TEMA 3. HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL (ORM)

2º DAM

Mª del Mar Pérez Ropero

Contenidos	Objetivos
Concepto de mapeo objeto relacional. Herramientas ORM. Características. Instalación y configuración de una herramienta ORM. Estructura de un fichero de mapeo. Elementos, propiedades. Clases persistentes. Sesiones; estados de un objeto. Carga, almacenamiento y modificación de objetos. Consultas SQL. Lenguajes propios de la herramienta ORM.	Instalar una herramienta ORM. Configurar la herramienta ORM. Definir los ficheros de mapeo. Aplicar mecanismos de persistencia a los objetos. Desarrollar aplicaciones que modifican y recuperan objetos persistentes. Desarrollar aplicaciones que realizan consultas usando el lenguaje OQL. Valorar la utilización de lenguajes propios de la herramienta ORM. Gestionar transacciones. Crear diagramas ORM que presentan el mapeo entre clases persistentes y entidades.

1. Concepto de Mapeo objeto-relacional.

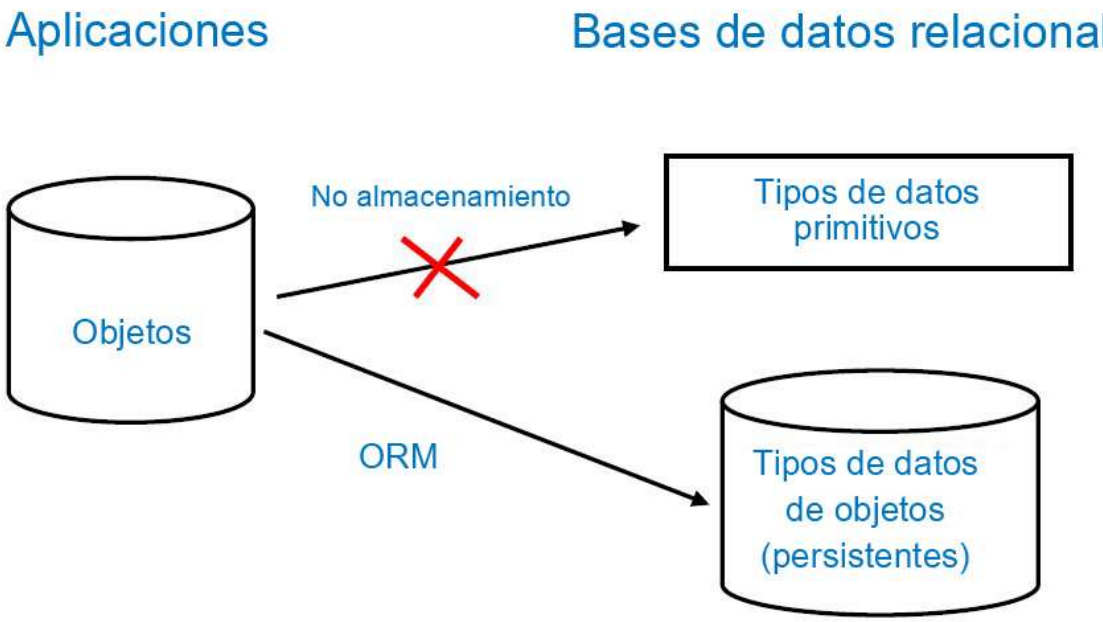
A la hora de almacenar los datos de un programa orientado a objetos en una base de datos relacional, surge un inconveniente debido a incompatibilidad de sistemas de tipos de datos. En el software orientado a objetos, la información se representa como clases y objetos. En las bases de datos relacionales, como tablas y sus restricciones. Por tanto, para almacenar la información tratada en un programa orientado a objetos en una base de datos relacional es necesaria una traducción entre ambas formas.

El mapeo objeto-relacional (ORM) soluciona este problema. Es una técnica de programación que se utiliza con el propósito de convertir datos entre el utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, gracias a la persistencia. Esto posibilita el uso en las bases de datos relacionales de las características propias de la programación orientada a objetos (básicamente herencia y polimorfismo).

La mayoría de las aplicaciones se construyen usando técnicas de programación orientada a objetos; sin embargo, los sistemas de bases de datos más extendidos son de tipo relacional.

Las bases de datos más extendidas son del tipo relacional y estas sólo permiten guardar tipos de datos primitivos (enteros, cadenas de texto, etc.) por lo que no se puede guardar de forma directa los objetos de la aplicación en las tablas. Por tanto, se debe convertir los valores de los objetos en valores simples que puedan ser almacenados en una base de datos (y poder recuperarlos más tarde).

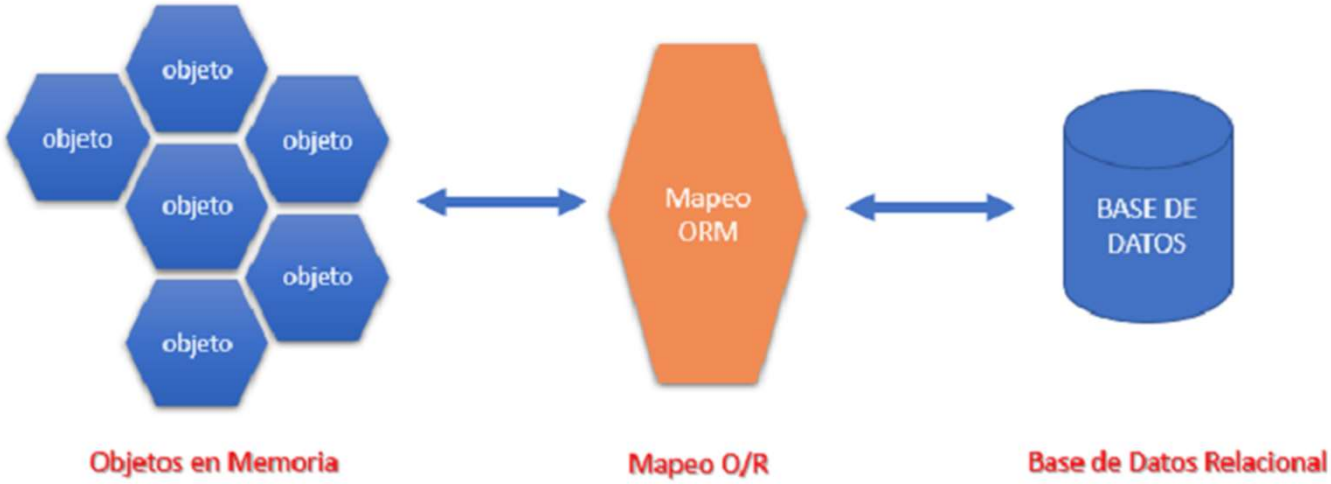
El mapeo objeto-relacional surge, pues, para dar respuesta a esta problemática: traducir los objetos a formas que puedan ser almacenadas en bases de datos preservando las propiedades de los objetos y sus relaciones; estos objetos se dice entonces que son persistentes.



El ORM se encarga, de forma automática, de convertir los objetos en registros y viceversa, simulando así tener una base de datos orientada a objetos.

2. Herramientas ORM. Características y herramientas más utilizadas.

Ya hemos dicho que las herramientas ORM se utilizan para dar solución al problema de que, en la programación orientada a objetos, la gestión de datos se implementa usando objetos; sin embargo, en los sistemas de gestión de bases de datos SQL sólo se pueden almacenar y manipular valores escalares organizados en tablas relacionales.



Object Relational Mapping (ORM) es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales, a representaciones (Modelos) de objetos. Dado que los RDBMS carecen de la flexibilidad para representar datos no escalares, la existencia de un ORM es fundamental para el desarrollo de sistemas de software robustos y escalables.

En el modelo relacional, cada fila en la tabla se mapea a un objeto y cada columna a una propiedad.

Las herramientas ORM pues, actúan como un puente que conecta las ventajas de los RDBMS con la buena representación de estos en un lenguaje Orientado a Objetos, o, dicho en otras palabras, nos lleva de la base de datos al lenguaje de programación.

2.1. Características.

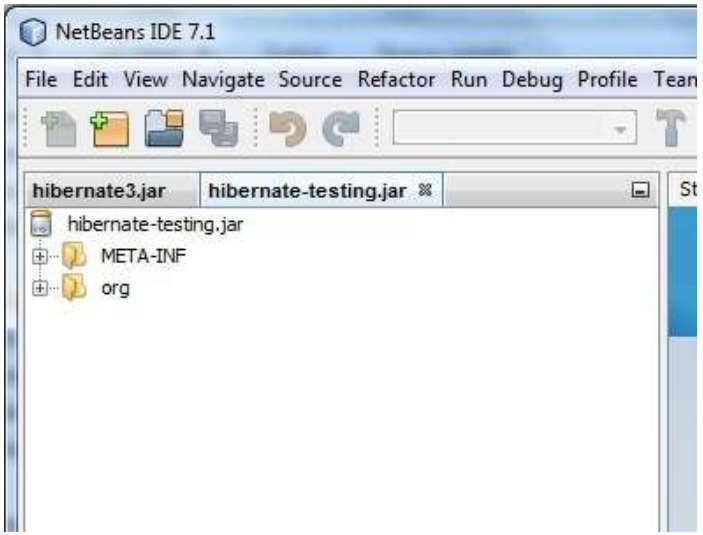
Las herramientas ORM facilitan el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o mediante anotaciones que permiten establecer estas relaciones.

Gracias a las ORM, podemos conectar con una base de datos relacional para extraer la información contenida en objetos de programas que están almacenados. Para ello, sólo tendremos que definir la forma en la que establecer la correspondencia entre las clases y las tablas una sola vez (indicando qué propiedad se corresponde con cada columna, qué clase con cada tabla, etc.). Una vez hecho esto, podremos utilizar POJOS de nuestra aplicación e indicar a la ORM que los haga persistentes, consiguiendo que una sola herramienta pueda leer o escribir en la base de datos utilizando VOs (Value Object, objeto Java Bean enfocado en la vista) directamente.

Una herramienta ORM permite tomar un objeto Java y hacerlo persistente, carga el objeto de la base de datos a memoria y permite hacer consultas a las tablas de la base de datos.

Ventajas de ORM.

- ✓ Ayudan a reducir el tiempo de desarrollo de software. La mayoría de las herramientas ORM disponibles, permiten la creación del modelo a través del esquema de la base de datos, es decir, el usuario crea la base de datos y la herramienta automáticamente lee el esquema de tablas y relaciones y crea un modelo ajustado.
- ✓ Abstracción de la base de datos.
- ✓ Reutilización.
- ✓ Permiten persistir objetos a través de un método Orm.Save y generar el SQL correspondiente.
- ✓ Permiten recuperar los objetos persistidos a través de un método Orm.Load.
- ✓ Lenguaje propio para realizar las consultas.



- ✓ Independencia de la base de datos.
- ✓ Incentivan la portabilidad y escalabilidad de los programas de software.

Desventajas de ORM.

- ✓ Tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización lleva un tiempo que hay que emplear en ver el funcionamiento correcto y ver todo el partido que se le puede sacar.

- ✓ Menor rendimiento (aplicaciones algo más lentas). Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá de transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

- ✓ Sistemas complejos. Normalmente la utilidad de ORM desciende con la mayor complejidad del sistema relacional.

2.2. Herramientas ORM más utilizadas.

Entre las herramientasORM más relevantes encontramos las siguientes:

Hibernate:

Hibernate es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación. Utiliza archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Java Persistence Api (JPA):

El Java Persistence (JPA) es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la Java Specification Request JSR 220.

Para su utilización, JPA requiere de J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características de lenguaje Java, como las anotaciones y los genéricos.

MyBatis:

MyBatis es un framework de persistencia desarrollado originalmente por Clinton Begin y otros bajo el nombre de iBatis dentro de la Apache software Foundation (ASF). Al igual que el resto de los proyectos desarrollados por la ASF, iBatis era una herramienta de código libre. Tras ser iBatis retirado de la ASF, los desarrolladores originales lo han continuado bajo el nombre de MyBatis. Puedes conocer más sobre este proceso en los enlaces que se proporcionan en "para saber más".

MyBatis sigue el mismo esquema de uso que Hibernate; se apoya en ficheros de mapeo XML para persistir la información contenida en los objetos en un repositorio relacional.

3. Arquitectura de Hibernate.

La arquitectura de Hibernate es en capas para mantenerlo aislado de tener que conocer las API subyacentes. Hibernate usa los datos de la base de datos y proporciona persistencia entre los diferentes objetos. Los interfaces que presenta Hibernate son los siguientes:

- **Configuration:** Representa un archivo de configuración o propiedades requerido por la hibernación. El objeto de configuración proporciona dos componentes claves: Conexión de bases de datos y configuración del mapeado de clase.

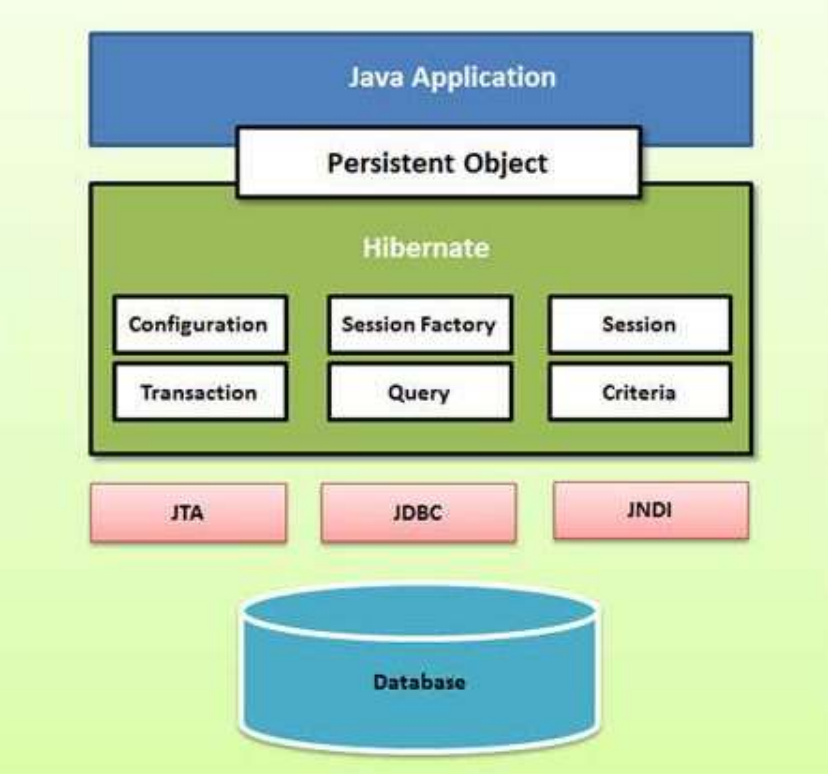
- **Session Factory:** Configura Hibernate para la aplicación que utiliza el archivo de configuración suministrado y permite un objeto Session que se crea una instancia.

