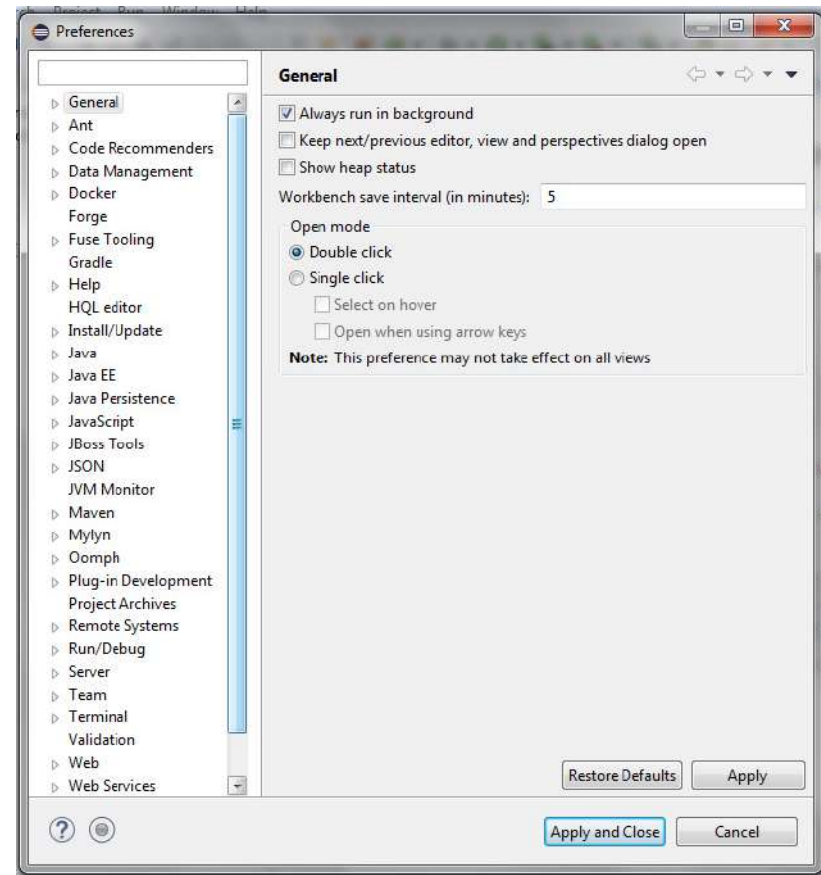
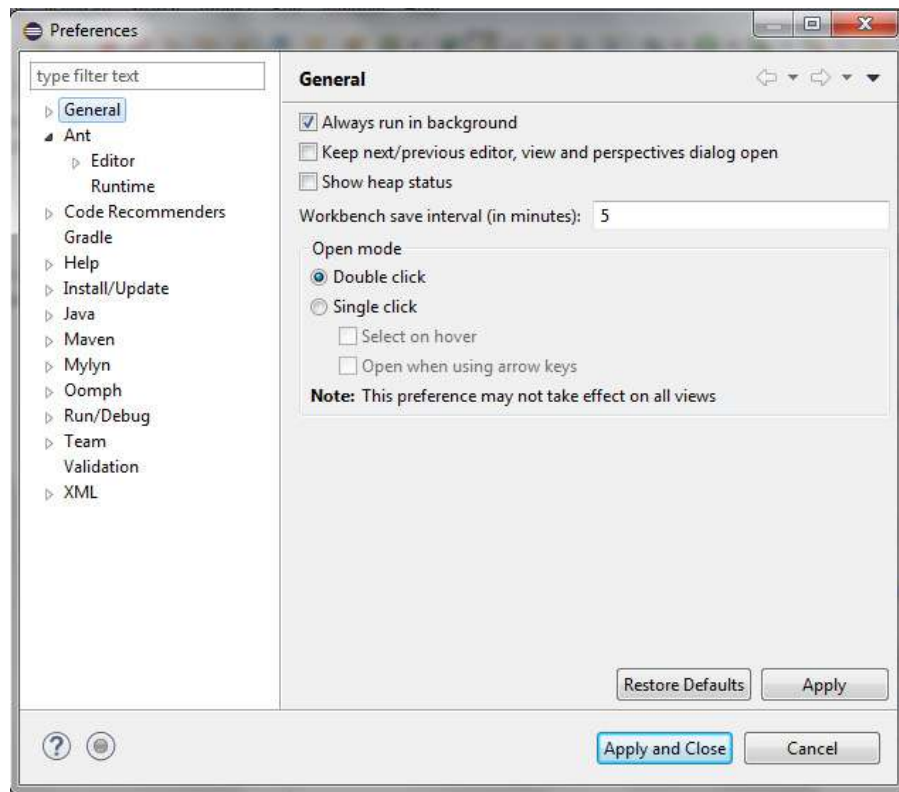
5. HIBERNATE CON EL PLUGIN JBOSS

Paso 1. Instalación del plugin para Eclipse “JBoss Tools”. Accedemos al menú **Help/Eclipse Marketplace**, buscamos el plugin **JBoss Tools** y lo instalamos (Requerirá reiniciar Eclipse)



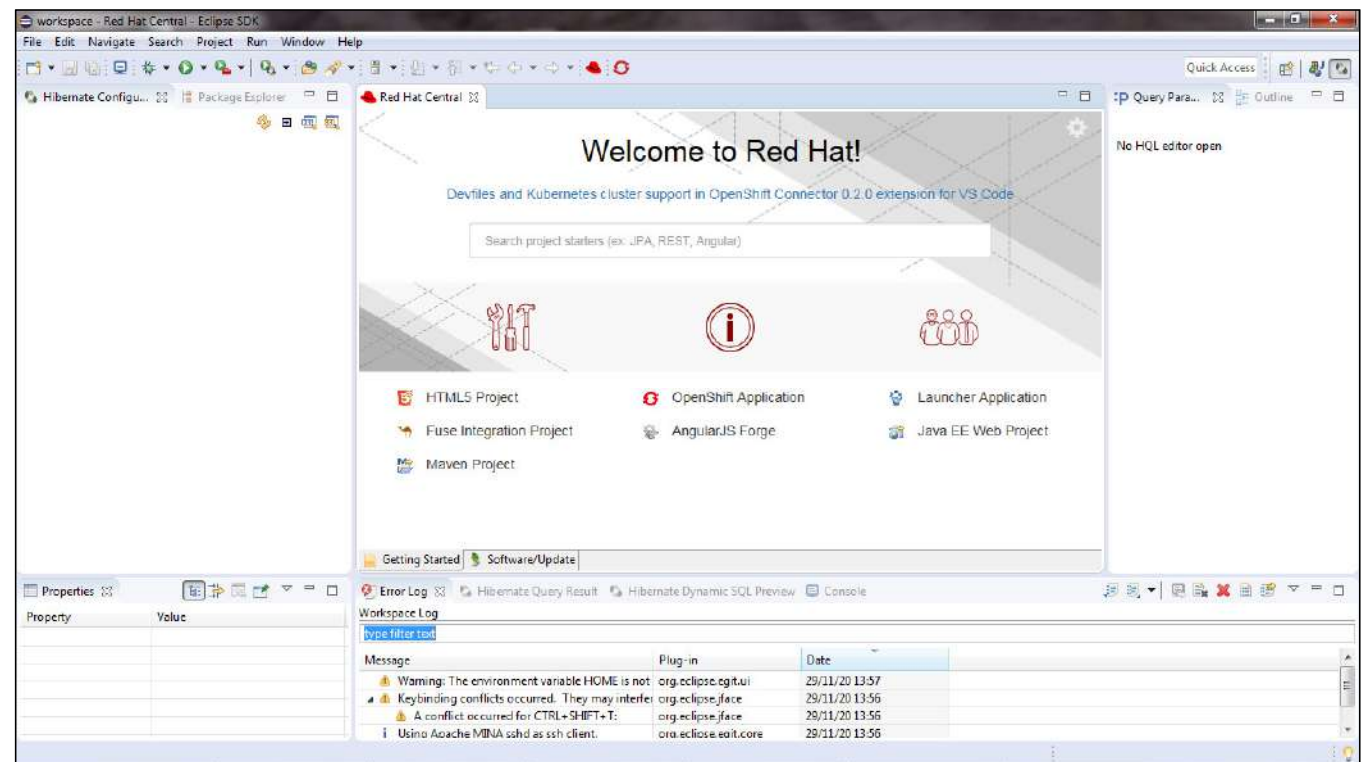
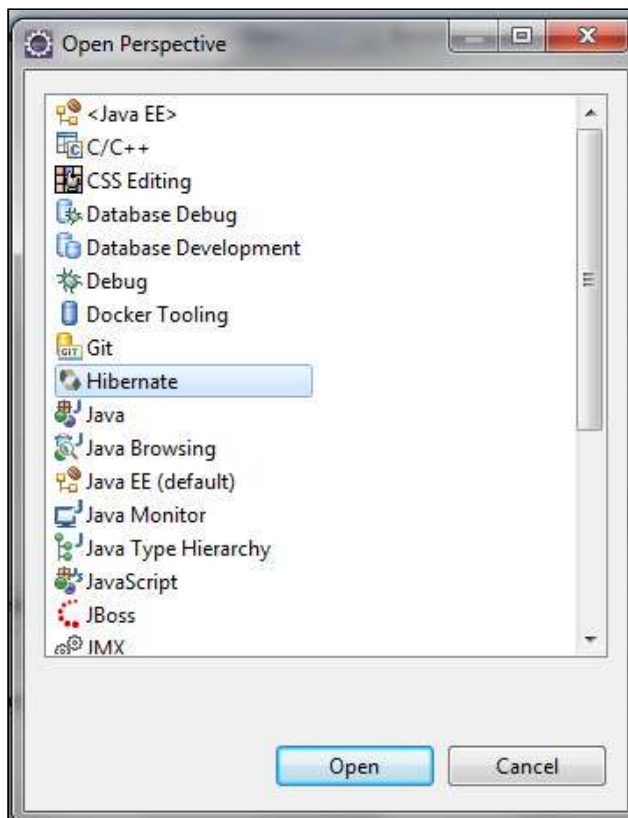
5. HIBERNATE CON EL PLUGIN JBOSS

Paso 2. Podemos comprobar que se ha instalado correctamente el plugin mirando que las opciones de preferencias se han duplicado:



5. HIBERNATE CON EL PLUGIN JBOSS

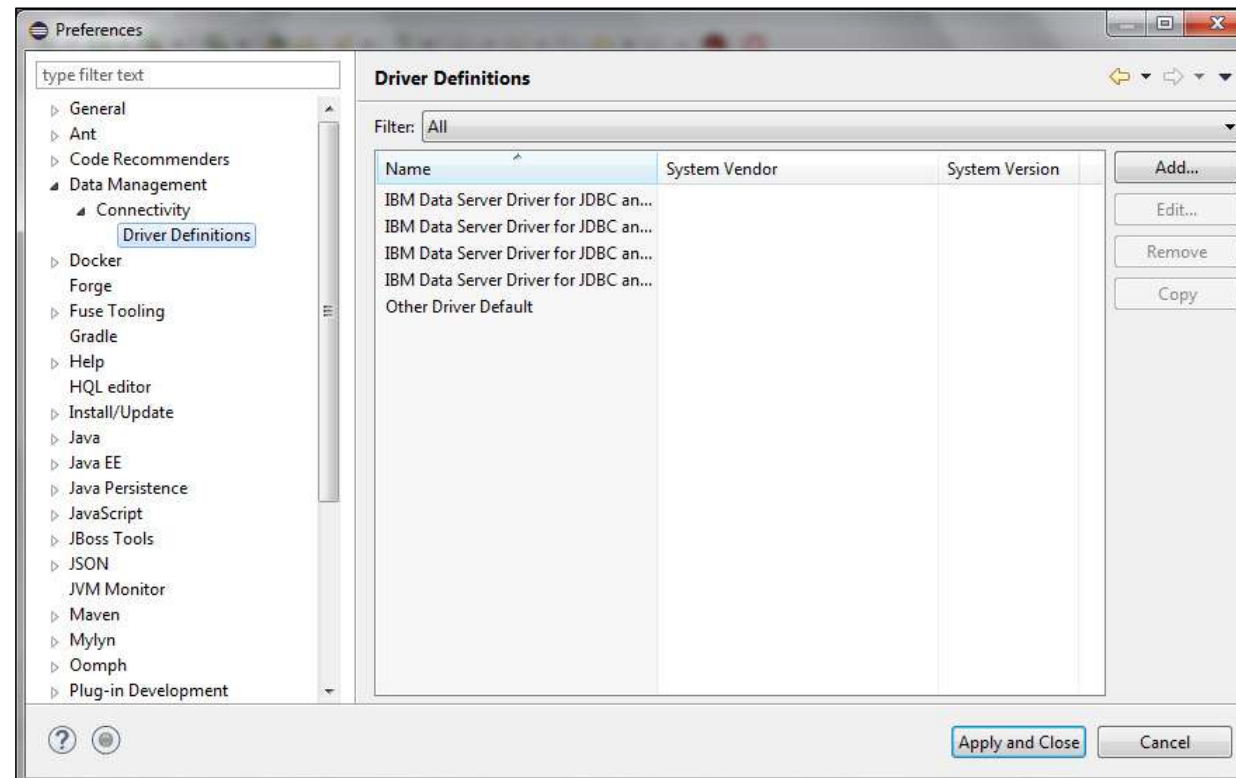
Paso 3. En **Window/perspective/other**, entre las opciones que aparecen debe estar la perspectiva **Hibernate**.



5. HIBERNATE CON EL PLUGIN JBOSS

Configuración driver Mysql en Eclipse

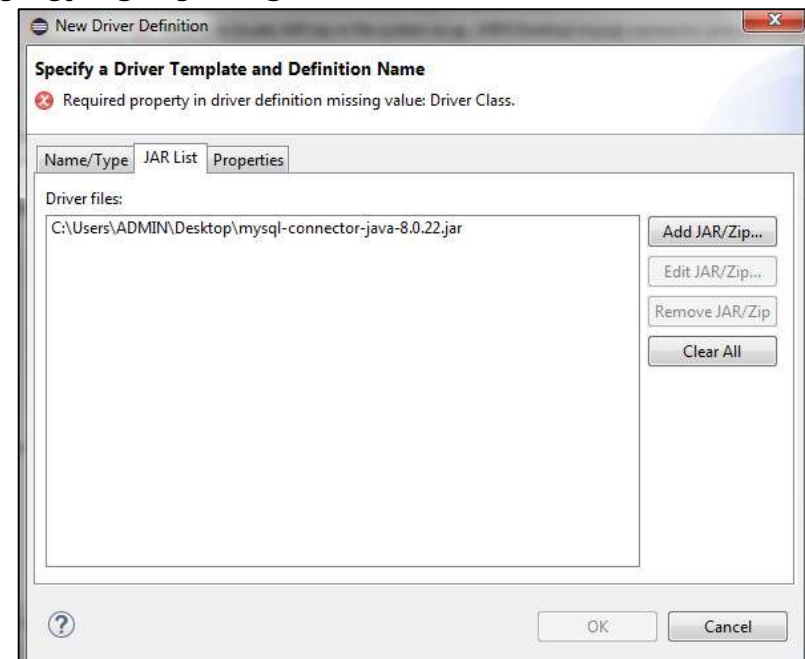
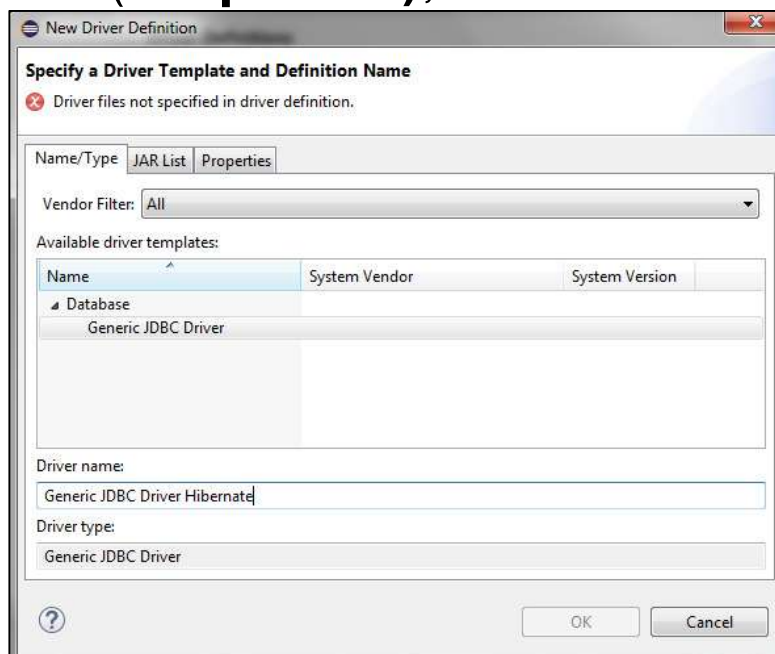
Paso 4. Vamos a **Window/preferences** y en el menú de la izquierda escogemos **DataManagement/Connectivity/Driver definitions**.



