UNIDAD DIDÁCTICA 3

HERRAMIENTAS DE MAPEO OBJETO RELACIONAL

MÓDULO PROFESIONAL: ACCESO A DATOS

Índice

RESUMEN INTRODUCTORIO 2
INTRODUCCIÓN 2
CASO INTRODUCTORIO 2
1. INTRODUCCIÓN AL ORM 4
1.1 Concepto de mapeo objeto-relacional (ORM, Object-Relational Mapping) 4
1.2 Características de las herramientas ORM 5
1.3 Herramientas ORM más utilizadas 7
2. INSTALACIÓN, CONFIGURACIÓN Y USO DE UNA HERRAMIENTA ORM. HIBERNATE. 11
2.1 Instalación y configuración de Hibernate 12
2.2 Estructura de un fichero de mapeo. Elementos, propiedades 16
2.3 Clases persistentes 19
2.4 Sesiones y estados de un objeto 22
2.5 Gestión de transacciones 25
2.6 Carga, almacenamiento y modificación de objetos 26
2.7 Relaciones entre clases 41
2.8 Consultas SQL (Standard Query Language) y HQL 45
RESUMEN FINAL 48

RESUMEN INTRODUCTORIO

En esta unidad se introducirán los conceptos sobre ORM, herramientas que nos permiten abstraernos de la capa más física de interacción con la base de datos, incluyendo una capa intermedia que permitirá el mapeo entre base de datos y objetos de nuestra aplicación.

Usaremos Hibernate como el paquete de herramientas que nos permite realizar todos los pasos necesarios, desde la configuración hasta la apertura de una sesión, el trabajo de interacción con nuestros datos y el cierre de los estados, y todo ello a través de capas intermedias que se abstraen de la conectividad contra una base de datos determinada.

Por último, introduciremos el concepto de lenguajes de consultas generalista y específico de las herramientas.

INTRODUCCIÓN

Insistimos en que los sistemas hoy en día han evolucionado enormemente, provocando que nuestras aplicaciones crezcan y dependan de más y más sistemas y bases de datos cambiantes.

Necesitamos de herramientas que nos permitan mapear nuestro negocio y la lógica programada en interacciones con la base de datos. Necesitamos, por lo tanto, poder abstraernos de esa capa más física que es la conexión contra la base de datos y de su lenguaje específico.

Veremos cómo con las herramientas ORM, en cualquier lenguaje de programación y en particular en Java, podremos realizar esa separación dentro de nuestra aplicación entre lógica y datos, proporcionándonos una capa añadida de abstracción, provocando más complejidad dentro de nuestra aplicación, pero, a su vez, produciendo una flexibilidad y una escalabilidad que de forma directa no la podríamos tener.

CASO INTRODUCTORIO

Acabas de entrar a trabajar en una pequeña empresa donde estáis como desarrolladores tu jefe y tú. La empresa, dedicada a la gestión deportiva de varios clubes, pretende seguir creciendo y realizar un cambio tecnológico de su base de datos, que actualmente era Access, para cambiarla a una base de datos más potente, tipo MySQL.

Tu jefe se plantea muchas cuestiones para que el desarrollo pueda ser más modular al existente actualmente, ya que los diseños con bases de datos relacionales son muy dependientes del esquema tradicional de base de datos (tablas), por lo que planteáis la adaptación de la base de datos a la forma en la que se trabaja con la programación orientada a objetos (ORM). Para ello, vas a adaptar el diseño a una base de datos en Hibernate, ya que el software está desarrollado en Java.

Al finalizar la unidad, conocerás el concepto del mapeo objeto-relacional, sus características y cuáles son sus herramientas más utilizadas, así como el proceso de instalación, configuración y uso de esas herramientas para estructurar ficheros, gestionar transacciones y otras muchas aplicaciones.

1. INTRODUCCIÓN AL ORM

Dentro de las preguntas que nos hacíamos al inicio del proyecto, nos encontramos con la posibilidad de buscar la existencia de herramientas que nos permitan trabajar de forma independiente a la base de datos que estemos manejando. Sabes que existen algunas herramientas para el desarrollo de ORM en Java, por lo que decides hacerlo con Hibernate.

A pesar de que los sistemas de conexión a las bases de datos relacionales como ODBC o JDBC permiten una gran expresividad, convirtiéndolas en herramientas muy potentes, a la vez son herramientas de bajo nivel que, desgraciadamente, necesitan un número importante de líneas de código para poder cubrir las necesidades de cada aplicación.

Los desarrolladores tienen que diseñar dos modelos (relacional y orientado a objetos) y tienen que usar herramientas de traducción entre ellos, con un coste realmente importante.

Las herramientas de mapeo objeto-relacional (ORM) intentan aprovechar la madurez y la eficiencia de las bases de datos relacionales, minimizando tanto como sea posible el desfase objeto-relacional. Se trata de bibliotecas y marcos de programación que definen un formato para expresar múltiples situaciones de transformación entre ambos paradigmas. Esto les permite automatizar procesos bidireccionales de un sistema al otro.

En cierta forma, podríamos decir que implementan una base de datos orientada a objetos virtual, porque aportan características propias del paradigma OO, pero el sustrato donde se acaban almacenando los objetos es un SGBD relacional.

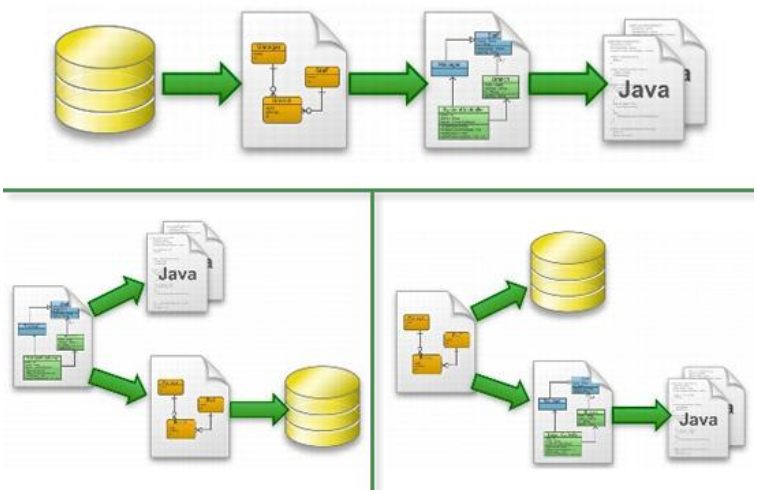
1.1 Concepto de mapeo objeto-relacional (ORM, Object-Relational Mapping)

El **mapeo de objeto-relacional** se puede definir como una técnica que permite convertir datos del sistema de tipos utilizado en el lenguaje OO con el que se desarrolla la aplicación al utilizado en las bases de datos relacionales. Para conseguir este objetivo, las herramientas crean una base de datos virtual orientada a objetos sobre la base de datos relacional. Así, se permite la utilización de las características del paradigma orientado a objetos.

Conviene recordar que las bases de datos relacionales trabajan con tipos de datos primitivos, por lo que, si se está desarrollando una aplicación utilizando un lenguaje OO, no se podrán almacenar dichos objetos en la base de datos de manera directa, sino que

estos deben ser convertidos antes en registros que, además, normalmente afectan a varias tablas.

Cuando se quieran recuperar esos objetos de la base de datos, tampoco se podrá hacer directamente, sino que habrá que realizar el proceso inverso al anterior, es decir, convertir los registros en objetos. Para estos procesos es para lo que se utilizan las herramientas ORM.



Esquema de funcionamiento ORM
Fuente: elaboración propia

1.2 Características de las herramientas ORM

Podemos resumir en tres las características de las herramientas ORM:

1. **Técnicas de mapeo:** entre las técnicas que estas herramientas usan para plasmar los mapas O-R (Objeto-Relación) distinguiremos las que incrustan las definiciones dentro del código de las clases y las que almacenan las definiciones en ficheros independientes. Las primeras suelen ser técnicas muy vinculadas al lenguaje de programación, así, por ejemplo, en C++ se suelen usar macros y en Java se utilizan anotaciones. Las técnicas de definición basadas en ficheros independientes del código suelen sustentarse en XML porque es un lenguaje muy expresivo y fácilmente extensible.

2. **Lenguaje de consulta:** el lenguaje de consulta más utilizado por la mayoría de las herramientas es el lenguaje llamado OQL (Object Query Language) o una variante del mismo. Se trata de un lenguaje especificado por el ODMG. Presenta cierta similitud con SQL dado que ambos son lenguajes de interrogación no procedimental, pero el OQL está totalmente orientado a objetos, es decir, los

componentes de la consulta se expresan usando la sintaxis propia de los objetos y los resultados obtenidos devuelven objetos o colecciones de objetos.

3. **Técnicas de sincronización:** la sincronización con la base de datos es seguramente uno de los aspectos más críticos de las herramientas de mapeo. Suelen ser procesos bastante complejos, donde encontramos implicadas sofisticadas técnicas de programación destinadas a descubrir los cambios que vayan sufriendo los objetos, a crear e inicializar las nuevas instancias que haya que posar en juego dentro de la aplicación de acuerdo con los datos almacenados, o también a extraer la información de los objetos para revertirla a las tablas del SGBD.

