

UT3. MAPEO OBJETO-RELACIONAL

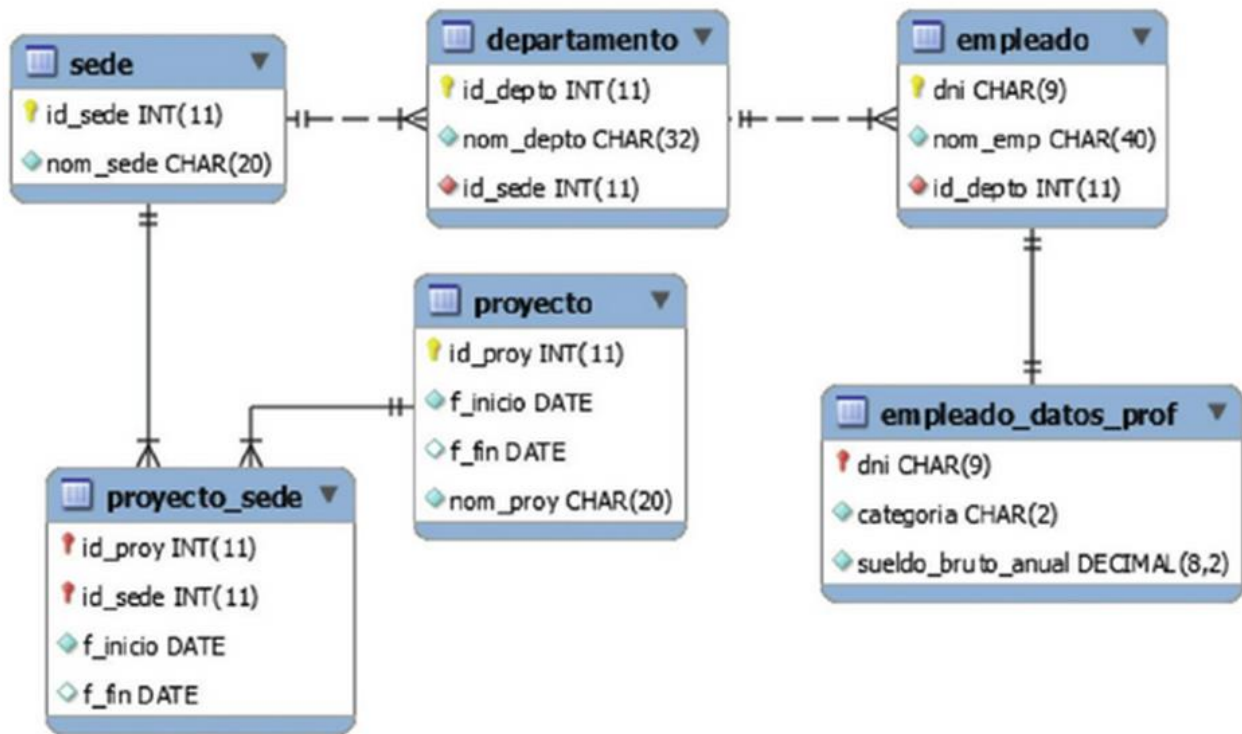
Proyecto Ejemplo 2: BD con varias tablas

1.	DEMO 3.2. BASE DE DATOS CON VARIAS TABLAS	2
1.1.	Crear base de datos y tablas en MySQL Workbench	2
1.2.	Crear o copiar proyecto Java Maven.....	3
1.3.	hibernate.cfg.xml.....	4
1.4.	Fichero hibernate.reveng.xml (Reverse Engineering)	7
1.5.	Mapeo de tablas.....	8
1.6.	Modelado de relaciones en los archivos de mapeo de Hibernate y con anotaciones JPA...	16
1.7.	Clase HibernateUtil.java.....	26
1.8.	Programa principal	27

1. DEMO 3.2. BASE DE DATOS CON VARIAS TABLAS

1.1. Crear base de datos y tablas en MySQL Workbench

Creamos una base de datos denominada **ut3demo2** y en ella las siguientes tablas:



```
create database ut3demo2;
```

```
use ut3demo2;
```

```
create table sede(
    id_sede integer auto_increment not null,
    nom_sede char(20) not null,
    primary key(id_sede)
);
```

```
create table departamento(
    id_depto integer auto_increment not null,
    nom_depto char(32) not null,
    id_sede integer not null,
    primary key(id_depto),
    foreign key fk_depto_sede(id_sede)
        references sede(id_sede)
        on delete cascade
);
```

```
create table empleado(  
    dni char(9) not null,  
    nom_emp char(40) not null,  
    id_depto integer not null,  
    primary key(dni),  
    foreign key fk_empleado_depto(id_depto)  
        references departamento(id_depto)  
        on delete cascade  
);  
  
create table empleado_datos_prof(  
    dni char(9) not null,  
    categoria char(2) not null,  
    sueldo_bruto_anual decimal(8,2),  
    primary key(dni),  
    foreign key fk_empleado_datosprof_empl(dni)  
        references empleado(dni)  
        on delete cascade  
);  
  
create table proyecto (  
    id_proy integer auto_increment not null,  
    f_inicio date not null,  
    f_fin date,  
    nom_proy char(20) not null,  
    primary key(id_proy)  
);  
  
create table proyecto_sede (  
    id_proy integer not null,  
    id_sede integer not null,  
    f_inicio date not null,  
    f_fin date,  
    primary key(id_proy, id_sede),  
    foreign key fk_proysede_proy (id_proy)  
        references proyecto(id_proy)  
        on delete cascade,  
    foreign key fk_proysede_sede (id_sede)  
        references sede(id_sede)  
        on delete cascade  
);
```

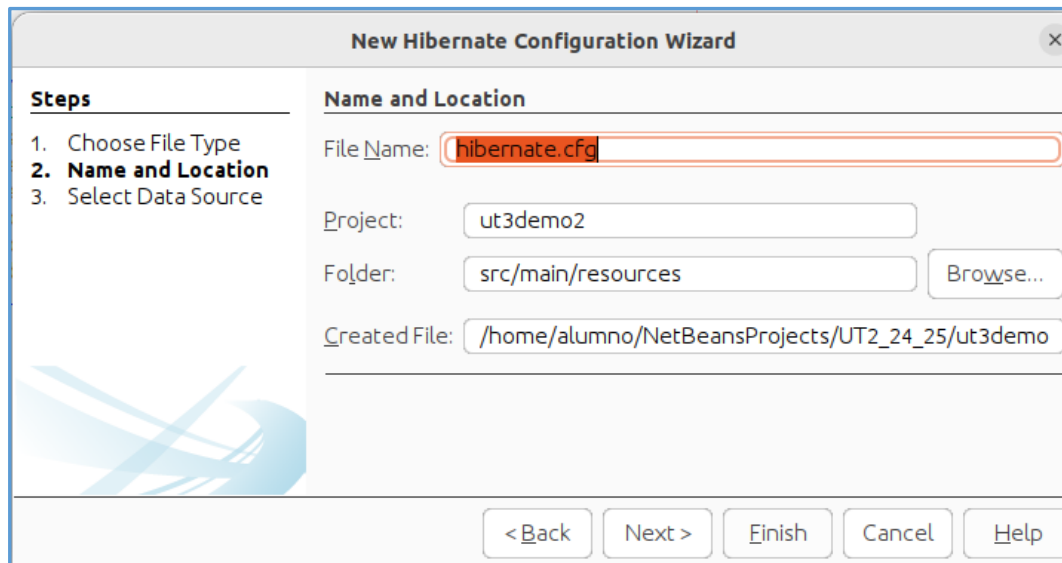
1.2. Crear o copiar proyecto Java Maven

Vamos a crear un nuevo proyecto Maven de nombre ut3demo2. También podemos hacer una copia de ut3demo1 y realizar las modificaciones necesarias, pero en este caso tendremos que borrar los ficheros de configuración, ingeniería inversa y los ficheros de mapeo (.hbm.xml y .java), para poder crear los nuevos. También tendremos errores en algunos de los ficheros que habrá que limpiar.

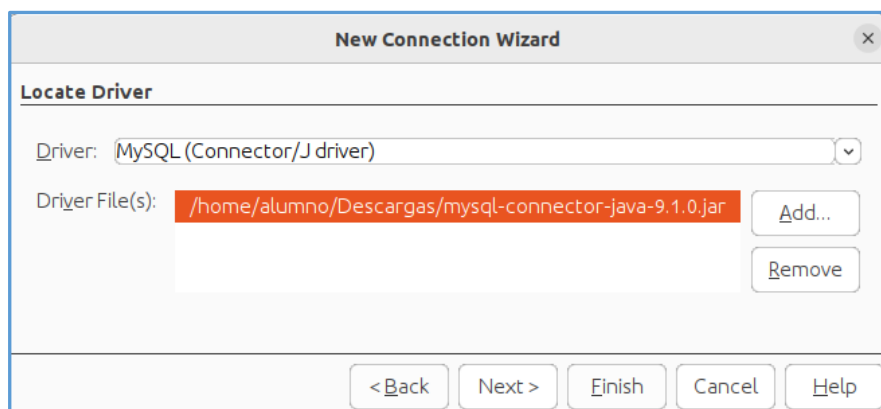
1.3. hibernate.cfg.xml

Una vez preparado el proyecto, creamos el fichero XML que contendrá los datos de conexión con la base de datos MySQL ut3demo2 creada al inicio de este apartado.