- **Session:** Se utiliza para obtener una conexión física con una base de datos.

- **Transaction:** Representa una unidad de trabajo con la base de datos y la mayor parte del RDBMS soporta la funcionalidad de transacción. Las transacciones en hibernación son manejadas por un gestor de transacciones subyacente y de transacción (de JDBC o JTA).

- **Query:** Objetos de consulta utilizan SQL o Hibernate Query Language (HQL) para recuperar datos de la base de datos y crear objetos.

De forma gráfica esto se puede ver en la siguiente imagen que se muestra a continuación:



4. Instalación y configuración de Hibernate.

La instalación de Hibernate sobre el IDE NetBeans requiere tener instalado previamente este entorno de desarrollo, junto al JDK. El proceso que se va a desarrollar a continuación se hace con NetBeans 7.2 aunque para la versión 18 el proceso es idéntico.

El proceso de instalación de Hibernate se muestra en el documento adjunto (“Instalación Hibernate). Sin embargo, en las últimas versiones de Hibernate no es posible descargarlo directamente, sino que hay que usar un gestor de dependencias, como Maven. Y si se quieren utilizar los asistentes y realizar un mapeo automático, en las últimas versiones de NetBeans no funciona. Por ello, aunque en este curso se indican los pasos de cómo realizarlo en versiones previas de NetBeans, se recomienda utilizar el IDE Eclipse, el cual también se explica.

Eclipse

En Eclipse Hibernate no viene instalado por defecto, y hay que realizar la instalación como veremos en el punto siguiente. Antes de instalarlo, es recomendable tener la última versión del JDK y de Eclipse. Para instalar la última versión del JDK:

- Descargamos la última versión de https://www.oracle.com/java/technologies/downloads/ e instalamos.

- Vamos a eclipse y entramos en el menú Windows --> Java --> Preferences, y seleccionamos la opción Installed JREs. Pulsamos Add y en cuadro de dialogo elegimos Standar VM.

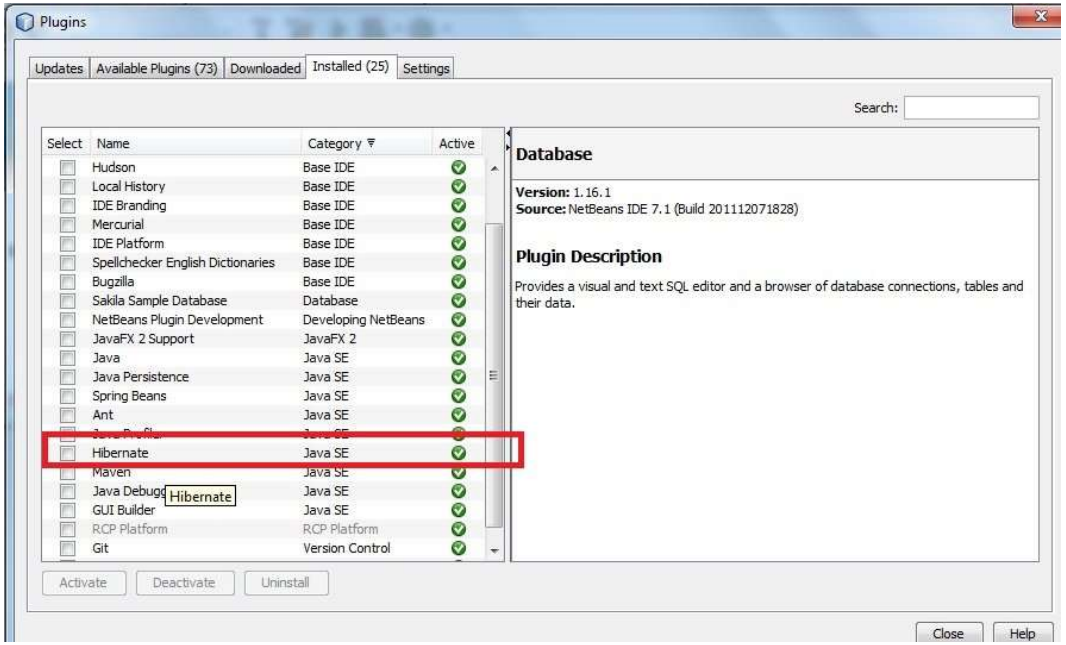
- En la opción JRE home seleccionaremos la carpeta donde hemos instalado el JDK y pulsaremos Finish.

- Marcaremos el check de nuestro JDK y pulsamos en Apply.

- En la opción Compiler elegiremos en Compiler compliance level, la versión de nuestro JDK y pulsamos en Apply and Close.

NetBeans

Una vez que inicializamos NetBeans, podemos encontrar la herramienta Hibernate como un plugin de este entorno de desarrollo. Lo único que tienes que hacer es ir a la lista de plugins de NetBeans y comprobar que lo tienes instalado; en caso contrario, deberás instalarlo desde la opción de plugins disponibles. No funciona en las últimas versiones.



Para utilizar Hibernate en una aplicación, es necesario conocer cómo configurarlo.

Hibernate puede configurarse y ejecutarse en la mayoría de aplicaciones Java y entornos de desarrollo. El archivo de configuración de Hibernate recibe el nombre de Hibernate.cfg.xml y contiene información sobre la conexión de la base de datos y otras propiedades. Al crearlo, hay que especificar la conexión a la base de datos.

En los siguientes puntos del tema veremos más detenidamente el proceso de configuración.

4.1. Instalación y configuración en NetBeans.

Aunque la descarga de Hibernate y el desarrollo de aplicaciones con Hibernate se puede realizar en NetBeans, en las últimas versiones de NetBeans no funciona el plugin de Hibernate, por lo que se recomienda utilizar Eclipse.

4.1.1. Instalación del plugin de Hibernate en NetBeans.

En las últimas versiones de NetBeans no funciona el plugin de Hibernate, por lo que se recomienda utilizar Eclipse.

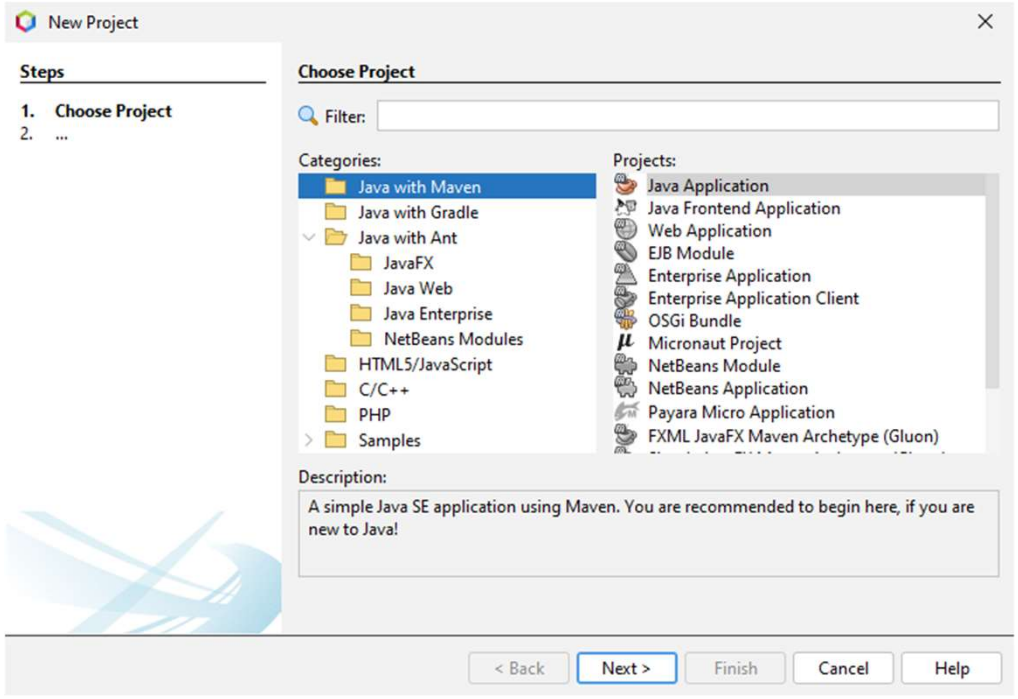
Para desarrollar fácilmente aplicaciones con Hibernate en NetBeans es necesario que tengas instalado el plugin de Hibernate. En las versiones antiguas este plugin venía instalado por defecto, pero en las nuevas versiones de NetBeans hay que instalarlo manualmente. En el siguiente video se explica cómo instalarlo.

https://www.youtube.com/embed/431fhzjUWf4

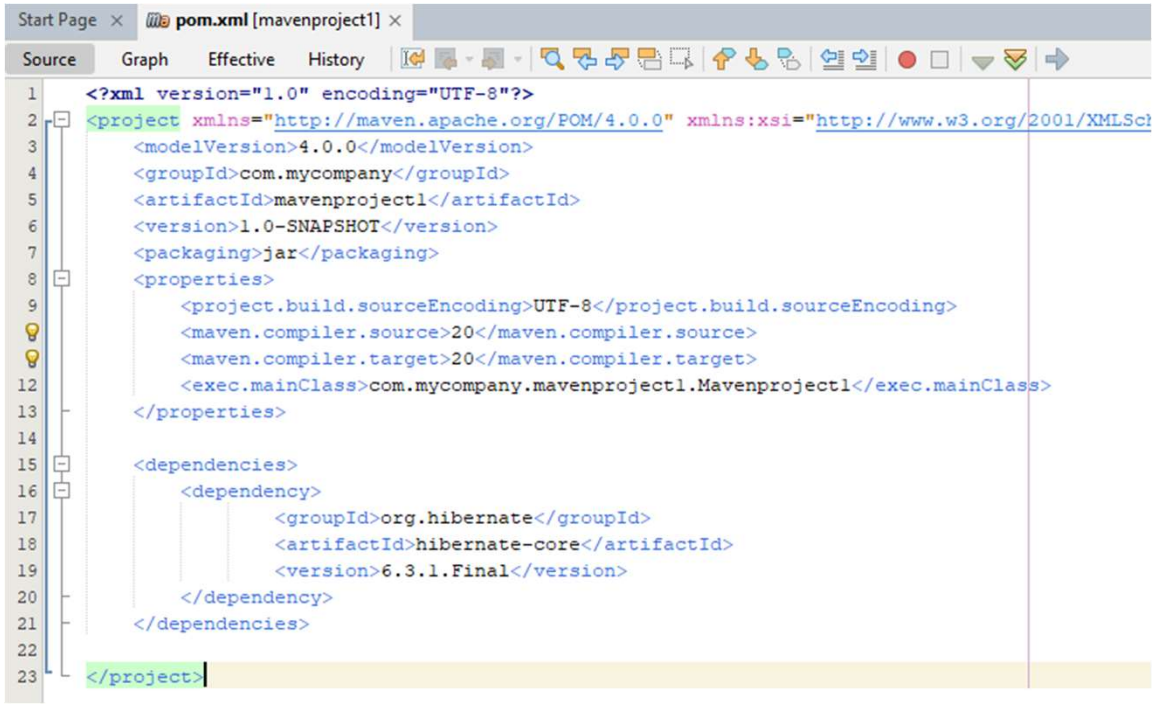
4.1.2. Descargar Hibernate.

Hasta la versión 6 de Hibernate se podía descargar y añadir como librerías a un proyecto. Desde entonces, hay que utilizar una herramienta de gestión de proyectos como Maven.

Para ello, en NetBeans hay que crear un proyecto con Maven:



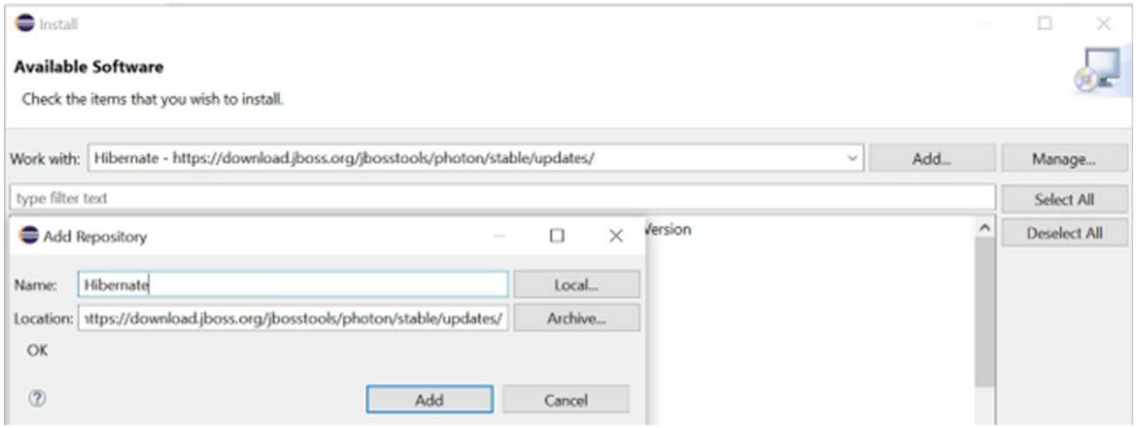
Y editar el fichero pom.xml, que se encuentra en Project files añadiendo las dependencias como se muestra en la siguiente imagen:



4.2. Instalación y configuración en Eclipse.

La instalación de Hibernate en Eclipse se compone de 2 partes: instalar el plugin, e instalar las librerías de Hibernate para poder utilizarlo. La instalación y uso del plugin es opcional, pero si queremos simplificar el trabajo de configuración de Hibernate en nuestro proyecto es altamente recomendable.