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 5. Añadiremos el nuevo conector haciendo click en “Add”. Nos aparecerá un cuadro de diálogo rotulado “**New Driver Definition**”, que dispone de 3 pestañas:

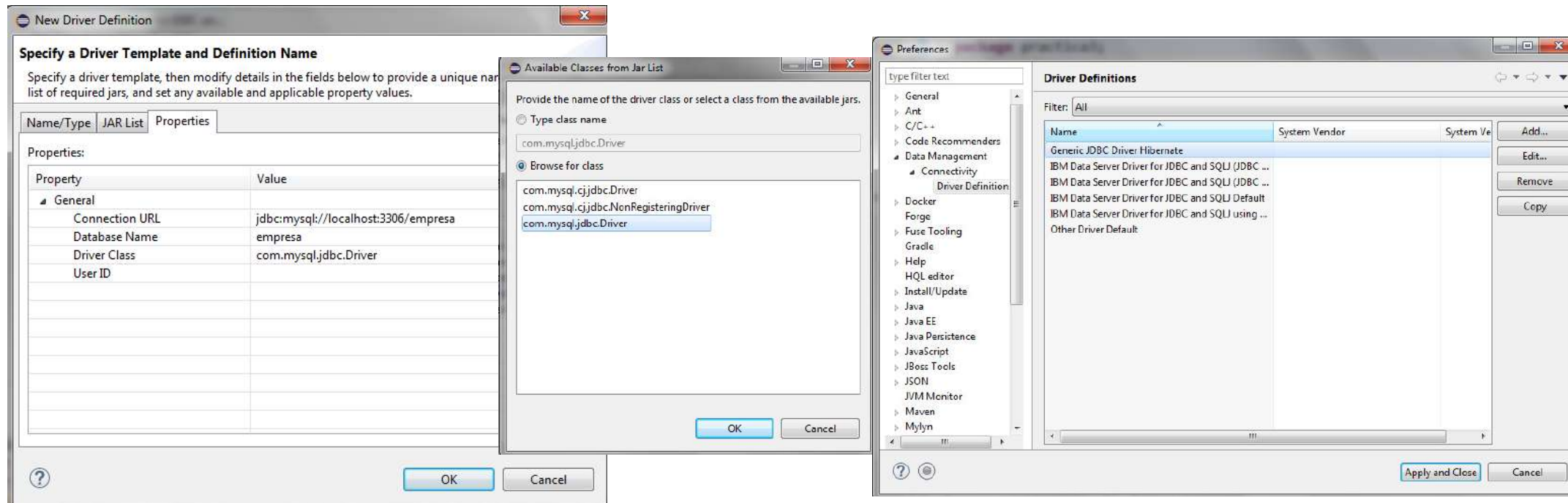
- La 1º (**Name/Type**) nos pide un nombre para el driver, el Vendor y el tipo conector.
- En la 2º (**JAR list**) se debe de indicar la ruta al paquete .jar del conector.
- Y en la 3º (**Properties**), se debe de configurar el driver.



5. HIBERNATE CON EL PLUGIN JBOSS

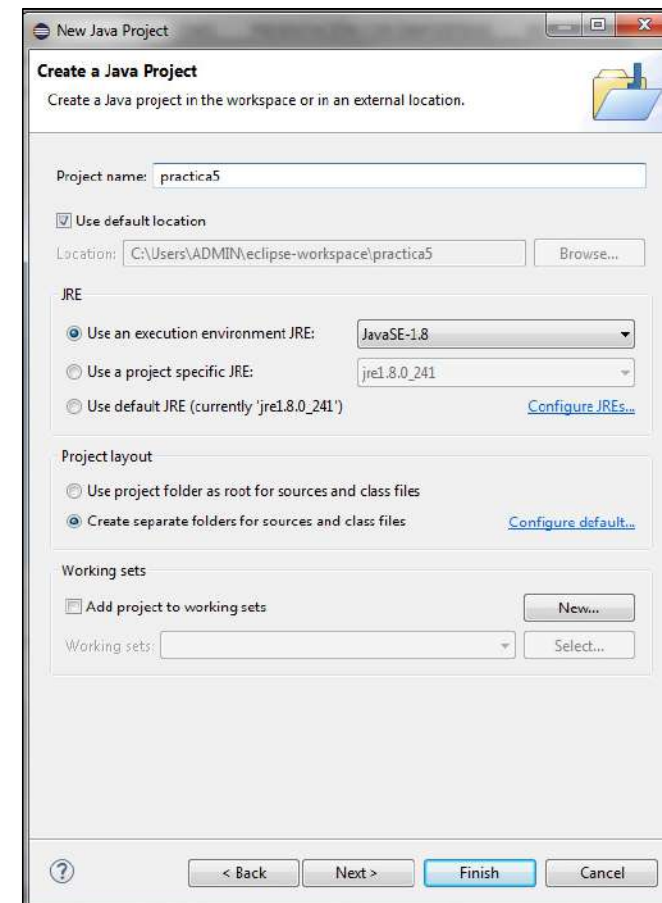
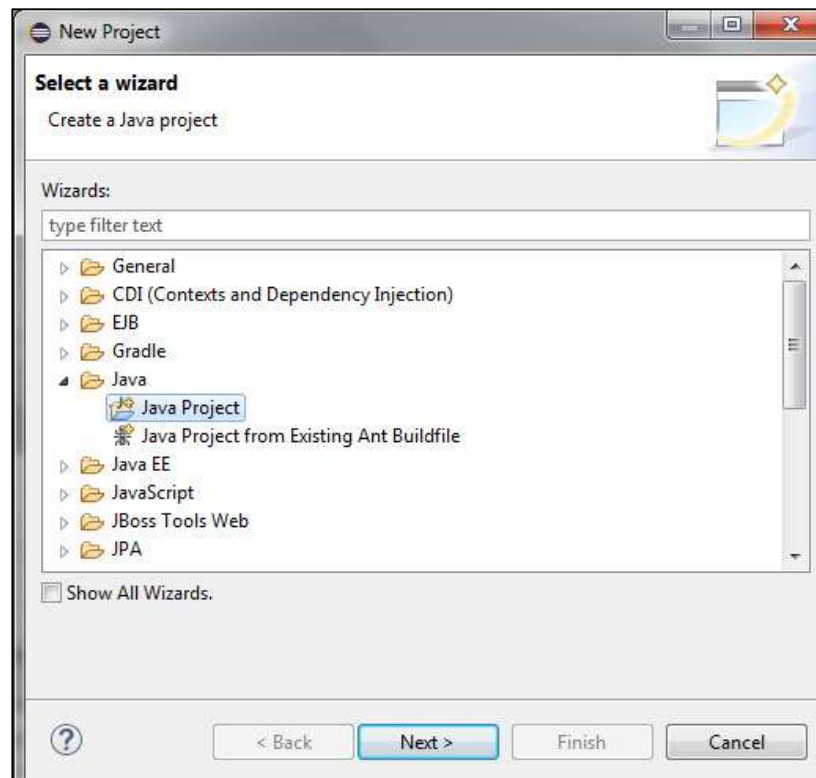
Paso 6. En la pestaña **properties** se realiza la configuración:

- Connection URL: Indicamos el localizador del recurso.
- Database Name: Indicamos el nombre de la bbdd.
- Driver Class: La clase del paquete que utilizaremos como driver



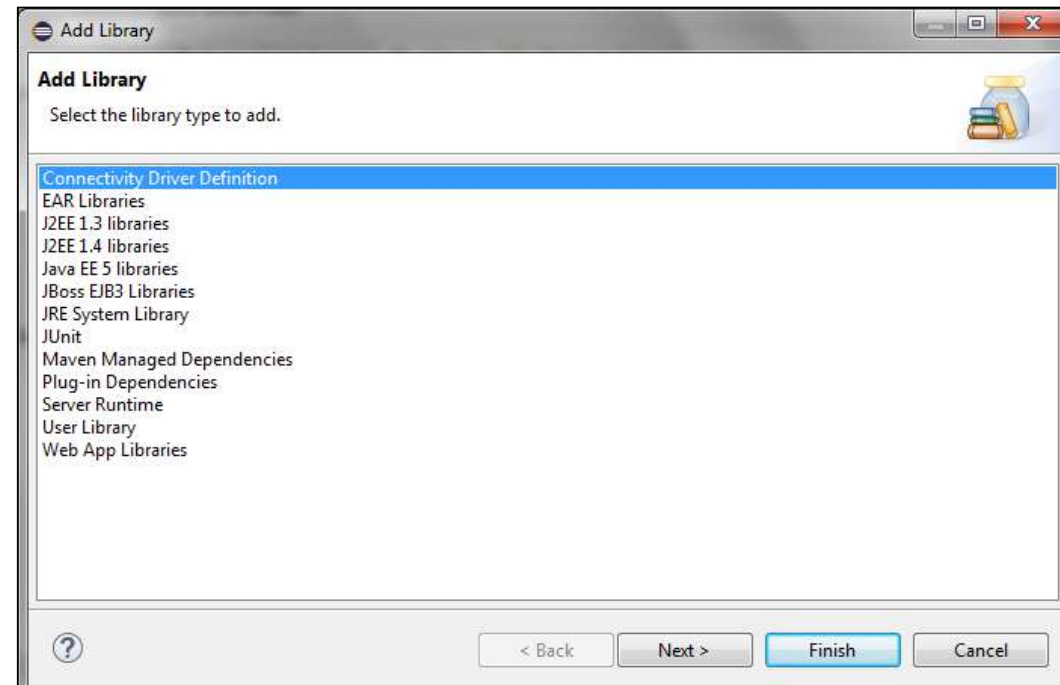
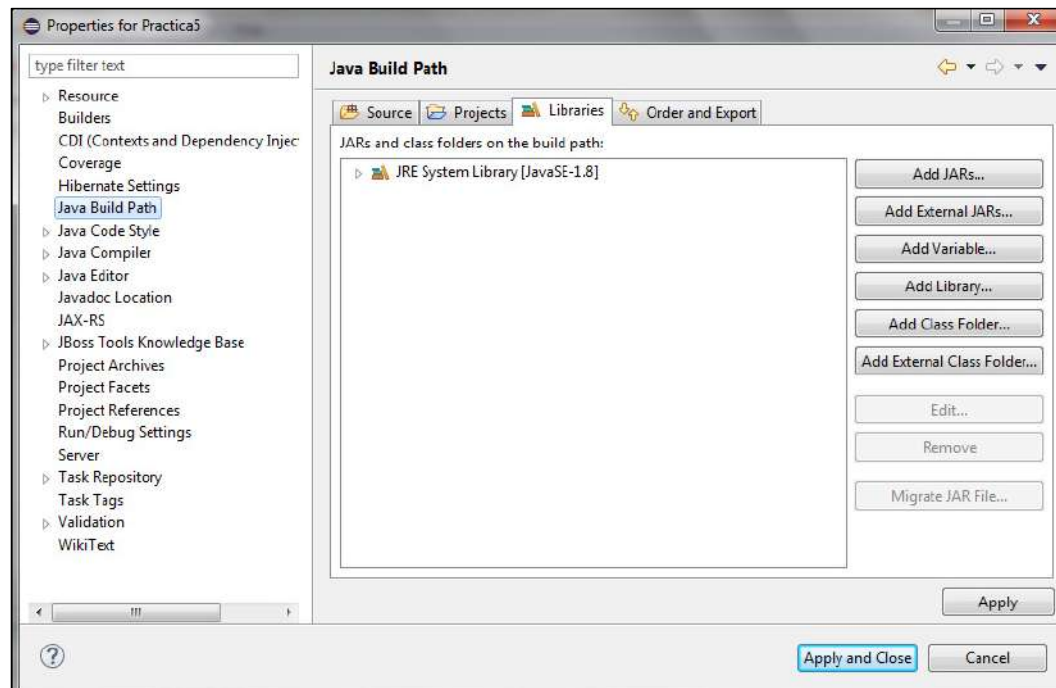
5. HIBERNATE CON EL PLUGIN JBOSS

Paso 7. Creamos un nuevo proyecto Java en Eclipse (por ejemplo practica5):



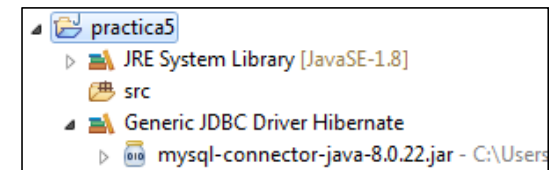
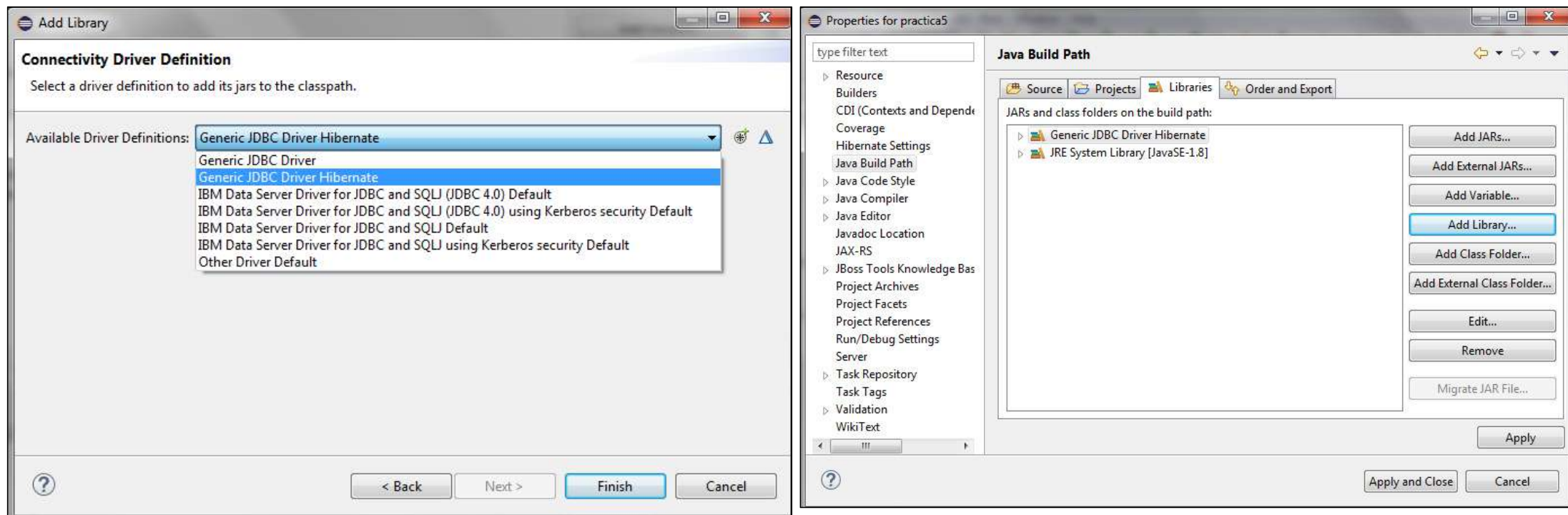
5. HIBERNATE CON EL PLUGIN JBOSS

Paso 8. Vamos a **Properties/Java Build Path** y hacemos click en **Add Librerries**. Se visualiza una ventana, elegimos **Connectivity Driver Definition**.