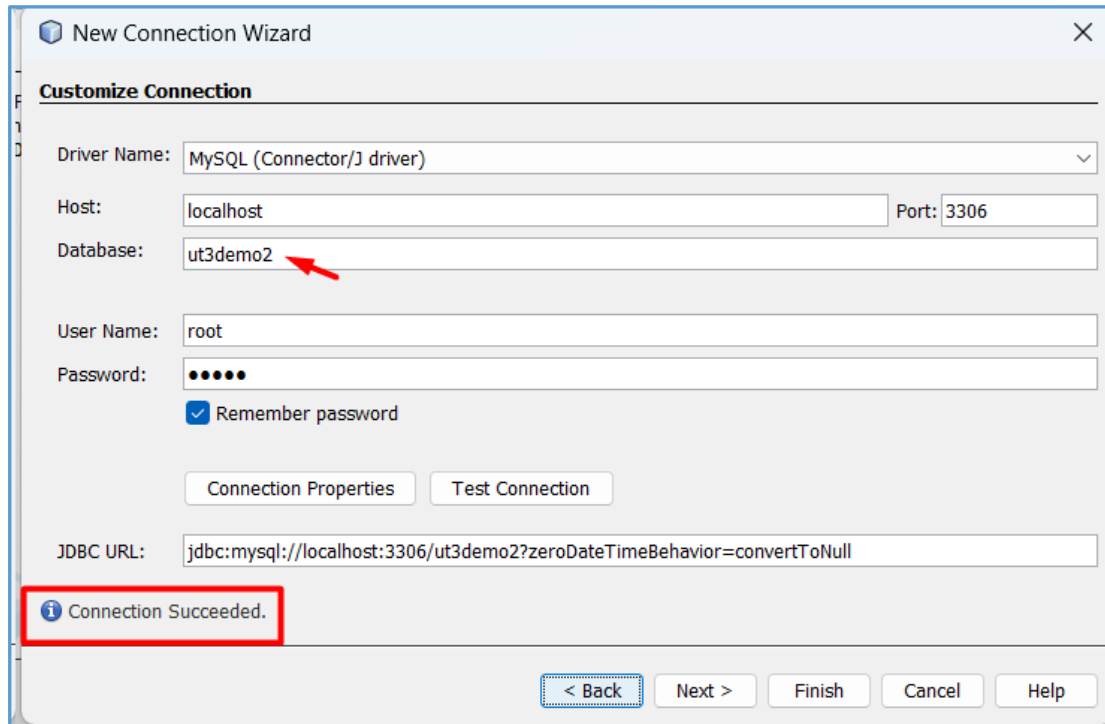
1. Hacemos clic derecho en la vista de Projects de NetBeans sobre Source Packages, New, Other.
2. Elegimos en Categories la carpeta Hibernate y a la derecha *Hibernate Configuration Wizard*.
3. Pulsamos **Next**.
4. El nombre por defecto es *hibernate.cfg*. Sin cambiar nada se pulsa de nuevo en **Next**.



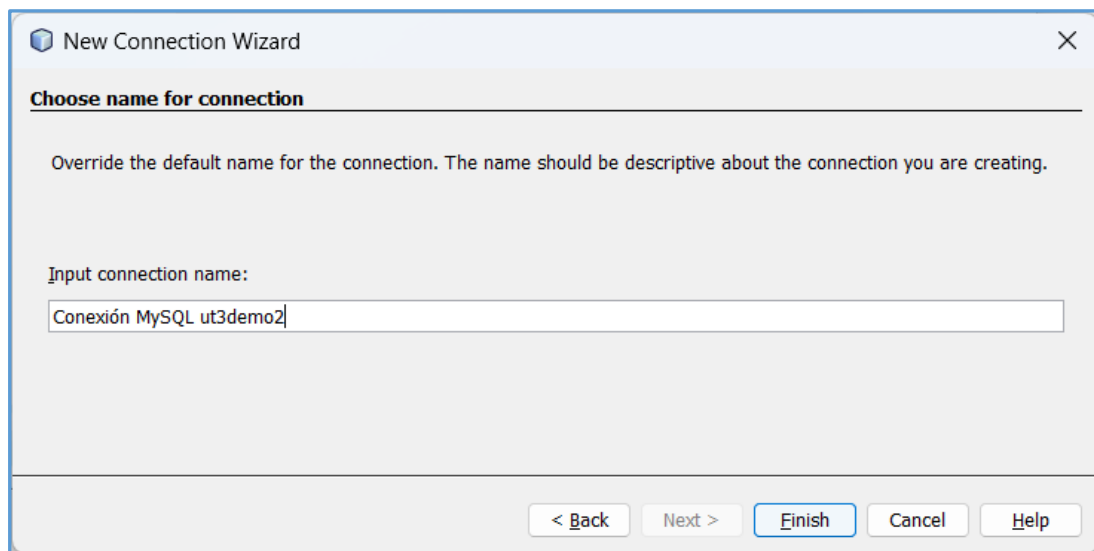
5. Pinchamos en la lista desplegable 'Database Connection' y seleccionamos *New Database Connection*....
6. En la ventana 'Locate Driver' seleccionamos el driver creado en la demo anterior o añadimos un jar nuevo con la **versión 9.1.0 del conector** de MySQL:



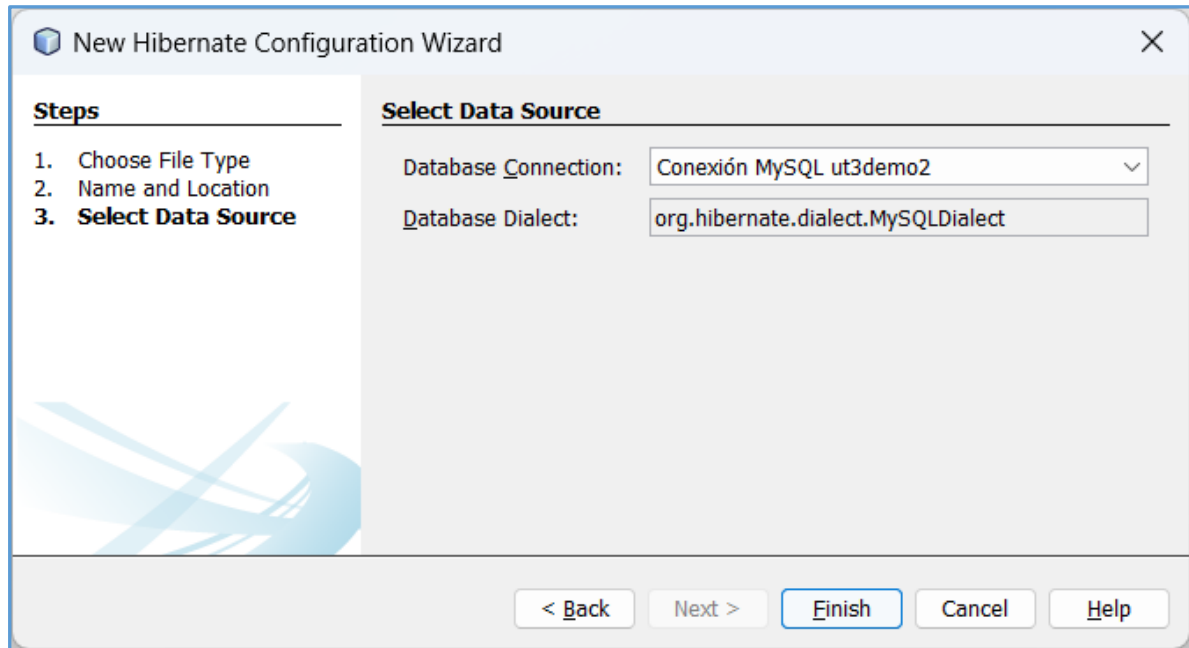
7. Pulsamos Next.
8. Aparece la ventana 'Customize Connection', con el driver elegido y los datos de conexión a nuestra base de datos. Rellenamos los campos, pulsamos **Test Connection** y si todo ha ido OK pulsamos Next:



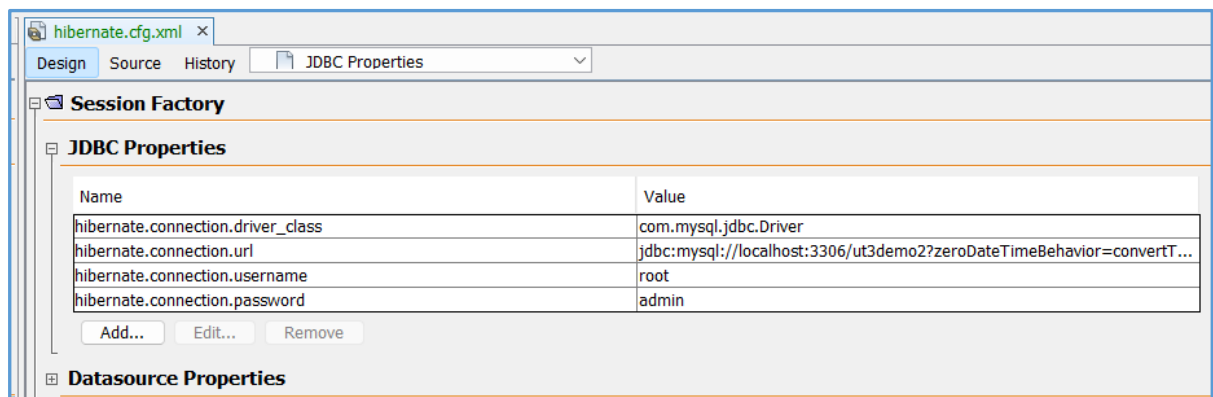
9. Pulsamos Next.
10. En la siguiente ventana (Choose Database Schema) no hace falta tocar nada. Simplemente pulsamos Next.
11. Cambiamos el 'Input connection name' por algo más intuitivo (por ejemplo, *Conexión MySQL ut3demo2*) y pulsamos Finish.



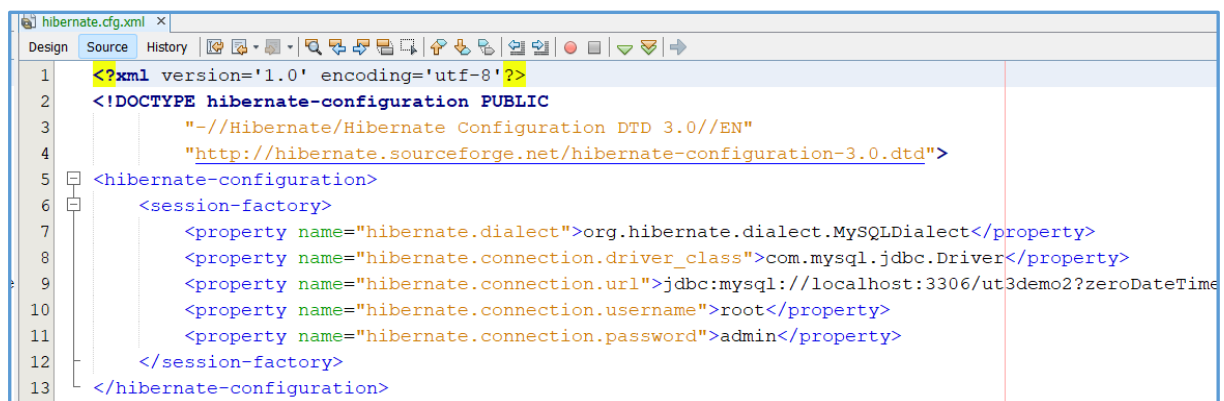
12. Si nos pide una Master Password, elegimos admin por ejemplo para que sea fácil de recordar y pulsamos ok.
13. Aparecerá de nuevo la ventana 'Select Data Source' con la nueva conexión creada (Conexión MySQL ut3demo2) seleccionada y el dialecto con el que nos comunicaremos con la base de datos:



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



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



16. Añadimos al fichero *hibernate.cfg.xml* 2 propiedades más que nos permitirán ver qué sentencias SQL está generando Hibernate por nosotros:

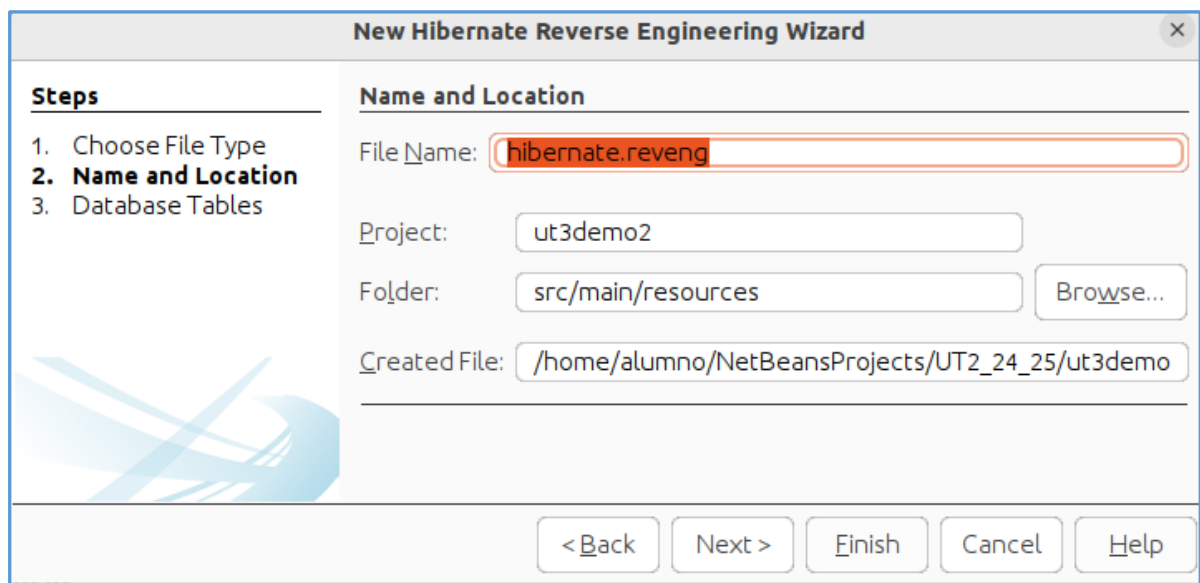
```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ut3demo2?zeroDat
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">admin</property>
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

1.4. Fichero hibernate.reveng.xml (Reverse Engineering)

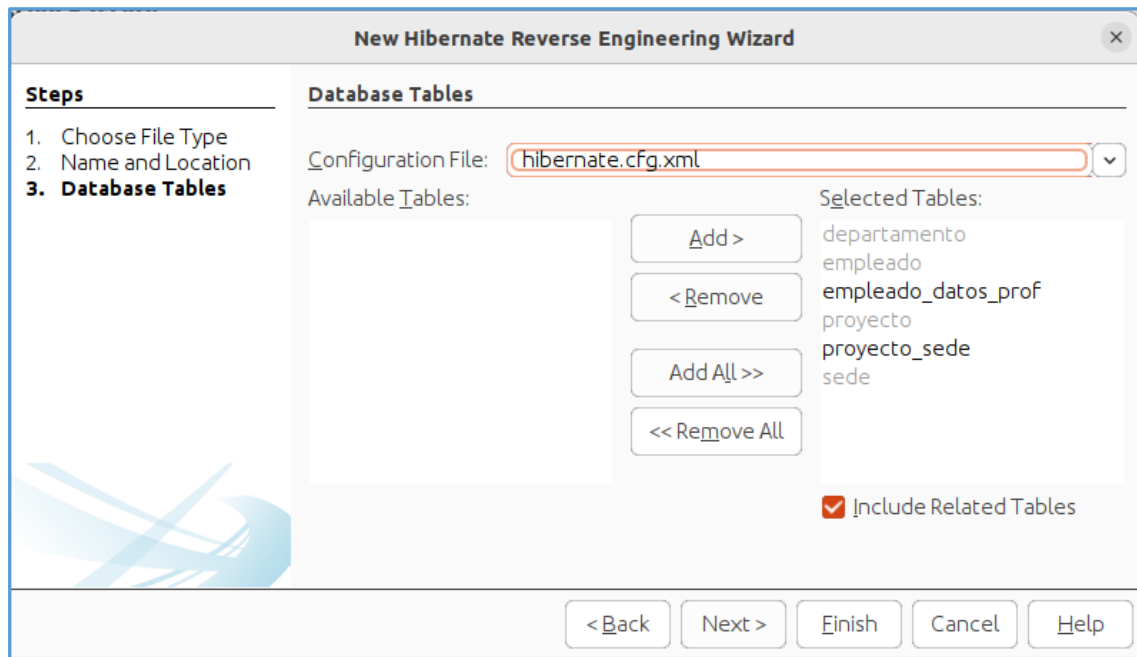
Este archivo indica el esquema (ROOT) en el que se encontrarán las tablas a mapear y el nombre de las mismas.

Para crearlo se siguen los siguientes pasos, ya vistos en la demo anterior:

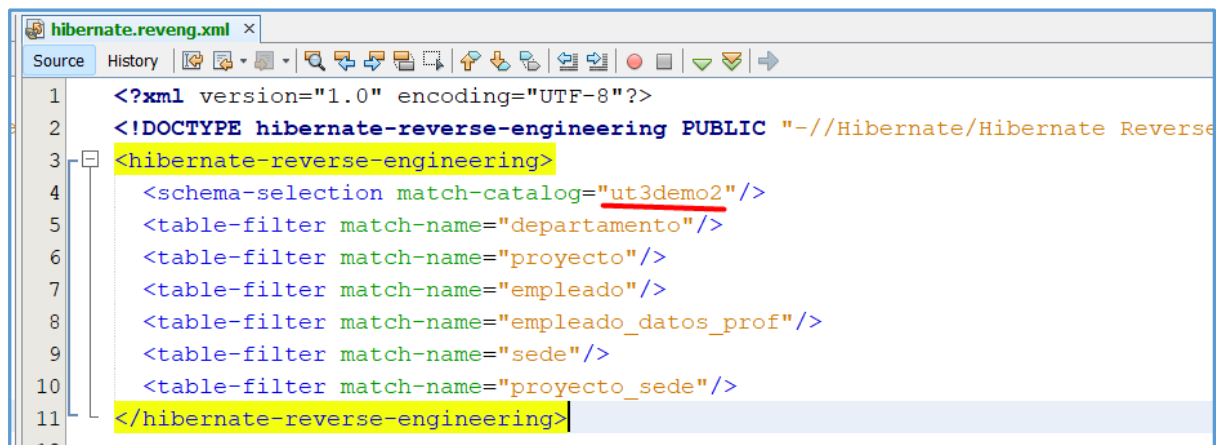
1. Seleccionando el paquete *default package* se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y, se selecciona dentro de las posibilidades (otros...), un tipo de archivo **Hibernate Reverse Engineering Wizard** dentro de la carpeta Hibernate.
2. Se pulsa Next.
3. El nombre por defecto del fichero será hibernate.reveng. Sin cambiar nada se pulsa en Next:



4. En la ventana 'Database Tables', deberían habernos aparecido todas las tablas que creamos al inicio en MySQL Workbench. Si es así, se seleccionan todas las tablas (o las que nos interesen) y pinchamos en **Add >** o directamente sin seleccionar pinchamos **Add All >>** para añadirlas al campo 'Selected Tables':



- Una vez seleccionadas pulsamos Finish. Con estos pasos se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero **hibernate.reveng.xml** con el nombre del esquema/base de datos (ut3demo2) y de las tablas que se desea mapear:



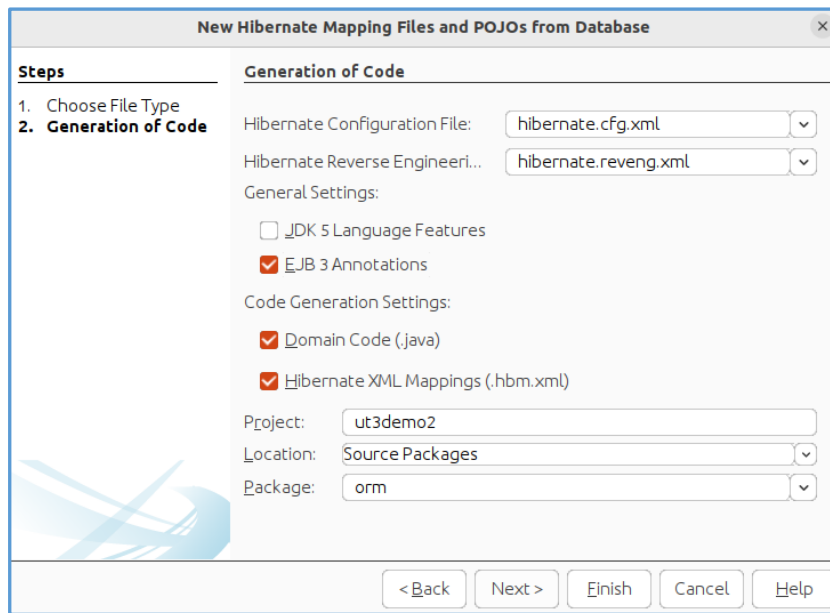
1.5. Mapeo de tablas

Una vez creados los ficheros hibernate.cfg.xml y hibernate.reveng.xml, procederemos a crear los ficheros necesarios para relacionar nuestros objetos Java con las tablas de la base de datos.

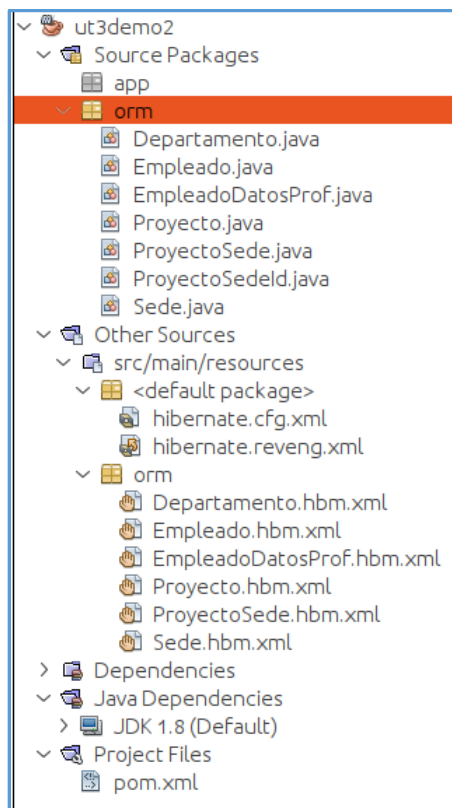
Pasos a seguir, ya vistos en la demo anterior:

- Seleccionando el nombre del proyecto en la vista Projects, pulsamos en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) > Others... > **Hibernate mapping files and POJO from Database** dentro de la carpeta Hibernate.
- Se pulsa Next.
- En la ventana que aparece se puede observar como ya salen seleccionados los ficheros de configuración creados en los pasos anteriores, que son los que necesita Hibernate para crear las clases Java (POJO, *Plain Old Java Object*) a partir de las tablas y los ficheros de mapeo entre las clases y las tablas. Para terminar de configurar esta parte solo es necesario indicar el

nombre de un paquete en el que almacenar los nuevos ficheros que se van a crear. Para nosotros **orm** (de *Object Relational Mapping*) y marcar (opcional) el checkbox 'EJB 3 Annotations':



4. Pulsamos Finish.
5. Con estos pasos se habrán creado en el paquete **orm** dos tipos de ficheros: las clases Java (POJOs) obtenidas a partir de las tablas de la base de datos (Departamento.java, Empleado.java,...) y los ficheros de mapeo entre las tablas y las clases (Departamento.hbm.xml, Empleado.hbm.xml).



6. A continuación, se muestra el código de los 2 ficheros creados por NetBeans para la tabla "proyecto": Proyecto. hbm.xml y Proyecto. java.

Proyecto.hbm.xml:

```

1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4  <!-- Generated 18-dic-2023 11:52:56 by Hibernate Tools 4.3.1 -->
5  <hibernate-mapping>
6      <class name="orm.Proyecto" table="proyecto" catalog="ut3demo2" optimistic-lock="version">
7          <id name="idProy" type="java.lang.Integer">
8              <column name="id_proy" />
9              <generator class="identity" />
10         </id>
11         <property name="FInicio" type="date">
12             <column name="f_inicio" length="10" not-null="true" />
13         </property>
14         <property name="FFin" type="date">
15             <column name="f_fin" length="10" />
16         </property>
17         <property name="nomProy" type="string">
18             <column name="nom_proy" length="20" not-null="true" />
19         </property>
20         <set name="proyectoSedes" table="proyecto_sede" inverse="true" lazy="true" fetch="select">
21             <key>
22                 <column name="id_proy" not-null="true" />
23             </key>
24             <one-to-many class="orm.ProyectoSede" />
25         </set>
26     </class>
27 </hibernate-mapping>

```

Ya conocíamos las etiquetas class, id y property:

- **class:** engloba la clase con sus atributos, indicando siempre el mapeo a la tabla de la base de datos. En **name** se indica el nombre de la clase (orm.Proyecto), en **table** el nombre de la tabla a la que representa este objeto (proyecto), y en **catalog** se indica el nombre de la base de datos (ut3demo2).
- Etiqueta **id** (dentro de class): indica en **name** el campo que representa el atributo clave en la clase (id_proy), en **column** su nombre en la tabla (id_proy) y en **type** el tipo de datos (Integer). En id tenemos la propiedad **<generator class="identity" />** que especifica que la clave primaria será generada automáticamente por la base de datos (se declaró como AUTO_INCREMENT al crear la tabla).
- El resto de atributos se indican en las etiquetas **property** asociando el nombre del campo de la clase con el nombre de la columna de la tabla y el tipo de datos.

Además, ahora tenemos otros elementos en el archivo que sirven para **representar la relación de uno-a-muchos** entre la tabla "proyecto" y la tabla "proyecto_sede":