Paso 1



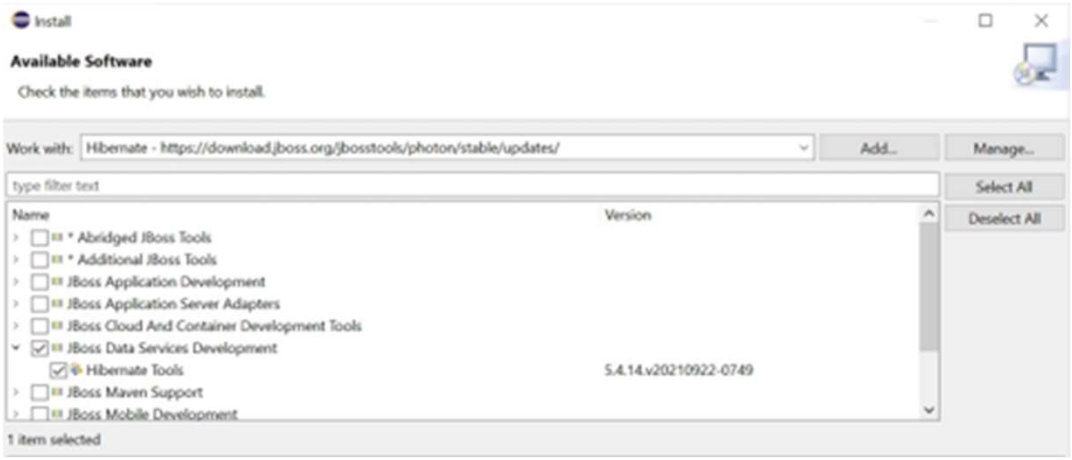
Para el proceso de instalación del plugin, vamos a necesitar tener conexión a internet. Haremos esta instalación desde el propio Eclipse:

Desde el **Menú Help** seleccionamos **Install New Software**.

Rellenaremos el campo **Work With** con la url **https://download.jboss.org/jbosstools/photon/stable/updates/** y pulsamos el botón Add.

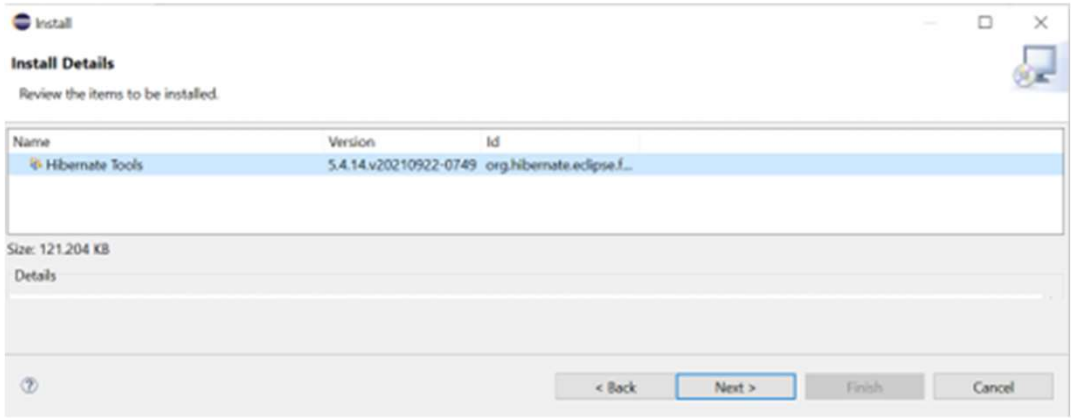
Pondremos el nombre, por ejemplo Hibernate, y pulsamos OK.

Paso 2



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

Paso 3



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

Durante el proceso de instalación nos pedirá confirmación para continuar ya que el plugin contiene software sin firmar.

Aceptaremos y, una vez instalado, procederemos a reiniciar Eclipse, tal como nos solicita.

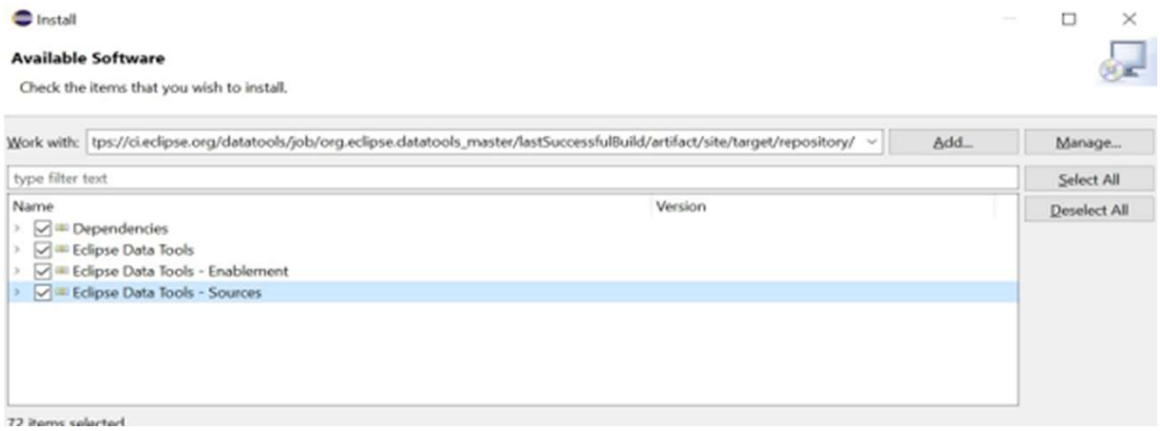
Paso 4

Para el proceso de instalación del Data Tools Platform, seguiremos unos pasos similares:

Desde el **Menú Help** seleccionamos **Install New Software**.

Rellenaremos el campo **Work With** con la url **http://download.eclipse.org/datatools/updates/release/latest** y pulsamos el botón Add.

Pondremos el nombre, por ejemplo, DTP, y pulsamos OK. En este caso, instalaremos todas las opciones disponibles.



Otra forma de instalar el plugin de Hibernate en Eclipse:

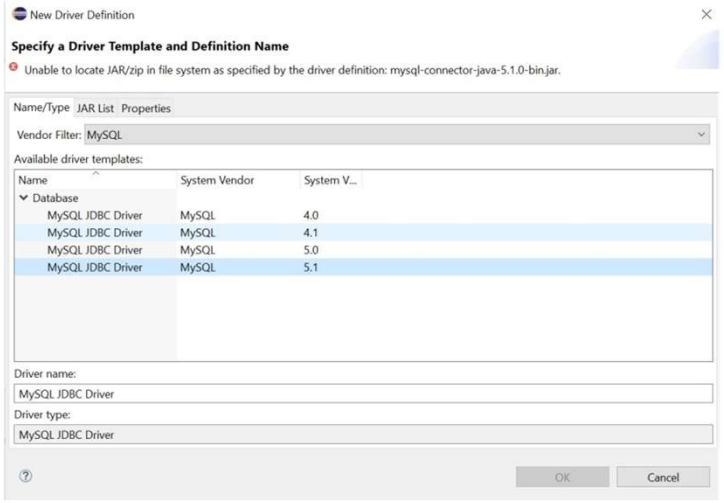
También se puede instalar desde **Eclipse Marketplace**, buscando JBoss y seleccionando la opción de Hibernate, como puedes ver en este vídeo.

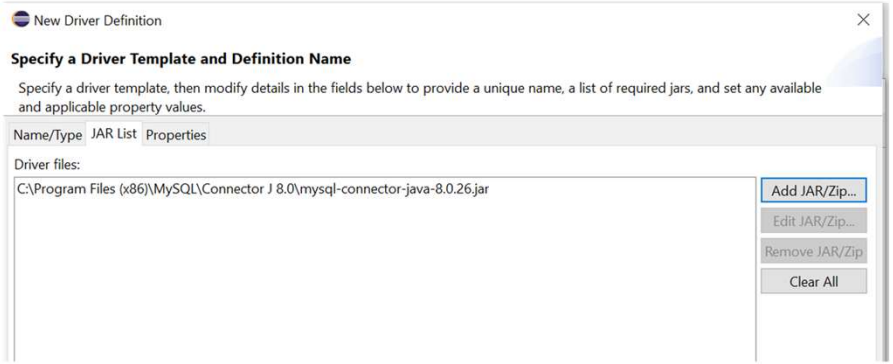
Una vez instalado Hibernate, el siguiente paso es configurarlo para que se comunique con nuestro Sistema Gestor de Base de Datos. En primer lugar, debemos añadir el driver, que es el mismo que utilizaríamos si hiciéramos una conexión por JDBC.

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

Tendríamos la opción de seleccionar en el Vendor Filter el correspondiente a nuestro SGBD (por ejemplo, “MySQL”), y elegimos el driver correspondiente a la versión de nuestro SGBD.

En el caso de MySQL, no está disponible el último driver existente, por lo que pinchamos sobre la pestaña JAR List, donde añadiremos el .JAR que podemos obtener en https://dev.mysql.com/downloads/connector/j/



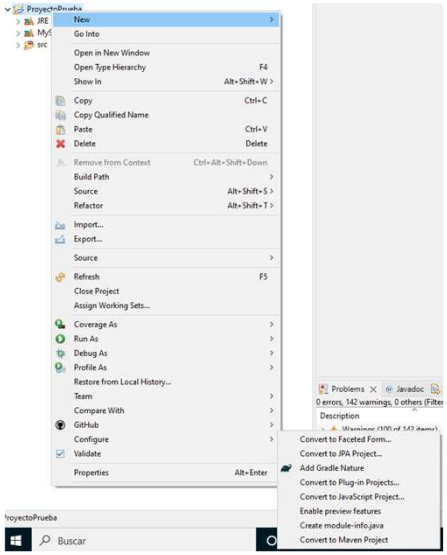


4.2.1. Descargar Hibernate.

Hasta la versión 6 de Hibernate se podía descargar y añadir como librerías a un proyecto. Desde entonces, hay que utilizar una herramienta de gestión de proyectos como Maven.

Para ello, en Eclipse hay que crear un proyecto con Maven o *mavenizar* un proyecto Java.

Para convertir un proyecto Java a un proyecto Maven, seleccionamos en **Configure**, la opción **Convert to Maven Project**.



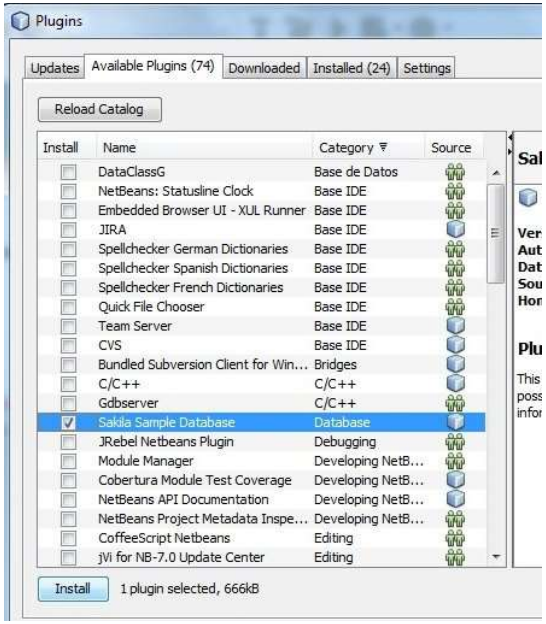
Se observa como si encima de la carpeta que está al lado del nombre del proyecto aparece una **J** es que es un proyecto Java, y si aparece una **M** es un proyecto Maven. Al convertir nuestro proyecto Java en un proyecto Maven, aparecen ambas letras.

Para poder utilizar Hibernate hay que editar el fichero **pom.xml**, que se encuentra en la raíz del proyecto, añadiendo las dependencias como se muestra en la siguiente imagen (se ha añadido el contenido en la etiqueta **dependencias**):

5. Ficheros de configuración y mapeo. Estructura y propiedades.

Para empezar a trabajar con Hibernate es necesario configurar la herramienta para que conozca qué objetos debe recuperar de la base de datos relacional y en qué lugar los hará persistir. Por tanto, el primer paso será tener una base de datos relacional con la que poder trabajar.

Tanto en NetBeans como en Eclipse, cuando se crea el archivo de configuración de Hibernate usando el asistente, podemos especificar la conexión a la base de datos, eligiendo de una lista de conexiones de bases de datos registradas en el IDE. Cuando se genera el archivo de configuración, el IDE añade de forma automática detalles de la conexión e información basada en la conexión de la base de datos seleccionada. El IDE añade también las bibliotecas de Hibernate en el proyecto. Después de crear el fichero de configuración, éste puede ser editado usando el editor interactivo, o editar directamente el código XML.



El fichero de configuración contiene información sobre la base de datos a la que vamos a conectar la aplicación. Si se la aplicación se va a conectar a varias bases de datos, sería necesario definir tantos archivos de configuración como bases de datos a las que nos queramos conectar.

Para tener toda esta información, en Hibernate surgen dos ficheros distintos:

- El archivo de propiedades de Hibernate (**Hibernate.properties**), que es el encargado de determinar todos los aspectos relacionados con el gestor de la base de datos y las conexiones con él.

- Los archivos que definen el emparejamiento (mapping) de propiedades con tablas y columnas (***.hbm.xml**).

Para utilizar Hibernate necesitamos tener una base de datos relacional instalada en NetBeans.

La seleccionamos desde la lista de plugins disponibles y la instalamos en nuestro IDE.

Como ya se ha visto, tras instalar el plugin podemos crear la base de datos Sakila desde la ventana Servicios de NetBeans. Para ello, seleccionaremos el servidor de MySQL y a continuación esta base de datos. Conectaremos con ella para poder tenerla disponible cuando empecemos a extraer la información que se nos pida.

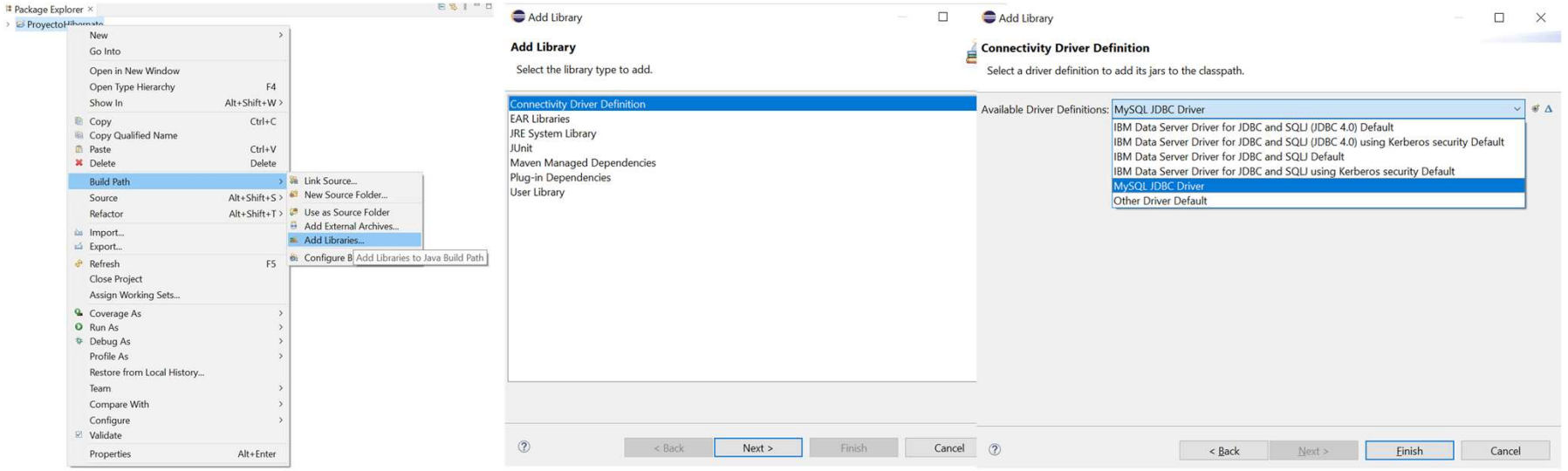
5.1. Ficheros de configuración. Propiedades.