5. HIBERNATE CON EL PLUGIN JBOSS

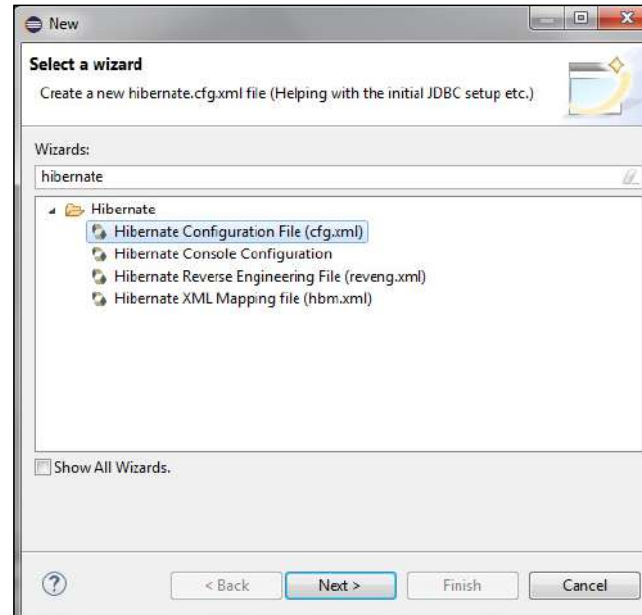
Paso 9. Tras el *Next* correspondiente elegimos el driver recién definido.



5. HIBERNATE CON EL PLUGIN JBOSS

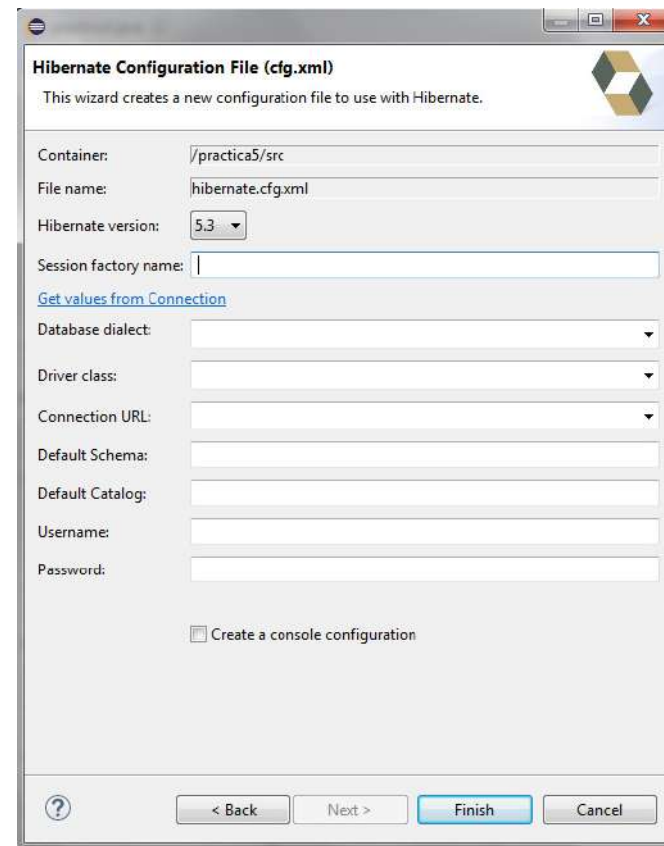
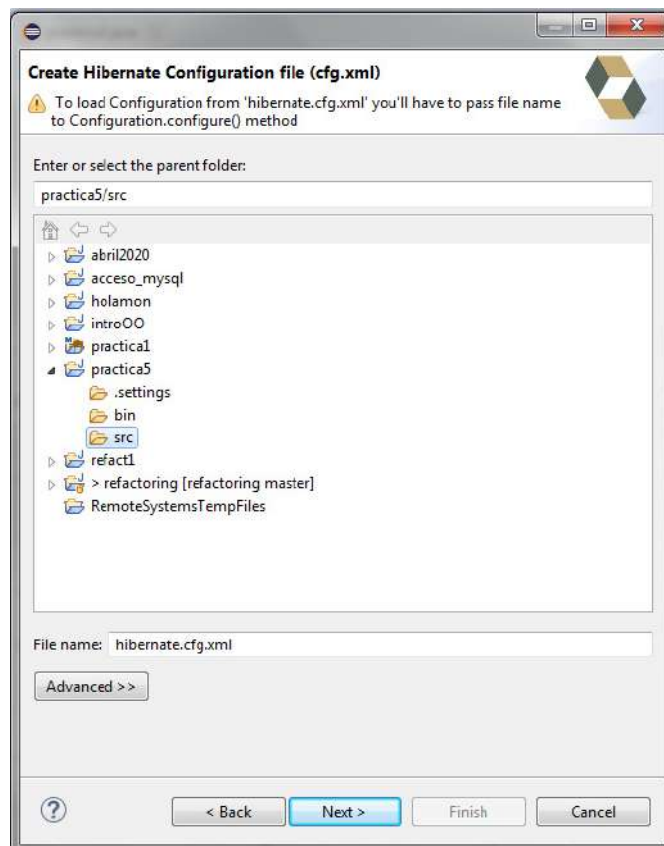
Configuración Hibernate

Paso 10. Una vez tenemos la librería MySql en nuestro proyecto hemos de crear el fichero de configuración de Hibernate llamado hibernate.cfg.xml. En nuestro proyecto, hacemos click botón derecho y seleccionamos **New/Other/Hibernate/Hibernate Configuration File (cfg.xml)**. Este fichero es un XML que contiene todo lo necesario para realizar la conexión a la base de datos.



5. HIBERNATE CON EL PLUGIN JBOSS

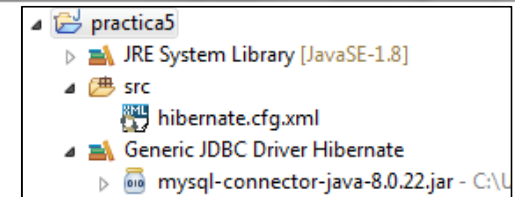
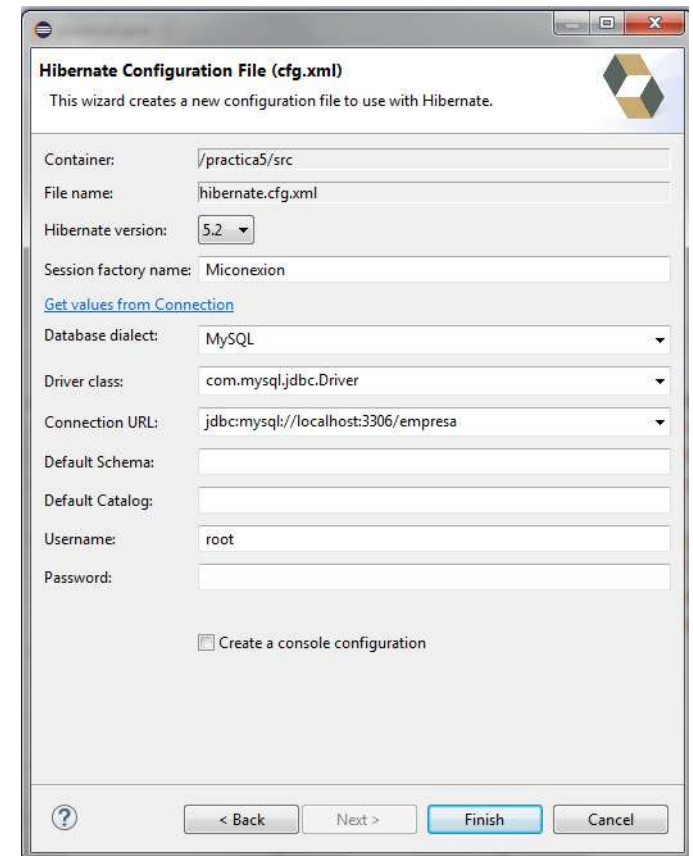
Paso 11. A continuación, nos pide dónde crear el fichero. Lo haremos en la carpeta src del proyecto. Otro *Next* y pasamos a configurar la conexión.



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 12. Los campos a rellenar son:

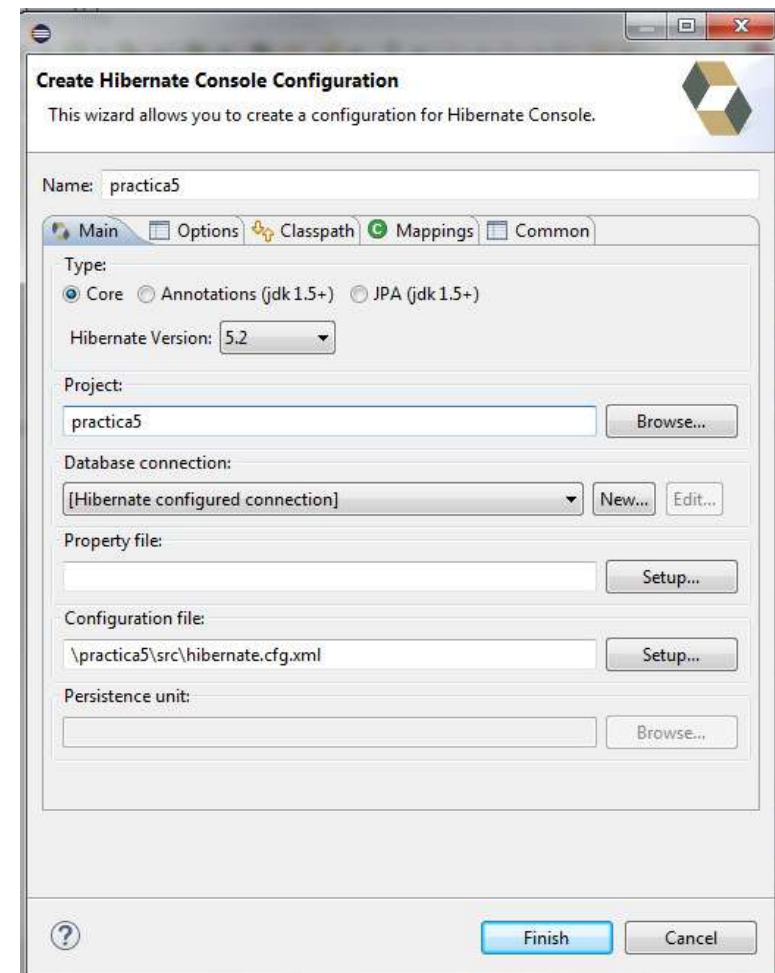
- **Session factory name** → nombre de la conexión. Pondremos “Miconexion”.
- **Database dialect** → elegimos tipo de comunicación JDBC. En concreto, “MySQL”
- **Driver Class** → clase JDBC para la conexión → **com.mysql.jdbc.Driver**
- **Conection URL** → ruta de conexión a la bbdd indicada cuando definíamos el driver
- **Username** → Usuario que se conectará a MySql. Debe de tener permisos
- **Password** → Contraseña correspondiente.



5. HIBERNATE CON EL PLUGIN JBOSS

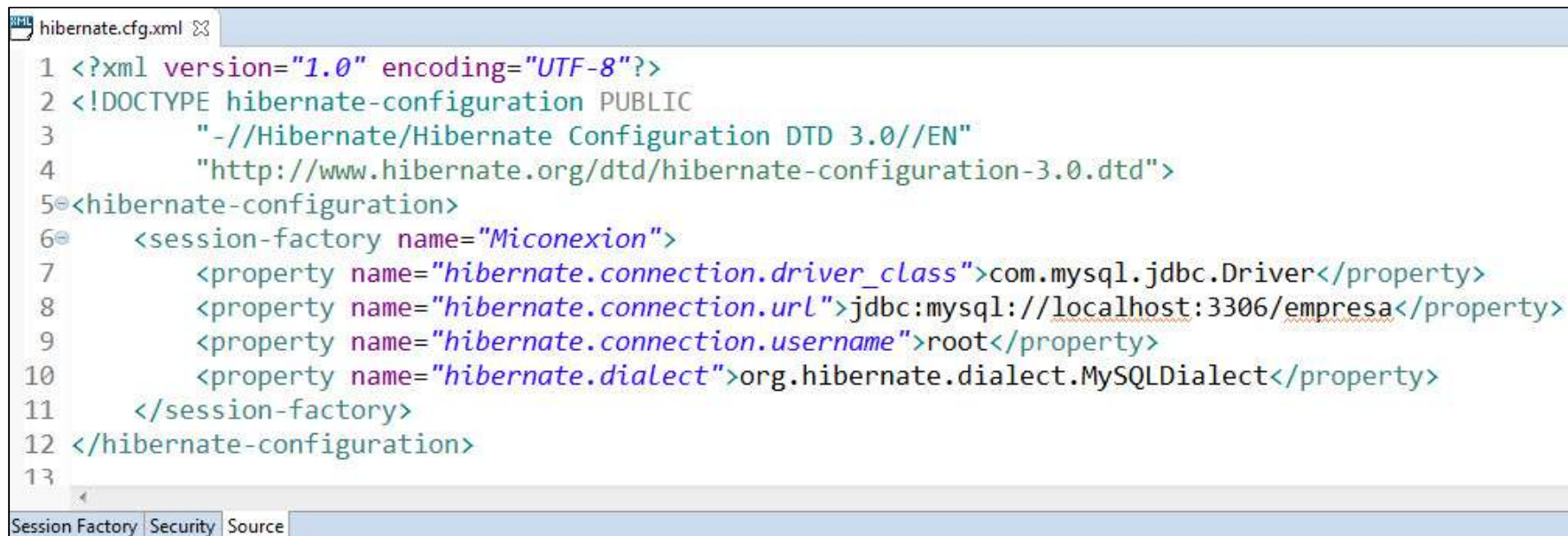
Paso 13. Una vez creado el **hibernate.cfg.xml**, hemos de crear el fichero **XML Hibernate Console Configuration**.

- Seleccionamos nuestro proyecto, botón derecho: **New / Other / Hibernate / Hibernate Console Configuration**.
- En el campo Name de la nueva ventana podemos dejar el nombre por defecto “practica5” para nuestra configuración de consola Hibernate.
- Finalmente hacemos click en Finish



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 14. Aparece el editor de configuración de Hibernate. Desde la pestaña **Source** se puede editar el fichero XML generado:

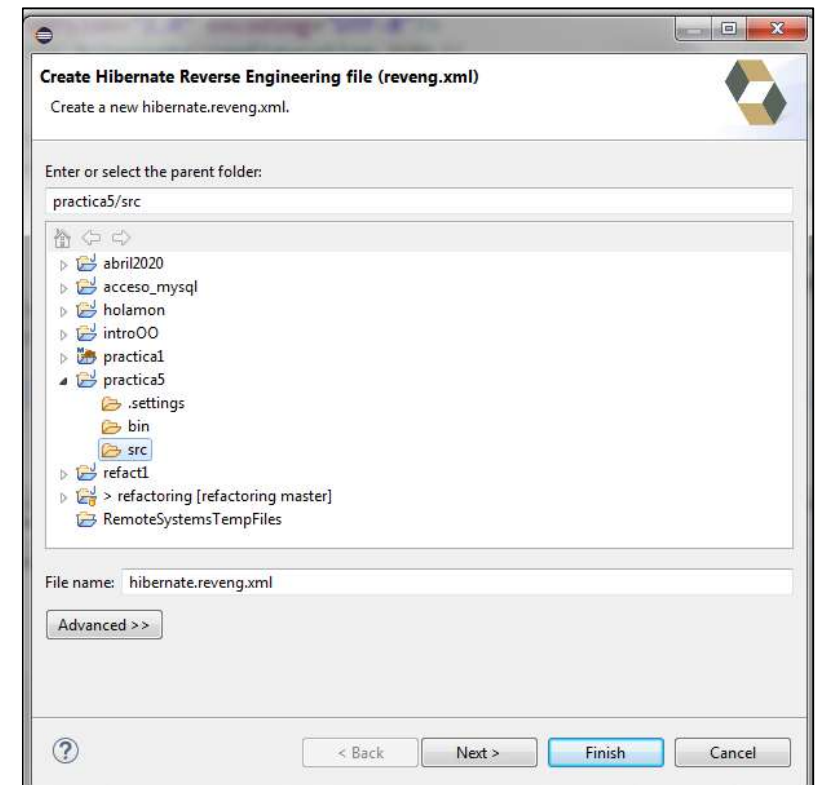
A screenshot of an IDE window titled 'hibernate.cfg.xml'. The window shows an XML configuration file for Hibernate. The XML content is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5 <hibernate-configuration>
6     <session-factory name="Miconexion">
7         <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
8         <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/empresa</property>
9         <property name="hibernate.connection.username">root</property>
10        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
11    </session-factory>
12 </hibernate-configuration>
13
```

The IDE interface includes a tab bar at the bottom with 'Session Factory', 'Security', and 'Source' tabs. The 'Source' tab is currently selected, displaying the XML code.

5. HIBERNATE CON EL PLUGIN JBOSS

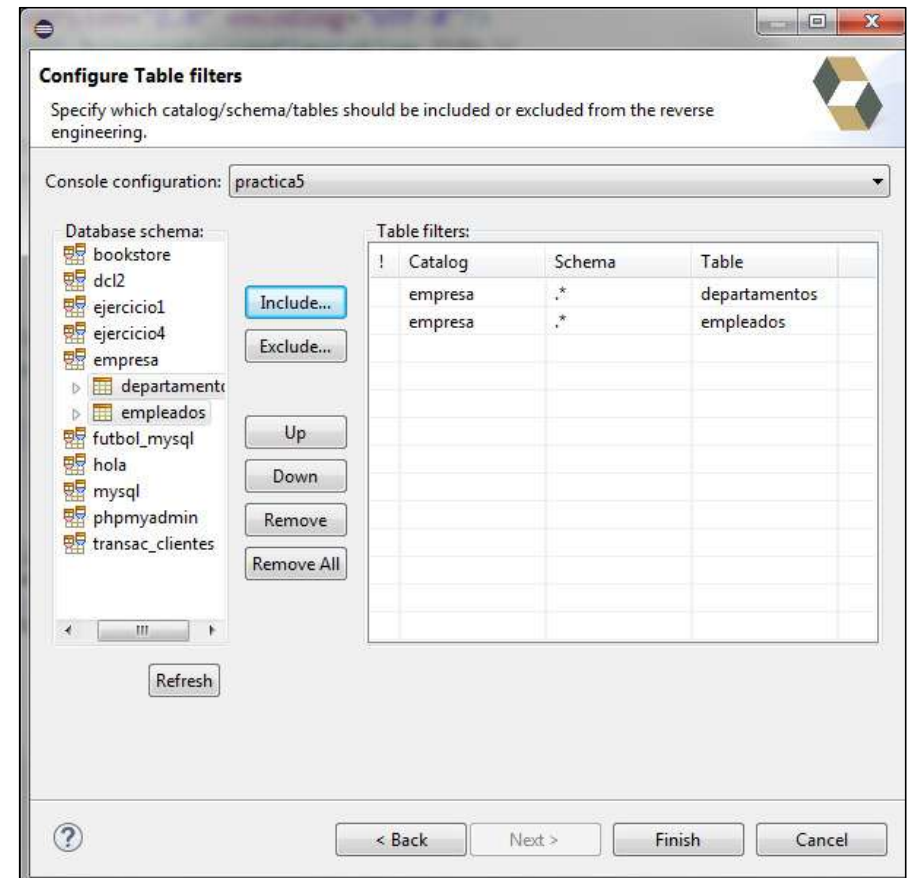
Paso 15. Por último, para acabar de configurar Hibernate, hemos de crear el fichero **XML Hibernate Reverse Engineering (reveng.xml)**. Vamos a **New / Other/ Hibernate / Hibernate Reverse Engineering File**. Primero nos pide la ubicación, se debe indicar la carpeta **src** del proyecto.



5. HIBERNATE CON EL PLUGIN JBOSS

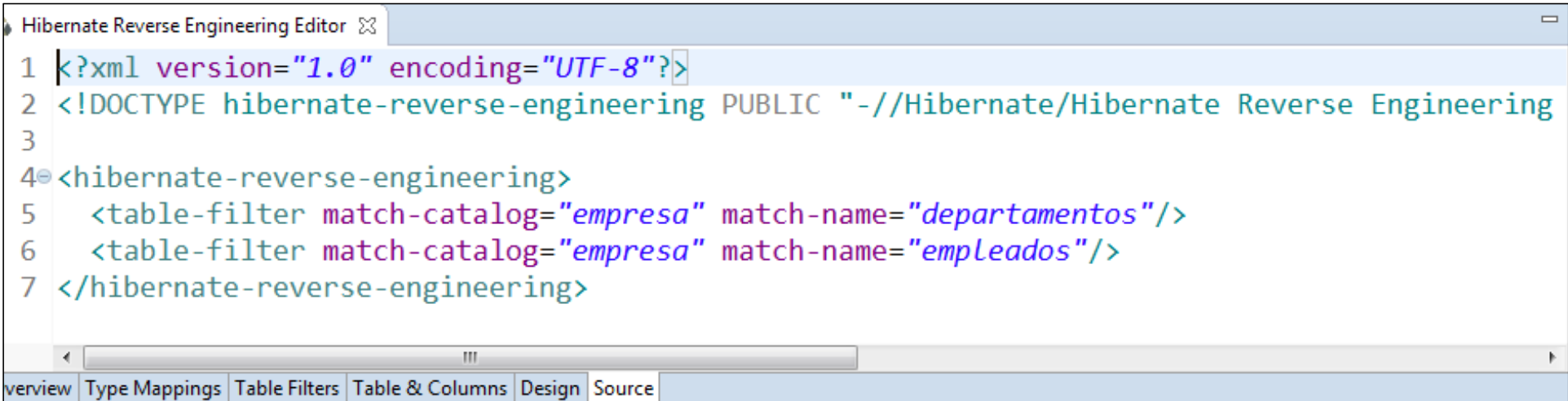
Paso 16. Al hacer click en Next, se visualiza una ventana desde donde indicaremos las tablas que queremos mapear.

- Primero elegimos nuestra **Console Configuration** creada en el paso anterior
- A continuación, pulsamos en *Refresh* para que muestre la base de datos “empresa” y sus tablas.
- Se seleccionan las tablas “departamentos” y “empleados” y pulsamos el botón *Include*.
- Finalmente hacemos click en *Finish*



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 17. Se abrirá el editor de Hibernate Reverse Engineering y en la pestaña **Source** podemos visualizar el XML generado

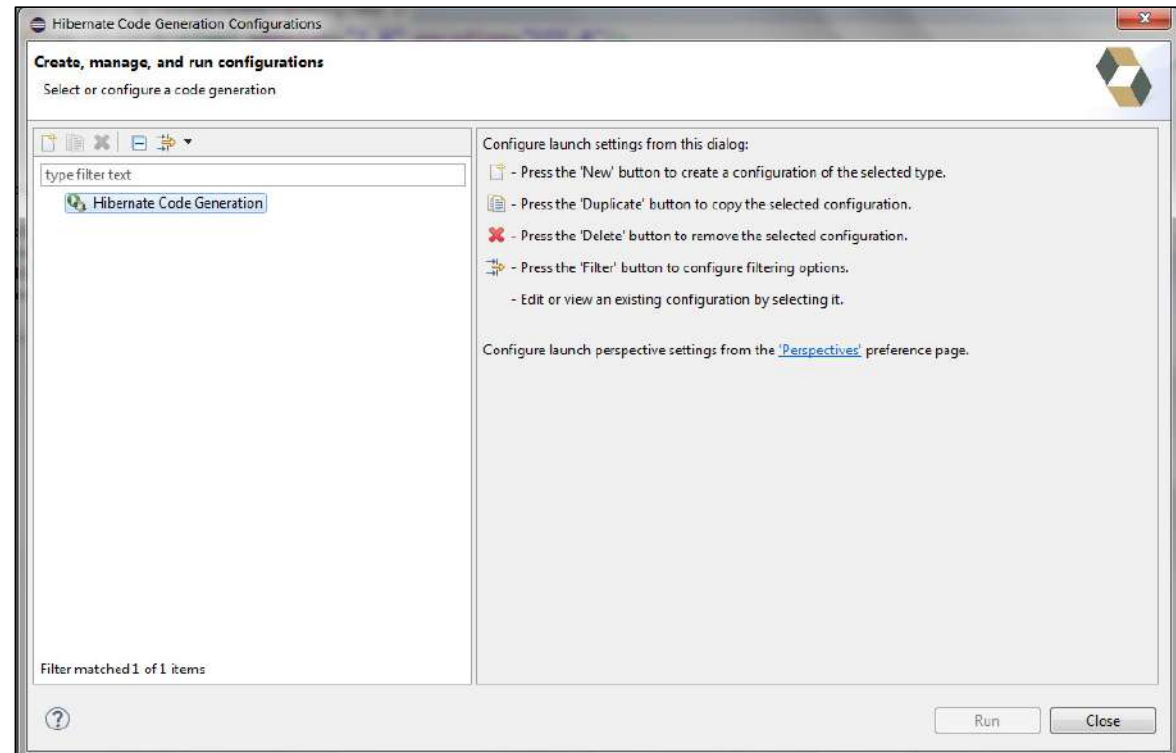
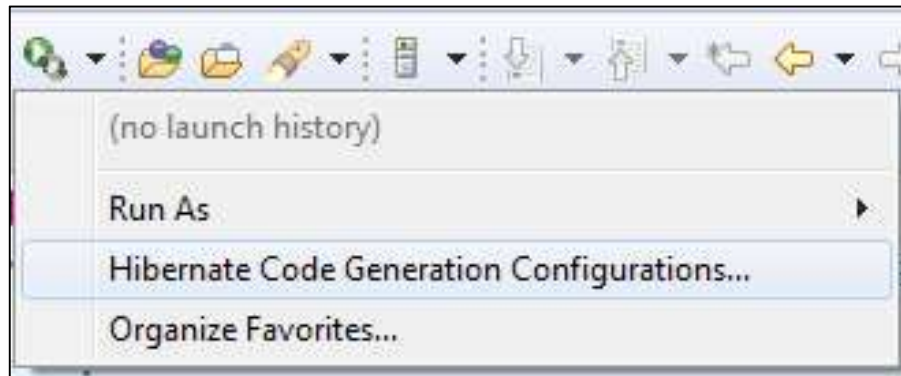


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engineering
3
4 <hibernate-reverse-engineering>
5   <table-filter match-catalog="empresa" match-name="departamentos"/>
6   <table-filter match-catalog="empresa" match-name="empleados"/>
7 </hibernate-reverse-engineering>
```

The screenshot shows the 'Hibernate Reverse Engineering Editor' window. The 'Source' tab is active, displaying the generated XML. The XML content is as follows:

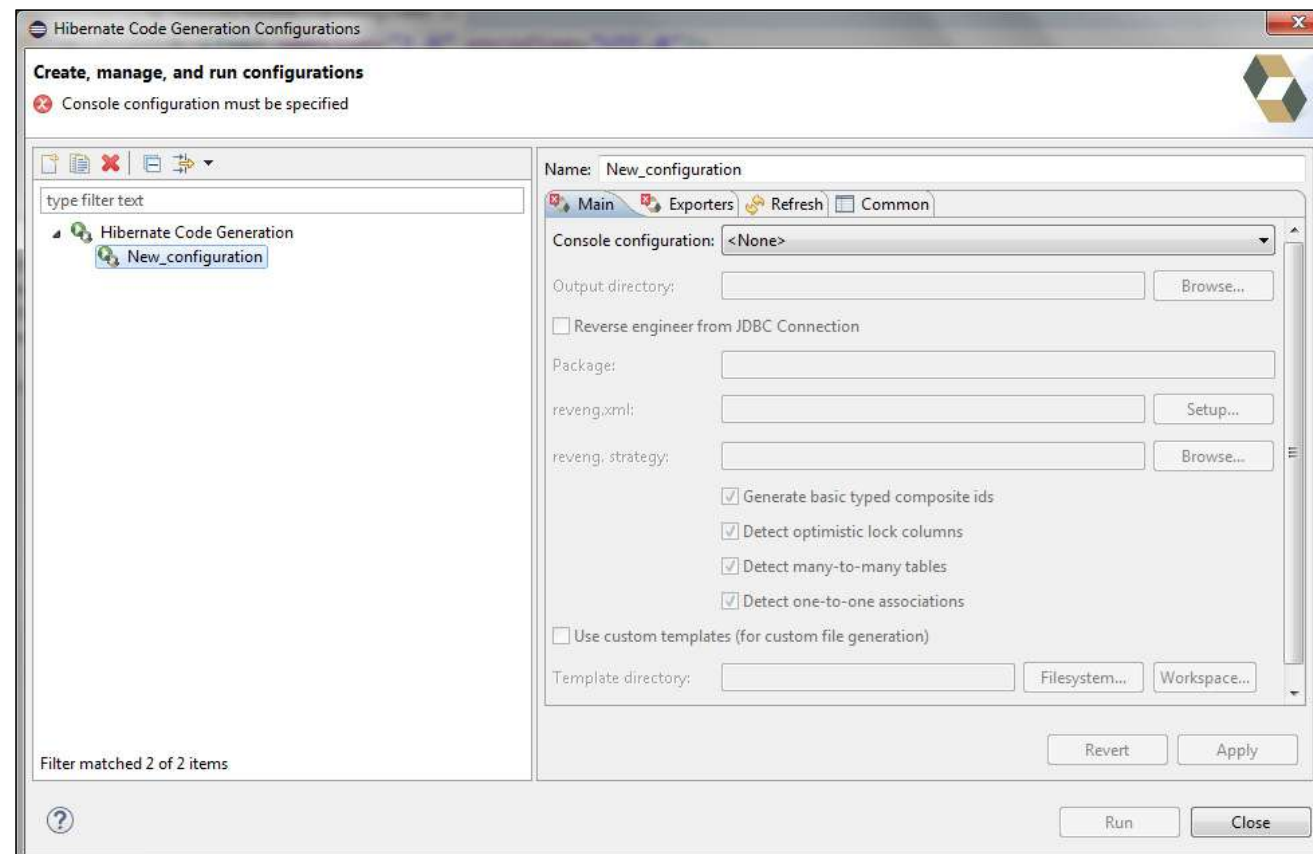
5. HIBERNATE CON EL PLUGIN JBOSS

Paso 18. Generamos las clases de nuestra base de datos “empresa”. Para ello pulsamos en la flechita situada a la derecha del botón **Run As** y seleccionamos **Hibernate Code Generation Configurations**.



5. HIBERNATE CON EL PLUGIN JBOSS

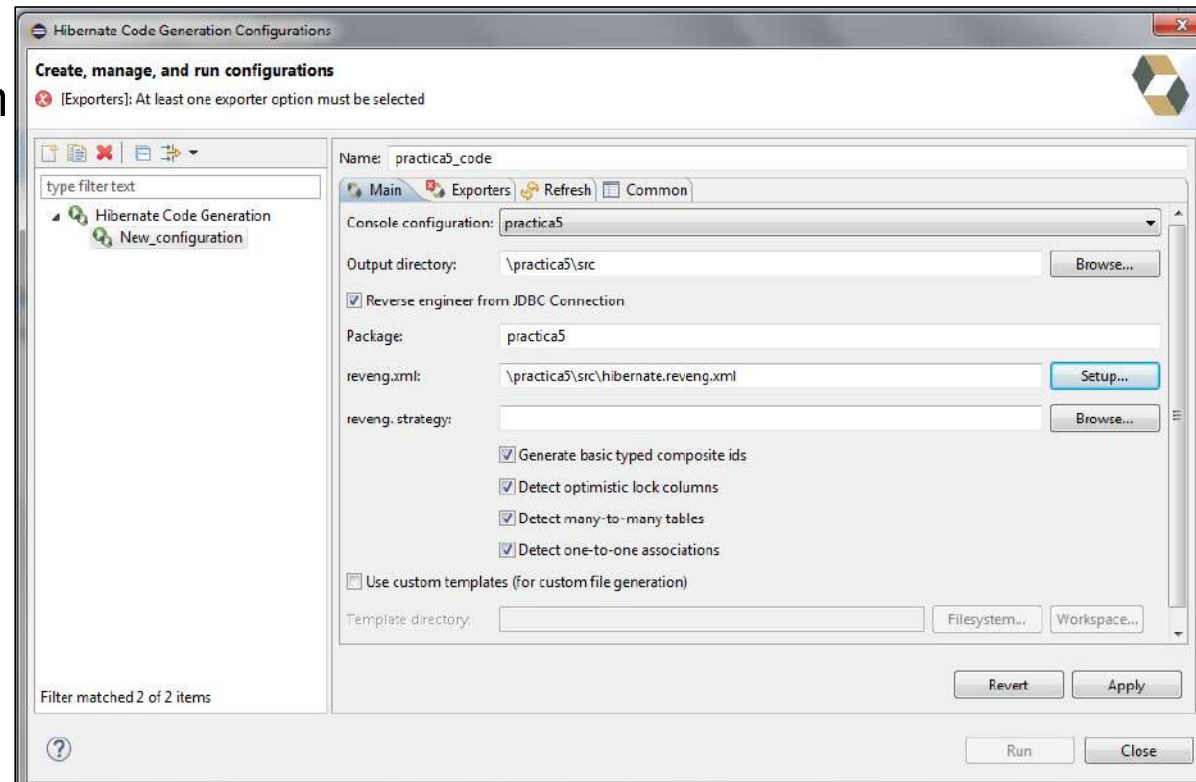
Paso 19. En la nueva ventana hacemos doble clic en la opción **Hibernate Code Generation** del marco de la izquierda. Aparecerán varias pestañas:



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 20. En la pestaña **Main** configuramos:

- **Name:** Nombre para la configuración
- **Console configuration:** Indicamos proyecto5
- **Output directory:** debe ser la carpeta **src**.
- **Package:** el nombre del paquete donde se crearán las clases, por ejemplo “practica5”
- **reveng.xml:** localizamos el fichero reveng.xml creado anteriormente.

