



JPA - Hibernate





# Índice

- Capítulo 1 – Introducción a la persistencia en Java
  - Capítulo 1.1 – Mapeo objeto relacional (ORM)
  - Capítulo 1.2 – Introducción JPA
    - Capítulo 1.2.1 – La API de JPA
    - Capítulo 1.2.2 – Versiones de JPA
    - Capítulo 1.2.3 – Entidades JPA
    - Capítulo 1.2.4 – El lenguaje JPQL
  - Capítulo 1.3 – Implementaciones JPA
    - Capítulo 1.3.1 – Hibernate
    - Capítulo 1.3.2 – EclipseLink



# Índice

- Capítulo 2 – Configuración JPA e Hibernate
  - Capítulo 2.1 – Dependencias necesarias
    - Capítulo 2.1.1 – Hibernate core
    - Capítulo 2.1.2 – Driver de base de datos
  - Capítulo 2.2 – Configuración xml para Hibernate
    - Capítulo 2.2.1 – Fichero hibernate.cfg.xml
    - Capítulo 2.2.2 – SessionFactory
    - Capítulo 2.2.3 – Session
  - Capítulo 2.3 – Configuración java para Hibernate



# Índice

- Capítulo 2 – Configuración JPA e Hibernate
  - Capítulo 2.4 – Configuración JPA
    - Capítulo 2.4.1 – La unidad de persistencia
    - Capítulo 2.4.2 – EntityManagerFactory
    - Capítulo 2.4.3 – EntityManager
  - Capítulo 2.5 – SessionFactory desde JPA
- Capítulo 3 – Entidades y asociaciones
  - Capítulo 3.1 – Introducción a las entidades
  - Capítulo 3.2 – Configuración de las entidades
  - Capítulo 3.3 – Clave primaria



# Índice

- Capítulo 3 – Entidades y asociaciones
  - Capítulo 3.4 – Tipos de datos en una entidad
    - Capítulo 3.4.1 – Tipos básicos
    - Capítulo 3.4.2 – Enumeraciones
    - Capítulo 3.4.2 – Colecciones
  - Capítulo 3.5 – Generación esquema base de datos
  - Capítulo 3.6 – Asociaciones entre entidades
    - Capítulo 3.6.1 – Asociación One-To-One
      - Capítulo 3.6.1.1 – Asociación con clave foránea
      - Capítulo 3.6.1.2 – Asociación con tabla join



# Índice

- Capítulo 3 – Entidades y asociaciones
  - Capítulo 3.6 – Asociaciones entre entidades
    - Capítulo 3.6.2 – Asociación One-To-Many
    - Capítulo 3.6.3 – Asociación Many-To-One
    - Capítulo 3.6.4 – Asociación Many-To-Many
    - Capítulo 3.6.5 – Ejemplo completo de asociaciones
  - Capítulo 3.7 – Operaciones en cascada
  - Capítulo 3.8 – Opción orphanRemoval
  - Capítulo 3.9 – Listeners



# Índice

- Capítulo 4 – HQL
  - Capítulo 4.1 – Introducción a HQL
  - Capítulo 4.2 – Sentencias select
  - Capítulo 4.3 – Sentencias insert
  - Capítulo 4.4 – Sentencias update
  - Capítulo 4.5 – Sentencias delete
- Capítulo 5 – Criteria API
  - Capítulo 5.1 – Introducción a Criteria API
  - Capítulo 5.2 – Estructura Criteria



# Índice

- Capítulo 5 – Criterios API
  - Capítulo 5.3 – Filtros
  - Capítulo 5.4 – Operaciones agregadas
  - Capítulo 5.5 – Actualizar datos
  - Capítulo 5.6 – Borrar datos





# 1. Introducción a la persistencia en Java

En toda aplicación empresarial uno de los principales requerimientos especificados que deben ser implementados son los relativos a la integración con las **bases de datos** para guardar, actualizar y recuperar la información que se utiliza.

Los **datos** son el principal activo de las empresas y necesitan por tanto ser almacenados en un sistema persistente, como las bases de datos.

**Persistencia** es la capacidad de los objetos para almacenarse y recuperarse desde un medio de almacenamiento.



## 1.1. Mapeo objeto relacional (ORM)

La dificultad en la codificación de la implementación de la persistencia reside en la complejidad de **convertir objetos a registros de la base de datos** subyacente y viceversa, lo que se puede denominar como traducción del modelo de objetos a modelo relacional y viceversa.

La relaciones entre estos dos modelos se denomina “**mapeo**” **ORM (Object Relational Mapping)**, tanto JPA como los frameworks que lo implementan tratan de simplificar y automatizar este proceso.



## 1.1. Mapeo objeto relacional (ORM)

Con **JDBC** es el programador quien tiene que extraer los resultados obtenidos en el objeto *ResultSet* y mapearlos a objetos Java con los que poder trabajar desde la aplicación.

En el desarrollo de software un **35% de tiempo** está dedicado al **mapeo** entre objetos y sus correspondientes relaciones, por tanto se vuelve necesaria una tecnología que facilite esta tarea.

Los **ORM** persiguen el objetivo de no perder las ventajas de la programación orientada a objetos al interactuar con las bases de datos, automatizando el proceso de mapeo entre ambos modelos objeto y relacional.



## 1.2. Introducción a JPA

**Java Persistence API** o **JPA** es una abstracción sobre JDBC que permite resolver las operaciones ORM de la conversión entre los objetos y las tablas de una base de datos de forma sencilla y viceversa, crear objetos a partir de las tablas de las bases de datos.

**JPA es una especificación**, es decir, es un conjunto de interfaces y clases que definen qué hay que hacer, pero no lo hacen. Para poder utilizarlo es necesaria una implementación de JPA, un framework de persistencia que sea el encargado de proporcionar todas las funcionalidades que JPA especifica.



## 1.2.1. La API de JPA

En el itinerario de paquetes **javax.persistence** se encuentran todas las clases e interfaces que conforman la especificación JPA.

Con motivo de la donación de Java EE a la Fundación Eclipse ahora JPA se conoce como **Jakarta Persistence API**.

Esto ocasiona que a partir de la versión JPA 3.0 se migra el itinerario de paquetes de `javax.persistence` a **`jakarta.persistence`**.