El archivo de configuración de Hibernate es el **Hibernate.cfg.xml** y contiene información sobre la conexión de base de datos, las asignaciones de recursos y otras propiedades de conexión.

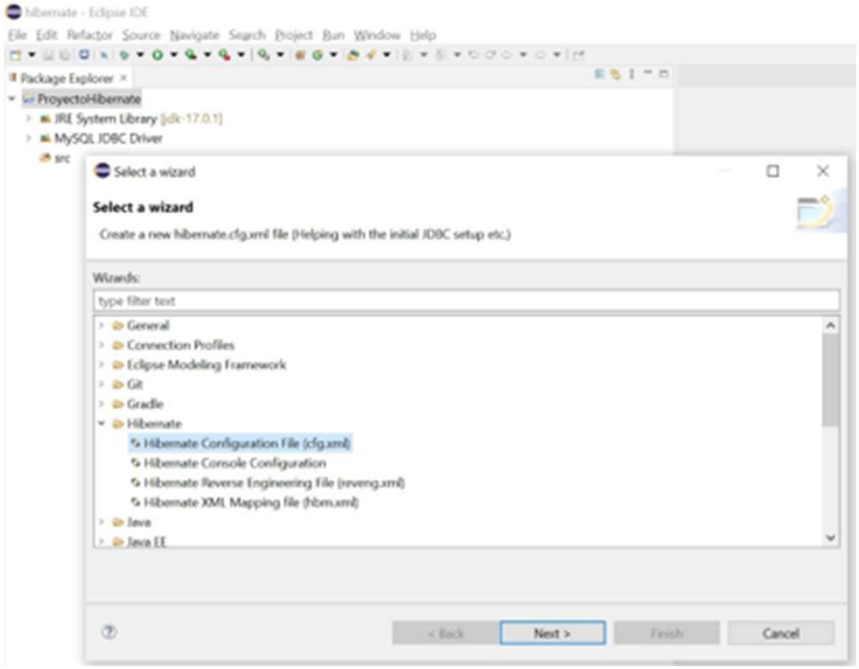
Eclipse

Una vez creado un proyecto de Java en Eclipse, debemos agregar el driver del SGBD (por ejemplo, el de MySQL), seleccionando en nuestro proyecto, con el botón derecho **Build Path**, y en **Add Libreries.**

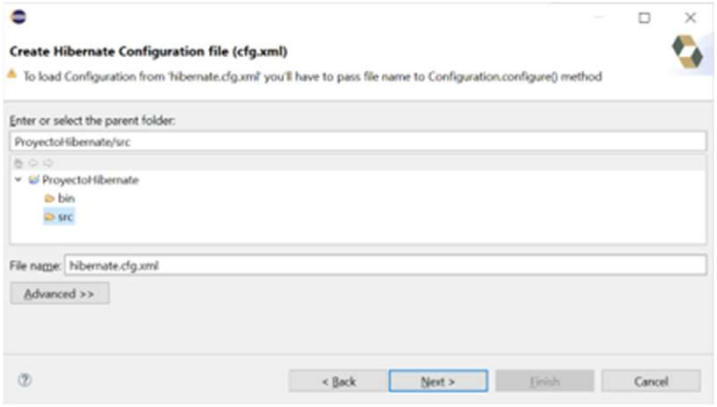
En la ventana que aparece, seleccionaremos **Connectivity Driver Definition** y pulsaremos “Next”. En la ventana se mostrará una lista opciones disponibles para conectarnos a una fuente de datos. Si el SGBD fuese MySQL, seleccionamos **MySQL JDBC Driver**.



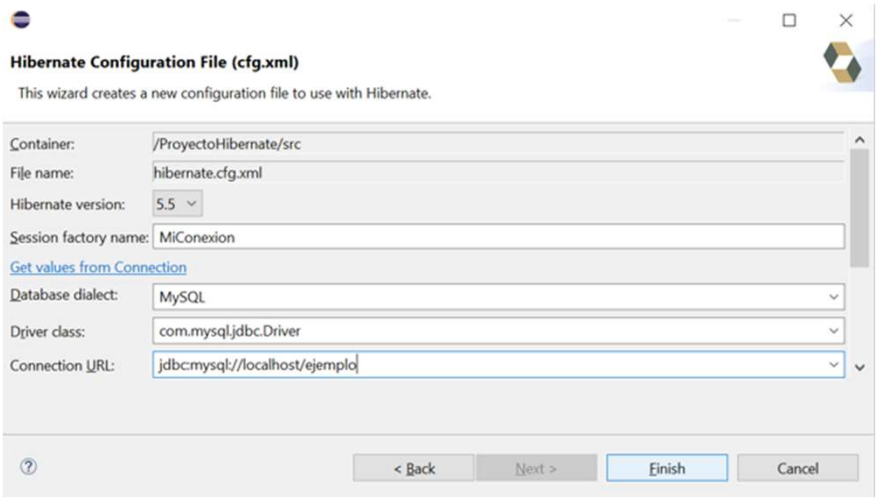
Una vez tenemos la librería en nuestro proyecto, hemos de crear el fichero de configuración de Hibernate, llamado **hibernate.cfg.xml**. Para ello, seleccionamos con el botón derecho, nuestro proyecto, y entramos en el menú **New**, en **Other**, en **Hibernate**, en **Hibernate Configuration File (cfg.xml)**.



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

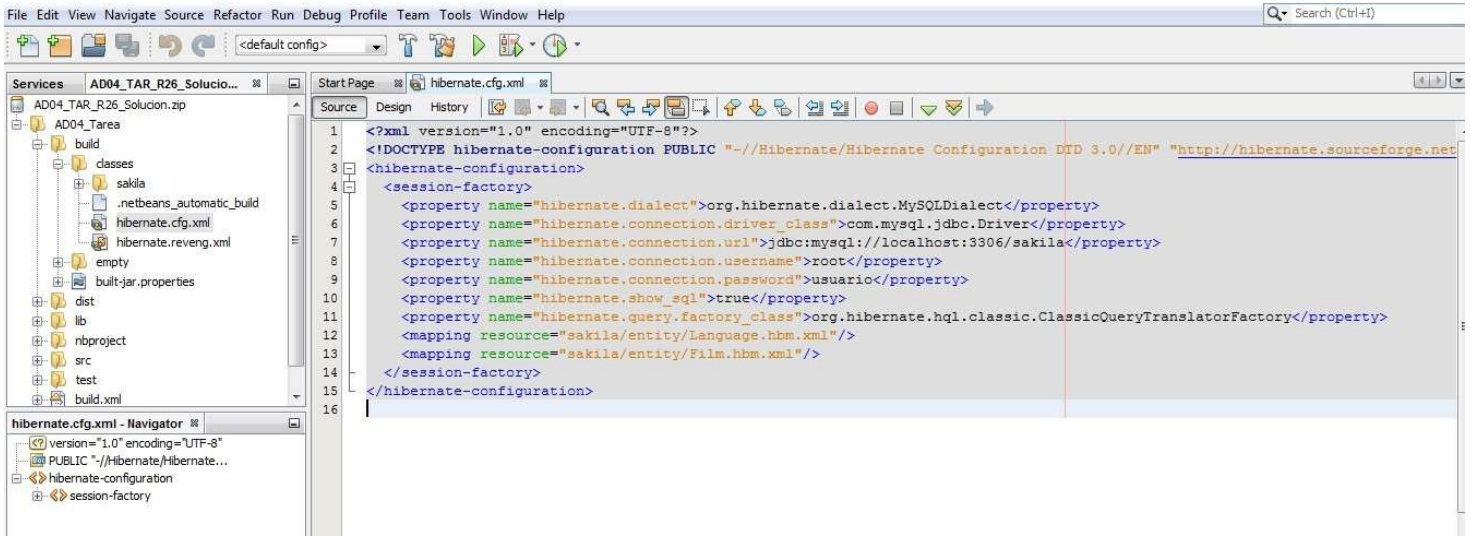


configuraremos los datos de conexión a nuestra base de datos:



NetBeans

Al seleccionar una conexión a una base de datos en NetBeans, se crea un archivo de configuración en Hibernate que guarda los detalles de esa conexión. NetBeans también añade de forma automática la biblioteca de Hibernate para la ruta de clase que tendrá nuestro proyecto. Después de la creación del archivo **Hibernate.cfg.xml**, podemos editarlo o modificar el código xml mediante el editor xml.



Las propiedades más importantes del fichero **Hibernate.cfg.xml** son:

- **Hibernate.dialect**: Dialecto o lenguaje empleado. Por ejemplo, MySQL.

- **Hibernate.connection.driver_class**. Driver utilizado para la conexión con la base de datos.

- **Hibernate.connection.url**. Dirección de la base de datos con la que se va a conectar Hibernate.

- **Hibernate.connection.username**. Nombre del usuario que va a realizar la extracción de información. Por defecto, el nombre de usuario es root.

- **Hibernate.connection.password**. Contraseña del root.

- **Hibernate.show_sql**. Para mostrar la herramienta. Por defecto, su valor es true.

Otro fichero importante es **hibernate.reveng.xml**, que contiene los parámetros de conexión a la base de datos. Su estructura es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate
Reverse Engineering DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
reverse-engineering-3.0.dtd">
<hibernate-reverse-engineering>
     <schema-selection match-schema="ANTONIO"/>
     <table-filter match-name="DEPARTAMENTOS"/>
     <table-filter match-name="EMPLEADOS"/>
</hibernate-reverse-engineering>
```

5.2. Ficheros de mapeo. Estructura, elementos y propiedades.

Fichero de mapeo ".hbm.xml‘’

Para cada clase que queremos persistir se creará un fichero xml con la información que permitirá mapear la clase a una base de datos relacional. Este fichero estará en el mismo paquete que la clase a persistir.

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Persistente.Departamentos" table="DEPARTAMENTOS" schema="ANTONIO">
    <id name="deptNo" type="byte">
      <column name="DEPT_NO" precision="2" scale="0" />
      <generator class="assigned" />
    </id>
    <property name="dnombre" type="string">
       <column name="DNOMBRE" length="15" />
    </property>
    <property name="loc" type="string">
      <column name="LOC" length="15" />
   </property>
    <set name="empleadoses" inverse="true">
    <key>
      <column name="DEPT_NO" precision="2" scale="0" not-null="true" />
    </key>
     <one-to-many class="Persistente.Empleados" />
    </set>
    </class>
</hibernate-mapping>
```

El significado del fichero anterior es el siguiente:

Hibernate-mapping: todos los ficheros de mapeo comienzan y acaban con esta etiqueta.

Class: engloba la clase con sus atributos e incluye el mapeo de la tabla de la base de datos. Name es el nombre de la clase, table es el nombre de la tabla de la base de datos que representa ese objeto y catálogo el nombre de la base de datos.

El Singleton

Antes de crear la clase main, una vez implementado hibernate, debemos crear una instancia que nos permita trabajar con la base de datos y que se utilizará a lo largo de toda la aplicación.

Un singleton es un patrón de diseño, realizado para restringir la creación de objetos pertenecientes a una clase. Su intención es garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado). Es una clase que ayuda y accede a **SessionFactory** para obtener un objeto de sesión. La clase se llama configure() y carga sobre el fichero de configuración hibernate.cfg.xml y entonces construye la **SessionFactory** para obtener el objeto de sesión. El nombre de la clase es **SessionFatoryUtil.java** y se incluirá en la clase main.

```
public class SessionFactoryUtil {
   private static final SessionFactory sessionFactory;
   static {
      try {
         sessionFactory= new
Configuration().configure().buildSessionFactory();
      } catch (Throwable ex) {
         System.err.println("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);
      }
   }
   public static SessionFactory getSessionFactory() {
      return sessionFactory;
   }
}
```

Para extraer un tabla concreta de la base de datos, la sintaxis en el mapeo requiere definir el POJO Nombre_Clase.hbm.xml, donde nombre_clase se corresponderá con el nombre la tabla que queremos extraer y donde se describe cómo se relacionan clases y tablas y propiedades y columnas.

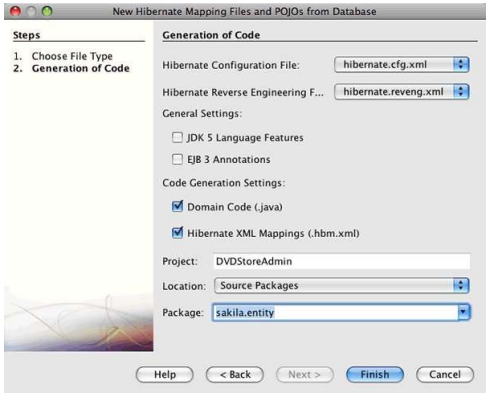
5.2.1. Creando ficheros de mapeo en NetBeans.

Mediante el asistente de NetBeans, seleccionamos Nuevo Mapeo en Hibernate y rellenamos los campos que nos piden, en función de la información que nos interese extraer de la base de datos.

De esta manera, NetBeans genera un POJO **nombre_clase.java** (siendo nombre_clase aquella tabla cuya información nos interesa extraer de la base de datos) con todos los campos necesarios.

Con ello, genera un fichero de mapeo de Hibernate, agregando su entrada a **Hibernate.cfg.xml**, entre las etiquetas .

Recuerda que en las últimas versiones de NetBeans no es posible realizar esto.