Son muchas las **ventajas** de los ORM, entre las que se pueden destacar:

- Rapidez de desarrollo.
- Abstracción de la base de datos.
- Seguridad.
- Reutilización y mantenimiento del código.
- Cuentan con un lenguaje de consulta específico.

Sin embargo, también es necesario citar las **desventajas** que estas herramientas conllevan, entre las que estarían:

- Necesidad de dedicar mucho tiempo y esfuerzo al aprendizaje de las herramientas, ya que suelen ser complejas.

- Las aplicaciones suelen ser más lentas, ya que las consultas sobre la base de datos implican realizar transformaciones al lenguaje que utiliza la herramienta, leer los registros y, finalmente, crear los objetos.

ENLACE DE INTERÉS

Profundiza sobre qué es ORM, sus aplicaciones y usos:

1.3 Herramientas ORM más utilizadas

Las herramientas de mapeo objeto-relacional (ORM) intentan aprovechar la madurez y la eficiencia de las bases de datos relacionales, minimizando tanto como sea posible el desfase objeto-relacional.

Se trata de bibliotecas y marcos de programación que definen un formato para expresar múltiples situaciones de transformación entre ambos paradigmas. Esto nos permite automatizar procesos de un sistema al otro.

En cierta forma, podríamos decir que implementan una base de datos orientada a objetos virtual porque aportan características propias del paradigma OO, pero el sustrato donde se acaban almacenando los objetos es un SGBD relacional.

1. Doctrine (PHP).

Doctrine es una herramienta ORM escrita en PHP que proporciona una capa de persistencia para objetos PHP que se sitúa encima del SGBD y permite trabajar con los esquemas de bases de datos como si se tratara de objetos en vez de tablas y registros.

Esto significa que, por ejemplo, si se dispone de una tabla llamada usuarios, se autogenerará una clase llamada Usuarios cuyas propiedades serán las columnas de la tabla.

Entre las principales características de Doctrine cabe citar el escaso nivel de configuración necesario para empezar a trabajar. Además, no es necesario que el desarrollador genere o mantenga complejos esquemas XML de bases de datos.

Por último, y al igual que ocurría con el ORM anterior, dispone de un lenguaje propio denominado Doctrine Query Language (DQL) que permite escribir consultas.

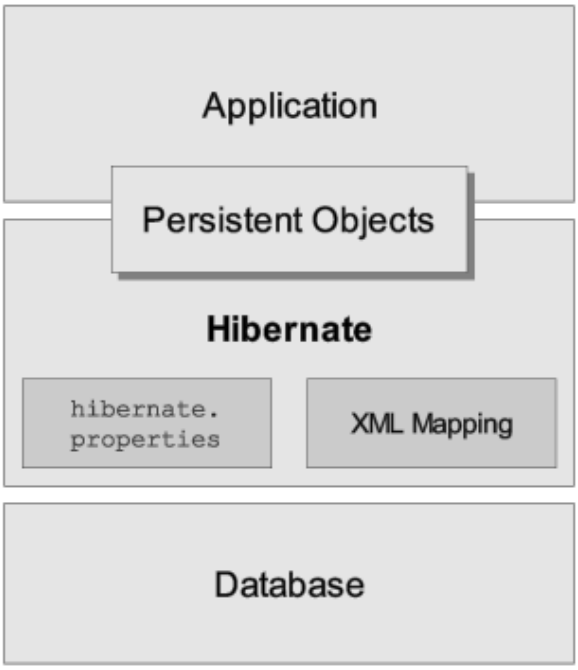
ENLACE DE INTERÉS

La web oficial ofrece completa información sobre Doctrine:

2. Hibernate (Java).

Hibernate es una herramienta de mapeo objeto-relacional escrita para la plataforma Java, cuyo objetivo es facilitar el mapeo de atributos entre una base de datos relacional y el modelo de objetos generado para una aplicación. Todo esto lo consigue mediante archivos declarativos escritos en XML.

Como se ha comentado anteriormente, en realidad, su funcionamiento se podría equiparar con el de un traductor que se sitúa entre la aplicación y la base de datos, y que permite abstraer el diseño de la base de datos para poder centrarse así solo en las funcionalidades de la aplicación.



Modelo de capas de Hibernate

Fuente: elaboración propia

ENLACE DE INTERÉS

La herramienta es muy grande y extensa, pero es tremendamente interesante visitar la web del fabricante:

VÍDEO DE INTERÉS

Aquí tienes una introducción sobre Hibernate y la importancia de la persistencia:

3. Entity Framework (C#).

Entity Framework (EF) es una herramienta ORM desarrollada por Microsoft para la plataforma .NET, lanzada por primera vez en 2008.

Entre sus principales características hay que destacar que incluye un potente motor de consultas basado en LINQ (Language Integrated Query), permitiendo a los desarrolladores escribir consultas de manera intuitiva en el lenguaje C#.

Con características como el enfoque de código primero, el enfoque de base de datos primero y la migración automática de esquemas.

ENLACE DE INTERÉS

Puedes ampliar información visitando la web oficial:

4. SQLAlchemy (Python).

SQLAlchemy es una de las herramientas ORM más usadas en el ecosistema Python. Es de código abierto y está disponible bajo la licencia MIT, lo que permite su uso y modificación libre.

Esta biblioteca ofrece un conjunto completo de herramientas para trabajar con bases de datos relacionales, proporcionando tanto un ORM como una interfaz de SQL de bajo nivel. Además, SQLAlchemy es capaz de integrarse con otras herramientas y frameworks en el ecosistema Python (Flask, Django).

ENLACE DE INTERÉS

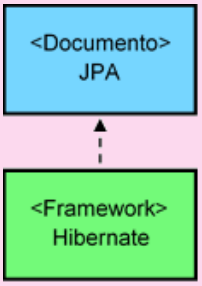
Aquí puedes acceder a su web oficial:

EJEMPLO PRÁCTICO

Nos planteamos un cambio en nuestra empresa, pasando de un modelo de desarrollo orientado a objetos con código insertado de conexión a base de datos a un uso de herramientas ORM.

El primer paso es seleccionar la herramienta que queremos usar. Nuestro jefe nos plantea una primera revisión de JPA e Hibernate. ¿Qué elegir? ¿Por qué?

1. El primer paso será visitar las páginas oficiales: https://hibernate.org/ y https://jakarta.ee/specifications/persistence/2.2/apidocs/.
2. La primera diferencia que encontramos es muy grande, ya que JPA no es un framework como Hibernate. No es una implementación, sino que es parte de un documento de Java EE/Jakarta.
3. Hibernate sí es una implementación real que tiene como referente, a su vez, la especificación de JPA. Por lo tanto, no existe ninguna disyuntiva en la elección de uno u otro, ya que, si pretendemos una implementación real, necesitamos elegir Hibernate directamente.



Relación JPA e Hibernate.
Fuente: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

2. INSTALACIÓN, CONFIGURACIÓN Y USO DE UNA HERRAMIENTA ORM. HIBERNATE

Una vez elegida Hibernate como la herramienta para el cambio tecnológico dentro de la aplicación, llega la parte del desarrollo de una prueba antes de acometer el proyecto final.

Una vez, por lo tanto, acotado el ámbito de trabajo, crearás un proyecto Maven en Eclipse y configurarás el fichero para importar las dependencias necesarias para utilizar Hibernate.

Una vez hecho esto, ya te puedes centrar en la programación de la gestión deportiva. Para ello, programarás las entidades (que se traducirán en el equivalente a tabla en la BBDD), los managers (para hacer el CRUD de las entidades) y la clase Principal que interactúe con el ORM.

En este apartado se presentará el procedimiento para instalar y configurar el ORM Hibernate. El proceso de instalación puede variar ligeramente según la versión usada, por lo que aquí se verán los pasos genéricos.

Para la configuración y gestión de los diferentes paquetes, usaremos un gestor de librerías denominado Maven.

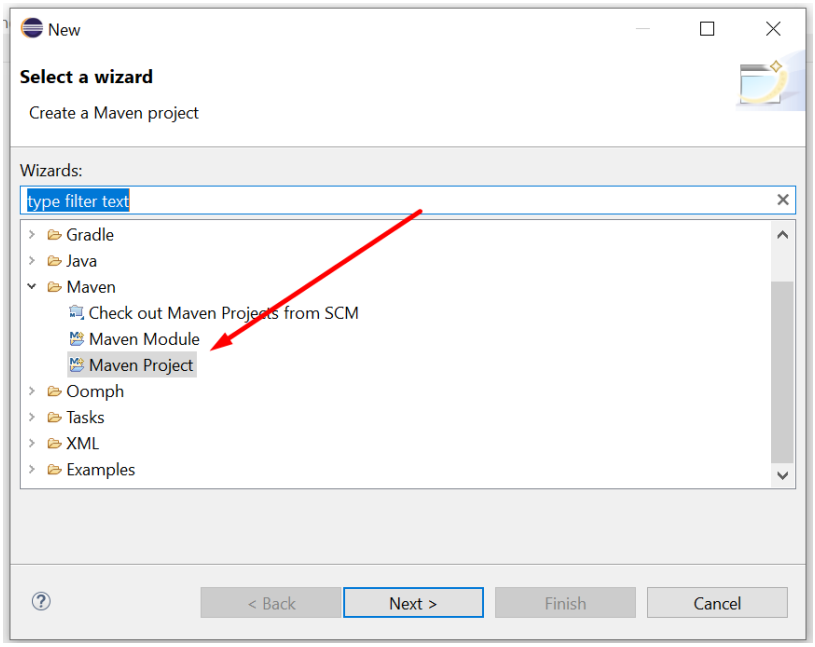
Hoy en día, los proyectos han crecido exponencialmente al necesitar cada vez más de librerías externas y dependencias con otros proyectos. La gestión manual se hace muy grande, por ese motivo, el uso de un gestor de paquetes como Maven o Gradle es perfecto para la gestión de nuestros proyectos.