## 1.2.2. Versiones de JPA

Las diferentes versiones de la especificación JPA son:

- **JPA 1.0** (2006): la propia capa de persistencia empleada en EJB se separa como una nueva especificación independiente, JPA 1.0.
- **JPA 2.0** (2009): introducción de Criteria API, etc.
- **JPA 2.1** (2013): generación de esquemas, introducción de Entity Graphs, etc.
- **JPA 2.2** (2017): soporte para clases del paquete `java.time` (jdk 1.8), etc.
- **JPA 3.0** (2020): cambio del espacio de nombres de paquete `javax.persistence` a `jakarta.persistence`.



## 1.2.3. Entidades JPA

Se considera una **entidad** (“entity”) al objeto que se va a persistir o recuperar de una base de datos. Se puede considerar una entidad como la representación de un registro de la tabla.

Para crear una entidad se utiliza la anotación **@Entity**, con ella se marca un POJO (Plain Old Java Object) como entidad.

```
7
8  @Entity
9  @Table(name = "project" )
10 public class Project implements Serializable {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15
16     private String name;
17
18     private String description;
```



## 1.2.4. El lenguaje JPQL

**Java Persistence Query Language** o **JPQL** (ahora Jakarta Persistence Query language), es un lenguaje inspirado en SQL pero que opera con entidades, siendo por tanto un lenguaje de consulta orientado a objetos y que forma parte de JPA.

JPA transforma las sentencias formuladas en JPQL a SQL. Existen múltiples formas de crear consultas con JPQL:

- **createQuery()**: permite crear sentencias en lenguaje JPQL.
- **createNamedQuery()**: permite crear sentencias nombradas que poder reutilizar.
- **createNativeQuery()**: permite crear sentencias nativas en lenguaje SQL.





## 1.3. Implementaciones JPA

JPA está compuesto por una serie de interfaces, clases, anotaciones y enumeraciones, entre todas ellas constituyen la **especificación JPA** de Java EE.

Para poder hacer uso de las mismas es necesaria una **implementación de JPA**. Actualmente existen varias:

- Hibernate
- EclipseLink
- Apache OpenJPA
- DataNucleus



## 1.3.1. Hibernate

Hibernate es un **conjunto de tecnologías** para la capa de persistencia en aplicaciones Java y que nace antes de la especificación JPA, en el año 2001.

Actúa como ORM sin perder las ventajas del acceso a SQL nativo. También provee mecanismos para recuperar y filtrar la información así como traducciones a SQL específico de la base de datos por medio de **dialectos**.





## 1.3.1. Hibernate

Hibernate se ha convertido en una **suite de herramientas** para la gestión de la persistencia en el ecosistema java. Su módulo principal es el ORM pero está compuesto por multitud de herramientas:

- Hibernate **ORM**: Object Relational Mapping.
- Hibernate **EntityManager**: implementación de JPA.
- Hibernate **Validator**: implementación de Bean Validation.
- Hibernate **Envers**: auditoría, histórico sobre datos, VCS a nivel de base de datos.
- Hibernate **Search**: búsquedas de texto utilizando índices Apache Lucene.
- Hibernate **OGM**: soporte para bases de datos NoSQL.



## 1.3.2. EclipseLink

**EclipseLink** es un framework de persistencia open source perteneciente a la Fundación Eclipse y basado en el producto TopLink de Oracle.

Cuenta con soporte para multitud de especificaciones java como JPA, JAXB, JCA, SDO. Es considerado como la implementación de referencia de JPA.





## 2. Configuración JPA e Hibernate

Para la configuración de un proyecto java con JPA e hibernate es necesario llevar a cabo una serie de pasos:

1. **Importar** las **dependencias** necesarias: hibernate y driver de base de datos.
2. **Configuración** Hibernate: propiedades de conexión a la base de datos y configuraciones concretas.
3. **Inicialización** de Hibernate por medio de los componentes principales: *SessionFactory* y *Session* o *EntityManagerFactory* y *EntityManager*.



## 2.1. Dependencias necesarias

Para las **dependencias** es posible descargarlas de la [página oficial](#) y agregarlas manualmente al proyecto. Por convención se empleará una herramienta de gestión de dependencias como puede ser **maven** o gradle.

Utilizando **Apache Maven** se agregarán las dependencias de hibernate y driver de base de datos correspondiente dentro del fichero **pom.xml** del proyecto.



## 2.1.1. Hibernate core

En primer lugar será necesario agregar la **dependencia de Hibernate**. [Dependencia Hibernate](#):

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.30.Final</version>
</dependency>
```



## 2.1.2. Driver de base de datos

En segundo lugar, Hibernate necesita conectarse a una **base de datos** para realizar la gestión de la misma. Hibernate utiliza JDBC para realizar sus funcionalidades, para ello es necesario agregar la [dependencia MySQL](#) driver base de datos:

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>8.0.23</version>  
</dependency>
```

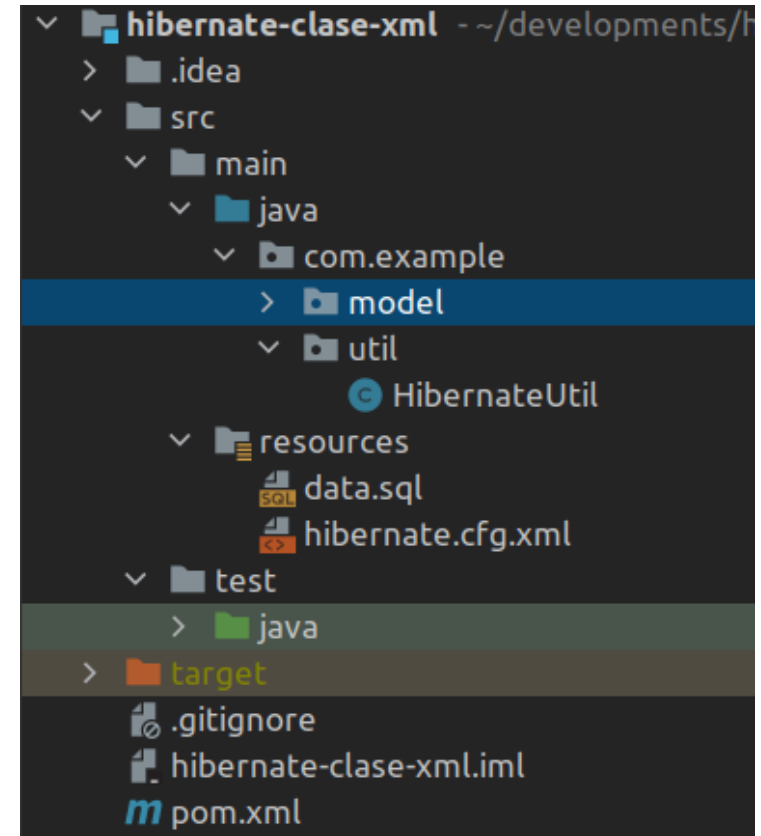




## 2.2. Configuración xml para Hibernate

Una vez importadas las dependencias necesarias en el proyecto hay múltiples maneras de **configurar Hibernate** y su funcionamiento.