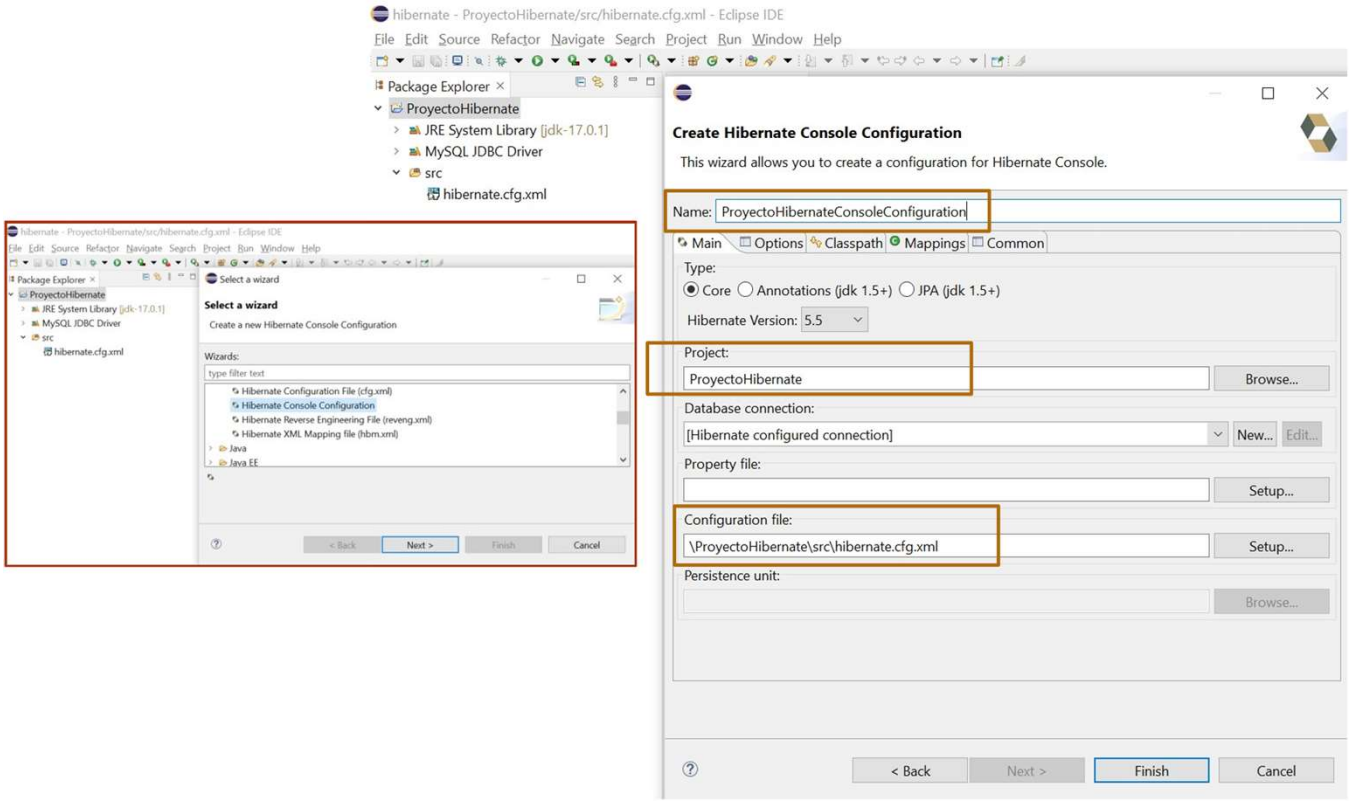


5.2.2. Creando ficheros de mapeo en Eclipse.

Eclipse permite realizar los ficheros de mapeo de forma automática, mediante el uso de asistentes, pero hay que realizar varios pasos.

Paso 1

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

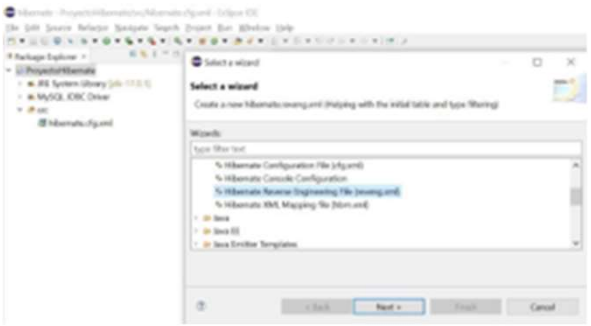


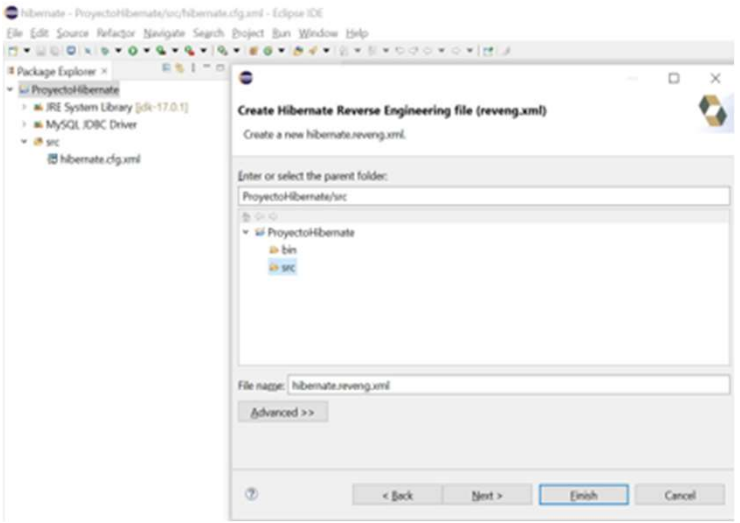
- **Name**: Por ejemplo “ProyectoHibernateConsoleConfiguration”.
- **Project**: Verificamos que corresponde con nuestro proyecto “ProyectoHibernate”.
- **Configuration File**: Verificamos que corresponde con el fichero que acabamos de crear en el paso anterior.

Paso 2

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

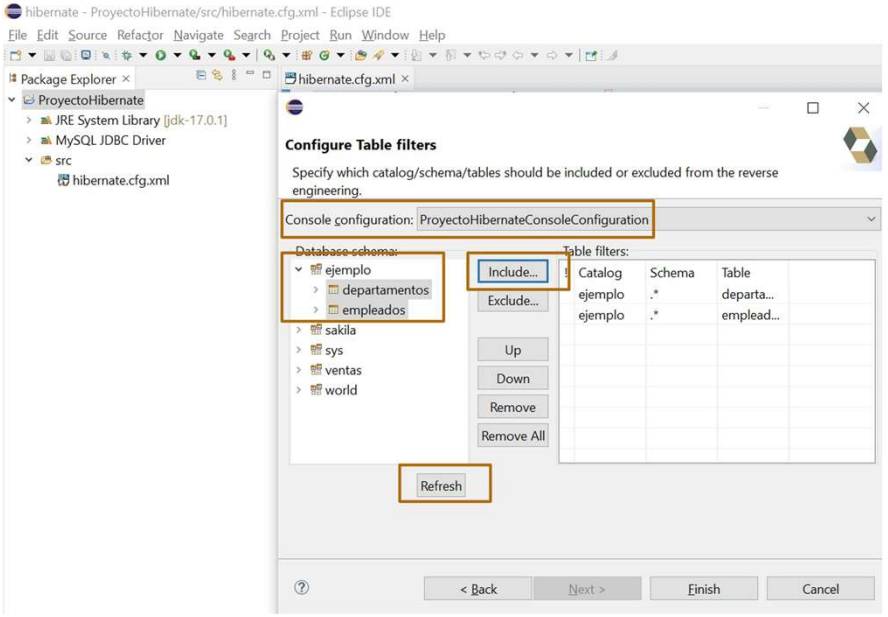
*Nota: Este fichero debemos guardarlo en el mismo lugar que el fichero de configuración, es decir, en nuestro caso en **src**. Una vez configurado, pulsaremos el botón **Next**.





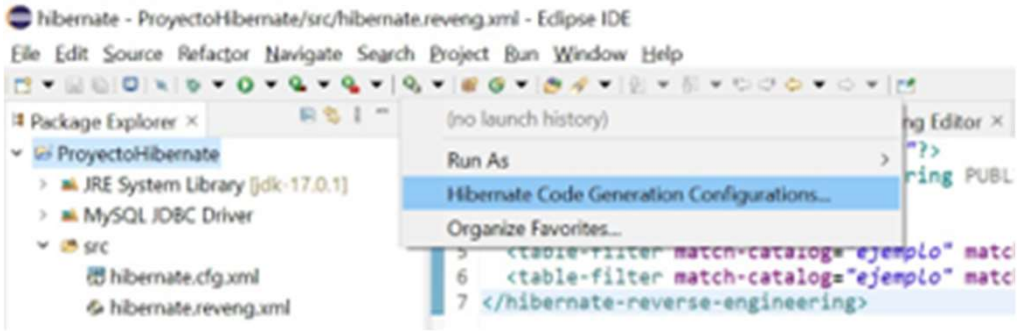
Desde la ventana **Configure Table Filters** que aparece, podemos configurar las tablas que queremos mapear. Para ello, seleccionaremos en el desplegable el **console configuration** creado anteriormente y pulsaremos **Refresh**.

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



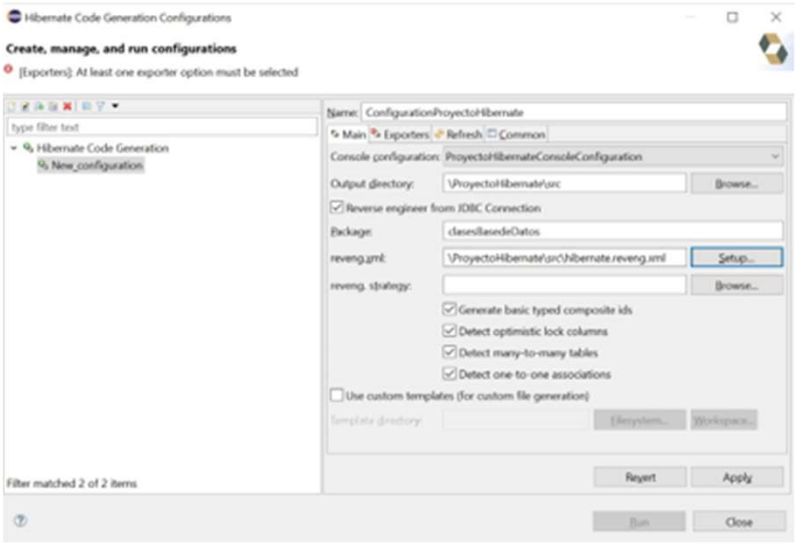
Paso 3

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



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

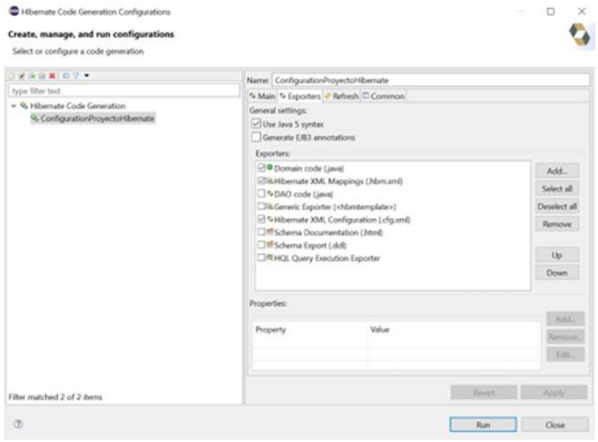
- **Name**: Por ejemplo “ConfigurationProyectoHibernate”
- **Console Configuration**: Elegimos el creado anteriormente.
- **Output directory**: Será “src” donde tenemos los ficheros.
- **Package**: Donde se crearán las clases
- **Reveng.xml**: configuraremos el creado anteriormente.



Tras la pestaña **Main**, configuraremos la pestaña **Exporters**, donde se indican los ficheros que queremos generar. Se marcarán las opciones: **Use Java 5 syntax**, **Domain code**, **Hibernate XML Mappings** e **Hibernate XML configuration**. Una vez seleccionados, pulsaremos **Apply** y después **Run.**

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

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



En este enlace hay un video que explica todos estos pasos.

6. Mapeo de colecciones, relaciones y herencia.

1. Mapeo de colecciones.

Hay bastantes rangos de mapeos que se pueden generar para colecciones que cubran diversos modelos relacionales.

El elemento de mapeo de Hibernate utilizado para mapear una colección depende del tipo de la interfaz; por ejemplo, un elemento se utiliza para mapear propiedades de tipo Set aunque, no obstante, existen además los elementos de mapeo etc.

Las instancias de colección de distinguen por la clave foránea de la entidad que posee la colección. El elemento mapea la columna clave de la colección.

Hay que resaltar que las colecciones pueden contener casi cualquier tipo de datos; esto implica que un objeto en una colección puede ser manejado con una semántica de "valor" o podría ser una referencia a otra entidad, con su propio ciclo de vida. Todos los mapeos de colección necesitan una columna índice en la tabla de colección: una columna índice es una columna que mapea a un índice de array o índice de **List**. Por último, una colección de valores o asociación muchos-a-muchos requiere una tabla de colección dedicada con una columna o columnas de clave foránea, columna de elemento de colección y probablemente una o varias columnas índice.

2. Mapeo de relaciones o asociaciones.

Para persistir, las relaciones usan las denominadas transacciones, ya que los cambios pueden incluir varias tablas. Una regla general para el mapeo es respetar el tipo de relación en el modelo de objetos y en el modelo relacional: así, una relación 1-1 en el modelo de objetos deberá corresponderse a una relación 1-1 en el modelo relacional.

Para mapear las relaciones, se usan los identificadores de objetos (OID). Son la llave primaria de la tabla relacionada y se agregan como una columna más en la tabla donde se quiere establecer la relación. Dicha columna es una clave foránea a la tabla con la que está relacionada.

3. Mapeo de herencia.

Como hemos comentado, las colecciones funcionan en ambos modelos, objeto y relacional. Para el caso de la herencia se presenta el problema que la base de datos relacionales no la soportan. Así es que somos nosotros quienes debemos modelar como se verá la herencia en el modelo relacional. Existen tres tipos de mapeos principales: modelar la jerarquía a una sola tabla, modelar la jerarquía completa en tablas, mapear cada tabla en tablas concretas. La decisión estará basada en el rendimiento y en la escalabilidad del modelo.

En el siguiente enlace hay acceso a un tutorial en línea de los mapeos en Hibernate, donde se detallan estos aspectos con más profundidad.

Mapeos en Hibernate

7. Clases persistentes.

Se denomina "persistencia" de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento. Las clases persistentes son clases en una aplicación que nos van a servir para representar entidades de la base de datos. No se considera que todas las instancias de una clase persistente puedan estar en estado persistente, ya que una instancia puede ser transitoria o separada.

El estándar Java Data Objects (JDO), define una clase con capacidad de persistencia, como aquella que implementa la interface javax.jdo.PersistenceCapable.

Las clases persistentes tienen la capacidad de definir objetos que pueden almacenarse y recuperarse y un almacén persistente de datos. La especificación JDO incorpora la figura del procesador de clases en código ejecutable Java, JDO Enhacer, que es un programa que modifica los archivos compilados de las clases, añadiendo el código ejecutable necesario para realizar la grabación y recuperación transparente de los atributos de las instancias persistentes.