ENLACE DE INTERÉS

Conoce más sobre Maven visitando su web oficial:

2.1 Instalación y configuración de Hibernate

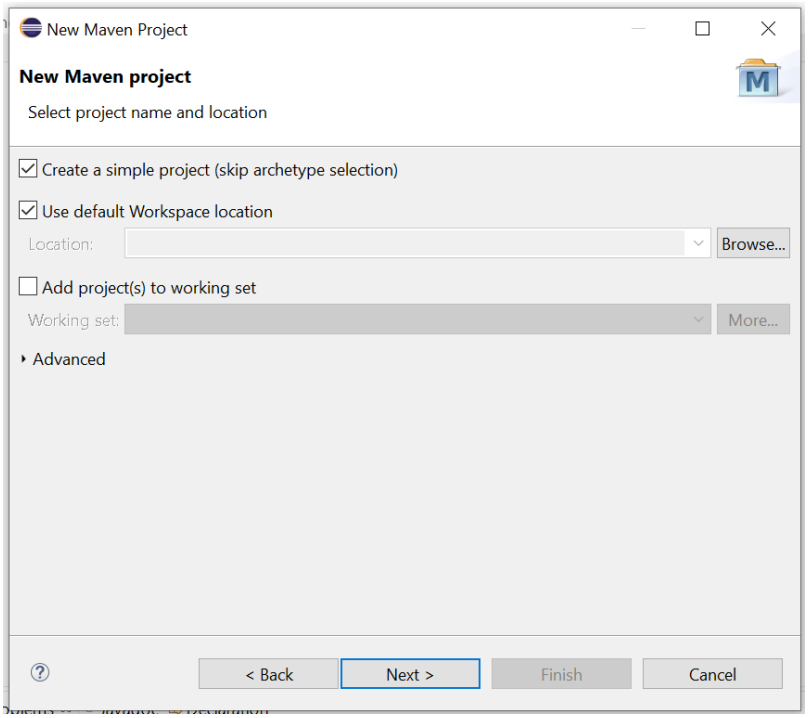
Crearemos un nuevo proyecto con Eclipse, teniendo en cuenta que la versión con la que estamos trabajando de Eclipse es Eclipse 4.15.



Creación de proyecto con Maven

Fuente: elaboración propia

Crearemos un proyecto simple, teniendo como referencia el workspace local.



Creación de proyecto con Maven

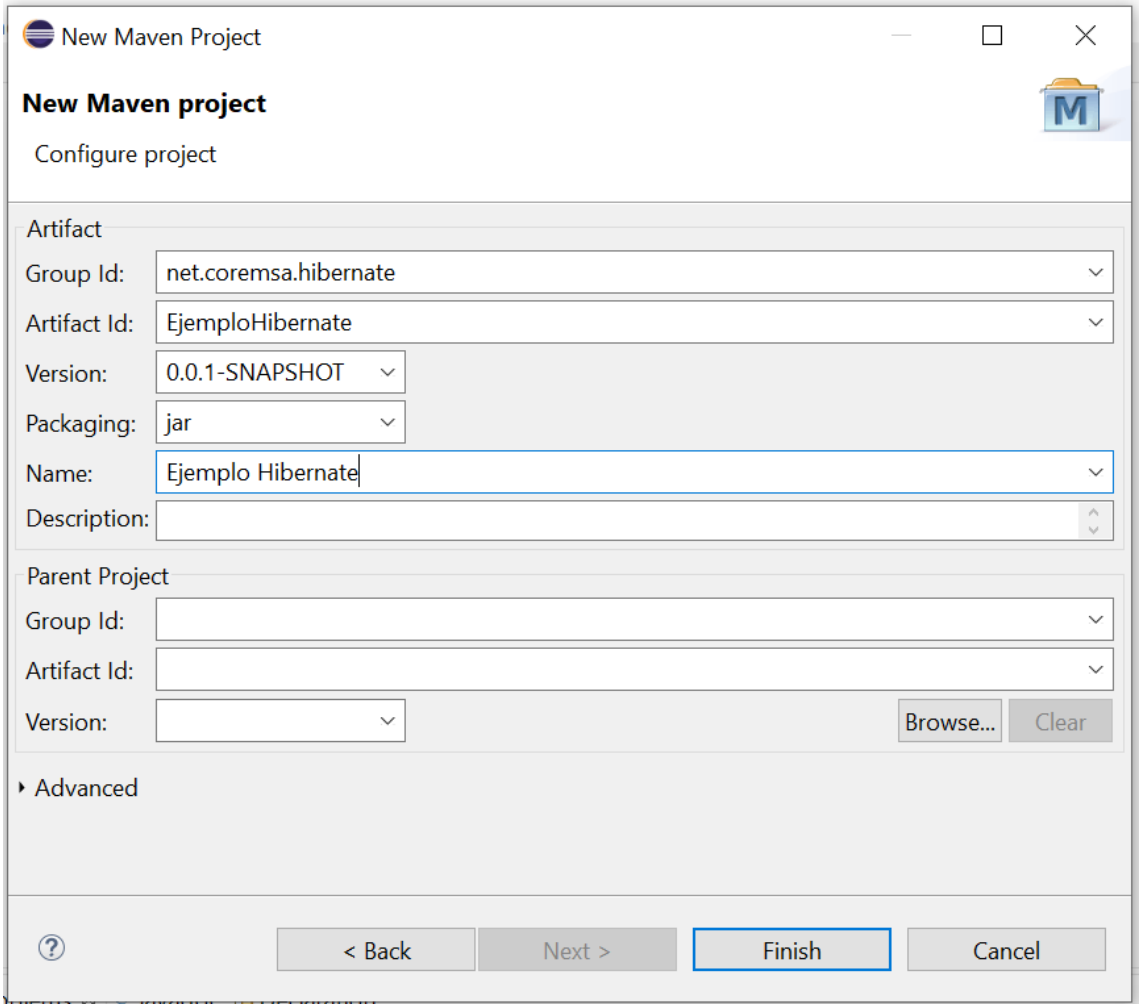
Fuente: elaboración propia

En las definiciones del proyecto Maven (pom.xml) elegiremos los siguientes parámetros:

- **Group Id:** el Group Id (identificador del grupo) es un identificador único que se utiliza para agrupar proyectos o módulos relacionados bajo un mismo nombre. Pondremos: net.coremsa.hibernate

- **Artifact Id:** el Artifact Id (identificador del artefacto) es un identificador único para el proyecto o módulo dentro del grupo especificado por el Group Id. Pondremos: EjemploHibernate

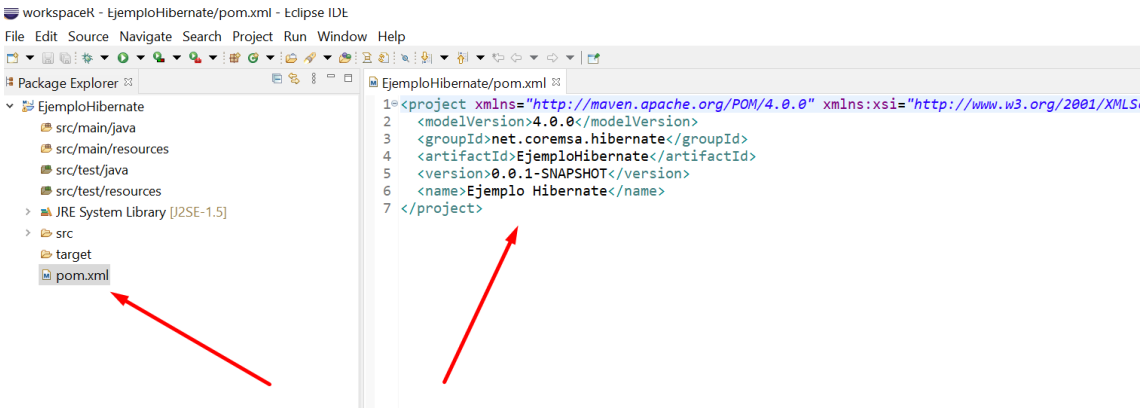
- **Name:** el Name es un campo opcional que proporciona una descripción legible para las personas del proyecto o módulo. Pondremos: Ejemplo Hibernate.



Parámetros con Maven

Fuente: elaboración propia

Esto nos genera un nuevo proyecto gestionado con Maven.