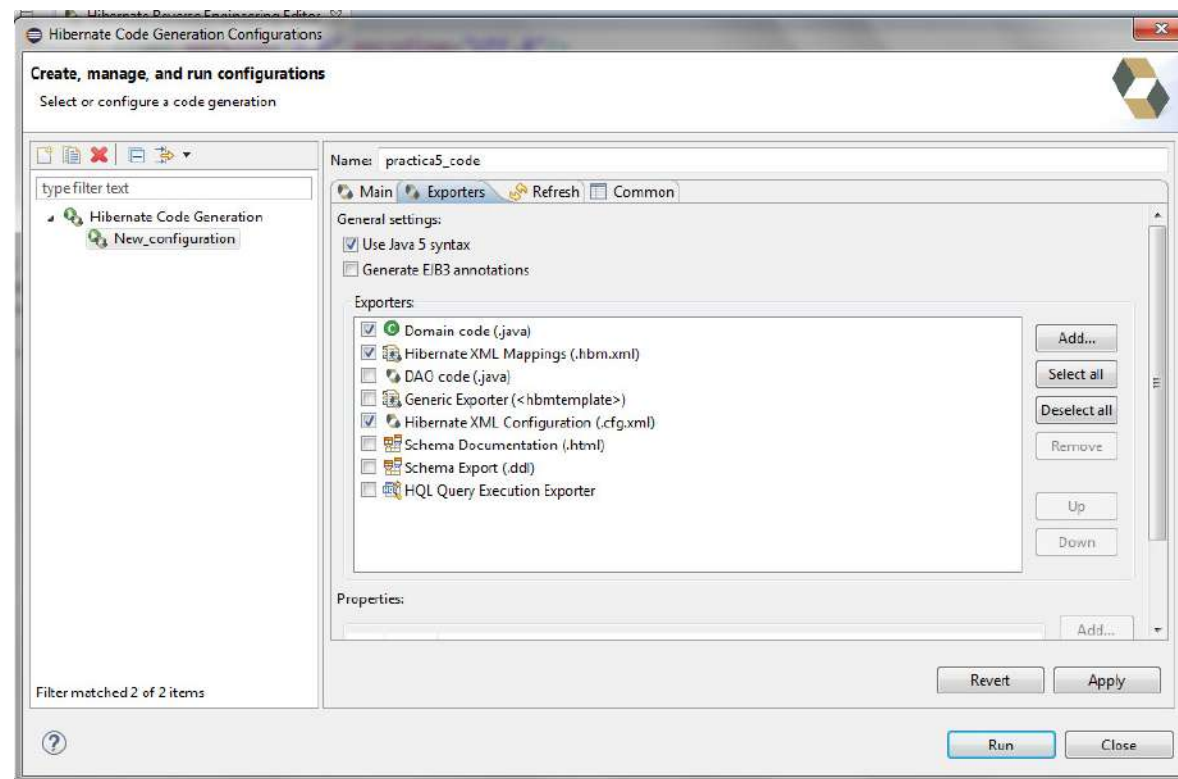


5. HIBERNATE CON EL PLUGIN JBOSS

Paso 21. En la pestaña **Exporters** se indica los fichero a generar. Se marcarán:

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

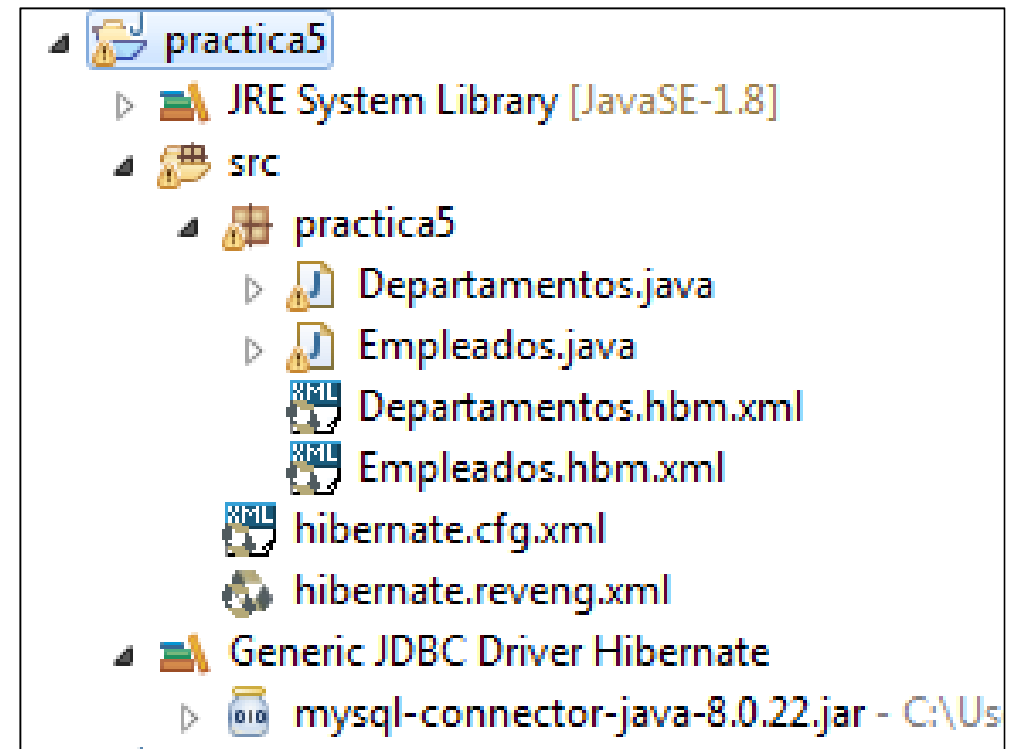
Clicamos en *Apply* y posteriormente en *Run*.



5. HIBERNATE CON EL PLUGIN JBOSS

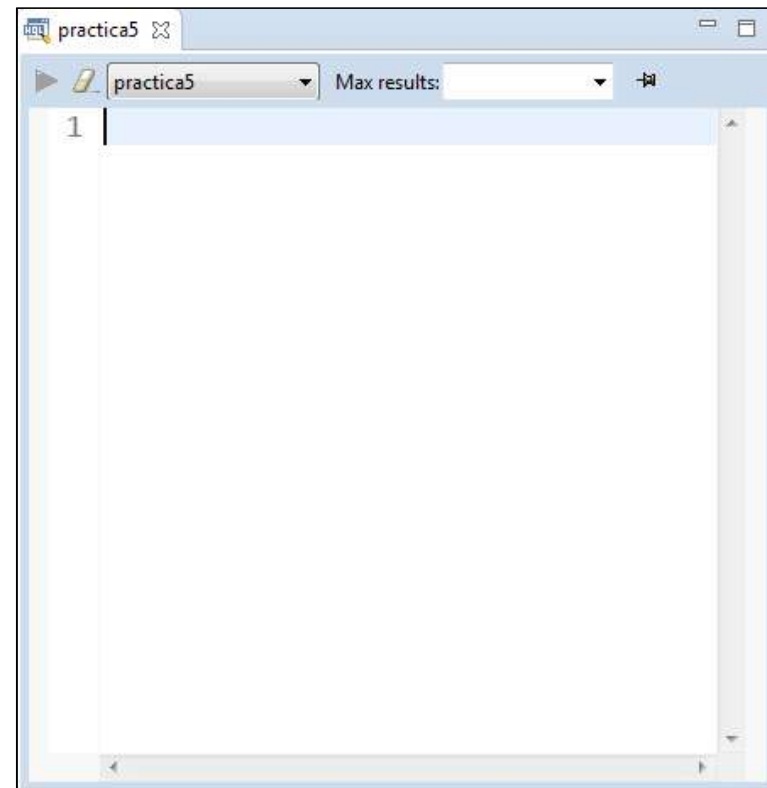
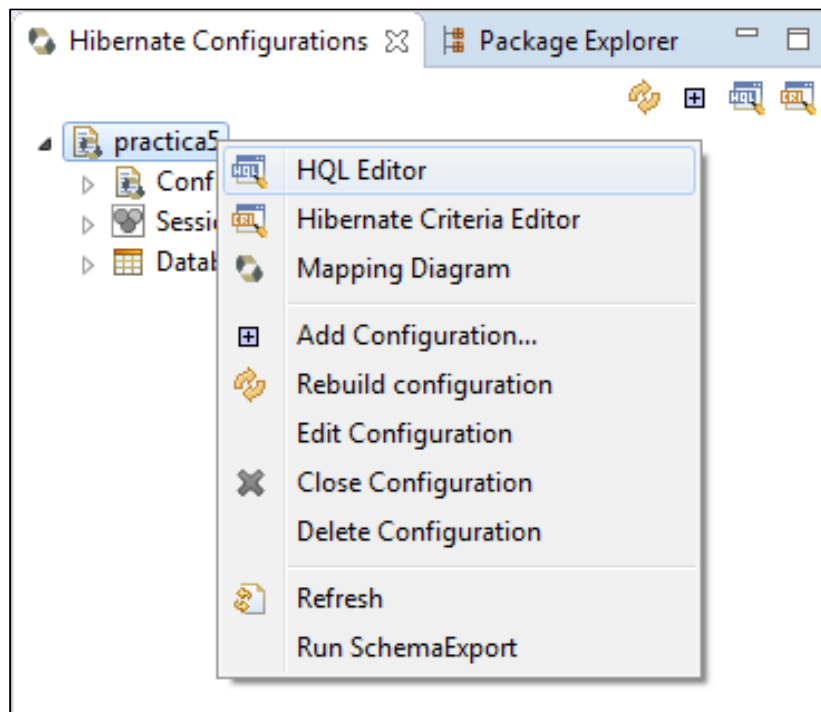
Paso 22. Al ejecutarse nos genera un paquete llamado primero con las clases Java de las tablas emple y depart que contienen los métodos getters y setters de cada campo de la tabla. También contendrá los XML con información del mapeo de cada tabla.

Conviene detenerse un poco y analizar cómo ha mapeado Hibernate cada tabla. Sobre todo es interesante la implementación de las claves ajenas.



5. HIBERNATE CON EL PLUGIN JBOSS

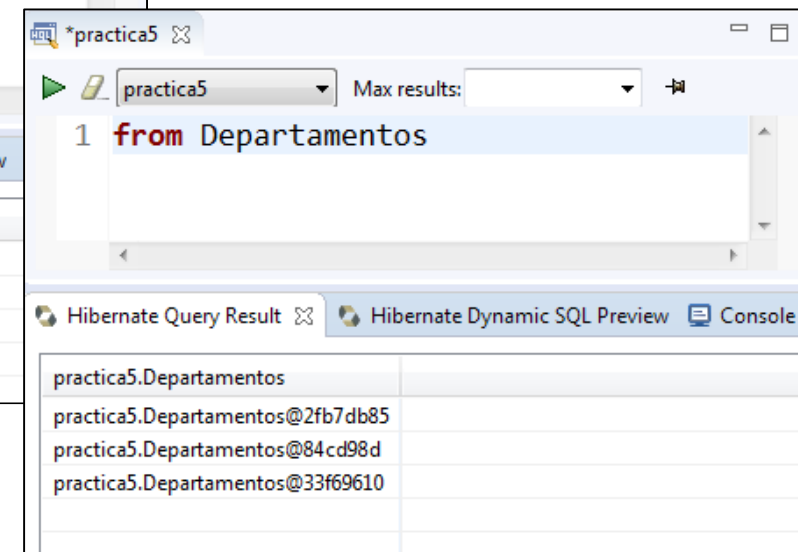
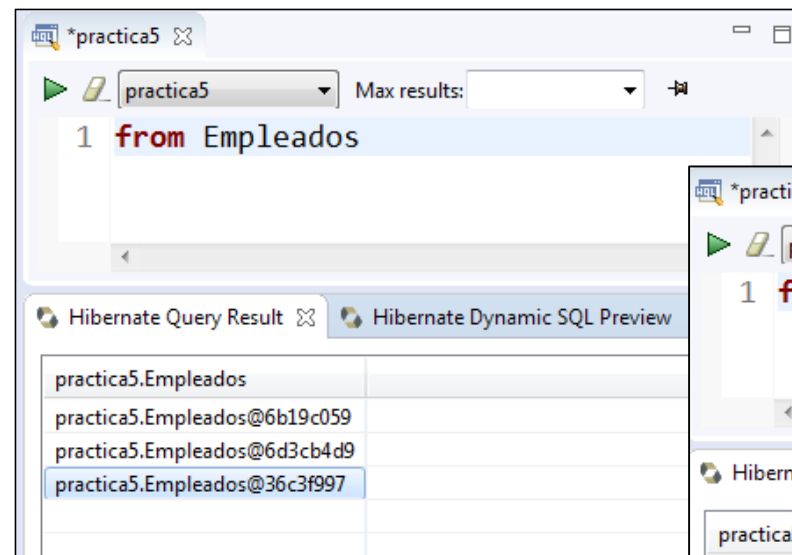
Paso 23. Para realizar la primera consulta en HQL (Huibernate Query language), debemos de abrir la vista Hibernate Configurations. En la nueva pestaña, se tiene que pulsar sobre nuestro proyecto con el botón derecho y seleccionar HQL Editor



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 24. En el editor HQL podemos realizar consultas. Por ejemplo, si se escribe "*from Empleados*" y pulsamos la flechita verde nos debe aparecer en la pestaña **Hibernate Query Result** el resultado de la consulta.

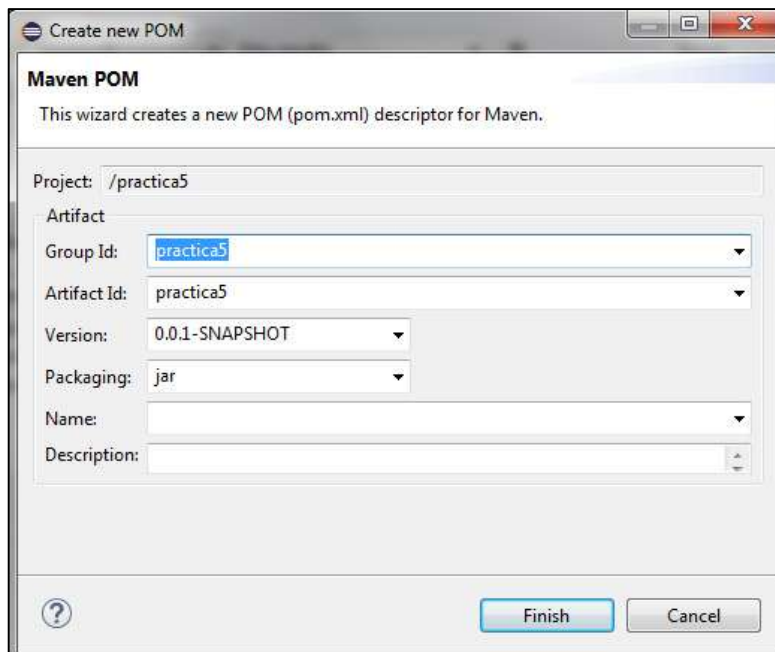
Desde este entorno también se pueden realizar consultas SQL aunque hay ciertas restricciones. Por ejemplo, no se puede utilizar * en el SELECT



5. HIBERNATE CON EL PLUGIN JBOSS

Librerías de Hibernate

Paso 25. Tenemos dos formas de agregar las librerías Hibernate a nuestro proyecto. Optaremos por la vía Maven. Convertimos nuestro proyecto a Maven y configuraremos el fichero pom.xml

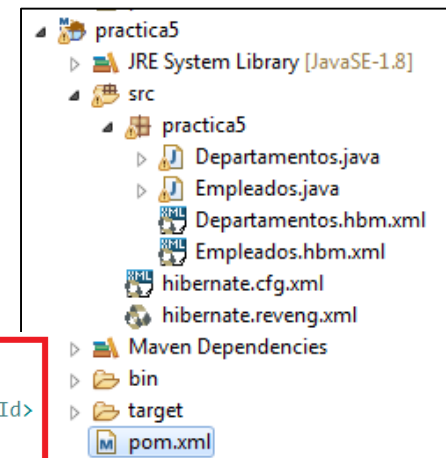


```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.6.Final</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.22</version>
  </dependency>
</dependencies>

</project>
```

```
<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.2</version>
</dependency>
```

</dependencies>



5. HIBERNATE CON EL PLUGIN JBOSS

Paso 26. Se debe de crear un **Singleton**: Es un patrón diseñado para restringir la creación de objetos pertenecientes a un clase. Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella (**es como el fichero de prácticas anteriores BaseDatos.java**):

- Nuestro Singleton será una clase de ayuda que accede a **SessionFactory** para obtener un objeto de sesión, hay una única **SessionFactory** para toda la aplicación.
- El nombre de la clase es *HibernateUtil.java* y se incluirá en el paquete de nuestro proyecto.
- Con esta clase podemos obtener la sesión actual desde cualquier parte de nuestro proyecto.