JDO permite a los programadores convertir sus clases en persistentes, de forma que los objetos pertenecientes a clases concretas definidas por el programador pueden mantener su estado, con la única limitación de que el estado esté compuesto por los atributos persistentes que sean independientes del contexto de ejecución: tipos primitivos, tipos de referencia e interfaz y algunas clases del sistema que permiten modelar el estado como por ejemplo la clase **Array, Date**, etc.

Para poder indicar las clases y atributos que son persistentes, se utiliza un fichero de configuración XML, que se denomina **descriptor de persistencia**. Para que las instancias de las clases persistentes pueden mantenerse en los sistemas gestores de bases de datos, es necesario establecer la correspondencia entre los objetos y su estado persistente.

Las clases Java deberán tener las siguientes características:

- Deben tener un constructor público sin ningún tipo de argumentos.
- Para cada propiedad que queramos persistir debe haber un método get/set asociado.
- Implementar el interfaz serializable.

Equivalen a la tabla de la base de datos y un registro o fila es un objeto persistente de esa clase. Un ejemplo podría ser el que se muestra a continuación:

```
package Persistente;
import java.util.HashSet;
import java.util.Set;
public class Departamentos implements java.io.Serializable {
private byte deptNo;

private String dnombre;
private String loc;
private Set empleadoses = new HashSet(0);
public Departamentos() {
}
public Departamentos(byte deptNo) {
    this.deptNo = deptNo;
}
```

```java
public Departamentos(byte deptNo, String dnombre, String loc, Set
    empleadoses) {
    this.deptNo = deptNo;
    this.dnombre = dnombre;
    this.loc = loc;
    this.empleadoses = empleadoses;
}
    public byte getDeptNo() {
    return this.deptNo;
}
    public void setDeptNo(byte deptNo) {
    this.deptNo = deptNo;
}
    public String getDnombre() {
    return this.dnombre;
}
    public void setDnombre(String dnombre) {
    this.dnombre = dnombre;
}
    public String getLoc() {
    return this.loc;
}
```

```
    public void setLoc(String loc) {
    this.loc = loc;
}
    public Set getEmpleadoses() {
    return this.empleadoses;
}
    public void setEmpleadoses(Set empleadoses) {
    this.empleadoses = empleadoses;
}
    public void setdeptNo(byte b) {
    throw new UnsupportedOperationException("Not yet implemented");
}
```

8. Sesiones. Estados de un objeto.

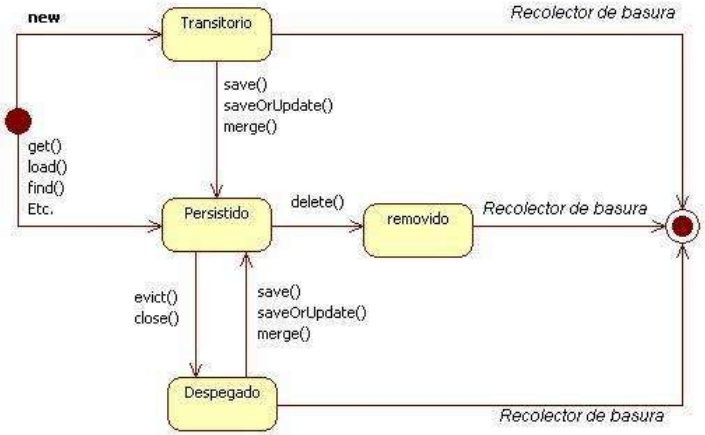
Para poder utilizar la persistencia en Hibernate es necesario definir un objeto **Session** utilizando la clase **SesssionFactory**. La sesión corresponde con un objeto que representa una unidad de trabajo con la base de datos. La sesión nos permite representar el gestor de persistencia, ya que dispone de una API básica que nos permite cargar y guardar objetos.

La sesión está formada internamente por una cola de sentencias SQL que son necesarias ejecutar para poder sincronizar el estado de la sesión con la base de datos.

Asimismo, la sesión contiene una lista de objetos persistentes. Una sesión corresponde con el primer nivel de caché.

Si para realizar el acceso a datos, usamos Hibernate, la sesión nos permite definir el alcance de un contexto determinado. Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto Session utilizando la clase **SessionFactory** de Hibernate. Un objeto Session Hibernate representa una única unidad-de-trabajo para un almacén de datos dado y lo abre un ejemplar de **SessionFactory**. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción.

Los estados en los que se puede encontrar un objeto son:



• **Transitorio (Transient).** En este estado estará un objeto recién creado que no ha sido enlazado con el gestor de persistencia.

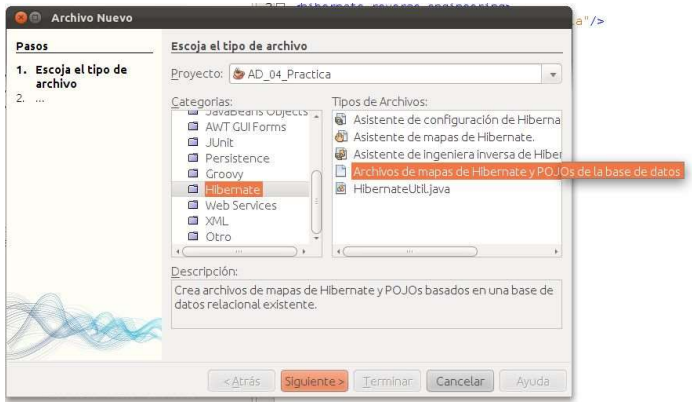
• **Persistente:** Ente este caso el objeto está enlazado con la sesión. Todos los cambios que se realicen serán persistentes.

• **Disociado (Detached):** En este caso nos encontramos con un objeto persistente que sigue en memoria después de que termine la sesión. En este caso existe en Java y en la base de datos.

• **Borrado (Removed):** En este caso el objeto está marcado para ser borrado de la base de datos. Existe en la aplicación Java y se borrará de la base de datos al terminar la sesión.

9. Carga, almacenamiento y modificación de objetos.

Para cargar un objeto de acceso a datos en la aplicación Java, el método **load()** de la clase **Session** suministra un mecanismo para capturar una instancia persistente, si conocemos su identificador. El método **load()** acepta un objeto **Class**, y cargará el estado de una nueva instancia de esa clase, inicializada en estado persistente.



El método **load()** lanzará una excepción irrecuperable si no existe la fila de base de datos correspondiente. Si no se está seguro de que exista una fila correspondiente, debe usarse el método **get()**, el cual consulta la base de datos inmediatamente y devuelve **null** si no existe una fila correspondiente.

Existen dos métodos que se encargan de recuperar un objeto persistente por identificador: **load()** y **get()**. La diferencia entre ellos radica en cómo indican que un objeto no se encuentra en la base de datos: **get()** devuelve un nulo y **load()** lanza una excepción **ObjectNotFoundException**.

Aparte de esta diferencia, **load()** intenta devolver un objeto proxy (durante la sesión de Hibernate se están cargando objetos y algunos de ellos se cargan como proxy debido a la carga diferida) siempre y cuando le sea posible (no esté en el contexto de persistencia). Con lo que es posible que la excepción sea lanzada cuando se inicialice el objeto proxy. Esto es conocido como **carga perezosa**.

En el caso que se obtenga un proxy, no tiene impacto sobre la base de datos (no se ejecuta ninguna consulta), hasta que no se inicializa el mismo. Muy útil cuando se obtiene una referencia de un objeto para asociarlo a otro. (No es necesario obtener el objeto).Se modifica un objeto persistente.

En el siguiente enlace se encuentra todo lo referente a la carga, almacenamiento y modificación de objetos.

9.1. Almacenamiento y modificando de objetos persistentes.

Para almacenar objetos persistentes se proceso siguiendo los siguientes pasos:

1. Se instancia un objeto nuevo (estado transitorio).

2. Se obtiene una sesión y se comienza la transacción, inicializando el contexto de persistencia.

3.Una vez obtenida da la sesión, se llama al método **save()**, el cual introduce el objeto en el contexto de persistencia. Este método devuelve el identificador del objeto persistido.

4. Para que los cambios sean sincronizados en las bases de datos, es necesario realizar el **commit** de la transacción. Dentro del objeto sesión se llama al método **flush()**. Es posible llamarlo explícitamente. En este momento, se obtiene la conexión JDBC a la base de datos para poder ejecutar la oportuna sentencia.

5.Finalmente, la sesión se cierra, con el objeto de liberar el contexto de persistencia, y, por tanto, devolver la referencia del objeto creado al estado disociado.

Los objetos cargados, grabados, creados o consultados por la sesión pueden ser manipulados por la aplicación, y cualquier cambio a su estado de persistencia será persistido cuando se le aplique "flush" a la sesión.

No hay que invocar ningún método en particular para que las modificaciones se vuelvan persistentes. Así que la manera más sencilla y directa de actualizar el estado de un objeto es cargarlo con load(), y luego manipularlo directamente, mientras la sesión esté abierta.

Para borrar objetos persistentes, podemos ejecutar **Session.delete()**, que quitará el estado de un objeto de la base de datos. Por supuesto, su aplicación podría aún contener una referencia al objeto quitado. Se puede borrar objetos en cualquier orden, no se van producir violaciones de llave externa, pero sí es posible violar **constraints NOT NULL** aplicadas a la columna de llave externa.

Muchas aplicaciones necesitan capturar un objeto en una transacción, mandarlo a la capa de interfaz de usuario para su manipulación, y grabar sus cambios en una nueva transacción. Las aplicaciones que usan este tipo de estrategia en entornos de alta concurrencia, normalmente usan datos versionados para garantizar aislamiento durante la "larga" unidad de trabajo.
Hibernate soporta este modelo, proveyendo "*reasociación*" de entidades desprendidas usando los métodos **Session.update()** o **Session.merge()**.

10. Consultas SQL.

Usando Hibernate, la ejecución de consultas SQL nativas se controla por medio de la interfaz **SQLQuery**, la cual se obtiene llamando a **Session.createSQLQuery()**. Las siguientes secciones describen cómo utilizar esta API para consultas.

La consulta SQL más básica es para obtener a una lista de escalares (valores).

- **sess.createSQLQuery("SELECT * FROM Personas").list();**
- **sess.createSQLQuery("SELECT ID,NOMBRE, EDAD FROM PERSONAS").list();**

Estas retornarán una lista de objetos arrays (**Object[]**) con valores escalares para cada columna en la tabla PERSONAS. Hibernate utilizará **ResultSetMetadata** para deducir el orden real y los tipos de los valores escalares retornados.

Otro tipo de consulta más compleja, es la consulta de entidades. Para obtener los objetos entidades desde una consulta sql nativa, se utiliza por medio de **addEntity()**.

- **sess.createSQLQuery("SELECT * FROM PERSONAS").addEntity(Persona.class);**
- **sess.createSQLQuery("SELECT ID,NOMBRE,EDAD FROM PERSONAS").addEntity(Persona.class);**

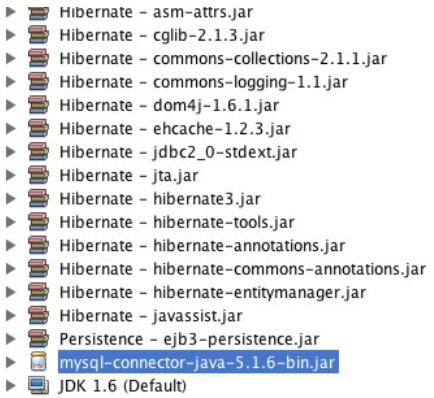
Se especifica esta consulta:

- La cadena de consulta SQL.
- La entidad devuelta por la consulta.

Asumiendo que Persona es mapeado como una clase con las columnas IDENTIFICACION, NOMBRE y FECHA DE NACIMIENTO las consultas anteriores devolverán una Lista en donde cada elemento es una entidad Persona.

11. Lenguajes propios de la herramienta ORM: HQL.

Hibernate utiliza un lenguaje de consulta potente (**HQL**) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente **orientado a objetos** y comprende nociones como herencia, polimorfismo y asociación. Las consultas se escriben en HQL y Hibernate se encarga de convertirlas al SQL usado por la base de datos con la que estemos trabajando y ejecutarla para realizar la operación indicada.



HQL es que es case-insensitive, o sea que sus sentencias pueden escribirse en mayúsculas y minúsculas. Por lo tanto "SeLeCt", "seleCT", "select", y "SELECT" se entienden como la misma cosa. Lo único con lo que debemos tener cuidado es con los nombres de las clases que estamos recuperando y con sus propiedades, ahí si se distinguen mayúsculas y minúsculas. O sea, en este caso "pruebas.Hibernate.Usuario" NO ES LO MISMO que "PrueBAs.Hibernate.UsuArio".

Entre las características más importantes de HQL:

- Soporte completo para operaciones relacionales: HQL permite representar consultas SQL en forma de objetos. HQL usa clases y atributos o propiedades en vez de tablas y columnas.

- Regresa sus resultados en forma de objetos: Las consultas realizadas usando HQL regresan los resultados de las mismas en la forma de objetos o listas de objetos, que son más fáciles de usar.

- Consultas Polimórficas: Podemos declarar el resultado usando el tipo de la superclase e Hibernate se encargará de crear los objetos adecuados de las subclases correctas de forma automática.

• Soporte para características avanzadas: HQL contiene muchas características avanzadas que son muy útiles y que no siempre están presentes en todas las bases de datos, o no es fácil usarlas, como paginación, fetch joins con perfiles dinámicos, inner y outer joins, etc. Además, soporta proyecciones, funciones de agregación (max, avg), y agrupamientos, ordenamientos, y subconsultas.

• Independiente del manejador de base de datos: Las consultas escritas en HQL son independientes de la base de datos (siempre que la base de datos soporte la característica que estamos intentando utilizar.

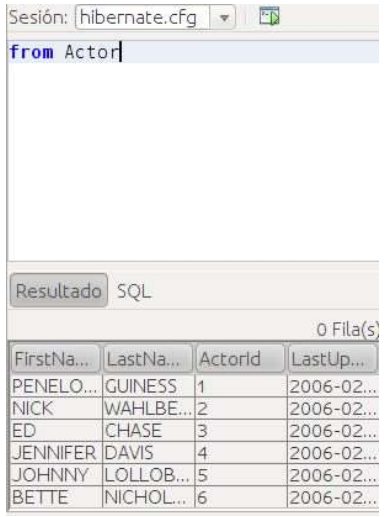
11.1. Sintaxis del lenguaje HQL.