Una de ellas es a través del fichero de configuración **hibernate.cfg.xml** en el directorio src/main/resources.





## 2.2.1. Fichero hibernate.cfg.xml

El contenido del fichero **hibernate.cfg.xml** define la configuración necesaria para que Hibernate pueda conectarse a una base de datos y realizar las funciones de ORM necesarias, dentro puede definir:

- Propiedades de **conexión** a la base de datos.
- Estrategia de generación del **esquema** de la base de datos.
- **Importar** ficheros sql.
- **Mostrar las sentencias** sql generadas por hibernate ( ideal para desarrollo).
- Mapeo de **entidades**.



## 2.2.1. Fichero hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3300/hibernatedb</property>
<property name="connection.username">root</property>
<property name="connection.password">admin</property>
<!-- Other properties: -->
<!-- .... -->
</session-factory>
</hibernate-configuration>
```



## 2.2.1. Fichero hibernate.cfg.xml

```
<!-- Select our SQL dialect -->
<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
<!-- Echo the SQL to stdout -->
<property name="show_sql">true</property>
<property name="format_sql">true</property>
<!-- Set the current session context -->
<property name="current_session_context_class">thread</property>
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create</property>
<!-- Import sql script-->
<property name="hibernate.hbm2ddl.import_files">data.sql</property>
```



## 2.2.1. Fichero hibernate.cfg.xml

Las entidades pueden configurarse mediante anotaciones sobre la propia clase o mediante otros ficheros xml vinculados al fichero hibernate.cfg.xml. Para la **configuración por anotaciones** se especifica la ruta a cada entidad al final del fichero hibernate.cfg.xml:

```
<!-- Entities-->
<mapping class="com.example.model.Direction" />
<mapping class="com.example.model.Employee" />
<mapping class="com.example.model.Company" />
<mapping class="com.example.model.Project" />
<mapping class="com.example.model.CreditCard" />
<mapping class="com.example.model.Department" />
```



## 2.2.2. SessionFactory

Una vez definido el fichero de configuración se suele crear por convención una **clase de utilidad** que genere una única instancia de tipo *SessionFactory*, el componente principal en Hibernate.

*SessionFactory* se encarga de gestionar el contexto de persistencia y las conexiones a base de datos.

```
public class HibernateUtil {  
    private static StandardServiceRegistry registry;  
    private static SessionFactory sessionFactory;  
    public static SessionFactory getSessionFactory() {  
        if (sessionFactory == null) {  
            try {  
                // Create registry  
                registry = new StandardServiceRegistryBuilder().configure().build();  
                // Create MetadataSources  
                MetadataSources sources = new MetadataSources(registry);  
                // Create Metadata  
                Metadata metadata = sources.getMetadataBuilder().build();  
                // Create SessionFactory  
                sessionFactory = metadata.getSessionFactoryBuilder().build();  
            } catch (Exception e) {  
                e.printStackTrace();  
                if (registry != null)  
                    StandardServiceRegistryBuilder.destroy(registry);  
            }  
        }  
        return sessionFactory;  
    }  
    public static void shutdown() {  
        if (registry != null)  
            StandardServiceRegistryBuilder.destroy(registry);  
    }  
}
```



## 2.2.3. Session

A partir del *SessionFactory* se pueden crear objetos **Session** que permitan realizar las **operaciones de persistencia** contra la base de datos. Ejemplo de extracción del objeto session y su uso para encontrar un elemento por su id:

```
Session session = HibernateUtil.getSessionFactory().openSession();  
Company company = session.find(Company.class, 1L);  
company.getEmployees().forEach(employee > System.out.println(employee.getName()));  
session.close();
```



## 2.3. Configuración java para Hibernate

Hibernate se puede configurar de otras maneras diferentes al uso del archivo **hibernate.cfg.xml**.

Una de ellas es directamente en la creación del componente *SessionFactory*, en la misma **clase de utilidad en Java**.