Fichero de configuración Maven

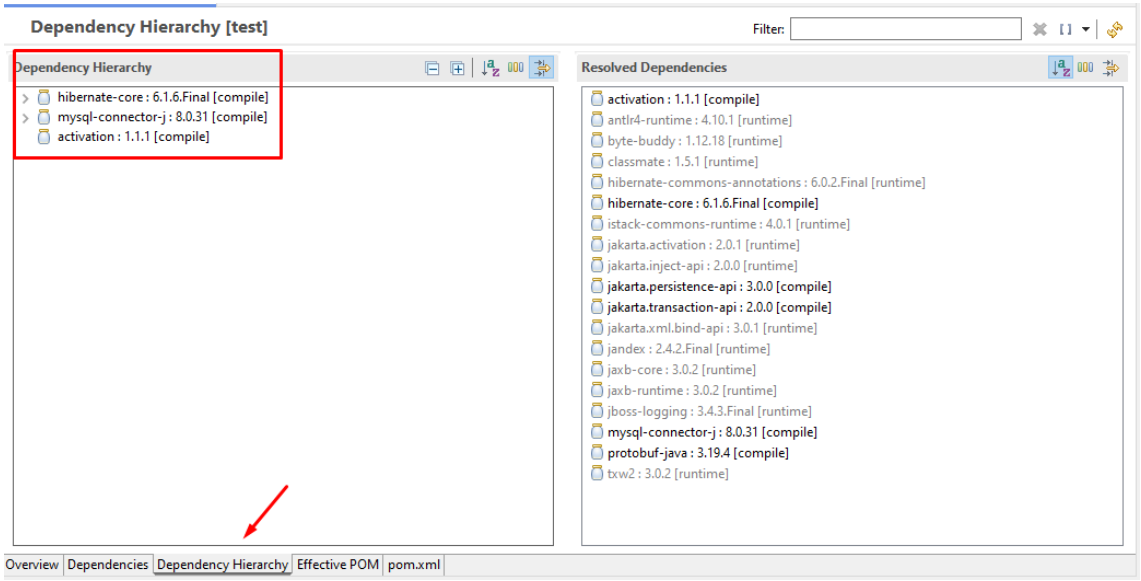
Fuente: elaboración propia

Cambiaremos el entorno de ejecución JRE por el que tengamos instalado en el sistema, haciendo clic derecho sobre JRE System y cambiándolo al adecuado, a través de las Properties.

Por último, añadiremos el JDBC de MYSQL y el de Hibernate, buscando las versiones en el repositorio de Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>net.coremsa.hibernate</groupId>
  <artifactId>EjemploHibernate</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Ejemplo Hibernate</name>

  <dependencies>
            <dependency>
                  <groupId>org.hibernate</groupId>
                  <artifactId>hibernate-core</artifactId>
                  <version>6.1.6.Final</version>
            </dependency>
            <dependency>
                  <groupId>mysql</groupId>
                  <artifactId>mysql-connector-java</artifactId>
                  <version>8.0.31</version>
            </dependency>
            <dependency>
                  <groupId>javax.activation</groupId>
                  <artifactId>activation</artifactId>
                  <version>1.1.1</version>
            </dependency>
      </dependencies>
</project>
```

Una vez que guardemos el fichero, Maven realiza la descarga e incorporación automática de las librerías necesarias que podemos ver a través de la interfaz gráfica del fichero pom.xml.

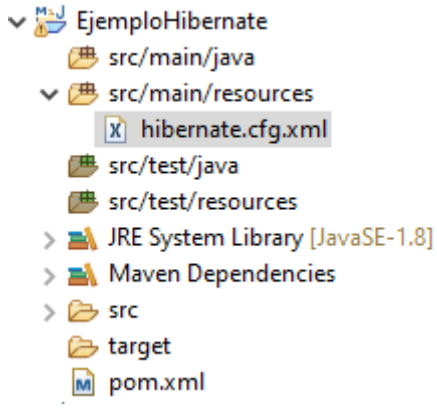


Dependencias

Fuente: elaboración propia

Para la configuración de la conexión con nuestra base de datos, generaremos un fichero de configuración XML con el nombre hibernate.cfg.xml dentro de la carpeta src/main/resources con el siguiente código.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
name="connection.url">jdbc:mysql://localhost:3306/ejemplo</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">create</property>

    <mapping class="net.coremsa.hibernate.Book" />

  </session-factory>
</hibernate-configuration>
```



Fichero de Configuración Hibernate

Fuente: elaboración propia

Cabe destacar que, con estos parámetros de configuración, nos estaríamos conectando a una base de datos llamada ejemplo en un SGBD de MySQL que lo tenemos lanzado en el ordenador local y puerto 3306. Por otra parte, los parámetros de conexión son usuario:root y sin contraseña (configuración por defecto de MySQL).

Veremos más adelante qué significado tiene la etiqueta mapping.

VÍDEO DE INTERÉS

Visualiza la instalación de Hibernate con Eclipse:

2.2 Estructura de un fichero de mapeo. Elementos, propiedades

Una entidad nos va a permitir definir una nueva clase que será el reflejo en nuestra base de datos y que después, mediante un mapeado, podremos establecer esta relación entre la clase de Java y la tabla de la base de datos.

Por lo tanto, una entidad tendrá dos componentes:

- La Clase en Java.

- El fichero de mapeado definido con XML o bien las anotaciones que nos permitirán, a su vez, definir otras características de las propiedades.

El primer paso será definir la clase, que en muchas ocasiones es una clase reflejo de la tabla en la base de datos.

```
public class Book {
    private long id;
    private String title;
    private String author;
    private float price;
```

Si desarrollamos completamente esta clase, tendremos:

```
public class Book implements Serializable{
    private long id;
    private String title;
    private String author;
    private float price;

    public Book() {
    }

    public long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public float getPrice() {
        return price;
    }
```

```
    public void setPrice(float price) {
        this.price = price;
    }
}
```

Para indicar a Hibernate cuál es la correspondencia con la base de datos, podemos usar dos mecanismos, tal y como hemos dicho antes. El primero es mediante un **fichero de mapeo:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
    <class name="ejemploHibernate.Book" table="Book" >
      <id column="Id" name="id" type="integer"/>
      <property name="title" />
      <property name="author" />
    </class>
</hibernate-mapping>
```

Lo primero que aparece en estos ficheros es la etiqueta hibernate-mapping, con su atributo package, que almacena el paquete Java que contiene la clase que se va a mapear. Dentro de esta etiqueta se colocan las clases de los objetos persistentes.

A continuación, en la etiqueta class, se indica la clase, poniendo su nombre como atributo name, y la tabla que se va a convertir, en el atributo table. Es decir, se asocia una clase con una tabla de la base de datos.

En el interior de la etiqueta class, se puede encontrar la etiqueta id que se utiliza para indicar el campo que representa el atributo clave en la clase. El nombre se indica mediante el atributo name, y con el atributo column se especifica su nombre en la tabla.

Es posible indicar también el tipo de datos Java a través del atributo type. Para el campo clave, existe, además, la etiqueta generator para especificar la naturaleza de dicho campo.

Hay que indicar que, aunque es posible declarar más de un elemento class en el mismo archivo XML, no es lo recomendado, ya que el hecho de hacer un documento XML para cada clase aporta más claridad a la aplicación.

El resto de atributos se indican con la etiqueta property, asociando los nombres de los campos de la clase con nombres de columna sobre la tabla de la base de datos. Esta etiqueta cuenta con los atributos name, column y type, al igual que en el caso anterior.

PARA SABER MÁS

Aquí encontrarás más ejemplos de archivos de mapeo:

2.3 Clases persistentes

La segunda, y más común actualmente, es definir las relaciones de las clases dentro de un proyecto Hibernate usando la propia clase como fichero de mapeo al utilizar anotaciones propias de Hibernate. Por ejemplo, para considerar una clase como una Entidad y su mapeo con una tabla, usaríamos las anotaciones @Entity y @Table.

Estas anotaciones provienen del API de persistencia que proviene de la plataforma JAVA EE, que se denomina Java Persistence API, cuyas siglas son JPA. Incorporando la librería jakarta.persistence.* podremos comenzar a usar las anotaciones comentadas:

```
@Entity
@Table(name = "book")
public class Book {
```

O para seleccionar el índice:

```
@Id
    @Column(name = "book_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public long getId() {
        return id;
    }
```

La clase finalmente quedaría de la siguiente manera:

```
package net.coremsa.hibernate;

import jakarta.persistence.*;

@Entity
@Table(name = "book")
```

```
public class Book {
    private long id;
    private String title;
    private String author;
    private float price;

    public Book() {
    }

    @Id
    @Column(name = "book_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

}
```

Como vemos, no es necesario anotar el resto de propiedades, ya que Hibernate hace el mapeo de forma automática. Retomando el fichero de configuración que antes definimos con la conexión de la base de datos, ahora mismo toma sentido la línea de mapeado que introdujimos.