```

    <set name="proyectoSedes" table="proyecto_sede"
        inverse="true" lazy="true" fetch="select">
        <key>
            <column name="id_proy" not-null="true" />
        </key>
        <one-to-many class="orm.ProyectoSede" />
    </set>

```

- Etiqueta **set:** se utiliza para implementar relaciones entre tablas, mediante el mapeo de colecciones. Con la etiqueta set se genera un nuevo atributo para la clase que es una colección tipo Set. Dentro de la etiqueta set se definen varios atributos, entre los que destacamos:

- En **name** se indica el nombre del atributo generado en la clase: en nuestro caso se genera el atributo proyectoSedes en la clase Proyecto.
- En **table** se indica el nombre de la tabla de donde se tomará la colección. En nuestro ejemplo "table=proyecto_sede", es decir, la relación se mapea con la tabla proyecto_sede.
- **inverse="true"** indica que esta es la parte inversa de la relación; es decir, el mantenimiento de la relación se realiza en el otro lado, en la clase ProyectoSede, que es donde está la clave ajena.
- **lazy="true"**: indica que la carga de la colección es perezosa (lazy loading), lo que significa que los elementos se cargarán bajo demanda, cuando se necesite acceder a ellos.
- **fetch="select"**: Especifica que la colección se recuperará mediante una consulta SELECT separada.
- En el elemento **key** se define el nombre de la columna identificadora en la asociación, es decir, la columna en la tabla proyecto_sede que referencia la clave primaria de Proyecto (id_proy), y se marca como no nula.
- El elemento **one-to-many** define la relación, en este caso es una asociación uno-a-muchos. Con el atributo **class** se indica de qué tipo son los elementos de la colección (orm.ProyectoSede). Es decir, un proyecto puede tener muchos "ProyectoSede".

Resumiendo, este mapeo indica que la clase Proyecto.java tiene un atributo de nombre "proyectoSedes" que es una lista de instancias de la clase "orm.ProyectoSede".

Nota: en el otro lado de la relación, es decir en la clase ProyectoSede, encontraremos una etiqueta "many-to-one" que representa la relación entre ProyectoSede y Proyecto.

Archivo Proyecto.java

```
Proyecto.java x
Source History
21 @Entity
22 @Table(name="proyecto"
23       ,catalog="ut3demo2"
24       )
25 public class Proyecto implements java.io.Serializable {
26
27     private Integer idProy;
28     private Date FInicio;
29     private Date FFin;
30     private String nomProy;
31     private Set proyectoSedes = new HashSet(0);
32
33     public Proyecto() {
34     }
35
36     public Proyecto(Date FInicio, String nomProy) {
37         this.FInicio = FInicio;
38         this.nomProy = nomProy;
39     }
40     public Proyecto(Date FInicio, Date FFin, String nomProy, Set proyectoSedes) {
41         this.FInicio = FInicio;
42         this.FFin = FFin;
43         this.nomProy = nomProy;
44         this.proyectoSedes = proyectoSedes;
45     }
46
47     @Id
48     @GeneratedValue(strategy=IDENTITY)
49     @Column(name="id_proy", unique=true, nullable=false)
50     public Integer getIdProy() {
51         return this.idProy;
52     }
}
```

```
55 public void setIdProy(Integer idProy) {
56     this.idProy = idProy;
57 }
58
59 @Temporal(TemporalType.DATE)
60 @Column(name="f_inicio", nullable=false, length=10)
61 public Date getFInicio() {
62     return this.FInicio;
63 }
64
65 public void setFInicio(Date FInicio) {
66     this.FInicio = FInicio;
67 }
68
69 @Temporal(TemporalType.DATE)
70 @Column(name="f_fin", length=10)
71 public Date getFFin() {
72     return this.FFin;
73 }
74
75 public void setFFin(Date FFin) {
76     this.FFin = FFin;
77 }
78
79 @Column(name="nom_proy", nullable=false, length=20)
80 public String getNomProy() {
81     return this.nomProy;
82 }
83
84 public void setNomProy(String nomProy) {
85     this.nomProy = nomProy;
86 }
87
88 @OneToMany(fetch=FetchType.LAZY, mappedBy="proyecto")
89 public Set getProyectoSedes() {
90     return this.proyectoSedes;
91 }
92
93 public void setProyectoSedes(Set proyectoSedes) {
94     this.proyectoSedes = proyectoSedes;
95 }
```

Observamos que se ha creado el atributo **proyectoSedes** (línea 31) como una colección del tipo Set, con una implementación específica HashSet. Set es una interfaz que representa una colección que no permite elementos duplicados, y HashSet es una implementación concreta de la interfaz Set. El valor 0 define la capacidad inicial del HashSet, que puede crecer dinámicamente a medida que se añaden elementos (aunque no es necesario porque HashSet no usa realmente el parámetro 0 para optimización en este caso).

IMPORTANTE: Hibernate genera el Set sin tipo genérico. Esto implica que cuando Hibernate analiza la clase Proyecto, encuentra la anotación @OneToMany en el getter del atributo ProyectoSedes, pero al ser un Set sin tipo genérico, no puede determinar qué entidad representa. Hibernate necesita esta información para gestionar la relación correctamente. De lo contrario se generan errores de compilación. En nuestro proyecto esto no ocurre porque además de las anotaciones tenemos los archivos de mapeo .hbm.xml, de donde Hibernate puede obtener la información. De lo contrario, tendríamos que modificar los POJOs donde se Hibernate genere atributos Set para incluir los tipos genéricos, tanto en la declaración del atributo como en los getter y setter.

Por ejemplo, en el archivo Proyecto.java habría que incluir las siguientes modificaciones para especificar el tipo genérico <ProyectoSede>:

- La línea 3, debería ser:

```
private Set<ProyectoSede> proyectoSedes = new HashSet<>();
```

- La línea 88, debería ser:

```
public Set<ProyectoSede> getProyectoSedes()
```

- La línea 92, debería ser:

```
public void setProyectoSedes(Set<ProyectoSede> proyectoSedes)
```

Veamos el significado de las anotaciones que aparecen en el archivo:

LINEAS 21 Y 22:

- **@Entity:** declara que esta clase es una entidad de persistencia, y está mapeada a una tabla en la base de datos.
- **@Table(name="proyecto", catalog="ut3demo2"):** indica que la entidad se corresponde con la tabla "proyecto" en el catálogo (o base de datos) "ut3demo2".

LINEAS 47 A 49

- **@Id:** identifica el atributo idProy como clave primaria.
- **@GeneratedValue(strategy=IDENTITY):** especifica que la clave primaria es autogenerada por la base de datos. En bases como MySQL, esta configuración corresponde a columnas AUTO_INCREMENT.
- **@Column (name="id_proy", unique=true, nullable=false):** define que esta propiedad se mapea a la columna id_proy. Es obligatorio ((nullable=false) y los valores deben ser únicos en toda la tabla.

LINEAS 58-59 Y 68-69

- **@Temporal(TemporalType.DATE):** especifica que la propiedad es una fecha (Date en Java) y se debe almacenar como un tipo de datos fecha en la base de datos.
- **@Column (name="f_inicio", nullable=false, length=10):** f_inicio es obligatorio (nullable=false)
- **@Column(name="f_fin", length=10):** f_fin no es obligatorio (nullable por defecto es true).

LINEA 78

- **@Column(name="nom_proy", nullable=false, length=20):** nom_proy es obligatorio y tiene un límite de 20 caracteres en la base de datos.

LINEAS 87 A 90:

```
@OneToMany(fetch=FetchType.LAZY, mappedBy="proyecto")
public Set getProyectoSedes() {
    return this.proyectoSedes;
}
```

- **La anotación se aplica al getter getProyectoSedes().** En Hibernate se pueden usar anotaciones JPA en los métodos getter. Hibernate considera la anotación al acceder o manipular datos mediante el método.
 - **@OneToMany:** declara que hay una relación uno-a-muchos entre la entidad Proyecto y la otra entidad (ProyectoSede).
 - **fetch=FetchType.LAZY:** especifica la estrategia de carga "perezosa". Cuando un "Proyecto" es consultado, los datos de proyectoSedes no se cargan automáticamente desde la base de datos. En su lugar, se cargan bajo demanda cuando se accede explícitamente al Set.
 - **mappedBy = "proyecto":** indica que la relación se gestiona en el otro lado (la entidad relacionada posee la propiedad "proyecto" que es la dueña de la relación).
7. Cabe destacar que en el fichero **hibernate.cfg.xml** se han añadido automáticamente varias líneas con la etiqueta 'mapping' y un atributo resource que apunta a los ficheros .hbm.xml generados:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3  <hibernate-configuration>
4  <session-factory>
5      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
6      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
7      <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ut3demo2?zeroDateTimeBehavior=convertToNull</property>
8      <property name="hibernate.connection.username">root</property>
9      <property name="hibernate.connection.password">admin</property>
10     <property name="show_sql">>true</property>
11     <property name="format_sql">>true</property>
12     <mapping resource="orm/ProyectoSede.hbm.xml"/>
13     <mapping resource="orm/Empleado.hbm.xml"/>
14     <mapping resource="orm/Departamento.hbm.xml"/>
15     <mapping resource="orm/Sede.hbm.xml"/>
16     <mapping resource="orm/EmpleadoDatosProf.hbm.xml"/>
17     <mapping resource="orm/Proyecto.hbm.xml"/>
18 </session-factory>
19 </hibernate-configuration>
20

```

- **EJERCICIO:** analizar el resto de archivos ".java" y ".hbm.xml" y observar y documentar cómo se han implementado todas las demás relaciones entre tablas.

1.6. Modelado de relaciones en los archivos de mapeo de Hibernate y con anotaciones JPA

En una relación entre entidades, la **parte propietaria** es la que define y controla la relación, normalmente la que contiene la columna de la clave foránea o, en el caso de una relación de clave compartida, es la que define cómo se genera el identificador.

La **parte inversa** (o no propietaria) es el lado que simplemente refleja la relación sin definir el mecanismo de persistencia (por ejemplo, no define la columna de la clave foránea).

Veamos cómo se identifican en cada tipo de asociación en los mapeos XML de Hibernate (entre paréntesis se muestran las equivalencias con anotaciones JPA):

1. Relación uno-a-uno: <one-to-one> (@OneToOne)

- Cada instancia de una entidad se asocia con exactamente una instancia de otra entidad.
- Puede definirse en ambos lados de la relación; uno de ellos se suele establecer como propietario (con la columna de clave foránea) y en el otro se marca la relación inversa.
- *Parte propietaria:* define cómo se genera o asocia la clave de la relación. En los archivos de mapeo de Hibernate esto se refleja mediante el uso de una etiqueta `<generator class="foreign">` dentro del elemento `<id>` y el atributo `constrained="true"` en la etiqueta `<one-to-one>` (o con `@JoinColumn` o similar en anotaciones).
- *Parte inversa:* es el lado que simplemente refleja la relación, sin definir el mecanismo de asociación o generación de clave. En el mapeo XML, se define la etiqueta `<one-to-one>` sin atributos como `constrained="true"` (o, en anotaciones, sin usar `@JoinColumn` o usando `mappedBy`), lo que indica que este lado es dependiente de la parte propietaria para la definición de la asociación.

- EJEMPLO: relación one-to-one entre Empleado y EmpleadoDatosProf.

Archivo de mapeo Empleado.hbm.xml

```

1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
   DTD 3.0//EN"
3 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <!-- Generated 27-ene-2025 20:44:55 by Hibernate Tools 4.3.1 -->
5 <hibernate-mapping>
6   <class name="orm.Empleado" table="empleado" catalog="ut3demo2"
       optimistic-lock="version">
7     <id name="dni" type="string">
8       <column name="dni" length="9" />
9       <generator class="assigned" />
10    </id>
11    <many-to-one name="departamento" class="orm.Departamento"
        fetch="select">
12      <column name="id_depto" not-null="true" />
13    </many-to-one>
14    <property name="nomEmp" type="string">
15      <column name="nom_emp" length="40" not-null="true" />
16    </property>
17    <one-to-one name="empleadoDatosProf"
        class="orm.EmpleadoDatosProf"></one-to-one>
18  </class>
19 </hibernate-mapping>

```

Archivo de mapeo EmpleadoDatosProf.hbm.xml:

```

1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
   DTD 3.0//EN"
3 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <!-- Generated 27-ene-2025 20:44:55 by Hibernate Tools 4.3.1 -->
5 <hibernate-mapping>
6   <class name="orm.EmpleadoDatosProf" table="empleado_datos_prof"
       catalog="ut3demo2" optimistic-lock="version">
7     <id name="dni" type="string">
8       <column name="dni" length="9" />
9       <generator class="foreign">
10        <param name="property">empleado</param>
11      </generator>
12    </id>
13    <one-to-one name="empleado" class="orm.Empleado"
        constrained="true"></one-to-one>
14    <property name="categoria" type="string">
15      <column name="categoria" length="2" not-null="true" />
16    </property>
17    <property name="sueldoBrutoAnual" type="big_decimal">
18      <column name="sueldo_bruto_anual" precision="8" />
19    </property>
20  </class>
21 </hibernate-mapping>

```

Archivo Empleado.java con anotaciones JPA

```
15  /* Empleado generated by hbm2java
16  */
17  @Entity
18  @Table(name="empleado"
19        ,catalog="ut3demo2"
20  )
21  public class Empleado implements java.io.Serializable {
22      private String dni;
23      private Departamento departamento;
24      private String nomEmp;
25      private EmpleadoDatosProf empleadoDatosProf;
26
27      public Empleado() {
28      }
29
30      public Empleado(String dni, Departamento departamento, String
31                      nomEmp) {
32          this.dni = dni;
33          this.departamento = departamento;
34          this.nomEmp = nomEmp;
35      }
36      public Empleado(String dni, Departamento departamento, String
37                      nomEmp, EmpleadoDatosProf empleadoDatosProf) {
38          this.dni = dni;
39          this.departamento = departamento;
40          this.nomEmp = nomEmp;
41          this.empleadoDatosProf = empleadoDatosProf;
42      }
43
44      @Id
45      @Column(name="dni", unique=true, nullable=false, length=9)
46      public String getDni() {
47          return this.dni;
48      }
49
50      public void setDni(String dni) {
51          this.dni = dni;
52      }
53  }
```

```
51
52     @ManyToOne(fetch=FetchType.LAZY)
53     @JoinColumn(name="id_depto", nullable=false)
54     public Departamento getDepartamento() {
55         return this.departamento;
56     }
57
58     public void setDepartamento(Departamento departamento) {
59         this.departamento = departamento;
60     }
61
62     @Column(name="nom_emp", nullable=false, length=40)
63     public String getNomEmp() {
64         return this.nomEmp;
65     }
66
67     public void setNomEmp(String nomEmp) {
68         this.nomEmp = nomEmp;
69     }
70
71     @OneToOne(fetch=FetchType.LAZY, mappedBy="empleado")
72     public EmpleadoDatosProf getEmpleadoDatosProf() {
73         return this.empleadoDatosProf;
74     }
75
76     public void setEmpleadoDatosProf(EmpleadoDatosProf
77         empleadoDatosProf) {
78         this.empleadoDatosProf = empleadoDatosProf;
79     }
```

Archivo EmpleadoDatosProf.java con anotaciones JPA

```
17 /**
18  * EmpleadoDatosProf generated by hbm2java
19  */
20 @Entity
21 @Table(name="empleado_datos_prof"
22       ,catalog="ut3demo2"
23 )
24 public class EmpleadoDatosProf implements java.io.Serializable {
25     private String dni;
26     private Empleado empleado;
27     private String categoria;
28     private BigDecimal sueldoBrutoAnual;
29
30     public EmpleadoDatosProf() {
31     }
32
33     public EmpleadoDatosProf(Empleado empleado, String categoria) {
34         this.empleado = empleado;
35         this.categoria = categoria;
36     }
37     public EmpleadoDatosProf(Empleado empleado, String categoria,
38                             BigDecimal sueldoBrutoAnual) {
39         this.empleado = empleado;
40         this.categoria = categoria;
41         this.sueldoBrutoAnual = sueldoBrutoAnual;
42     }
43     @GenericGenerator(name="generator", strategy="foreign",
44                     parameters=@Parameter(name="property",
45                                           value="empleado"))@Id
46     @GeneratedValue(generator="generator")
47     @Column(name="dni", unique=true, nullable=false, length=9)
48     public String getDni() {
49         return this.dni;
50     }
51
52     public void setDni(String dni) {
53         this.dni = dni;
54     }
55
56     @OneToOne(fetch=FetchType.LAZY)@PrimaryKeyJoinColumn
57     public Empleado getEmpleado() {
58         return this.empleado;
59     }
60
61     public void setEmpleado(Empleado empleado) {
62         this.empleado = empleado;
63     }
64 }
```

```

61
62     @Column(name="categoria", nullable=false, length=2)
63     public String getCategoria() {
64         return this.categoria;
65     }
66
67     public void setCategoria(String categoria) {
68         this.categoria = categoria;
69     }
70
71     @Column(name="sueldo_bruto_anual", precision=8)
72     public BigDecimal getSueldoBrutoAnual() {
73         return this.sueldoBrutoAnual;
74     }
75
76     public void setSueldoBrutoAnual(BigDecimal sueldoBrutoAnual) {
77         this.sueldoBrutoAnual = sueldoBrutoAnual;
78     }
79 }