5. HIBERNATE CON EL PLUGIN JBOSS

```
HibernateUtil.java  ✖
1 package practica5;
2
3 import org.hibernate.SessionFactory;
4 import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
5 import org.hibernate.cfg.Configuration;
6
7 public class HibernateUtil {
8     private static final SessionFactory sessionFactory = buildSessionFactory();
9
10    private static SessionFactory buildSessionFactory(){
11        try{
12            return new Configuration().configure().
13                buildSessionFactory(new StandardServiceRegistryBuilder().
14                    configure().build());
15        }
16        catch (Throwable ex) {
17            System.err.println("Initial SessionFactory creation failed."+ex);
18            throw new ExceptionInInitializerError(ex);
19        }
20    }
21    public static SessionFactory getSessionFactory(){
22        return sessionFactory;
23    }
24 }
```

PRACTICA 5

Paso 27. Para finalizar esta primera parte de instalación del plugin JBOSS y de validación, crea en nuestro proyecto una nueva clase que llamaremos *Main.java*. Haremos un caso de uso que insertará una fila en la tabla *departamentos*

```
Main.java
1 package practica5;
2
3 import org.hibernate.Session;
4 import org.hibernate.SessionFactory;
5 import org.hibernate.Transaction;
6
7 public class Main {
8     public static void main(String[] args) {
9         //En primer lugar se obtiene la sesión creada por el Singleton.
10        SessionFactory session = HibernateUtil.getSessionFactory();
11        //Abrimos sesión e iniciamos una transacción
12        Session session = session.openSession();
13        Transaction tx = session.beginTransaction();
14
15        System.out.println("Inserto una fila en departamentos");
16        //Creamos un nuevo objeto Departamentos y damos valor a sus atributos
17        Departamentos dep = new Departamentos();
18        dep.setCodDept(4);
19        dep.setNombre("MARKETING");
20        dep.setDireccion("GUADALAJARA");
21        dep.setObjetivos(10000);
22
23        //Guardamos en la base de datos y comprometemos la información
24        session.save(dep);
25        tx.commit();
26        session.close();
27    }
28 }
```

6. HIBERNATE

Mapeo objeto-relacional (ORM Object-Relational Mapping)

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 bbdd relacional

- **Ventajas del uso de un ORM:**

- Ayuda a reducir tiempo de desarrollo
- Abstracción de la base de datos. Reutilización
- Independencia de la BD → Migrar con facilidad
- Lenguaje propio para realizar consultas

- **Inconvenientes:**

- Aplicaciones más lentas
- Configuración del entorno más compleja

- **Herramientas:** Doctrine, Propel, ADOdb Active Record, Hibernate, Oracle Toplink, iPersist, etc.

6. HIBERNATE

Que es Hibernate

- Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java (disponible para .NET con el nombre de Nhibernate) que facilita el mapeo de atributos mediante ficheros declarativos (XML)
- Se está convirtiendo en el estándar de facto para almacenamiento persistente cuando queremos independizar la capa de negocio del almacenamiento de la información.
- Con Hibernate no emplearemos habitualmente SQL para acceder a datos, sino que el propio motor de Hibernate, mediante el uso de factorías (patrón de diseño **Factory**) construirá esas consultas para nosotros.
- Hibernate pone a disposición del diseñador un lenguaje llamado **HQL (Hibernate Query Language)** que permite acceder a los datos mediante POO.

6. HIBERNATE

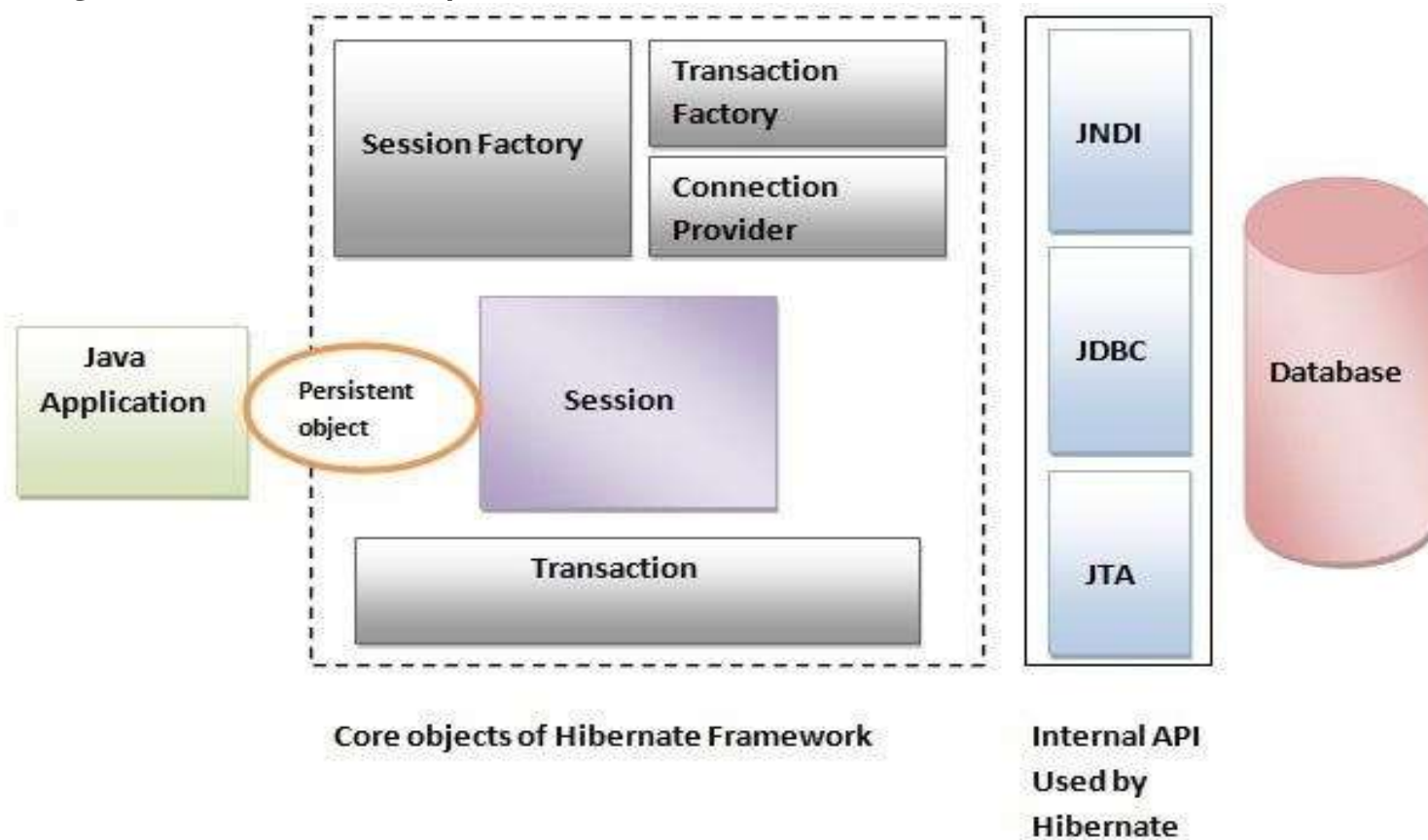
Interfaces de Hibernate

- La interfaz **SessionFactory** (org.hibernate.SessionFactory): permite obtener instancias de **Session**. Esta interfaz debe compartirse entre muchos hilos de ejecución. Normalmente hay una única **SessionFactory** para toda la aplicación. Si la aplicación accede a varias bases de datos se necesitará una **SessionFactory** por cada base de datos. Por otro lado, la clase `Session` nos ofrece métodos como **save(Object)**, **createQuery(String)**, **beginTransaction()**, **close()**, etc.
- La interfaz **Configuration** (org.hibernate.cfg.Configuration): se utiliza para configurar Hibernate. La aplicación utiliza una instancia de **Configuration** para especificar la ubicación de los documentos que indican el mapeado de los objetos y a continuación crea la **SessionFactory**
- La interfaz **Query** (org.hibernate.Query): permite ejecutar consultas y controla cómo se realizan. Las consultas se escriben en **HQL**
- La interfaz **Transaction** (org.hibernate.Transaction): permite realizar modificaciones o consultas en la base de datos según el paradigma ACID

6. HIBERNATE

Arquitectura Hibernate

Una forma de imaginarse esta arquitectura es:

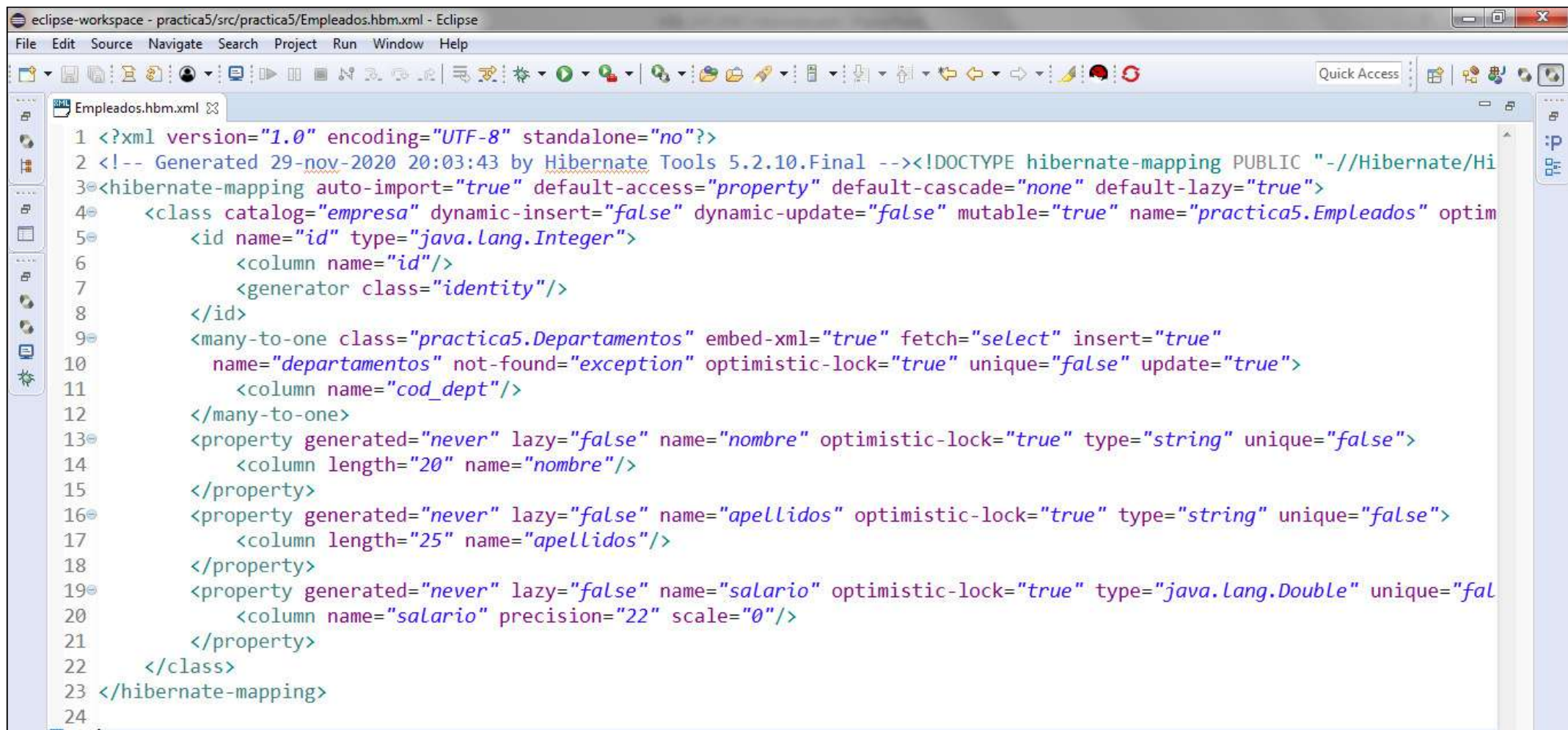


6. HIBERNATE

Estructura de los ficheros de mapeo (hbm.xml)

- Hibernate utiliza unos ficheros de mapeo para relacionar las tablas de la base de datos con los objetos Java. Estos ficheros están en formato XML y tienen la extensión **.hbm.xml**.
- En el proyecto anterior se han creado los ficheros **Empleados.hbm.xml** y **Departamentos.hbm.xml** asociados a las tablas emple y depart respectivamente.
- Veamos la estructura de estos ficheros

6. HIBERNATE



```
eclipse-workspace - practica5/src/practica5/Empleados.hbm.xml - Eclipse
File Edit Source Navigate Search Project Run Window Help

Empleados.hbm.xml
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!-- Generated 29-nov-2020 20:03:43 by Hibernate Tools 5.2.10.Final --><!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hi
3 <hibernate-mapping auto-import="true" default-access="property" default-cascade="none" default-lazy="true">
4   <class catalog="empresa" dynamic-insert="false" dynamic-update="false" mutable="true" name="practica5.Empleados" optim
5     <id name="id" type="java.lang.Integer">
6       <column name="id"/>
7       <generator class="identity"/>
8     </id>
9     <many-to-one class="practica5.Departamentos" embed-xml="true" fetch="select" insert="true"
10       name="departamentos" not-found="exception" optimistic-lock="true" unique="false" update="true">
11       <column name="cod_dept"/>
12     </many-to-one>
13     <property generated="never" lazy="false" name="nombre" optimistic-lock="true" type="string" unique="false">
14       <column length="20" name="nombre"/>
15     </property>
16     <property generated="never" lazy="false" name="apellidos" optimistic-lock="true" type="string" unique="false">
17       <column length="25" name="apellidos"/>
18     </property>
19     <property generated="never" lazy="false" name="salario" optimistic-lock="true" type="java.lang.Double" unique="fal
20       <column name="salario" precision="22" scale="0"/>
21     </property>
22   </class>
23 </hibernate-mapping>
24
```

6. HIBERNATE

- **hibernate-mapping**: todos los ficheros de mapeo comienzan y acaban con esta etiqueta.
- **class**: esta etiqueta engloba la clase con sus atributos indicando el mapeo a la tabla de la base de datos.
 - **name**: nombre de la clase
 - **table**: nombre de la tabla que representa el objeto
 - **catalog**: nombre de la base de datos
- Dentro de class distinguimos la etiqueta **id** en la cual se indica en **name** el campo que representa al atributo clave y en **column** su nombre sobre la tabla, en **type** el tipo de datos. También tenemos la propiedad **generator** que indica la naturaleza del campo clave. En este caso es “identity” porque es un campo generado por la base de datos. Este atributo se correspondería con el campo id de la tabla Empleados.