```
public class HibernateUtil {  
    private static SessionFactory sessionFactory;  
    public static SessionFactory getSessionFactory(){  
        if (sessionFactory == null){  
            Configuration settings = new Configuration();  
            Properties properties = new Properties();  
            properties.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");  
            properties.put(Environment.URL, "jdbc:mysql://localhost:3300/hibernatedb");  
            properties.put(Environment.USER, "root");  
            properties.put(Environment.PASS, "admin");  
            properties.put(Environment.CURRENT_SESSION_CONTEXT_CLASS, "thread");  
            properties.put(Environment.DIALECT, "org.hibernate.dialect.MySQL8Dialect");  
            properties.put(Environment.SHOW_SQL, "true");  
            properties.put(Environment.FORMAT_SQL, "true");  
            properties.put(Environment.HBM2DDL_AUTO, "create");  
            properties.put(Environment.HBM2DDL_IMPORT_FILES, "data.sql");  
            settings.setProperties(properties);  
            // añadir entidades  
            settings.addAnnotatedClass(Company.class);  
            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()  
                .applySettings(settings.getProperties()).build();  
            sessionFactory = settings.buildSessionFactory(serviceRegistry);  
        }  
        return sessionFactory; }}  
}
```





## 2.4. Configuración JPA

Se utiliza *SessionFactory* y *Session* cuando se necesita una **interacción directa** con la **API de Hibernate**. Esto genera acoplamiento con el framework de persistencia, si se quiere reemplazar Hibernate por otro framework habrá que cambiar el código de la aplicación que utiliza las interfaces de Hibernate.

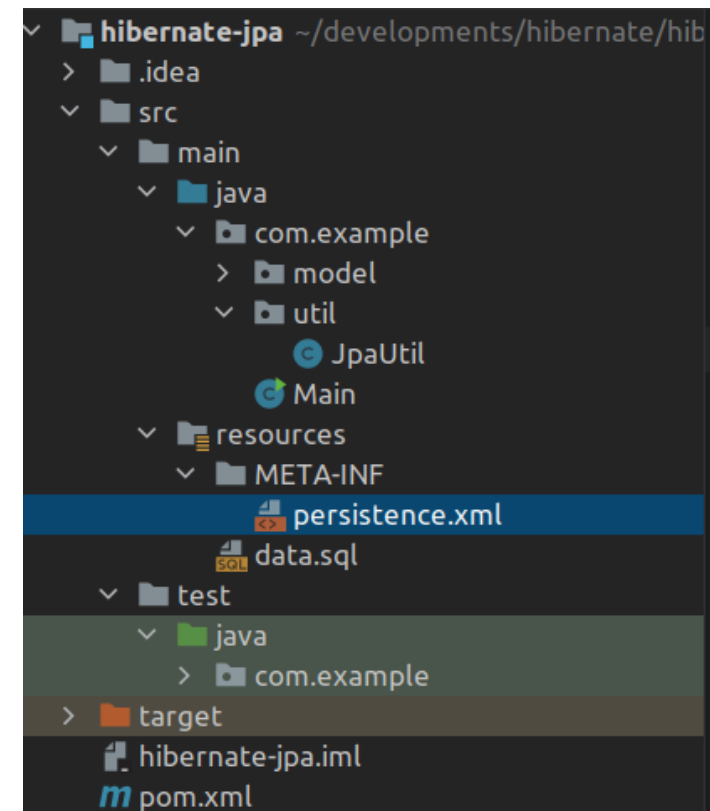
En cambio, si se utiliza la **abstracción de JPA** a través de las clases *EntityManagerFactory* y *EntityManager*, es posible cambiar el framework de persistencia sin cambiar el código, ya que todos deben implementar las mismas interfaces JPA.



## 2.4.1. La unidad de persistencia

Para utilizar la interfaces de JPA primero se configura la unidad de persistencia en el fichero **persistence.xml** dentro de la carpeta *META-INF* en **src/main/resources**.

En este fichero se definen las propiedades necesarias para que el proveedor de persistencia utilizado en el proyecto pueda conectarse a la base de datos y gestionarla.





## 2.4.1. La unidad de persistencia

La estructura del fichero **persistence.xml** es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:x
si="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns
.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
<persistence-unit name="hibernate-jpa" transaction-type="RESOURCE_LOCAL">
<properties>
<!-- Propiedades -->
</properties>
</persistence-unit>
</persistence>
```



## 2.4.1. La unidad de persistencia

Las **propiedades** a definir en el fichero **persistence.xml** son las siguientes:

```
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://127.0.0.1:3300/hibernatedb" />
<property name="javax.persistence.jdbc.user" value="root" />
<property name="javax.persistence.jdbc.password" value="admin" />
<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
<property name="javax.persistence.schema-
generation.database.action" value="create" />
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
<property name="javax.persistence.sql-load-script-source" value="data.sql" />
<property name="hibernate.show_sql" value="true"/>
```



## 2.4.2. EntityManagerFactory

Una vez definido el fichero de configuración se suele crear por convención una **clase de utilidad** que genere una única instancia de tipo *EntityManagerFactory*, el componente principal en JPA que se encarga de gestionar el contexto de persistencia y las conexiones a base de datos.

```
public class JpaUtil {  
  
    private static final EntityManagerFactory ENTITY_MANAGER_FACTORY =  
        Persistence.createEntityManagerFactory(  
            persistenceUnitName: "hibernate-jpa");  
  
    public static EntityManager getEntityManager() {  
        return ENTITY_MANAGER_FACTORY.createEntityManager();  
    }  
}
```



## 2.4.2. EntityManagerFactory

El nombre utilizado en el parámetro `persistenceUnitName` dentro del método `createEntityManagerFactory` debe ser **el nombre de la unidad de persistencia** definido al comienzo del archivo `persistence.xml`.

```
public class JpaUtil {  
  
    private static final EntityManagerFactory ENTITY_MANAGER_FACTORY =  
        Persistence.createEntityManagerFactory(  
            persistenceUnitName: "hibernate-jpa");  
  
    public static EntityManager getEntityManager() {  
        return ENTITY_MANAGER_FACTORY.createEntityManager();  
    }  
}
```



## 2.4.3. EntityManager

A partir de la clase **JpaUtil** se pueden crear objetos **EntityManager** que permitan realizar las operaciones de persistencia contra la base de datos. Ejemplo de extracción del objeto manager y su uso para encontrar un elemento por su id.

```
EntityManager manager = JpaUtil.getEntityManager();  
Employee employee = manager.find(Employee.class, 1L);  
System.out.println(employee);  
manager.close();
```



## 2.5. SessionFactory desde JPA

Cuando se trabaja con las interfaces de JPA puede haber determinados casos en los que es necesario acceder a la API de Hibernate de forma directa. En estas situaciones se puede **extraer** el *SessionFactory* a partir de *EntityManager*:

```
EntityManager manager = factory.createEntityManager();  
Session session = entityManager.unwrap(Session.class);  
// Es posible extraer SessionFactory a partir de session  
SessionFactory sessionFactory = session.getSessionFactory();
```





### 3. Entidades y asociaciones

Una **entidad** es una clase java que representa una tabla en la base de datos y sus atributos representan columnas en la tabla.

Cada instancia de la clase entidad representa una **fila en la base de datos**.

id	modelo	fabricante	num_cilindros	num_cv
1 (Edited)	458 Italia	Ferrari	12	588.99
3 (Edited)	Múltipla	Fiat	4	55.00
5 (Edited)	Ford	Mondeo	4	432.50
7 (Edited)	Alfa	Romeo	5	180.50



## 3.1. Introducción a las entidades

Los **atributos de una entidad** se mapean contra las columnas de la tabla que representa dicha entidad.