```
<mapping class="net.coremsa.hibernate.Book" />
```

Ya que, con esa línea, mapeamos la clase Book con la tabla Book que tenemos en nuestra base de datos ejemplo.

PARA SABER MÁS

Obtén más información sobre Hibernate y las clases persistentes en este enlace:

EJEMPLO PRÁCTICO

Queremos realizar una primera prueba con un proyecto en Hibernate. Para ello, elegimos MySQL y la tabla usuarios como primera tabla de ejemplo.

Nuestro jefe nos plantea un primer ejemplo de proyecto. ¿Cuáles son los pasos que hay que realizar? ¿Cuál sería la primera clase para el mapeado y la configuración con la base de datos?

1. Una vez instalado Hibernate mediante Maven, comenzaríamos con el proyecto de prueba.

2. Creamos la clase de Mapeado User.

```
import jakarta.persistence.*;

@Entity(name = "User")
@Table(name = "users")
public class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String phoneNumber;

}
```

3. Crearíamos el fichero de configuración para el mapeado con la base de datos.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
name="connection.url">jdbc:mysql://localhost:3306/gym</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="show_sql">true</property>

    <mapping class="com.empresa.hibernate.User" />

  </session-factory>
</hibernate-configuration>
```

2.4 Sesiones y estados de un objeto

Una vez creada la Entidad que mapea la clase con la base de datos, necesitamos crear el "Manager" que manejará los diferentes estados del objeto Book cuando se cree. Dentro del mismo paquete Hibernate, crearemos la siguiente clase.

```
package net.coremsa.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class BookManager {
    protected SessionFactory sessionFactory;

    protected void setup() {
        // code to load Hibernate Session factory
    }

    protected void exit() {
        // code to close Hibernate Session factory
    }

    protected void create() {
        // code to save a book
    }

    protected void read() {
        // code to get a book
    }
```

```
    protected void update() {
        // code to modify a book
    }

    protected void delete() {
        // code to remove a book
    }

}
```

Con Hibernate se realizan las operaciones mediante una Session, la cual se puede obtener a través de un SessionFactory, como el que hemos creado dentro de nuestra clase.

La SessionFactory realiza las siguientes tareas:

1. Carga el fichero de configuración de Hibernate.
2. Analiza el mapeado.
3. Crea la conexión con la base de datos.

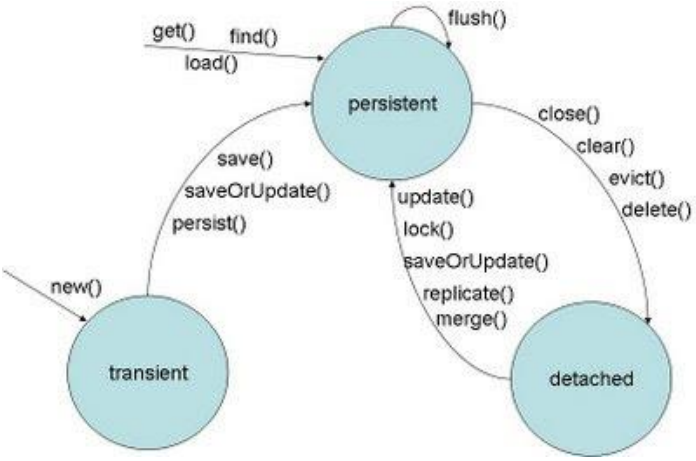
Para realizar estas tareas, debemos configurarlas en el método setup.

```
        final StandardServiceRegistry registry = new
StandardServiceRegistryBuilder()
                .configure() // configures settings from
hibernate.cfg.xml
                .build();
        try {
            sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
        } catch (Exception ex) {
            StandardServiceRegistryBuilder.destroy(registry);
        }
```

Ahora configuraremos el método exit.

```
protected void exit() {
        if (sessionFactory != null && !sessionFactory.isClosed()) {
            sessionFactory.close();
        }
    }
```

A continuación, se muestra un **diagrama** con los **estados** de un objeto en una aplicación Hibernate:



Estados en las sesiones
Fuente: elaboración propia

Su significado es el siguiente:

- **Transient:** objetos que acaban de ser instanciados utilizando el operador new, pero no están asociados a sesiones de Hibernate.

- **Persistent:** un objeto persistente es aquel que tiene una representación en la base.

- **Detached:** un objeto separado es aquel que se ha hecho persistente, pero su sesión ha sido cerrada de datos y, además, un valor identificador.

Una vez que Hibernate crea la SessionFactory, se puede abrir una Session y comenzar una transacción como la que sigue:

```
Session session = sessionFactory.openSession();
session.beginTransaction();
```

Por último, podemos probar nuestro código añadiendo una clase de inicio de la aplicación y añadiendo el siguiente código al método main:

```
package net.coremsa.hibernate;

public class Principal {

    public static void main(String[] args) {

        BookManager manager = new BookManager();
        manager.setup();
        manager.exit();
    }

}
```

2.5 Gestión de transacciones

La gestión de transacciones es importante en cualquier aplicación que interactúa con una base de datos. Las transacciones aseguran que un conjunto de operaciones sobre la base de datos cumpla las propiedades ACID:

- **Atomicidad:** todas las operaciones dentro de una transacción se completan exitosamente o ninguna de ellas lo hace.
- **Consistencia:** la transacción lleva a la base de datos de un estado válido a otro estado válido.
- **Aislamiento:** las transacciones concurrentes no interfieren entre sí.
- **Durabilidad:** una vez confirmada (commit), la transacción persiste incluso en caso de fallos del sistema.

A	C	I	D
• Atomicity (atomicidad)	• Consistency (consistencia)	• Isolation (aislamiento)	• Durability (durabilidad)

Propiedades ACID
Fuente: elaboración propia

En Hibernate, estos son los pasos para gestionar transacciones:

1. **Inicio de la transacción:** una transacción en Hibernate se inicia mediante el método beginTransaction() de la Session. Esto marca el inicio de un conjunto de operaciones que deben ejecutarse de forma atómica.

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
```

2. **Operaciones dentro de la transacción:** una vez iniciada la transacción, se pueden realizar operaciones de CRUD (Create, Read, Update, Delete) sobre la base de datos. Todas estas operaciones estarán dentro del contexto de la transacción.

```
Book book = new Book();
book.setTitle("Aprende Hibernate");
book.setAuthor("Patricio Fernández");
session.save(book);
```

3. **Confirmación de la transacción (commit):** si todas las operaciones se ejecutan correctamente, se realiza un commit de la transacción para confirmar los cambios en la base de datos.

```
transaction.commit();
```

4. **Deshacer la transacción (rollback):** en caso de que alguna operación falle, se puede deshacer la transacción completa con rollback().

```
transaction.rollback();
```

5. **Cierre de la sesión:** finalmente, se cierra la sesión para liberar los recursos.

```
session.close();
```

PARA SABER MÁS

Aquí podrás ampliar información sobre la gestión de transacciones en Hibernate:

2.6 Carga, almacenamiento y modificación de objetos

Vamos a conocer cuáles son las operaciones para la carga, almacenamiento y modificación de objetos:

Usaremos el método create() para **crear un nuevo libro** en nuestra base de datos. Para ello, introduciremos la siguiente modificación dentro de nuestro BookManager como ejemplo de esa creación de un nuevo libro.

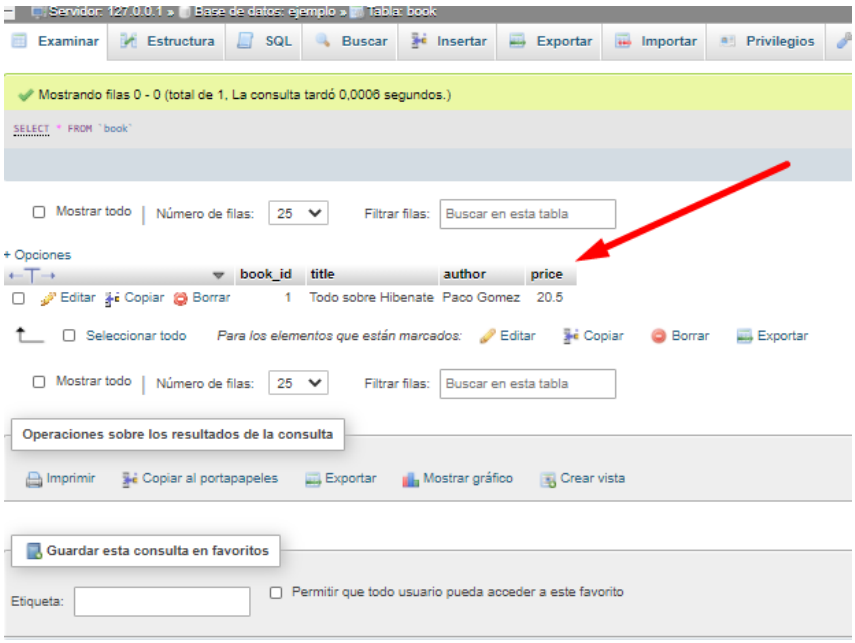
```
protected void create() {
            Book book = new Book();
            book.setTitle("Todo sobre Hibenate");
            book.setAuthor("Paco Gomez");
            book.setPrice(20.5f);
```

```
            Session session = sessionFactory.openSession();
            session.beginTransaction();

            session.save(book);

            session.getTransaction().commit();
            session.close();

    }
```

Para comprobar el correcto funcionamiento de este método, introduciremos las siguientes modificaciones en nuestro main.