El Hibernate Query Language (HQL) es el lenguaje de consultas que usa Hibernate para obtener los objetos desde la base de datos. Su principal particularidad es que las consultas se realizan sobre los objetos java que forman nuestro modelo de negocio, es decir, las entidades que se persisten en Hibernate. Esto hace que HQL tenga las siguientes características:

- Los tipos de datos son los de Java.
- Las consultas son independientes del lenguaje de SQL específico de la base de datos.
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible tratar con las colecciones de Java.
- Es posible navegar entre los distintos objetos en la propia consulta.

En Hibernate las consultas HQL se lanzan (o se ejecutan) sobre el modelo de entidades que hemos definido en Hibernate, esto es, sobre nuestras clases de negocio.

De forma poco ortodoxa se podría ver cómo que nuestro *modelo de tablas* en HQL son las clases Java y **NO** las tablas de la base de datos. Es decir que cuando hagamos "**SELECT columna FROM nombreTabla**" , el "**nombreTabla**" será una clase Java y "**columna**" será una propiedad Java de dicha clase y **nunca** una tabla de la base de datos ni una columna de una tabla.



Sentencias consulta (SELECT)

• **Cláusula from**: La consulta más simple que se puede realizar con Hibernate, es utilizando la cláusula **from**, la siguientes sería una consulta que mostraría todos los datos de una tabla de nombre Alumnos: **from Alumnos.**

• **Cláusula select**: La cláusula select escoge qué objetos y propiedades devolver en el conjunto de resultados de la consulta. Un ejemplo de consulta podría ser **select alumno.nombre from Alumnos alumno where alumno.nombre like 'A%'.**

• **La cláusula where**: La cláusula **where** nos permite refinar la lista de instancias retornadas. Si no existe ningún alias, puede referirse a las propiedades por nombre: **from Alumnos where nombre='Francisco'**. Si existe un alias, usaremos un nombre de propiedad calificado: **from Alumnos as alumnos where alumnos.nombre='Francisco'**. Esto retorna instancias de Alumnos llamados "Francisco".

• **Funciones de agregación**. Las consultas HQL pueden retornar resultados de funciones de agregación sobre propiedades: **select avg(alumnos.nota), sum(alumnos.nota), max(alumnos.nota), count(alumnos) from Alumnos alumnos**.

• **Expresiones**. Las expresiones utilizadas en la cláusula where incluyen lo siguiente: operadores matemáticos, operadores de comparación binarios, operadores lógicos , paréntesis () que indican agrupación, funciones Java, etc.

• **La cláusula order by**. La lista retornada por una consulta se puede ordenar por cualquier propiedad de una clase retornada o componentes. La palabra **asc** o **desc** opcionales indican ordenamiento ascendente o descendente respectivamente.

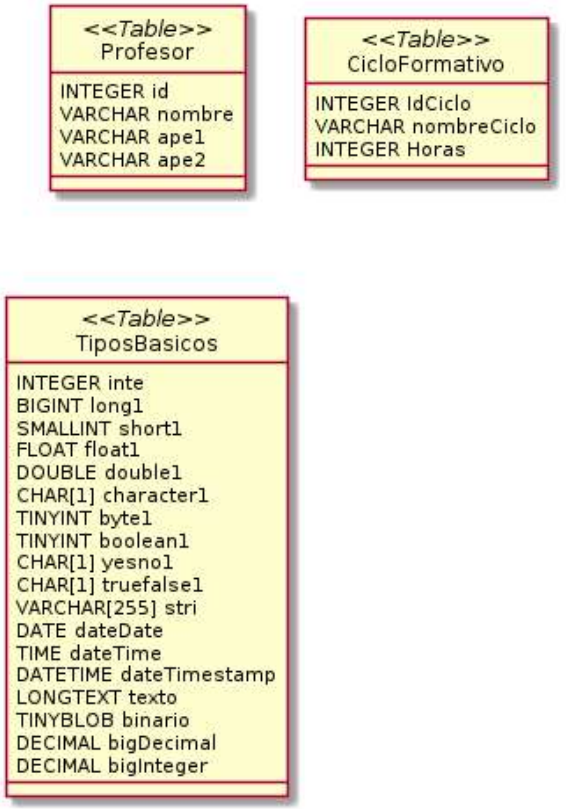
• **La cláusula group by**. Una consulta que retorna valores agregados se puede agrupar por cualquier propiedad de una clase retornada o componentes.

• **La cláusula having**: restringe grupos.

La sintaxis HQL para las sentencias de consultas es la siguiente:

```
[select_clause] from_clause [where_clause] [groupby_clause] [having_clause]
[orderby_clause]
```

Sobre el siguiente modelo de datos que se expone a continuación se harán todos los ejemplos de HQL:



Vemos en la sintaxis que la única clausula obligatoria es ***from_clause.***

No obstante, Hibernate recomienda incluir **select_clause**, por portabilidad con JPQL (java persistence query language).

Veamos ahora un sencillo ejemplo de consulta en HQL:

```
SELECT c FROM Ciclo c ORDER BY nombre
```

¿Qué diferencias podemos ver entre HQL y SQL?

- **Ciclo** hace referencia a la clase Java**ejemplo02.Ciclo** y **NO** a la tabla **CicloFormativo**. Nótese que la clase Java y la tabla tienen distinto nombre.
- Es necesario definir el alias **c** de la clase Java **Ciclo**.
- Tras la palabra **SELECT** se usa el alias en vez del "*".
- Al ordenar los objetos se usa la propiedad **nombre** de la clase **Ciclo** en vez de la columna **nombreCiclo** de la tabla **CicloFormativo**.

Recuerda incluir el alias en la consulta HQL. Si no se hace y se deja la consulta de la siguiente forma:

```
SELECT Ciclo FROM Ciclo
```

se producirá la siguiente excepción:

```
java.lang.NullPointerException
```

Hibernate soporta **no** incluir la parte del **SELECT** en la consulta HQL, quedando entonces la consulta de la siguiente forma:

```
FROM Ciclo
```

pero en la propia documentación se recomienda no hacerlo ya que de esa forma se mejora la portabilidad en caso de usar el lenguaje de consultas de JPA .

Se ha hecho mención de esta característica ya que en muchos tutoriales que se encuentran por Internet se hace uso de ella.

Mayúsculas

Respecto a la sensibilidad de las mayúsculas y minúsculas, el lenguaje HQL sí que lo es, pero con matices.

- Las palabras clave del lenguaje **NO** son sensibles a las mayúsculas o minúsculas.
  - Las siguientes 2 consultas son equivalentes.

```
select count(*) from Ciclo
SELECT COUNT(*) FROM Ciclo
```

- El nombre de las clases Java y sus propiedades **SI** son sensibles a las mayúsculas o minúsculas.
  - La siguiente consulta HQL es correcta:

```
SELECT c.nombre FROM Ciclo c WHERE nombre='Desarrollo de aplicaciones Web’
```

• La siguiente consulta HQL es **errónea** ya que la propiedad **nombre** está escrita con la “N” en mayúsculas.

```
SELECT c.Nombre FROM Ciclo c WHERE Nombre='Desarrollo de aplicaciones Web'
```

- La siguiente consulta HQL es **errónea** ya que el nombre de la clase Java **Ciclo** está escrita con la “c” en minúsculas.

```
SELECT c.nombre FROM ciclo c WHERE nombre='Desarrollo de aplicaciones Web’
```

• Al realizar comparaciones con los valores de las propiedades, éstas **NO** son sensibles a las mayúsculas o minúsculas.

- Las siguientes 2 consultas retornan los mismos objetos.

```
SELECT c.nombre FROM Ciclo c WHERE nombre='Desarrollo de aplicaciones Web’
SELECT c.nombre FROM Ciclo c WHERE nombre='DESARROLLO DE APLICACIONES WEB'
```

Filtrando

Al igual que en SQL en HQL también podemos filtrar los resultados mediante la cláusula **WHERE**. La forma de usarla es muy parecida a SQL.

```
SELECT p FROM Profesor p WHERE nombre='ISABEL' AND ape1<>'ORELLANA'
```

Al igual que con el nombre de la clase, el nombre de los campos del **WHERE** siempre hace referencia a las propiedades Java y nunca a los nombres de las columnas. De esa forma seguimos independizando nuestro código Java de la estructura de la base de datos.

Literales

- Texto.

El carácter para indicar un literal de texto es la comilla simple no pudiéndose usar la doble comilla.

```
SELECT p FROM Profesor p WHERE nombre='juan’
```

Si se quiere usar la comilla dentro de un literal deberemos duplicarla.

```
SELECT p FROM Profesor p WHERE ape1='perez l''andreu’
```

- Integer.

Para incluir un número del tipo **integer** simplemente se escribe dicho número.

SELECT tb FROM TiposBasicos tb WHERE inte=4

• Long.

Para incluir un número del tipo **long** se escribe dicho número y se añade una **L** mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE long1=4L
```

• double.

Para representar un **double** se escribe el número separando la parte decimal con un punto o se puede usar la notación científica.

```
SELECT tb FROM TiposBasicos tb WHERE double1=1.45
SELECT tb FROM TiposBasicos tb WHERE double1=1.79769313486231570E308
```

• float.

Para representar un **float** se escribe el número separando la parte decimal con un punto o se puede usar la notación científica, pero se le añade el carácter **F** en mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE float1=1.45F
SELECT tb FROM TiposBasicos tb WHERE float1=3.4028235E38F
```

• Fecha.

Para indicar una fecha la incluiremos entre comillas simples con el formato **yyyy-mm-dd.**

```
SELECT tb FROM TiposBasicos tb WHERE dateDate='2012-07-25'
```

• Hora.

Para indicar una hora la incluiremos entre comillas simples con el formato **hh:mm:ss**

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='02:05:10’
```

• fecha y hora.

Para indicar una fecha y hora la incluiremos entre comillas simples con el formato **yyyy-mm-dd hh:mm:ss.millis**, siendo optativos el último punto y los milisegundos .

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='2012-07-25 02:05:10'
```

• Operadores de comparación.

Para comparar los datos en una expresión se pueden usar las siguientes Operadores:

- Signo **igual ”=”:** La expresión será verdadera si los dos datos son iguales. En caso de comparar texto, la comparación no es sensible a mayúsculas o minúsculas.
- Signo **mayor que ”>”:** La expresión será verdadera si el dato de la izquierda es mayor que el de la derecha.
- Signo **mayor o igual que ”>=”:** La expresión será verdadera si el dato de la izquierda es mayor o igual que el de la derecha.
- Signo **menor que ”<”:** La expresión será verdadera si el dato de la izquierda es menor que el de la derecha.
- Signo **menor o igual que ”<=”:** La expresión será verdadera si el dato de la izquierda es menor o igual que el de la derecha.
- Signo **distinto ”<>”:** La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- Signo **distinto ”!=”:** La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.

- Operador ”between”: La expresión será verdadera si el dato de la izquierda está dentro del rango de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte BETWEEN 1 AND 10
```

- Operador ”in”: La expresión será verdadera si el dato de la izquierda está dentro de la lista de valores de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte IN (1,3,5,7)
```

- Operador ”like”: La expresión será verdadera si el dato de la izquierda coincide con el patrón de la derecha. Se utilizan los mismos signos que en SQL ”%” y ”_”.

```
SELECT tb FROM TiposBasicos tb WHERE stri LIKE 'H_la%'
```

- Operador ”not”: Niega el resultado de una expresión.
- Expresión ”is null”: Comprueba si el dato de la izquierda es null.

```
SELECT tb FROM TiposBasicos tb WHERE dataDate IS NUL
```

- Operadores lógicos.

  Se puede hacer uso de los típicos operadores lógicos como en SQL: AND, OR y NOT

```
SELECT p FROM Profesor p WHERE nombre='ANTONIO' AND (ape1='LARA' OR ape2='RUBIO’)
```

- Operadores Aritméticos.

  Se puede hacer uso de los típicos operadores aritméticos: suma (+), resta (-), multiplicación (*) y división (/).

```
SELECT tb FROM TiposBasicos tb WHERE (((inte+1)*4)-10)/2=1
```

• Funciones de agregación.

Las funciones de agregación que soporta HQL son:

- **AVG()**: Calcula el valor medio de todos los datos.
- **SUM()**: Calcula la suma de todos los datos.
- **MIN()**: Calcula el valor mínimo de todos los datos.
- **MAX()**: Calcula el valor máximo de todos los datos.
- **COUNT()**: Cuanta el número de datos.

```
SELECT AVG(c.horas),SUM(c.horas),MIN(c.horas),MAX(c.horas),COUNT(*)
FROM Ciclo c
```

• Funciones sobre escalares.

Algunas de las funciones que soporta HQL sobre datos escalares son:

- **UPPER(s)**: Transforma un texto a mayúsculas.
- **LOWER(s)**: Transforma un texto a minúsculas.
- **CONCAT(s1, s2)**: Concatena dos textos
- **TRIM(s)**: Elimina los espacios iniciales y finales de un texto.
- **SUBSTRING(s, offset, length)**: Retorna un substring de un texto. El **offset** empieza a contar desde 1 y no desde 0.
- **LENGTH(s)**: Calcula la longitud de un texto.
- **ABS**: Calcula el valor absoluto de un número.
- **SQRT**: Calcula la raíz cuadrada del número
- Operador "**||**" : Permite concatenar texto.

```
SELECT p.nombre || ' ' || p.ape1 || ' ' || p.ape2 FROM Profesor p
WHERE Id=1001
```

- Ordenación.

Como en SQL también es posible ordenar los resultados usando **ORDER BY**. Su funcionamiento es como en SQL.

```
SELECT p FROM Profesor p ORDER BY nombre ASC,ape1 DESC
```

Las palabras **ASC** y **DESC** son opcionales al igual que en SQL.

El uso de funciones escalares y funciones de agrupamiento en la cláusula **ORDER BY** sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula **ORDER BY**.

- Agrupaciones.

Al igual que en SQL se pueden realizar agrupaciones mediante las palabras claves **GROUP BY** y **HAVING.**

```
SELECT nombre,count(nombre) FROM Profesor p GROUP BY nombre HAVING count(nombre)>1
ORDER BY count(nombre)
```

Los nombres de profesores que se repiten mas de una vez.

El uso de funciones escalares y funciones de agrupamiento en la cláusula **HAVING** sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula **GROUP BY**.

- Subconsultas.

  HQL también soporta subconsultas como en SQL.

```
SELECT c.nombre,c.horas FROM Ciclo c WHERE c.horas >
(SELECT AVG(c2.horas) FROM Ciclo c2)
Ciclos que duran más horas que la media de duración de todos los ciclos
```

- Combinaciones (JOIN).