Al igual que ocurre con JDBC, los tipos de datos se convierten de **tipos SQL** a **tipos de datos Java**.

coche

InnoDB

10

utf8

utf8\_general\_ci

Column Name	#	Data Type	Not Null	Auto Increment	Key
<a href="#">123</a> id	1	int	[X]	[X]	PRI
<a href="#">ABC</a> modelo	2	varchar(255)	[ ]	[ ]	
<a href="#">ABC</a> fabricante	3	varchar(255)	[ ]	[ ]	
<a href="#">123</a> num_cilindros	4	int	[ ]	[ ]	
<a href="#">123</a> num_cv	5	decimal(7,2)	[ ]	[ ]	



## 3.2. Configuración de las entidades

El **mapeo de entidades** se pueden configurar por medio de archivos xml o por medio de anotaciones sobre las clases java. Por defecto se optará por la segunda opción.

Para crear una entidad se utiliza la anotación **@Entity**, con ella se marca un POJO como entidad. Para que una clase sea considerada entidad debe cumplir con los siguientes requisitos:

- Implementar la **interfaz Serializable**.
- Debe tener un método **constructor sin argumentos**, con visibilidad pública.
- Métodos **getter** y **setter** para acceso a sus atributos.



## 3.3. Clave primaria

**@Id:** anota una propiedad como clave primaria tanto si es *PK (Primary Key)* simple o si forma parte de una PK compuesta, de varios campos. Todas las entidades tienen que tener una identidad que las diferencie del resto, por lo que deben contener una propiedad marcada con esta anotación. Es recomendable que las propiedades sean de clase envoltorio en vez de tipos primitivos, para que puedan contener el valor null.

**@GeneratedValue:** anota una propiedad cuyo valor va a ser generado por el proveedor de persistencia, este será quien le asigne un valor la primera vez que se almacene la entidad en la base de datos.



## 3.3. Clave primaria

Ejemplo de entidad con **clave primaria** con las anotaciones indicadas. La anotación **@Table** permite definir el nombre de la tabla en la base de datos.

```
@Entity
@Table(name="employees")
public class Employee implements Serializable{

    // atributos - columnas de base de datos
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // ... otras columnas ...
}
```



## 3.4. Tipos de datos en una entidad

Los **atributos de una entidad** son mapeados contra las columnas de la tabla que representa dicha entidad. La configuración de cada atributo se hace por medio de la anotación **@Column**:

```
@Entity
@Table(name="employees")
public class Employee implements Serializable{
    // ... Clave primaria ...
    @Column(name="name", length = 300)
    private String name;

    @Column(name="age")
    private Integer age;
```



## 3.4. Tipos de datos en una entidad

Los **datos** pueden de distinto tipo en Hibernate:

- **Tipos valor:** datos que no definen su propio ciclo de vida, forman parte de la entidad a la que pertenecen.
  - **Tipos básicos:** representan una columna
  - **Tipos incrustables** (embeddables): suele ser una clase separada pero que representa una agrupación de columnas dentro de la entidad
  - **Tipos colección:** colecciones de datos utilizando las interfaces del framework Java Collections.
- **Tipos entidad:** tienen su propia clave primaria y ciclo de vida independiente de la entidad en la que se asocia, consisten en otras entidades con sus configuraciones propias.



## 3.4.1. Tipos básicos

Los **tipos básicos** no requieren de ninguna configuración especial. Mediante la anotación **@Column** se puede definir el nombre de la columna así como **restricciones** de longitud, si es nullable o no, etc:

```
@Column(name = "postal_code", length = 5)
private String postalCode;

@Column(name="expiration_date")
private LocalDate expirationDate;

@Column(name="married")
private Boolean married;
```





## 3.4.2. Enumeraciones

Las **enumeraciones** se almacenan por defecto como el número ordinal de la enumeración. Mediante anotación **@Enumerated** se puede cambiar a String:

```
public enum ProjectType { SMALL, MEDIUM, LARGE } // Archivo ProjectType.java
@Entity // Archivo Project.java
@Table(name="projects")
public class Project {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Enumerated(EnumType.STRING)
    private ProjectType type;
    // ...
}
```



## 3.4.3. Colecciones

Una entidad puede tener como atributos **colecciones de elementos**, lo cual genera una nueva tabla donde poder almacenar la asociación en base de datos.

```
// Ejemplo colección de Strings
@ElementCollection
@CollectionTable(
    name = "project_tags",
    joinColumns = @JoinColumn(name = "id_project")
)
private List<String> tags = new ArrayList<>();
// Ejemplo colección de enumeraciones
@ElementCollection
private List<ProjectType> projectTypes = new ArrayList<>();
```



## 3.5. Generación esquema base de datos

Una vez creada una entidad, para generar la tabla correspondiente a dicha entidad es posible indicarlo en la configuración de Hibernate o JPA. De esta forma se puede configurar que se **genere automáticamente el esquema de la base de datos** con todas las tablas necesarias en base a las entidades creadas.

En Hibernate se define en el fichero hibernate.cfg.xml mediante la propiedad **hbm2ddl.auto**:

```
<property name="hbm2ddl.auto">create</property>
```



## 3.5. Generación esquema base de datos

Existen diferentes valores para la estrategia de generación de **esquema de base de datos**:

- **validate**: valida el esquema pero no ejecuta cambios en la base de datos.
- **create**: crea de nuevo el esquema, borrando los datos anteriores.
- **create-drop**: borra el esquema cuando se cierra el *SessionFactory*, crea el esquema cuando se crea el *SessionFactory*.
- **update**: actualiza el esquema actual con los nuevos cambios en las entidades.
- **none**: no realiza ningún cambio en el esquema.



## 3.6. Asociaciones entre entidades

Una entidad puede tener atributos cuyo tipo sea otra entidad con su propio ciclo de vida y anotación *@Entity*. Cuando esto ocurre se produce una **asociación entre entidades**. Las asociaciones pueden ser de diferente tipo:

- **Uno a uno** (One-To-One): se mapean con la anotación *@OneToOne*.
- **Uno a muchos** (One-To-Many): se mapean con la anotación *@OneToMany*.
- **Muchos a uno** (Many-To-One): se mapean con la anotación *@ManyToOne*.
- **Muchos a muchos** (Many-To-Many): se mapean con la anotación *@ManyToMany*.



## 3.6.1. Asociación One-To-One

La relación de **One-To-One** representa un tipo de asociación de **uno a uno**, en el que una entidad tiene un atributo cuyo tipo es otra entidad diferente.