6. HIBERNATE

- 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.
- La etiqueta **set** se utiliza para mapear colecciones. Dentro de set se definen varios atributos:
 - **name**: indica el nombre del atributo generado
 - **table**: el nombre de la tabla de donde se tomará la colección
 - El elemento **key** define el nombre de la columna identificadora en la asociación
 - El elemento **one-to-many** define la relación (un departamentos puede tener muchos empleados)
 - **class**: indica de qué tipo son los elementos de la colección.
- Los tipos que declaramos en los ficheros de mapeo no son tipos de datos Java ni SQL. Se llaman **tipos de mapeo Hibernate**.

6. HIBERNATE

Clases persistentes

- Las clases referenciadas en el elemento **class** de los ficheros de mapeo hacen referencia a las clases generadas en nuestro proyecto como Empleados.java y Departamentos.java. A estas clases se les llama **clases persistentes**.
- Las clases persistentes son las clases que implementan las entidades del problema, deben implementar la interfaz **Serializable**. Equivalen a una tabla de la base de datos, y un registro o fila es un objeto persistente de esa clase
- Estas clases representan un objeto emple y un objeto depart, por lo tanto, podemos crear objetos empleados y departamentos a partir de ellas. Tienen unos atributos privados y unos métodos públicos (*getters* y *setters*) para acceder a los mismos.
- A estas reglas se les suele llamar modelo de programación **POJO** (Plain Old Java Objects)

6. HIBERNATE

```
Empleados.java ✖
1 package practica5;
2 // Generated 29-nov-2020 20:03:33 by Hibernate Tools 5.2.10.Final
3
4 /**
5  * Empleados generated by hbm2java
6  */
7 public class Empleados implements java.io.Serializable {
8
9     private Integer id;
10    private Departamentos departamentos;
11    private String nombre;
12    private String apellidos;
13    private Double salario;
14
15    public Empleados() {
16    }
17
18    public Empleados(Departamentos departamentos, String nombre,
19                     String apellidos, Double salario) {
20        this.departamentos = departamentos;
21        this.nombre = nombre;
22        this.apellidos = apellidos;
23        this.salario = salario;
24    }
25
```

```
26    public Integer getId() {
27        return this.id;
28    }
29
30    public void setId(Integer id) {
31        this.id = id;
32    }
33
34    public Departamentos getDepartamentos() {
35        return this.departamentos;
36    }
37
38    public void setDepartamentos(Departamentos departamentos) {
39        this.departamentos = departamentos;
40    }
41
42    public String getNombre() {
43        return this.nombre;
44    }
45
46    public void setNombre(String nombre) {
47        this.nombre = nombre;
48    }
49
```


6. HIBERNATE

Sesiones y objetos Hibernate

- Como ya hemos visto, para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto **Session** utilizando la clase **SessionFactory**.

```
//Inicializa el entorno Hibernate: carga el fichero hibernate.cfg.xml
Configuration cfg = new Configuration().configure();

//Crea el ejemplar de SessionFactory. Se necesita crear un objeto
//StandardServiceRegistry que contiene la lista de servicios de Hibernate
//El ejemplar de SessionFactory se crea normalmente solo una vez y se utiliza
//para crear todas las sesiones relacionadas con el contexto dado (singleton)
StandardServiceRegistry registry=
    new StandardServiceRegistryBuilder().configure().build();

final SessionFactory sessionFactory = cfg.buildSessionFactory (registry) ;

//Obtiene un objeto Session
Session session = sessionFactory.openSession();
```

6. HIBERNATE

Transacciones

- Un objeto **Session** de Hibernate representa una única unidad de trabajo. Al crear la sesión se crea la transacción para dicha sesión. Se deben cerrar las sesiones.

```
//Abrimos sesión e iniciamos una transacción
Session session = session.openSession();
Transaction tx = session.beginTransaction();

// ... código de persistencia ...//

//Hacemos el commit de la transaccion y cerramos la sesión
tx.commit();
session.close();
```

- El método **beginTransaction()** marca el inicio de una transacción. El método **commit()** la valida; y el método **rollback()** deshace los cambios.

6. HIBERNATE

Estados de un objeto Hibernate

- **Transitorio** (Transient) : Un objeto es transitorio si ha sido recién instanciado y no está asociado a una Session de Hibernate. No tiene una representación persistente en la bbdd y no se le ha asignado un identificador.
- **Persistente** (Persistent) : Un objeto estará en este estado cuando ya está almacenado en la bbdd. Puede haber sido guardado o cargado pero se encuentra en el ámbito de una **Session**.
- **Separado** (Detached) : Una instancia separada es un objeto que se ha hecho persistente pero su sesión ha sido cerrada. La referencia al objeto todavía es válida, y podría ser modificado. Para hacer persistentes dichas modificaciones debemos asociar el objeto a una nueva **Session**

6. HIBERNATE

Carga de objetos

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

- **<T> T load (Class<T> Clase, Serializable id)** → Devuelve la instancia persistente de la clase indicada con el identificador dado. Si la instancia no existe el método lanza una excepción.
- **Object load (String nombreClase, Serializable id)** → Lo mismo que antes pero indicando nombre de la clase
- **<T> T get (Class<T> Clase, Serializable id)** → Lo mismo pero si la instancia no existe devuelve *null*
- **Object get (String nombreClase, Serializable id)** → Lo mismo que en el caso anterior pero indicando el nombre de la clase.

6. HIBERNATE

Carga de objetos: Ejemplos

```
Departamentos dep = new Departamentos();  
try {  
    dep = (Departamentos) session.load(Departamentos.class, id);  
    System.out.println(dep);  
} catch (ObjectNotFoundException o) {  
    System.out.println ("No existe el departamento");  
}
```

El método load funciona con excepciones si no existe el objeto

```
Departamentos dep = (Departamentos) session.get (Departamentos.class, id);  
if (dep==null) {  
    System.out.println ("No existe el departamento");  
}  
else {  
    System.out.println("Nombre Dep: "+dep.getNombre());  
}
```

El método get no necesita de excepciones. Si no existe devuelve un null

6. HIBERNATE

Almacenamiento, modificación y borrado de objetos

Disponemos de los siguientes métodos:

- `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 obj)` → Actualiza en la bbdd el objeto que se pasa como argumento. El objeto a modificar debe ser cargado con el método `load()` o `get()`
- `void delete (Object obj)` → Elimina de la bbdd el objeto que se pasa como argumento. El objeto a eliminar debe ser cargado con el método `load()` o `get()`.

6. HIBERNATE

Almacenamiento, modificación y borrado de objetos: Ejemplos

```
//Departamentos dep = new Departamentos(nombre, direccion, objetivo);
Departamentos dep = new Departamentos();
dep.setNombre(nombre);
dep.setDireccion(direccion);
dep.setObjetivos(objetivo);
session.save(dep);
```

Generacion y guardado de un nuevo departamento con el método save

```
Departamentos dep = session.get(Departamentos.class, id);
if (dep==null) System.out.print("No existe departamento con ese id");
else {
    System.out.print("Introduce nombre nuevo de departamento (" + dep.getNombre() + "):");
    dep.setNombre(reader.next());
    System.out.print("Introduce objetivos nuevos (" + dep.getObjetivos() + "):");
    dep.setObjetivos(reader.nextInt());
    session.update(dep);
}
```

Para modificar un objeto, primero hemos de cargarlo, realizar la modificaciones y, por último, utilizar el método update().

6. HIBERNATE

Almacenamiento, modificación y borrado de objetos: Ejemplos

```
Empleados emp = session.get (Empleados.class, id);  
if (emp==null)  
    System.out.print("Empleado no existe");  
else  
    session.delete (emp);
```

Borrado de un empleado
usando previamente el
método get

```
Departamentos dep = new Departamentos();  
try {  
    dep = (Departamentos) session.load(Departamentos.class, id);  
    session.delete(dep);  
    tx.commit();  
    System.out.println ("Departamento eliminado");  
}catch (ObjectNotFoundException c) {  
    System.out.println ("No existe el departamento");  
}catch (ConstraintViolationException c) {  
    System.out.println ("No se puede eliminar. Tiene empleados");  
}  
  
}catch (Exception e) {  
    e.printStackTrace();  
}
```

Eliminación de un departamento
con el método load, controlando
las distintas excepciones: que el
departamento no exista y que
tenga empleados

7. CONSULTAS HIBERNATE

Consultas Hibernate

- Hibernate soporta un lenguaje de consulta orientado a objetos denominado **HQL** (Hibernate Query Language)
- Las consultas HQL y SQL son representadas con una instancia de **org.hibernate.Query**.
- La interfaz **Query** ofrece métodos para ligar parámetros, manejo del conjunto resultado y para la ejecución de la consulta real.
- Siempre se obtiene una Query utilizando el objeto **Session** actual.

7. CONSULTAS HIBERNATE

Métodos de Query

- **Iterator iterate()** → Devuelve en un objeto Iterator el resultado de la consulta
- **List list()** → Devuelve el resultado de la consulta en un List
- **Query setFetchSize (int size)** → Fija el número de resultados a recuperar en cada acceso a la base de datos al valor indicado en size
- **int executeUpdate()** → Ejecuta la sentencia de modificación o borrado. Devuelve el número de entidades afectadas
- **String getQueryString()** → Devuelve la consulta en un string
- **Query setCharacter(int posición, char valor)** o también **Query setCharacter(String nombre, char valor)** → Asigna el valor indicado en el método a un parámetro de tipo char. Posición indica la posición del parámetro empezando en 0. Nombre es el nombre (:nombre) del parámetro

7. CONSULTAS HIBERNATE

- **Object uniqueResult()** → Devuelve un objeto (cuando sabemos que la consulta devuelve un objeto) o nulo si la consulta no devuelve resultados
- **Query setDate (int posición, Date fecha)** o también, **Query setDate (String nombre, Date fecha)** → Asigna la fecha a un parámetro de tipo DATE
- **Query setDouble (int posición, double valor)** o también **Query setInteger (String nombre, double valor)** → Asigna valor a un parámetro de tipo FLOAT
- **Query setInteger(int posición, String valor)** o también **Query setInteger(String nombre, String valor)** → Asigna valor a un parámetro de tipo entero
- **Query setString(int posición, String valor)** o también **Query setString(String nombre, String valor)** → Asigna valor a un parámetro de tipo VARCHAR

7. CONSULTAS HIBERNATE

- **Query setParameterList (String nombre, Collection valores):** Asigna una colección de valores al parámetro cuyo nombre se indica en nombre
- **Query setParameter (int posicion, 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 UPDATE o DELETE, devuelve el número de entidades afectadas.
- API de Hibernate: <https://docs.jboss.org/hibernate/orm/current/javadocs/>

7. CONSULTAS HIBERNATE

- Para realizar una consulta usaremos el método **createQuery()** de la interfaz **SharedSessionContract**. Se le pasará en un String la consulta HQL. Por ejemplo, para hacer una consulta sobre la tabla departamentos, mapeada con la clase Departamentos, se escribe de la siguiente manera:

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

- Para recuperar los datos de la consulta usaremos el método **getResultList()** (el método **list()** está deprecated). Devuelve una colección con el conjunto de todos los resultados de la consulta.

```
List <Departamentos> lista = q.getResultList();  
System.out.println("Número de departamentos: "+ lista.size());  
for(Departamentos dep:lista){  
    System.out.println(dep.getCodDept() + " " + dep.getNombre());  
}
```

7. CONSULTAS HIBERNATE

- Se puede obtener de un determinado departamento, la lista de sus empleados. Basta con usar el método **getEmpleadoses()** de la clase Departamentos:

```
System.out.print("Introduce id de departamento para ver sus empleados: ");
int id=reader.nextInt();

Departamentos dep = session.get(Departamentos.class, id);
Set<Empleados> listaemp = dep.getEmpleadoses();
for(Empleados emp:listaemp){
    System.out.println(emp.getId() + " " + emp.getNombre());
}
```

PRACTICA 6

Realiza un programa en Java que ofrezca las siguientes opciones:

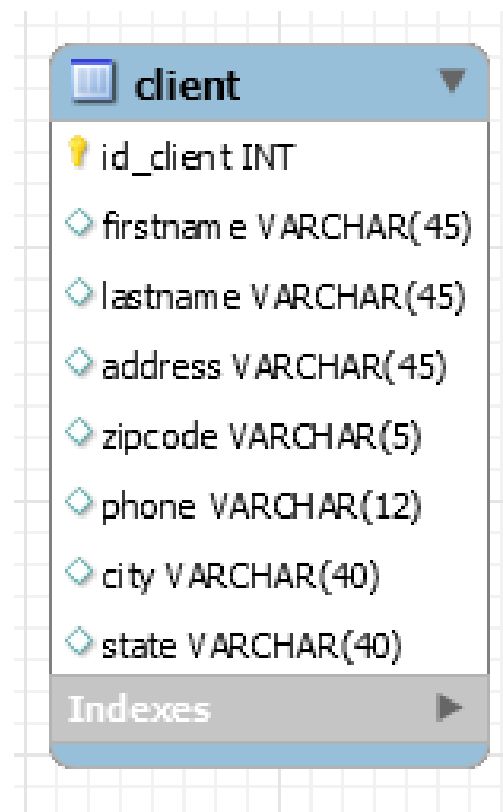
1. Listado de departamentos → **CreateQuery**
2. Listado de empleados → **CreateQuery**
3. Consulta los datos de un determinado departamento por su id → **get**
4. Consulta los datos de un empleado determinado por su id → **load**
5. Consulta los datos de un empleado determinado por el nombre → **CreateQuery**
6. Consulta los datos de los empleados de un departamento, mediante consulta HQL → **CreateQuery**
7. Consulta los datos de los empleados de un departamento, sin consulta HQL. → **get departamento + lista empleados**
8. Consulta el salario medio, máximo y mínimo de todos los empleados. → **createQuery**

PRACTICA 6

9. Consulta los nombres de los empleados junto con el nombre de su departamento → **createQuery**
10. Consulta por cada departamento el número de departamento, el nombre, el número de empleados que tiene y el salario medio. → **createQuery**
11. Inserte un departamento. El programa recibirá los datos de un departamento
12. Inserte un empleado en la tabla empleados. El programa recibe del usuario los valores a insertar (incluido el id). Se deben de comprobar los siguientes casos:
 - Que el departamento no exista en la tabla departamentos
 - Que el empleado ya exista
13. Suba el salario en 10000 a los empleados de un departamento. El programa recibirá el número de departamento y el incremento.
14. Realiza un programa que modifique el salario y el departamento de un empleado determinado, sumando 1000 al salario y asignándole a otro departamento.

DISEÑO BASE DE DATOS PIZZERIA

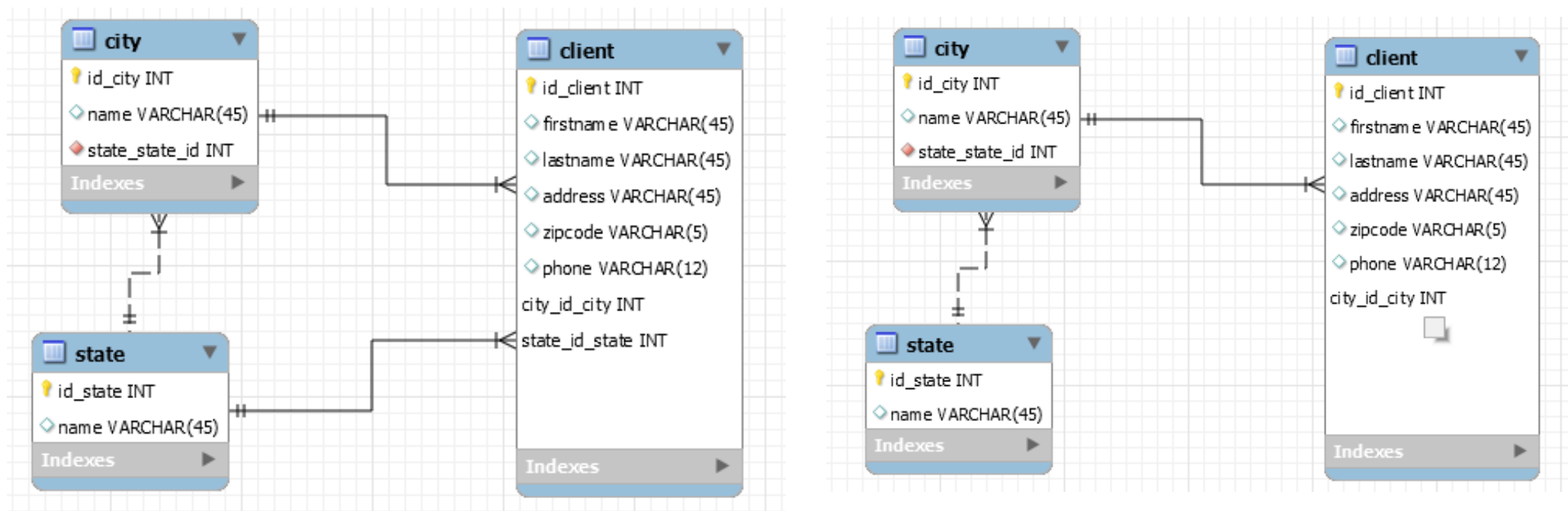
Pas 1. Per a cada client emmagatzemem un identificador únic, nom, cognoms, adreça, codi postal, localitat, província i número de telèfon.