```

- **EmpleadoDatosProf** es la parte propietaria:
 - En su archivo de mapeo XML se observa el uso de <generator class="foreign"> (línea 9) dentro del <id> para generar el identificador a partir de la propiedad empleado, y la declaración <one-to-one name="empleado" class="orm.Empleado" constrained="true"/> (línea 13).
 - En las anotaciones JPA se utiliza un generador definido con @GenericGenerator) (línea 43) con estrategia foreign, que toma el identificador de Empleado a través de la propiedad empleado y se emplea @PrimaryKeyJoinColumn en la asociación @OneToOne (línea 53), lo que indica que la relación se une a través de su clave primaria.
- **Empleado** es la parte inversa:
 - En su archivo de mapeo XML la declaración <one-to-one name="empleadoDatosProf" class="orm.EmpleadoDatosProf"/> (línea 17) (sin atributos como constrained="true") refleja la asociación definida en EmpleadoDatosProf.
 - En las anotaciones JPA se utiliza mappedBy="empleado" en la anotación @OneToOne (línea 71), lo que indica que la propiedad que controla la relación se encuentra en la entidad **EmpleadoDatosProf**, concretamente en su propiedad empleado).

Relaciones uno-a-uno con restricción de existencia

Existen otro tipo de relaciones uno-a-uno, que son las relaciones con restricción de existencia (una clase sólo tiene sentido dentro de otra clase). Para estas relaciones en JPA se utilizan las anotaciones @Embedded y @Embeddable, que permiten modelar la composición de objetos dentro de una entidad sin que la clase embebida tenga una identidad propia en la base de datos. Es decir, permiten reutilizar estructuras de datos dentro de múltiples entidades sin necesidad de crear una tabla separada.

@Embeddable

- Se usa en una clase para indicar que su instancia puede ser incrustada (embebida) dentro de una entidad.
- No tiene una identidad propia en la base de datos (no tiene una clave primaria).
- Sus atributos se almacenan en la misma tabla de la entidad que la contiene.

@Embedded

- Se usa en el atributo de una entidad para indicar que este es una instancia de una clase @Embeddable.
- Los atributos de la clase embebida se mapearán en la misma tabla que la entidad contenedora.

Ejemplo: clase Direccion embebida en clase Persona

```
@Embeddable
public class Direccion {
    private String calle;
    private String ciudad;
    private String codigoPostal;

    // Getters y setters
}
```

```
@Entity
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @Embedded
    private Direccion direccion; // Se almacena en la misma tabla de Persona

    // Getters y setters
}
```

@AttributeOverrides y @AttributeOverride

Si la entidad contiene más de un objeto embebido del mismo tipo, se pueden personalizar los nombres de las columnas en la base de datos usando @AttributeOverrides.

Ejemplo con dos direcciones en la misma entidad:

```

@Entity
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "calle", column = @Column(name = "facturacion_calle")),
        @AttributeOverride(name = "ciudad", column = @Column(name = "facturacion_ciudad")),
        @AttributeOverride(name = "codigoPostal", column = @Column(name = "facturacion_codigo_postal"))
    })
    private Direccion direccionFacturacion;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "calle", column = @Column(name = "envio_calle")),
        @AttributeOverride(name = "ciudad", column = @Column(name = "envio_ciudad")),
        @AttributeOverride(name = "codigoPostal", column = @Column(name = "envio_codigo_postal"))
    })
    private Direccion direccionEnvio;

    // Getters y setters
}

```

```

@Entity
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "calle", column = @Column(name = "facturacion_calle")),
        @AttributeOverride(name = "ciudad", column = @Column(name = "facturacion_ciudad")),
        @AttributeOverride(name = "codigoPostal", column = @Column(name = "facturacion_codigo_postal"))
    })
    private Direccion direccionFacturacion;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "calle", column = @Column(name = "envio_calle")),
        @AttributeOverride(name = "ciudad", column = @Column(name = "envio_ciudad")),
        @AttributeOverride(name = "codigoPostal", column = @Column(name = "envio_codigo_postal"))
    })
    private Direccion direccionEnvio;

    // Getters y setters
}

```

2. Relación uno-a-muchos / muchos-a-uno

La relación uno-a-muchos y muchos-a-uno son dos vistas de la misma asociación. La relación se define en dos lados. En el lado “muchos” se identifica con <many-to-one> (o @ManyToOne en anotaciones), y en el lado “uno” mediante <one-to-many> (o @OneToMany).

Estas relaciones se suelen implementar mediante un atributo en la entidad “uno” que es una colección de elementos de la entidad “muchos”.

Etiqueta <many-to-one> (@ManyToOne)

- Mapea una relación muchos a uno, en la que varias instancias de una entidad hacen referencia a una única instancia de otra entidad.
- Se utiliza en el lado “muchos” para indicar la propiedad que contiene la clave foránea que apunta a la entidad “uno”.
- Este lado es la parte propietaria, ya que es el que contiene la clave foránea que apunta a la entidad “uno”. En el mapeo, se usa <many-to-one> **sin** atributo “inverse”.

Etiqueta <one-to-many> (@OneToMany)

- Esta etiqueta se define en el lado “uno”. Representa la relación inversa de “muchos a uno”: una única instancia de una entidad (lado “uno”) está asociada a una colección de instancias de otra entidad (lado “muchos”). En la clase del lado “uno” se define un atributo tipo colección (por ejemplo Set) de elementos de la clase del lado “muchos”.
- En XML se utiliza la etiqueta <one-to-many> dentro de un elemento <set> (u otro tipo de colección) con el atributo inverse="true" (en anotaciones JPA se usa @OneToMany junto con el atributo mappedBy para indicar cuál es el campo en la entidad “muchos” que posee la relación).

➤ EJEMPLO: Relación entre Departamento (lado “uno”) y Empleados (lado “muchos”).