Se utiliza normalmente para **agrupar datos relacionados** y separarlos a una tabla diferente en lugar de acumularlos todos en una misma entidad.

**Ejemplo:** la entidad *Employee* puede tener una asociación con la entidad *Direction* de tipo uno a uno, en la que un empleado solo tiene una dirección y una dirección únicamente puede pertenecer al mismo empleado.



## 3.6.1. Asociación One-To-One

La **relación de uno a uno** se produce cuando existen dos clases con la anotación *@Entity* y una de ellas tiene un atributo cuyo tipo es de la otra clase.

Se puede implementar de múltiples maneras, siendo las dos más comunes:

- **Asociación con clave foránea:** en una de las dos tablas se crea una columna que almacena el identificador de la otra tabla.
- **Asociación con tabla join:** se crea una nueva tabla auxiliar para establecer la relación entre las claves primarias de las dos tablas de las entidades.



## 3.6.1.1. Asociación con clave foránea

Se implementa con la anotación **@OneToOne**. Genera una nueva columna en la tabla **Employee**, el nombre de la nueva columna se especifica con la anotación **@JoinColumn**. Se genera una relación **unidireccional**.

```
@OneToOne  
@JoinColumn(name = "id_direction") // nueva columna  
private Direction direction;
```





## 3.6.1.1. Asociación con clave foránea

A nivel de base de datos se genera una **nueva columna** en una de las dos tablas apuntando hacia la **clave primaria** de la otra tabla.

Properties Datos Diagrama ER							
employees Enter a SQL expression to filter results (use Ctrl+Space)							
	123 id	123 age	123 married	ABC name	123 salary	123 id_direction	
1	1	32	1	Anthony	40.000	1	
2	2	32	1	Anthony2	40.000	2	



## 3.6.1.1. Asociación con clave foránea

Si únicamente desde un lado de la asociación podemos acceder a la otra entidad, entonces es un **asociación unidireccional**.

Para lograr una **asociación bidireccional** y poder acceder a la otra entidad desde ambos lados de la asociación, se utiliza el argumento *mappedBy*:

```
@OneToOne(mappedBy = "direction") // bidireccional  
private Employee employee;
```



## 3.6.1.1. Asociación con clave foránea

Ejemplo One To One con **clave foránea** para clases *Employee.java* y *Direction.java*:

```
@Entity
@Table(name="employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="name", length = 300)
    private String name;
    private Integer age;
    private Boolean married;
    private Double salary;
    @Column(name="created_date")
    private Instant createdAt;
    @OneToOne
    @JoinColumn(name = "id_direction")
    private Direction direction;
    // ...
}
```

```
@Entity
@Table(name="direction")
public class Direction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String province;
    @Column(name = "postal_code", length = 5)
    private String postalCode;
    private String country;
    @OneToOne(mappedBy = "direction") // bidireccional
    private Employee employee;
    // ...
}
```



## 3.6.1.2. Asociación con tabla join

Se implementa con las anotaciones **@OneToOne** y **@JoinTable**, lo cual genera una nueva tabla auxiliar en la base de datos:

```
@OneToOne
@JoinTable(name="employee_direction",
    joinColumns = {@JoinColumn(name="employee_id", referencedColumnName = "id")},
    inverseJoinColumns = {@JoinColumn(name="direction_id", referencedColumnName="id")}
)
private Direction direction;
```



## 3.6.1.2. Asociación con tabla join

Para que sea una **asociación bidireccional** se mantiene la anotación **@OneToOne** y el argumento *mappedBy* en el otro lado de la asociación:

```
@OneToOne(mappedBy = "direction") // bidireccional  
private Employee employee;
```



## 3.6.1.2. Asociación con tabla join

Ejemplo One To One **con tabla join** para clases *Employee.java* y *Direction.java*:

```
@Entity
@Table(name="employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="name", length = 300)
    private String name;
    private Integer age;
    private Boolean married;
    private Double salary;
    @Column(name="created_date")
    private Instant createdAt;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name="employee_direction",
        joinColumns = {
            @JoinColumn(name="employee_id", referencedColumnName = "id")},
        inverseJoinColumns = {
            @JoinColumn(name="direction_id", referencedColumnName="id")})
    private Direction direction;
    // ...
}
```

```
@Entity
@Table(name="direction")
public class Direction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String province;
    @Column(name = "postal_code", length = 5)
    private String postalCode;
    private String country;
    @OneToOne(mappedBy = "direction") // bidireccional
    private Employee employee;
    // ...
}
```



## 3.6.1.2. Asociación con tabla join

El resultado de asociación One To One con **tabla join** produce una **nueva tabla auxiliar** en la base de datos que mapea las claves primarias de las dos tablas asociadas.

Para el ejemplo de asociación entre *Employee* y *Direction* se genera la nueva tabla **employee\_direction**.

	direction_id	employee_id
1	3	1
2	4	2



## 3.6.2. Asociación One-To-Many

La relación **One-To-Many** establece una asociación de **uno a muchos**. Se produce cuando una entidad tiene un atributo de tipo estructura de datos. Esa estructura de datos almacena uno o más objetos de otra entidad.

Ejemplo: una empresa tiene muchos departamentos, un departamento pertenece a una empresa. En la entidad Company habrá un atributo que será una lista de departamentos:

```
@OneToMany()  
@JoinColumn(name = "company_id")  
List<Department> departments = new ArrayList<>();
```





## 3.6.2. Asociación One-To-Many

Como resultado del ejemplo anterior se genera **un nueva columna** *company\_id* en la tabla de la entidad **Department**:

	id	name	company_id
1	1	Marketing	1
2	2	Development	1
3	3	Management	1



## 3.6.2. Asociación One-To-Many

Ejemplo de relación **One To Many** clases *Company.java* y *Department.java*:

```
@Entity
@Table(name="companies")
public class Company {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String cif;
    @OneToMany()
    @JoinColumn(name = "company_id")
    List<Department> departments = new ArrayList<>();
}
```

```
@Entity
@Table(name="departments")
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    public Department() { }

    // getters and setters ...
}
```



## 3.6.3. Asociación Many-To-One

La relación **Many-To-One** es el lado inverso de la relación **One-to-Many**.