DISEÑO BASE DE DATOS PIZZERIA

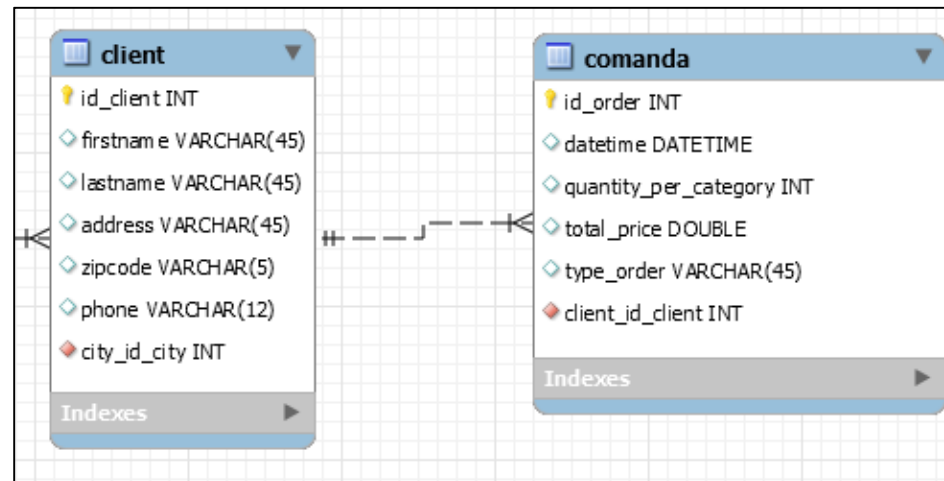
Pas 2. Les dades de localitat i província estaran emmagatzemats en taules separades. Sabem que una localitat pertany a una única província, i que una província pot tenir moltes localitats.

Pas 3. Per a cada localitat emmagatzemem un identificador únic i un nom. Per a cada província emmagatzemem un identificador únic i un nom.



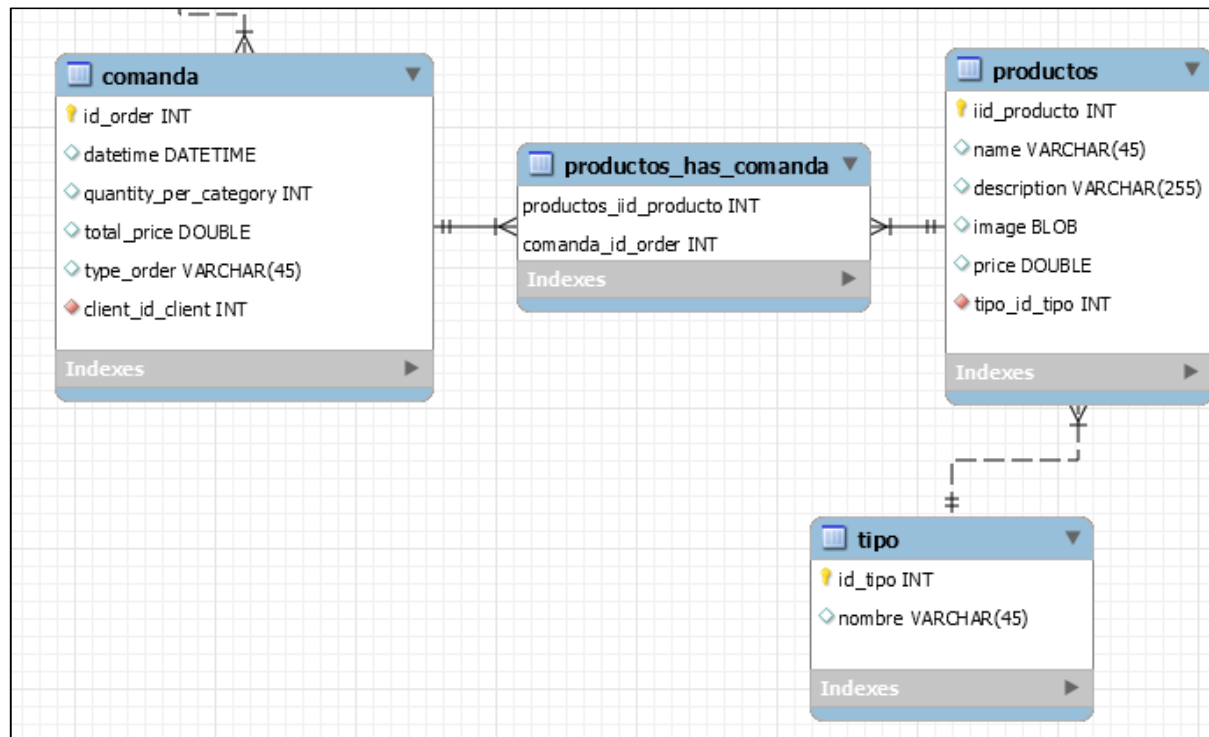
DISEÑO BASE DE DATOS

Pas 4. Un client pot realitzar moltes comandes, però una única comanda només pot ser realitzat per un únic client. De cada comanda s'emmagatzema un identificador únic, data/hora, si la comanda és per a repartiment a domicili o per a recollir en botiga, la quantitat de productes que s'han seleccionat de cada tipus i el preu total.



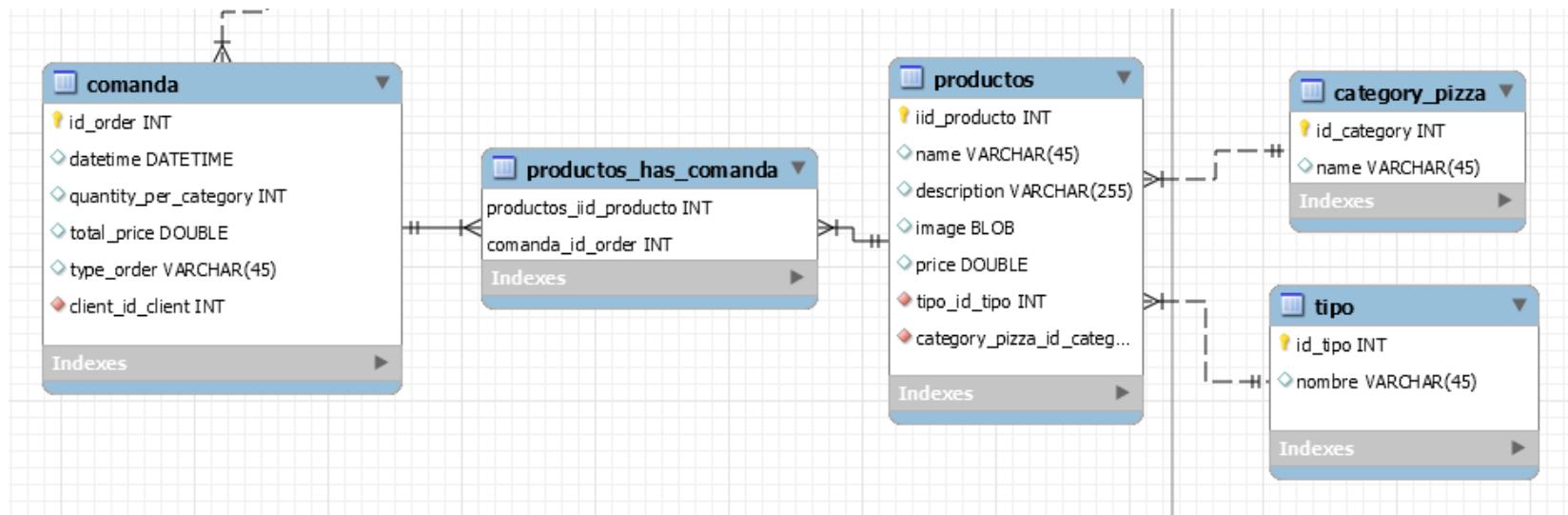
DISEÑO BASE DE DATOS

Pas 5. Una comanda pot constar d'un o diversos productes. Els productes poden ser pizzas, hamburgueses i begudes. De cada producte s'emmagatzema: un identificador únic, nom, descripció, imatge i preu.



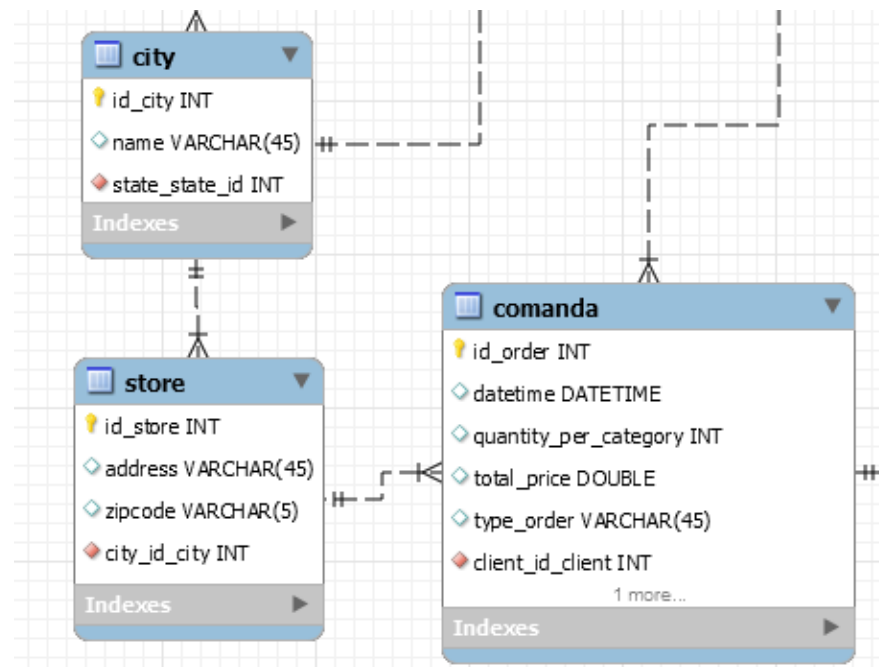
DISEÑO BASE DE DATOS

Pas 6. En el cas de les pizzes existeixen diverses categories que poden anar canviant de nom al llarg de l'any. Una pizza només pot estar dins d'una categoria, però una categoria pot tenir moltes pizzes. De cada categoria s'emmagatzema un identificador únic i un nom.



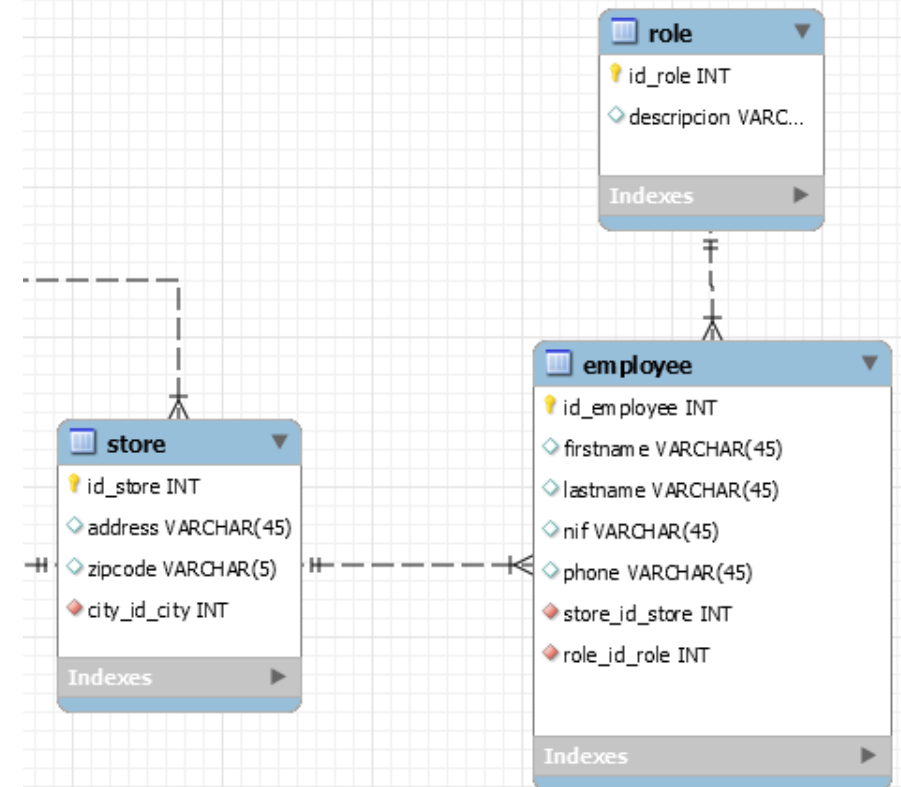
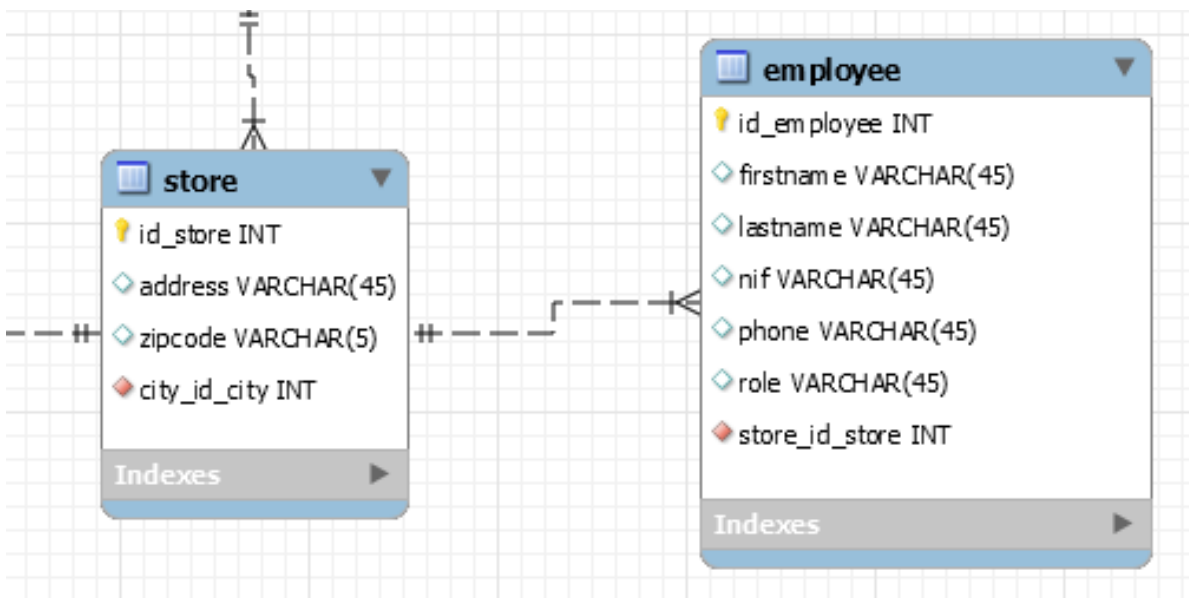
DISEÑO BASE DE DATOS

Pas 7. Una comanda és gestionada per una única botiga i una botiga pot gestionar moltes comandes. De cada botiga s'emmagatzema un identificador únic, adreça, codi postal, localitat i província.



DISEÑO BASE DE DATOS

Pas 8. En una botiga poden treballar molts empleats i un empleat només pot treballar en una botiga. De cada empleat s'emmagatzema un identificador únic, nom, cognoms, nif, telèfon i si treballa com a cuiner o repartidor.



DISEÑO BASE DE DATOS

Pas 9. Per a les comandes de repartiment a domicili interessa guardar qui és el repartidor que realitza el lliurament de la comanda i la data/hora del moment del lliurament.