Veamos cómo se modela la relación muchos-a-uno en Empleado.hbm.xml

Archivo Empleado.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 27-ene-2025 20:44:55 by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
  <class name="orm.Empleado" table="empleado" catalog="ut3demo2" optimistic-
    lock="version">
    <id name="dni" type="string">
      <column name="dni" length="9" />
      <generator class="assigned" />
    </id>
    <many-to-one name="departamento" class="orm.Departamento" fetch="select">
      <column name="id_depto" not-null="true" />
    </many-to-one>
    <property name="nomEmp" type="string">
      <column name="nom_emp" length="40" not-null="true" />
    </property>
    <one-to-one name="empleadoDatosProf" class="orm.EmpleadoDatosProf"></one-to-one>
  </class>
</hibernate-mapping>
```


Elemento <many-to-one>: representa una relación unidireccional muchos-a-uno entre dos clases Java (en este caso, muchos empleados pertenecen a un departamento). En esta relación, la tabla de la entidad "muchos" (empleado) almacena una clave foránea que apunta a la clave primaria de la entidad "uno" (departamento).

- **Atributos principales:**

- **name:** Define el nombre del atributo en la clase Java que representa la relación.
- **class:** Especifica la clase a la que se refiere la relación. En este caso, departamento es una instancia de `orm.Departamento`.
- **fetch:** Define la estrategia de carga de la relación:
 - **select (por defecto en Hibernate):** Ejecuta una consulta independiente solo cuando se accede a la propiedad relacionada.
 - **join:** Usa un JOIN en la consulta inicial para cargar la relación de inmediato, lo que puede mejorar el rendimiento en algunos casos.

Nota: En JPA, el equivalente de <many-to-one> es `@ManyToOne`, y el valor por defecto de fetch es `FetchType.EAGER`, lo que significa que la relación se carga de inmediato.

Comportamiento de fetch y lazy

El atributo fetch se usa junto con lazy para definir cómo y cuándo se cargan las relaciones en Hibernate.

Hibernate hbm.xml	Anotaciones JPA	Comportamiento
fetch="select" lazy="true" (por defecto)	fetch=FetchType.LAZY	Ejecuta una consulta adicional cuando se accede al objeto referenciado, con carga "perezosa"
fetch="select" lazy="false"	fetch= FetchType.EAGER	Hibernate carga la relación de inmediato, pero sigue usando consultas adicionales (una para la entidad principal y otra para la relacionada).
fetch="join"	FetchType.EAGER	Usa un JOIN en la consulta inicial y carga la relación inmediatamente.

Importante: En Hibernate, fetch="join" **implica** lazy="false", por lo que no es necesario especificarlo.

Elemento <column> dentro de <many-to-one>

- Define la clave foránea en la base de datos que establece la relación con la entidad "uno".
- El atributo not-null="true" indica que esta clave no puede ser nula, evitando que haya empleados sin departamento asignado.
- **Nota:** Aunque Hibernate gestiona esta restricción, es recomendable asegurarse de que la base de datos también la imponga con NOT NULL.

3. Relación muchos-a-muchos: <many-to-many> (@ManyToMany)

- Es una relación en la que ambas entidades pueden estar asociadas a múltiples instancias de la otra.
 - Generalmente, para gestionar la relación se utiliza una tabla intermedia (tabla de unión) en la que se definen las columnas que actúan como claves foráneas de ambas entidades.
- **EJEMPLO:** en el proyecto ut3demo2 la base de datos contiene la tabla ProyectoSede para modelar la relación muchos-a-muchos entre Proyecto y Sede. En caso de no existir dicha tabla, Hibernate puede generar esta entidad intermedia para modelar la relación muchos-a-muchos.

4. Etiquetas para relaciones basadas en colecciones (uno-a-muchos y muchos-a-muchos)

Las etiquetas utilizadas para mapear colecciones permiten definir cómo se almacenan y se relacionan los conjuntos de elementos en la base de datos. Por ejemplo, al mapear una colección sin elementos duplicados se puede usar la etiqueta <set>. Dentro de esta etiqueta se suelen incluir:

- Un elemento <key>, que especifica la columna de la tabla que actuará como clave foránea.
- Un elemento que indica el tipo de relación para cada elemento de la colección, como <one-to-many> o <many-to-many>, según corresponda.

1.7. Clase HibernateUtil.java

Este fichero se utiliza para gestionar las conexiones (sesiones en Hibernate) que se hacen a las bases de datos y que permitirán mapear los objetos en las tablas correspondientes. Podríamos en este caso crear el fichero a través de NetBeans (New > Other > Hibernate > HibernateUtil) o reutilizar el archivo ya creado en la demo anterior.

Fichero HibernateUtil.java resultante:

```

package app;

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

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

    private static final SessionFactory sessionFactory;

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

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

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

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

    //Método abrirSession
    public static Session abrirSession() {
        try {
            return sessionFactory.openSession();
        } catch (HibernateException he) {
            System.err.println("Error al abrir Session: " + he.getMessage());
            return null;
        }
    } //Fin método abrirSession
}

```

1.8. Programa principal

A continuación, vamos a crear una sede, un departamento de esa sede y un empleado de ese departamento. Todas las operaciones se realizan en una sesión y, dentro de ella, en una transacción.

La clase **Principal.java** queda de la siguiente manera:

```
package app;

import java.util.logging.Level;
import java.util.logging.Logger;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import orm.Departamento;
import orm.Empleado;
import orm.Sede;

public class Principal {
    public static void main(String[] args) {

        // Para que no salgan los mensajes de INFO al ejecutar el programa
        Logger.getLogger("org.hibernate").setLevel(Level.SEVERE);

        SessionFactory sf = null;
        Session s = null;
        Transaction t = null;

        try {
            // Creamos la factoría de sesiones
            sf = HibernateUtil.getSessionFactory();
            // Creamos la sesión (conexión) con la base de datos
            s = HibernateUtil.abrirSesion();

            // Creamos una transacción (finaliza cuando hacemos commit())
            t = s.beginTransaction();

            // Creamos sede, departamento y empleado
            Sede sede1 = new Sede("MÁLAGA");
            s.save(sede1);

            Departamento dept = new Departamento(sede1, "I+D");
            s.save(dept);

            Empleado e = new Empleado("11222333i", dept, "SAMPER");
            s.save(e);

            // Hacemos commit
            t.commit();
        } catch (HibernateException he) {
            System.err.println("Error Hibernate: " + he.getMessage());
            if (t != null) {
                t.rollback();
            }
        } finally {
            // Cerramos la sesión y la factoría de sesiones
            HibernateUtil.cerrarSesion(s);
            HibernateUtil.cerrarSessionFactory();
        }
    }
}
```

Ejecutamos y veremos que en nuestra base de datos se habrán creado los elementos deseados:

#	id_sede	nom_sede
1	1	MÁLAGA
*	NULL	NULL

#	id_depto	nom_depto	id_sede
1	1	I+D	1
*	NULL	NULL	NULL

#	dni	nom_emp	id_depto
1	11222333A	SAMPER	1
*	NULL	NULL	NULL

Ejercicio. Añadir la siguiente funcionalidad en el programa principal

a) sin usar HQL:

- Insertar una nueva sede en SEVILLA
- Insertar el departamento I+D en la sede de SEVILLA
- Insertar 3 empleados este departamento creado
- Mostrar el nombre de todos los empleados del departamento

b) Usando HQL (créate la clase OperacionesHQL igual que en el proyecto ut3demo1 parte 3):

- Mostrar el id, nombre y sede de todos los departamentos. Sobreescribe el método toString en la clase Departamento para que devuelva esta información. Ejemplo de salida:

```

----- LEAMOS TODOS LOS DEPARTAMENTOS -----
***** DATOS *****
Dept Id: 5      NOMBRE: I+D      SEDE: MÁLAGA
Dept Id: 12     NOMBRE: I+D      SEDE: SEVILLA
-----
BUILD SUCCESS
-----
Total time: 2.569s
Finished at: Mon Feb 03 23:44:53 CET 2025
Final Memory: 7M/153M
-----

```

Nuevos elementos que podemos/debemos incorporar en el programa:

1. **Método refresh()** de la interfaz Session: recarga un objeto desde la base de datos y descarta los cambios no guardados en memoria, es decir, se utiliza para actualizar un objeto que ha sido modificado en la base de datos. En este ejemplo necesitamos usarlo para para los objetos sede, (después de añadir un departamento a esa sede), y departamento (después de añadir empleados al departamento).

Posibles soluciones si refresh() no funciona correctamente

Si tras ejecutar el programa no se muestran los empleados aunque se use refresh(), puede deberse al uso de carga perezosa o diferida (lazy loading), ya que en este caso la colección no se cargará automáticamente. A continuación se muestran alternativas para solucionar este problema.

a) Forzar la carga de la colección con Hibernate.initialize()

Es posible que el uso de refresh() no solucione el problema del Lazy Loading en las colecciones, por lo que necesitas inicializarlas antes de cerrar la sesión. En el ejemplo, usaremos:

```
Hibernate.initialize(dept.getEmpleados());
```

Esto fuerza la carga de los empleados sin necesidad de refresh().

b) Cambiar el tipo de carga a "eager"

Podemos forzar a que Hibernate haga carga directa de todos los elementos de Departamento:

- En el archivo de mapeo Departamento.hbm.xml, modificando el valor del atributo lazy: cambiar lazy="true" por lazy="false" en la etiqueta <set ... >
- En el POJO Departamento.java modificando las anotaciones: cambiar la línea

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "departamento") por
```

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "departamento")
```

Esto hace que los empleados se carguen junto con el departamento, evitando problemas de carga diferida (lazy loading).

c) Cerrar y reabrir la sesión antes de mostrar los empleados

A veces, Hibernate no actualiza correctamente las relaciones dentro de la misma sesión. Puedes probar cerrando y abriendo una nueva sesión antes de iterar sobre dept.getEmpleados().

2. **Ajustar el nivel de log:** si no queremos que salgan los mensajes de INFO al ejecutar el programa podemos poner la instrucción:

```
Logger.getLogger("org.hibernate").setLevel(Level.SEVERE);
```

3. Propiedades en hibernate.cfg.xml para **ver las sentencias SQL generadas por Hibernate:**

- **hibernate.show_sql** es una propiedad útil para depuración que nos permite ver en consola la traducción a SQL de las sentencias Hibernate que ejecute nuestro programa. Para esto tenemos que añadir
- Para complementar la propiedad show_sql de Hibernate, podemos utilizar **hibernate.format_sql** e **hibernate.use_sql_comments**, que nos mostrarán la salida algo más legible

Para activar estas tres propiedades, añadimos estas líneas en nuestro archivo hibernate.cfg.xml:

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<property name="hibernate.use_sql_comments">true</property>
```