```
package net.coremsa.hibernate;

public class Principal {

    public static void main(String[] args) {

        BookManager manager = new BookManager();
        manager.setup();

        manager.create();

        manager.exit();
    }

}
```

Comprobaremos en nuestra base de datos que la operación se ha realizado correctamente.



Creación de tupla

Fuente: elaboración propia

De igual forma, podemos implementar la **lectura** desde la base de datos incorporando el siguiente código al método read.

```
protected void read() {
    Session session = sessionFactory.openSession();

    long bookId = 20;
    Book book = session.get(Book.class, bookId);

    System.out.println("Title: " + book.getTitle());
    System.out.println("Author: " + book.getAuthor());
    System.out.println("Price: " + book.getPrice());

    session.close();
```

En el caso de la **actualización**:

```
    protected void update() {
        Book book = new Book();
        book.setId(1);
        book.setTitle("Java e Hibernate");
        book.setAuthor("Noelia Sanchez");
        book.setPrice(19.99f);

        Session session = sessionFactory.openSession();
        session.beginTransaction();

        session.update(book);

        session.getTransaction().commit();
        session.close();
    }
```

Por último, en el caso del **borrado** de un objeto:

```
    protected void delete() {
        Book book = new Book();
        book.setId(1);

        Session session = sessionFactory.openSession();
        session.beginTransaction();

        session.delete(book);

        session.getTransaction().commit();
        session.close();
    }
```

Al final, la clase BookManager quedaría así:

```
package net.coremsa.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class BookManager {
    protected SessionFactory sessionFactory;

    protected void setup() {
        final StandardServiceRegistry registry = new
StandardServiceRegistryBuilder()
                .configure()
                .build();
        try {
            sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
        } catch (Exception ex) {
            StandardServiceRegistryBuilder.destroy(registry);
        }

    }

    protected void exit() {
        if (sessionFactory != null && !sessionFactory.isClosed()) {
            sessionFactory.close();
        }
    }


    protected void create() {
            Book book = new Book();
            book.setTitle("Todo sobre Hibenate");
            book.setAuthor("Paco Gomez");
            book.setPrice(20.5f);

            Session session = sessionFactory.openSession();
            session.beginTransaction();

            session.save(book);

            session.getTransaction().commit();
            session.close();

    }


    protected void read() {
        Session session = sessionFactory.openSession();

        long bookId = 20;
        Book book = session.get(Book.class, bookId);
```

```
            System.out.println("Title: " + book.getTitle());
            System.out.println("Author: " + book.getAuthor());
            System.out.println("Price: " + book.getPrice());

            session.close();
        }


        protected void update() {
            Book book = new Book();
            book.setId(1);
            book.setTitle("Java e Hibernate");
            book.setAuthor("Noelia Sanchez");
            book.setPrice(19.99f);

            Session session = sessionFactory.openSession();
            session.beginTransaction();

            session.update(book);

            session.getTransaction().commit();
            session.close();
        }


        protected void delete() {
            Book book = new Book();
            book.setId(1);

            Session session = sessionFactory.openSession();
            session.beginTransaction();

            session.delete(book);

            session.getTransaction().commit();
            session.close();
        }

}
```

Si queremos que la clase Principal llame a todas estas funciones, quedaría así:

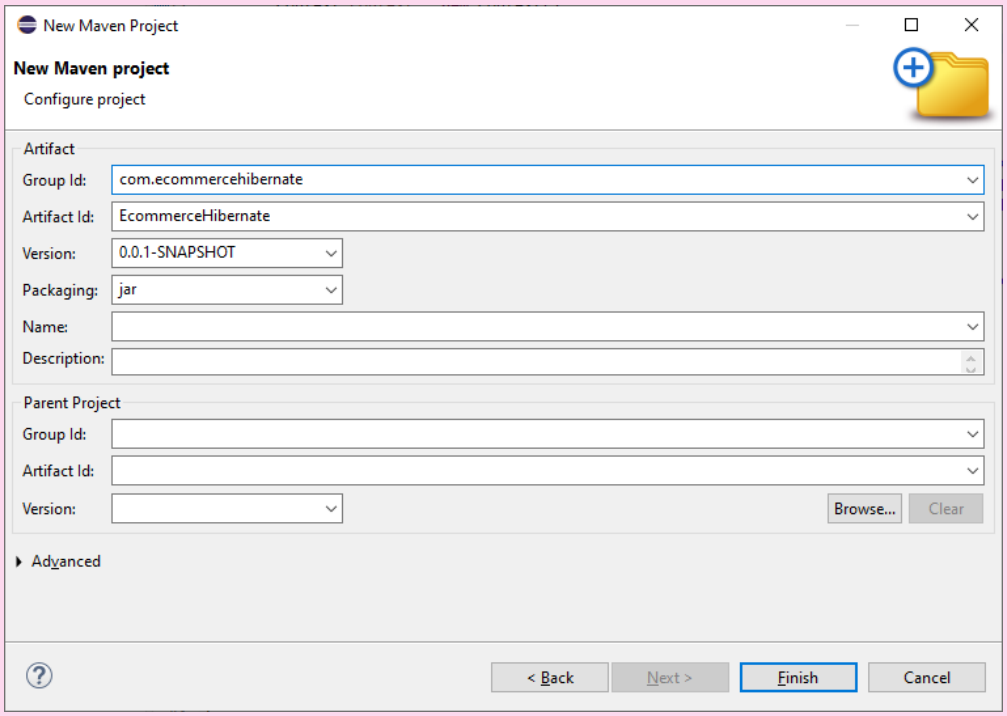
```
package net.coremsa.hibernate;

public class Principal {

    public static void main(String[] args) {
        // Crear una instancia de BookManager
        BookManager manager = new BookManager();

        try {
            // Configurar la SessionFactory
            manager.setup();
```

```
            // Crear un nuevo libro
            manager.create();

            // Leer el libro que acabamos de crear
            manager.read();

            // Actualizar el libro que acabamos de leer
            manager.update();

            // Leer el libro después de la actualización para
verificar los cambios
            manager.read();

            // Eliminar el libro
            manager.delete();

            // Intentar leer el libro después de eliminarlo para
confirmar que ha sido eliminado
            manager.read();

        } catch (Exception e) {
            // Manejar excepciones (el error al leer el libro
eliminado)
            e.printStackTrace();
        } finally {
            // Cerrar la SessionFactory en el bloque finally para
asegurar que se liberen los recursos
            manager.exit();
        }
    }
}
```

EJEMPLO PRÁCTICO

Trabajas en una empresa de comercio electrónico que ha decidido mejorar su sistema de gestión de productos y pedidos. Actualmente, el sistema utiliza JDBC para interactuar directamente con una base de datos MySQL. Tu equipo ha decidido migrar a Hibernate para aprovechar las ventajas de las herramientas ORM.

1. **El primer paso es crear el proyecto en Maven, al que llamaremos EcommerceHibernate.**



Creación del proyecto
Fuente: elaboración propia

2. **Configuramos el pom.xml:**

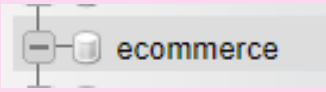
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ecommercehibernate</groupId>
  <artifactId>EcommerceHibernate</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
            <dependency>
                  <groupId>org.hibernate</groupId>
                  <artifactId>hibernate-core</artifactId>
                  <version>6.1.6.Final</version>
            </dependency>
            <dependency>
```

```
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>8.0.31</version>
                </dependency>
                <dependency>
                    <groupId>javax.activation</groupId>
                    <artifactId>activation</artifactId>
                    <version>1.1.1</version>
                </dependency>
            </dependencies>

</project>
```

3. Creamos el hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
name="connection.url">jdbc:mysql://localhost:3306/ecommerce</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">create</property>

    <!-- Mapeo de las clases -->
    <mapping class="Entidades.Producto" />
    <mapping class="Entidades.modelo.Pedido" />

  </session-factory>
</hibernate-configuration>
```

4. Lanzamos un servicio MySQL (por ejemplo, como se explica en la Unidad 2 con XAMPP) y creamos la base de datos ecommerce.



Base de datos
Fuente: elaboración propia

5. Creamos las entidades en el paquete: Entidades.

```
package Entidades;

import jakarta.persistence.*;

@Entity
@Table(name = "producto")
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(nullable = false)
    private String nombre;

    private String descripcion;

    @Column(nullable = false)
    private Double precio;

    // Constructores, getters y setters

    public Producto() {}

    public Producto(String nombre, String descripcion, Double precio) {
        this.nombre = nombre;
        this.descripcion = descripcion;
        this.precio = precio;
    }

    public Long getId() {
        return id;
    }

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

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public Double getPrecio() {
        return precio;
    }

    public void setPrecio(Double precio) {
        this.precio = precio;
    }
}


package Entidades;

import jakarta.persistence.*;
import java.util.Date;
import java.util.Set;

@Entity
@Table(name = "pedido")
public class Pedido {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date fechaPedido;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "pedido_producto",
        joinColumns = @JoinColumn(name = "pedido_id"),
        inverseJoinColumns = @JoinColumn(name = "producto_id")
    )
    private Set<Producto> productos;

    // Constructores, getters y setters

    public Pedido() {}

    public Pedido(Date fechaPedido, Set<Producto> productos) {
        this.fechaPedido = fechaPedido;
        this.productos = productos;
    }

    public Long getId() {
        return id;
    }

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

    public Date getFechaPedido() {
        return fechaPedido;
    }

    public void setFechaPedido(Date fechaPedido) {
        this.fechaPedido = fechaPedido;
    }

    public Set<Producto> getProductos() {
        return productos;
    }

    public void setProductos(Set<Producto> productos) {
        this.productos = productos;
    }
}
```

6. Creamos los manager en el paquete: Manager.