Ejemplo: muchos empleados pertenecen a una misma empresa, una misma empresa tiene muchos empleados. Asociación en la entidad **Employee**:

```
@ManyToOne()  
@JoinColumn(name = "id_company")  
private Company company;
```



## 3.6.3. Asociación Many-To-One

Cada vez que hay una anotación `@JoinColumn` se genera una nueva columna. En el caso de **la entidad Employee** se genera una nueva columna con el id de la empresa. De esta forma más de un empleado puede tener asociado el mismo id de empresa, por tanto se cumple la asociación muchos (empleados) a una (empresa):

Properties Datos Diagrama ER						
employees Enter a SQL expression to filter results (use Ctrl+Space)						
	id	age	married	name	salary	id_company
1	1	34	0	Empleado1	300.000	3
2	2	34	0	Empleado2	300.000	3
3	3	34	0	Empleado3	300.000	3



## 3.6.3. Asociación Many-To-One

En el lado inverso de la relación **Many-To-One** se crea una lista de elementos del tipo de la otra entidad, siendo una asociación de tipo **One-To-Many**.

El dueño de la relación es la entidad **Employee**, el argumento *mappedBy* en la entidad **Company** indica a Hibernate que el mapeo se establece en el otro lado de la asociación, logrando una asociación **bidireccional**.

```
@OneToMany(mappedBy = "company") // Por defecto es LAZY  
private List<Employee> employees = new ArrayList<>();
```



## 3.6.4. Asociación Many-To-Many

En la relación **Many-To-Many** se establece una asociación de **muchos a muchos**.

Ejemplo: un empleado puede trabajar en muchos proyectos, mientras que en un proyecto pueden trabajar muchos empleados. En la entidad *Employee* habrá un atributo **lista de proyectos**, mientras que en la entidad *Project* habrá un atributo **lista de empleados**.

Para implementar este tipo de asociación se genera una **nueva tabla auxiliar** mediante la anotación *@JoinTable*.



## 3.6.4. Asociación Many-To-Many

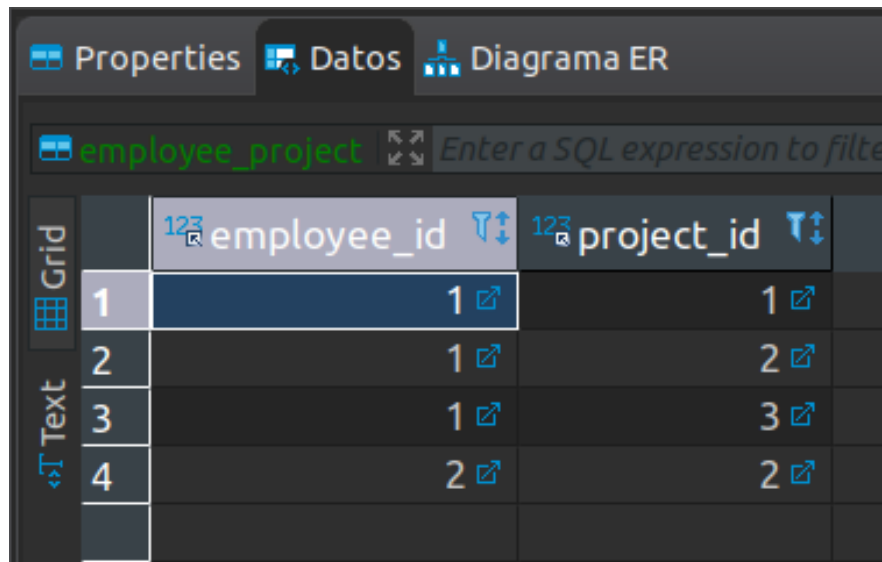
En la entidad *Employee* habrá un atributo de tipo lista donde se alojen objetos de tipo *Project*. Mediante la anotación **@ManyToMany** se establece la **relación de Muchos a Muchos**. La anotación **@JoinTable** permite generar la tabla auxiliar:

```
@ManyToMany
@JoinTable(
    name = "employee_project", // nombre de la nueva tabla auxiliar
    // nombre de la primera nueva columna de la tabla de la entidad actual (Employee):
    joinColumns = {@JoinColumn(name="employee_id", referencedColumnName = "id")},
    // nombre segunda nueva columna de la tabla de la entidad relacionada (Project):
    inverseJoinColumns = {@JoinColumn(name="project_id", referencedColumnName = "id")}
)
private List<Project> projects = new ArrayList<>();
```



## 3.6.4. Asociación Many-To-Many

El resultado de la asociación **Many-To-Many** es una **nueva tabla** con claves foráneas hacia las claves primarias de las dos tablas de las entidades relacionadas.



	employee_id	project_id
1	1	1
2	1	2
3	1	3
4	2	2





## 3.6.4. Asociación Many-To-Many

Para que la asociación **Many To Many** sea **bidireccional** en la otra entidad (*Project*) hay que añadir el atributo con la anotación y el argumento *mappedBy*. De esta manera Hibernate sabrá que la asociación se mapea en la entidad *Employee*, que es donde están los datos para la nueva tabla auxiliar:

```
// Bidireccional
@ManyToMany(mappedBy = "projects")
private List<Employee> employees = new ArrayList<>();
```