  En la clausula from podemos incluir múltiples clases, lo que causa un producto cartesiano o una unión "cruzada" (cross join).

```
from Profesor as profesor, Ciclo as ciclo
```

Para poder relacionar los profesores con los ciclos que imparten tenemos la opción de especificar la condición de combinación, de forma similar a como se hace en SQL.

```
from Profesores as profesor, Ciclo as ciclo
where profesor = ciclo.profesor
```

También podemos utilizar la clausula JOIN:

```
from Profesor as profesor INNER JOIN profesor.ciclo as ciclo
```

La palabra reservada INNER e opcional, también es posible incluir la clausula *on condición.*

Ejemplo:

```
from Profesor as profesor INNER JOIN Ciclo as ciclo on
```

Los tipos de uniones soportadas se tomaron prestados de ANSI SQL:

- inner join.
- left outer join.
- right outer join.
- full join.

• Parámetros nombrados.

En ejemplos anteriores incluíamos directamente el valor de los parámetros del **WHERE**. Eso nos llevaría desde Java a tener un código similar al siguiente:

```
String nombre="ISIDRO";
String ape1="CORTINA";
String ape2="GARCIA";
Query query = session.createQuery("SELECT p FROM Profesor p where nombre='" + nombre +"' AND
ape1='"
List<Profesor> profesores = query.list();
for (Profesor profesor : profesores) {
System.out.println(profesor.toString());
}
```

Hibernate soporta parámetros nombrados en sus consultas HQL. Esto facilita la escritura de consultas HQL que aceptan la entrada del usuario. A continuación, se muestra la sintaxis simple de usar parámetros nombrados:

```
String nombre="ISIDRO";
String ape1="CORTINA";
String ape2="GARCIA";
Query query = session.createQuery("SELECT p FROM Profesor p where nombre=:nombre +"' AND
ape1=:ape1 + List<Profesor> profesores =
query.setParameter(1,nombre).setParameter(2,ape1).setParameter(3,ape2
for (Profesor profesor : profesores) {
System.out.println(profesor.toString());
}
```

La asignación de valores a los parámetros se realiza con el método **setParameter**, los métodos **setString**, **setByte**, etc. están obsoletos. También podemos utilizar parámetros con ?.

En la siguiente dirección Web, encontrarás una amplia información sobre el lenguaje de consulta HQL. Tutorial de HQL

11.2. Consultas por criterios.

Hibernate tiene el objeto **Query** que nos da acceso a todas las funcionalidades para poder leer objetos desde la base de datos. Veamos ahora un sencillo ejemplo de cómo funciona y posteriormente explicaremos más funcionalidades de la clase **Query**.

```
Query query = session.createQuery("SELECT p FROM Profesor p");
Lis<Profesor> profesores = query.list();
or (Profesor profesor : profesores)
{   System.out.println(profesor.toString());
}
```

Lanzar una consulta con Hibernate es bastante simple. Usando la **session** llamamos al método **createQuery(String queryString)** con la consulta en formato HQL y nos retorna un objeto **Query** (Línea 1). Después, sobre el objeto **Query** llamamos al método **list()** que nos retorna una lista de los objetos que ha retornado (Línea 2).

Por último, en las líneas de la 4 a la 6 podemos ver cómo usar la lista de objetos **Profesor,** aunque este código ya es simplemente el uso de la clase **java.util.List** que no tiene nada que ver con Hibernate.

1. Query

Pasemos ahora a ver las funcionalidades de la clase **Query.**

A. List()

• Lista de Objetos:

El método **list()** nos retorna una lista con todos los objetos que ha retornado la consulta. En caso de que no se encuentre ningún resultado se retornará una lista sin ningún elemento.

```
Query query = session.createQuery("SELECT p FROM Profesor p");
List<Profesor> profesores = query.list();
for (Profesor profesor : profesores) {
System.out.println(profesor.toString())
}
```

• Lista de Array de Objetos:

Las consultas pueden devolver datos escalares en vez de clases completas, por ejemplo, **SELECT p.id,p.nombre FROM Profesor p**.

En la consulta podemos ver cómo en vez de retornar un objeto Profesor se retorna únicamente el código del profesor y su nombre.

En estos casos el método **list()** retorna una lista con una Array de objetos con tantos elementos como propiedades hayamos puesto en la **SELECT.**

Veamos ahora un ejemplo:

```
Query query = session.createQuery("SELECT p.id,p.nombre FROM Profesor p");
List<Object[]> listDatos = query.list();
for (Object[] datos : listDatos)
{
System.out.println(datos[0] + "--" + datos[1]);
}
```

En la línea 1 vemos cómo se crea el objeto **Query** con la consulta de datos escalares.

En la línea 2 se ve que el método **list()** retorna una lista de array de Objetos. Es decir '**List<Object[]>**'.

En la línea 3 se inicia el bucle para recorrer cada una de las filas de datos escalares.

En la línea 5 finalmente se accede a los 2 datos de cada fila mediante datos[0] y datos[1].

• Lista de Objeto:

Hay otro caso cuando hay una única columna en el SELECT de datos escalares. Es ese caso, como el array a retornar dentro de la lista solo tendría un elemento, no se retorna una lista de arrays **List<Object[]>** sino únicamente una lista de elementos **List<Object>**.

Si modificamos la anterior consulta de forma que sólo se retorne el nombre, el código quedará de la siguiente forma:

```
Query query = session.createQuery("SELECT p.nombre FROM Profesor p");
List<Object> listDatos = query.list();
for (Object datos : listDatos)
{
System.out.println(datos);}
```

En la línea 1 ahora la consulta sólo tiene un único dato escalar.

En la línea 2 el método list() ya no retorna un List<Object[] sino un List<Object>.

En la línea 3 se inicia el bucle para recorrer cada una de las filas de datos escalares, pero ahora el tipo es Object en vez de Object[].

En la línea 5 finalmente se accede al dato sin el índice del array ya que ha dejado de serlo.

B. uniqueResult()

En muchas ocasiones una consulta únicamente retornará cero o un resultado. En ese caso es poco práctico que nos retorne una lista con un único elemento. Para facilitarnos dicha tarea Hibernate dispone del método **uniqueResult().**

Este método retornará directamente el único objeto que ha obtenido la consulta. En caso de que no encuentre ninguno se retornará **null**.

```
Profesor profesor = (Profesor) session.createQuery("SELECT p FROM Profesor p WHERE id=1001").uniqueResult();
System.out.println("Profesor con Id 1001=" + profesor.getNombre());
```

Vemos como gracias al método **uniqueResult()** se simplifica el código, aunque siempre se debe comprobar si ha retornado o no **null**.

Al igual que ocurre con **list()** el método **uniqueResult()** puede retornar tanto un objeto de una entidad , un array de objetos escalares **Object[]** o un único objeto escalar **Object**.

Si el método **uniqueResult()** retorna más de un resultado se producirá la excepción: **org.hibernate.NonUniqueResultException: query did not return a unique result.**

C. Paginación

La paginación es parte fundamental de cualquier aplicación ya que una consulta puede tener miles de resultados y no queremos mostrarlos todos a la vez.

Para conseguir paginar el resultado de una consulta la clase **Query** dispone de los siguientes métodos:

- **setMaxResults(int maxResults):** Establece el número máximo de objetos que van a retornarse.
- **setFirstResult(int firstResult):** Establece el primer de los objetos que se van a retornar.

Al realizar la paginación son necesarios al menos 2 valores:

- El tamaño de la página.
- El número de la página a mostrar.

Con esos 2 valores hemos creado el siguiente código Java que muestra una única página en función del **tamañoPagina** y **paginaAMostrar**.

```
 1: int tamanyoPagina = 10;
 2: int paginaAMostrar = 7;
 3:
 4: Query query = session.createQuery("SELECT p FROM Profesor p Order By p.id");
 5: query.setMaxResults(tamanyoPagina);
 6: query.setFirstResult(paginaAMostrar * tamanyoPagina);
 7: List<Profesor> profesores = query.list();
 8:
 9: for (Profesor profesor : profesores) {
10:     System.out.println(profesor.toString());
11: }
```

Las líneas 1 y 2 establecen los valores necesarios para poder mostrar la página, que son el tamaño de la página y el número de la página a mostrar.

En la línea 4 se crea la **Query**.

En la línea 5 se llama al método **setMaxResults(int maxResults)** para indicar que sólo se retornen como máximo tantos objetos como tamaño tiene la página.

En la línea 6 se llama al método **setFirstResult(int firstResult)** para indicarle cuál es el primer objeto a retornar. Este valor coincidirá con el primer objeto de la página que se quiere mostrar. Para calcular dicho valor se multiplica el número de la página a mostrar por el tamaño de página. Para que esta fórmula funcione el número de página debe empezar por 0, es decir, que la primera página deberá ser la número 0, la segunda la número 1, y así sucesivamente.

Por fin, en la línea 7 se obtienen únicamente los resultados de la página solicitada.

Por último, en las líneas 9,10 y 11 se muestran los datos.

El mayor problema que tiene la paginación es determinar el número de páginas para poder mostrárselo al usuario, para saber el número de páginas es necesario conocer el número de objetos que retorna la consulta y la forma mas rápida y sencilla es mediante una consulta que las cuente.

El siguiente código Java calcula el número de páginas de la consulta.

```
long numTotalObjetos = (Long) session.createQuery("SELECT count(*) FROM Profesor p").uniqueResult();
int numPaginas =(int) Math.ceil((double)numTotalObjetos / (double)tamanyoPagina);
```

En la línea 1 realizamos la consulta de **count(*)** para obtener el número de objetos que retorna la consulta.

En la línea 3 de divide el número total de objetos entre el tamaño de la página obteniéndose el número total de páginas.

Al hacer la división para calcular el número de páginas es necesario hacer el cast de los 2 valores a double ya que si no Java automáticamente redondea el resultado a un valor entero con lo que el valor que se le pasa a **ceil** ya no será el correcto.

2. Consultas con nombre

En cualquier libro sobre arquitectura del software siempre se indica que las consultas a la base de datos no deberían escribirse directamente en el código, sino que deberían estar en un fichero externo para que puedan modificarse fácilmente.

Hibernate provee una funcionalidad para hacer esto mismo de una forma sencilla. En cualquier fichero de mapeo de Hibernate se puede incluir anotaciones **@NamedQuery and @NamedQueries Annotations** con las consultas HQL que deseamos lanzar.

En el siguiente ejemplo se ha definido una **query** nombrada para la clase Profesor:

```
@NamedQuery(name="Profesores.findAll", query="SELECT p FROM Profesor p")
```

La anotación @NamedQuery contiene cuatro parámetros, dos de los cuales son obligatorios y dos opcionales. Los dos parámetros requeridos, ***name*** y ***query*** definen el nombre de la consulta y la propia cadena de consulta y se muestran arriba. Los dos elementos opcionales, ***lockMode*** y ***hints***, proporcionan un reemplazo estático para los métodos **setLockMode** y **setHint.**

La entidad Profesor quedaría:

```
@Entity
@NamedQuery(name="Profesores.findAll", query="SELECT p FROM Profesor p")
public class Profesor {
  ...
}
```

Para añadir varias consultas nombradas , tiene que haber una consulta nombrada por cada una de ellas y estas dentro de una anotación envoltorio @NamedQueries:

```
@Entity
@NamedQueries({
@NamedQuery(name="Profesores.findAll", query="SELECT p FROM Profesor p")
@NamedQuery(name="Profesores.ByName", query="SELECT p FROM Profesor p where p.nombre =:nombre and p.ape1=:ape1 and p.ape2=:ape2")
})
public class Profesor {
  ...
}
```

Estas anotaciones tienen los siguientes datos:

- **name**: Este atributo define el nombre de la consulta. Es el nombre que posteriormente usaremos desde el código Java para acceder a la consulta.

- **query** es la consulta en formato HQL que ejecutará Hibernate.

- :nombre, :ape1, :ape2 son parámetros nombrados, también se podrían especificar con parámetros posicionales ? como en JDBC:

```
"SELECT p FROM Profesor p where p.nombre =? and p.ape1=? and p.ape2=?"
```

3. Código Java

Para hacer uso de una consulta con nombre usaremos el método **getNamedQuery(String qyeryString)** en vez de **createQuery(String queryString)** para obtener el objeto **Query**. Por lo tanto, sólo se ha modificado la línea 1 y el resto del código queda exactamente igual.

```
1: Query query = session.getNamedQuery("Profesores.findAll");
2: List<Profesor> profesores = query.list();
3: for (Profesor profesor : profesores) {
4:     System.out.println(profesor.toString());
5: }
```

Como podemos ver el uso de consultas con nombre es muy sencillo al usar Hibernate.

```
 1: String nombre="ISIDRO";
 2: String ape1="CORTINA";
 3: String ape2="GARCIA";
 4:
 5: Query query = session.getNamedQuery(Profesores.ByName)
 6: query.setString("nombre",nombre);
 7: query.setString("ape1",ape1);
 8: query.setString("ape2",ape2);
 9:
10: List<Profesor> profesores = query.list();
11: for (Profesor profesor : profesores) {
12:     System.out.println(profesor.toString());
13: }
```

12. Gestión de transacciones.

Una transacción es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, en forma indivisible o atómica.

Para la gestión de transacciones en Hibernate, no se produce bloqueo de objetos en la memoria. La aplicación puede esperar el comportamiento definido por el nivel de aislamiento de sus transacciones de las bases de datos. Gracias a la **Session**, la cual también es un caché con alcance de transacción.

Para realizar con éxito la gestión de transacciones, estas se van a basar en el uso del objeto **Session**.

El objeto **Session** se obtiene a partir de un objeto **SessionFactory**, invocando el método **openSession**. Un objeto **SessionFactory** representa una configuración particular de un conjunto de metadatos de mapping objeto/relacional.

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // Utilizar la Session para saveOrUpdate/get/delete/...tx.commit();
} catch (Exception e) {
   if (tx != null) {
      tx.rollback();
      throw e;
   }
} finally {
   session.close();
}// Al finalizar la aplicación ...HibernateUtil.shutdown( );
```

Cuando se crea el objeto **Session**, se le asigna la conexión de la base de datos que va a utilizar. Una vez obtenido el objeto **Session**, se crea una nueva unidad de trabajo (Transaction) utilizando el método **beginTransaction**. Dentro del contexto de la transacción creada, se pueden invocarlos métodos de gestión de persistencia proporcionados por el objeto **Session**, para recuperar, añadir, eliminar o modificar el estado de instancias de clases persistentes. También se pueden realizar consultas. Si las operaciones de persistencia no han producido ninguna excepción, se invoca el método **commit** de la unidad de trabajo para confirmar los cambios realizados. En caso contrario, se realiza un **rollback** para deshacer los cambios producidos. Sobre un mismo objeto Session pueden crearse varias unidades de trabajo. Finalmente se cierra el objeto **Session** invocando su método **close**.