```
package Managers;

import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class ManagerPrincipal {

      protected static SessionFactory sessionFactory;
```

```
	public static void setup() {

		final StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().configure().build();
		try {
			sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
		} catch (Exception ex) {
			StandardServiceRegistryBuilder.destroy(registry);
			System.out.println(ex);

		}
	}

	public static void exit() {
		sessionFactory.close();
	}
}


package Managers;

import org.hibernate.Session;
import org.hibernate.Transaction;
import Entidades.Pedido;
import Entidades.Producto;

import java.util.Date;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class PedidoManager {

    public void crear() {
        // Supongamos que ya existen productos con IDs 1, 2 y 3
        ProductoManager productoManager = new ProductoManager();
        Producto producto1 = productoManager.obtener(1L);
        Producto producto2 = productoManager.obtener(2L);
        Producto producto3 = productoManager.obtener(3L);

        Set<Producto> productos = new HashSet<>();
        productos.add(producto1);
        productos.add(producto2);
        productos.add(producto3);

        Pedido pedido = new Pedido(new Date(), productos);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.save(pedido);
        session.getTransaction().commit();
        session.close();
    }

    public Pedido obtener(long id) {
        Session session = ManagerPrincipal.sessionFactory.openSession();
        Pedido pedido = session.get(Pedido.class, id);
        session.close();
        return pedido;
    }
```

```
    public void informacion(long id) {
        Pedido pedido = obtener(id);
        System.out.println("ID Pedido: " + pedido.getId());
        System.out.println("Fecha Pedido: " + pedido.getFechaPedido());
        System.out.println("Productos en el Pedido:");
        for (Producto p : pedido.getProductos()) {
            System.out.println(" - " + p.getNombre() + " ($" + p.getPrecio()
+ ")");
        }
    }

    public void actualizar(long id, Date fechaPedido, Set<Producto>
productos) {
        Pedido pedido = obtener(id);

        if (fechaPedido != null)
            pedido.setFechaPedido(fechaPedido);

        if (productos != null && !productos.isEmpty())
            pedido.setProductos(productos);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.update(pedido);
        session.getTransaction().commit();
        session.close();
    }

    public void eliminar(long id) {
        Pedido pedido = obtener(id);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.delete(pedido);
        session.getTransaction().commit();
        session.close();
    }

    public List<Pedido> obtenerTodos() {
        Session session = ManagerPrincipal.sessionFactory.openSession();
        List<Pedido> pedidos = session.createQuery("FROM Pedido",
Pedido.class).list();
        session.close();
        return pedidos;
    }
}


package Managers;

import org.hibernate.Session;
import org.hibernate.Transaction;
import Entidades.Producto;

import java.util.List;

public class ProductoManager {

    public void crear() {
```

```
        Producto producto1 = new Producto("Portatil", "Dell Inspiron",
800.00);
        Producto producto2 = new Producto("Smartphone", "Samsung Galaxy",
500.00);
        Producto producto3 = new Producto("Tablet", "iPad", 300.00);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.save(producto1);
        session.save(producto2);
        session.save(producto3);
        session.getTransaction().commit();
        session.close();
    }


    public Producto obtener(long id) {
        Session session = ManagerPrincipal.sessionFactory.openSession();
        Producto producto = session.get(Producto.class, id);
        session.close();
        return producto;
    }

    public void informacion(long id) {
        Producto producto = obtener(id);
        System.out.println("ID: " + producto.getId());
        System.out.println("Nombre: " + producto.getNombre());
        System.out.println("Descripción: " + producto.getDescripcion());
        System.out.println("Precio: " + producto.getPrecio());
    }

    public void actualizar(long id, String nombre, String descripcion,
Double precio) {
        Producto producto = obtener(id);

        if (nombre != null)
            producto.setNombre(nombre);

        if (descripcion != null)
            producto.setDescripcion(descripcion);

        if (precio != null && precio > 0)
            producto.setPrecio(precio);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.update(producto);
        session.getTransaction().commit();
        session.close();
    }

    public void eliminar(long id) {
        Producto producto = obtener(id);

        Session session = ManagerPrincipal.sessionFactory.openSession();
        session.beginTransaction();
        session.delete(producto);
        session.getTransaction().commit();
        session.close();
    }
```

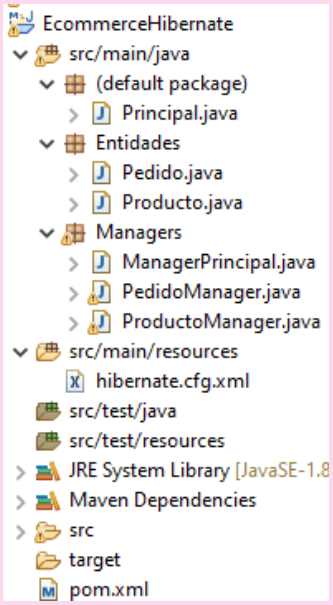
```
    public List<Producto> obtenerTodos() {
        Session session = ManagerPrincipal.sessionFactory.openSession();
        List<Producto> productos = session.createQuery("FROM Producto",
Producto.class).list();
        session.close();
        return productos;
    }
}
```

7. Creamos la clase Principal.

```
import Managers.ManagerPrincipal;
import Managers.ProductoManager;
import Managers.PedidoManager;
import Entidades.Producto;
import Entidades.Pedido;

import java.util.List;

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

        // Configurar la SessionFactory
        ManagerPrincipal.setup();

        // Instanciar los managers
        ProductoManager productoManager = new ProductoManager();
        PedidoManager pedidoManager = new PedidoManager();

        // Crear Productos
        productoManager.crear();

        // Mostrar información de un Producto
        System.out.println("Información del Producto con ID 1:");
        productoManager.informacion(1L);

        // Actualizar un Producto
        productoManager.actualizar(1L, "Portatil Gaming", "Dell Alienware",
1200.00);

        // Mostrar todos los Productos
        System.out.println("\nLista de Todos los Productos:");
        List<Producto> productos = productoManager.obtenerTodos();
        for (Producto p : productos) {
            System.out.println("ID: " + p.getId() + ", Nombre: " +
p.getNombre() + ", Precio: " + p.getPrecio());
        }

        // Crear un Pedido
        pedidoManager.crear();

        // Mostrar información de un Pedido
        System.out.println("\nInformación del Pedido con ID 1:");
        pedidoManager.informacion(1L);


        // Mostrar todos los Pedidos
        System.out.println("\nLista de Todos los Pedidos:");
        List<Pedido> pedidos = pedidoManager.obtenerTodos();
        for (Pedido o : pedidos) {
```

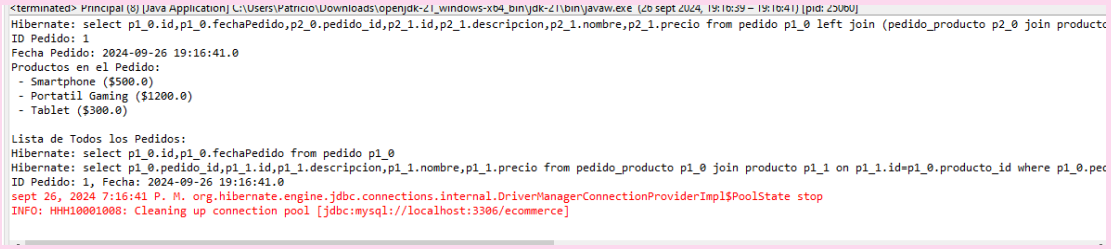
```
            System.out.println("ID Pedido: " + o.getId() + ", Fecha: " +
o.getFechaPedido());
        }


        // Cerrar la SessionFactory
        ManagerPrincipal.exit();

    }
}
```

8. Al final, la estructura del proyecto quedará así:



Estructura del Proyecto
Fuente: elaboración propia

9. Si lo ejecutamos, veremos esto:



Ejecución del proyecto
Fuente: elaboración propia

Pedido_Producto BBDD
Fuente: elaboración propia

2.7 Relaciones entre clases

Cuando se realizan proyectos en los que intervienen bases de datos e información, es obligatorio relacionar las tablas. Dentro de un proyecto con Hibernate, tenemos anotaciones que nos harán mucho más sencillo el usar y trabajar con dichas relaciones.

Las relaciones que se pueden establecer entre clases se dividen en:

- **One-to-one (uno a uno):** es la relación más simple, donde una clase (A) tiene una referencia a una instancia de otra clase (B), y ambas se encuentran relacionadas bajo una misma clave.

Relación 1 a 1

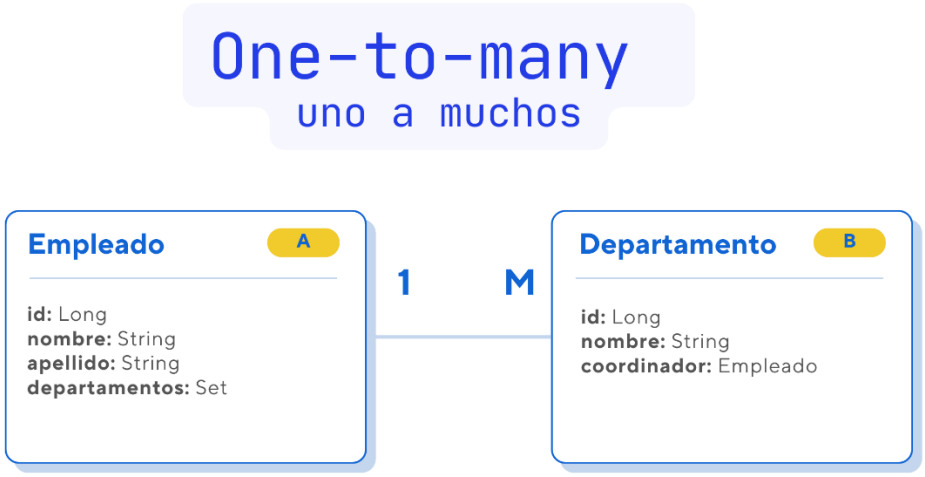
Fuente: elaboración propia

- **Many-to-one (muchos a uno):** relación donde la clave foránea de una tabla está referenciando la clave primaria de la otra tabla.

Relación muchos a 1

Fuente: elaboración propia

- **One-to-many (uno a muchos):** relación donde una de las clases (A) debe tener una colección que referencie a la otra (B).



Relación 1 a muchos
Fuente: elaboración propia

- **Many-to-many (muchos a muchos):** relación donde una de las clases (A) posee una colección que referencia a la otra clase (B), como en el caso de la relación one-to-many, pero, además, la segunda clase puede tener múltiples clases del tipo A.

Relación muchos a muchos
Fuente: elaboración propia

ENLACE DE INTERÉS

Aquí podrás comprobar la documentación oficial del fabricante al respecto de las asociaciones:

PARA SABER MÁS

También puede ser interesante que visualices un proyecto básico creado con Hibernate con asociaciones:

VÍDEO DE INTERÉS

Conoce cómo establecer las relaciones entre diferentes tablas con Hibernate:

EJEMPLO PRÁCTICO

En el proyecto que estamos trabajando sobre la gestión de usuarios dentro de un gimnasio, tenemos creado el esqueleto de interacción con la base de datos y los usuarios.

Nuestro jefe nos plantea avanzar en el mismo e incluir la gestión de los contactos de los usuarios. ¿Cómo realizaríamos esta gestión de los contactos? ¿Cuál sería la relación con los usuarios?

1. El primer paso sería crear la clase de mapeado Contacts.

```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    //Getters and setters are omitted for brevity
}
```

2. Como podemos imaginar, un usuario puede tener múltiples contactos, pero un contacto solo puede tener un usuario, por lo que la relación es uno a muchos. En la clase de Contacts, añadiríamos la siguiente propiedad:

```
@ManyToOne
    @JoinColumn(name = "user_id",
            foreignKey = @ForeignKey(name = "USER_ID_FK")
    )
    private User user;


    public User getUser () {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }
```

3. Además, deberíamos modificar la base de datos:

```
ALTER TABLE Contact
ADD CONSTRAINT USER_ID_FK
FOREIGN KEY (user_id) REFERENCES User
```

2.8 Consultas SQL (Standard Query Language) y HQL

Como se comentó en el segundo punto de la unidad, las herramientas ORM cuentan con un lenguaje de consultas propio que, en el caso de Hibernate, es el Hibernate Query Language (HQL).

El funcionamiento es muy sencillo, las consultas son escritas en HQL e Hibernate se encarga de traducirlas a SQL y ejecutarlas para obtener los resultados deseados.

Entre sus principales características cabe destacar:

- Similitud con SQL y, por tanto, facilidad de uso y aprendizaje.
- Lenguaje orientado a objetos que, por tanto, usa clases y atributos en vez de tablas y columnas y, además, permite el uso de mecanismos de herencia, polimorfismo y asociaciones.
- Los tipos de datos que utiliza son los mismos que los del lenguaje de programación Java.
- Las consultas son independientes del lenguaje SQL utilizado y del modelo de tablas de la base de datos.
- Es case insensitive, lo que quiere decir que se puede escribir las sentencias en mayúscula o minúscula y se obtendrá el mismo resultado.
- La información resultante se obtiene en forma de objetos.

Al igual que ocurre con SQL, para realizar consultas en HQL, se utilizan las sentencias SELECT, FROM, WHERE, HAVING, GROUP BY, etc.

En HQL, la consulta más sencilla que se puede encontrar es:

```
FROM clase
```

Esta consulta devuelve todas las instancias de la clase. Es posible añadir más clases a la consulta separándolas por comas de la siguiente forma:

```
FROM clase1, clase2.
```

Al igual que ocurre con SQL, si se quiere, se pueden especificar condiciones utilizando la sentencia WHERE. Por ejemplo:

```
FROM clase WHERE condición
```

Además, si se quieren indicar qué objetos y propiedades devolver en el conjunto de resultados de la consulta, se puede utilizar la sentencia SELECT, cuya sintaxis es:

```
SELECT objeto
FROM clase
WHERE condicion
```

En la cláusula where se pueden utilizar operadores matemáticos, de comparación binarios, lógicos, paréntesis para agrupar, concatenación de cadenas, así como las cláusulas IN, NOT IN, BETWEEN, IS NULL, IS NOT NULL, IS EMPTY, IS NOT EMPTY, MEMBER OF, NOT MEMBER OF, etc.

Si se quieren utilizar funciones de agregación en las consultas, igualmente se hará como en el caso de SQL, utilizando las funciones: avg, sum, min, max, count, entre otras.

EJEMPLO PRÁCTICO

Tu jefe te solicita el paso final de una migración de proyectos de gestión tradicional (bases de datos puramente relacionales) a gestión con el ORM. Para ello, ya todo está hecho, pero el equipo no conoce el lenguaje HQL, por lo que te pide transformar las siguientes sentencias SQL a HQL. Además, te solicita una pequeña explicación por cada una de ellas. Las sentencias son:

```
SELECT *
FROM Empleado
WHERE departamento = 'TI';

SELECT p.*
FROM Proyecto p
JOIN Empleado e ON p.equipo_id = e.equipo_id
WHERE e.lider_nombre = 'Carlos';

SELECT AVG(sueldo) AS promedio_sueldo
FROM Empleado;
```

¿Cómo lo llevarías a cabo?

1. `SELECT e FROM Empleado e WHERE e.departamento = 'TI'`. Selecciona todos los objetos de la clase Empleado cuyo atributo departamento es igual a 'TI' (Tecnologías de la Información).

2. `SELECT p FROM Proyecto p JOIN p.equipo e WHERE e.lider.nombre = 'Carlos'`. Selecciona todos los objetos de la clase Proyecto que están asociados a un equipo cuyo líder tiene

el nombre 'Carlos'. Aquí se realiza un join entre las entidades Proyecto y Equipo, y se utiliza la condición e.lider.nombre = 'Carlos' para filtrar los proyectos asociados a un equipo liderado por alguien llamado 'Carlos'.

3. `SELECT AVG(sueldo) FROM Empleado.` AVG calcula el promedio de los valores del atributo sueldo de todos los objetos de la clase Empleado.

RESUMEN FINAL

Una aplicación, hoy en día, depende de la información y los sistemas de gestión de bases de datos (SGBD) para poder funcionar y presentar la información al usuario final.

Las bases de datos suelen tener sus propios lenguajes y estándares de trabajo, por lo que limitan enormemente la futura escalabilidad de la aplicación en el caso de querer cambiar de SGBD o, simplemente, ampliar la BD.

Por este motivo, surgen y existen las herramientas de ORM, que nos permiten mapear los objetos de nuestra aplicación con las tablas de la BD, encargándose ellos de realizar esa pasarela.

Esta estructura hace que nuestra aplicación sea más escalable, ya que estamos dividiendo la misma en configuración contra la base de datos, objetos de mapeado, sesiones de interacción y gestores de toda esta estructura.

En concreto, en Java disponemos de varias herramientas ORM, entre ellas Hibernate, una herramienta muy madura que se puede integrar en los proyectos mediante un gestor de paquetes como Maven para su administración. Mediante Hibernate podremos definir las características de conexión con la base de datos, la persistencia mediante clases y orientación a objetos, el mapeado con el diseño de la base datos y, por último, la interacción de creación, modificación, lectura y borrado de datos.

Una estructura final que introduce mayor complejidad inicial en el diseño de la aplicación, pero que permite a su vez hacer crecer la misma de una forma más rápida en el tiempo y, sobre todo, independiente de la tecnología de base de datos que subyace por debajo.