## 3.6.4. Asociación Many-To-Many

Ejemplo asociación **Many To Many** entre clases *Employee.java* y *Project.java*:

```
@Entity
@Table(name="employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // ... otros campos ...

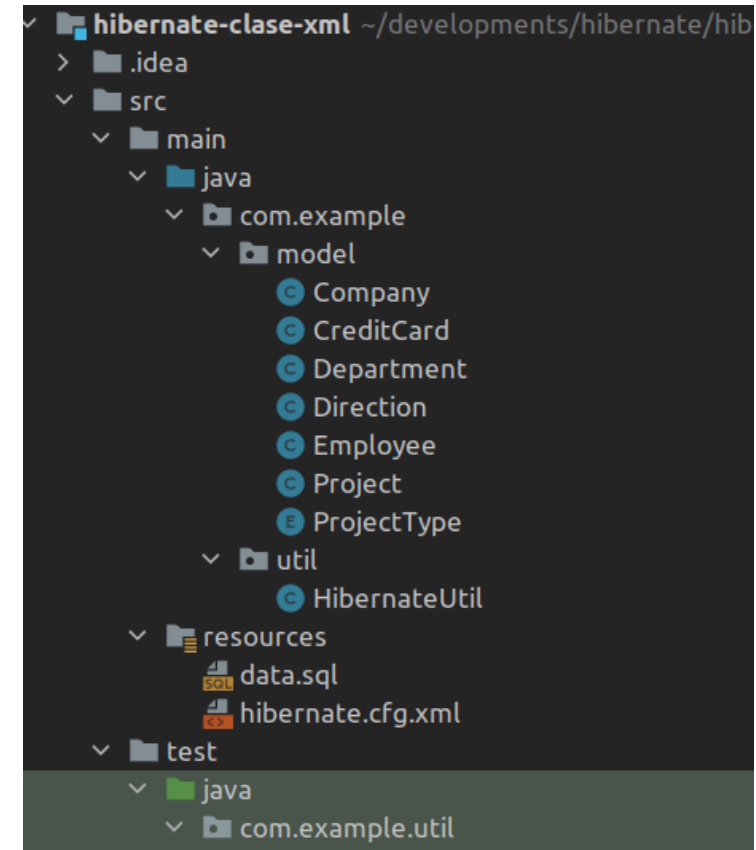
    @ManyToMany
    @JoinTable(
        // nombre de la nueva tabla auxiliar
        name = "employee_project",
        // nombre de la primera nueva columna
        // de la tabla de la entidad actual (Employee)
        joinColumns = {
            @JoinColumn(name="employee_id", referencedColumnName = "id")},
        // nombre de la segunda nueva columna
        // de la tabla de la entidad relacionada (Project)
        inverseJoinColumns = {
            @JoinColumn(name="project_id", referencedColumnName = "id")}
    )
    private List<Project> projects = new ArrayList<>();
}
```

```
@Entity
@Table(name="projects")
public class Project {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String code;
    private String description;
    @Enumerated(EnumType.STRING)
    private ProjectType type;
    // Bidireccional
    @ManyToMany(mappedBy = "projects")
    private List<Employee> employees = new ArrayList<>();
}
```



## 3.6.5. Ejemplo completo de asociaciones

En el proyecto **hibernate-clase-xml** subido a la plataforma se puede observar las anteriores asociaciones puestas en práctica y casos de testing para verificar los datos insertados en la base de datos al guardar las entidades.





## 3.7. Operaciones en cascada

Las **operaciones en cascada** permiten propagar una operación de persistencia hacia una entidad asociada. Ejemplo de configuración de una cascada en la entidad **Employee**:

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "id_direction") // nueva columna
private Direction direction
```

De esta forma cuando se guarda un **Employee** se guarda la **Direction**, si se borra un **Employee** se borra la **Direction** asociada.



## 3.7. Operaciones en cascada

Con la opción **CascadeType.ALL** se propagan todas las operaciones, si se quiere propagar únicamente una operación concreta y no todas entonces se debe especificar la operación:

```
@OneToOne(cascade = CascadeType.REMOVE) // solo operación borrado  
@JoinColumn(name = "id_direction") // nueva columna  
private Direction direction
```



## 3.8. Opción orphanRemoval

El argumento **orphanRemoval** permite configurar el borrado automático de un elemento cuando es desasociado de otro. Al desasociar una **CreditCard** de una **Company**, es lógico que se borre la **CreditCard** para evitar que existan tarjetas de crédito no vinculadas a ninguna empresa. Asociación en la entidad **Company**:

```
@OneToMany(orphanRemoval = true, cascade = CascadeType.ALL)
@JoinTable(
    name = "company_creditcard",
    joinColumns = @JoinColumn(name = "company_id", referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name="creditcard_id", referencedColumnName = "id")
)
private List<CreditCard> creditCards = new ArrayList<>();
```



## 3.9. Listeners

Los **listeners** permiten ejecutar fragmentos de código antes o después de que se realicen determinadas operaciones en la base de datos. Ejemplo de listener en la propia clase entidad que se ejecuta antes de crear o actualizar una instancia **Employee** para asignar un tipo de categoría profesional en caso de que no se haya asignado:

```
@PreUpdate
@PrePersist
void prePersist() {
    if(Objects.isNull(this.type))
        this.type = EmployeeType.JUNIOR;
}
```



## 4. El lenguaje HQL

**Hibernate Query Language** o **HQL** es un lenguaje similar al lenguaje SQL pero orientado a objetos. En lugar de utilizar los nombres de las tablas y de las columnas, HQL utiliza el nombre de la entidad y de sus atributos para construir las sentencias.

De esta forma se aprovechan las ventajas de la programación orientada a objetos al interactuar con la base de datos. Finalmente Hibernate traduce las **sentencias HQL a código SQL** para su ejecución en la base de datos correspondiente.





## 4.1. Introducción a HQL

Al utilizar **HQL** se mejora la **portabilidad entre bases de datos** debido a que Hibernate traduce las sentencias HQL a sentencias SQL específicas de la base de datos con la que opera. Como desventaja puede ser más lento que el propio SQL al tener que traducirse a este.

Para crear sentencias HQL se utiliza la **API Query**. Desde la versión Hibernate 5.2 [org.hibernate.query.Query](https://hibernate.org/orm/javadocs/org/hibernate/query/Query.html) extiende de [javax.persistence.Query](https://docs.oracle.com/javase/8/docs/api/javax/persistence/Query.html).



## 4.2. Sentencias select

Para ejecutar una sentencia HQL se crea una query por medio del objeto Session. Ejemplo de sentencia que devuelve una **lista de objetos Employee**:

```
Session session = HibernateUtil.getSessionFactory().openSession();

List<Employee> employees =
session.createQuery("from Employee", Employee.class ).list();

session.close();
```



## 4.2. Sentencias select

Para ejecutar una sentencia HQL se crea una sentencia por medio del objeto Session. Ejemplo de sentencia que devuelve un único resultado filtrando por el id del empleado, utilizando **parámetro por nombre**:

```
Session session = HibernateUtil.getSessionFactory().openSession();
String hql = "from Employee where id = :idEmpleado"; // named parameters
Query query = session.createQuery(hql);
query.setParameter("idEmpleado", 1L);
Employee employee = (Employee) query.getSingleResult();
```



## 4.2. Sentencias select

Para ejecutar una sentencia HQL se crea una sentencia por medio del objeto Session. Ejemplo de sentencia que devuelve un único resultado filtrando por el id del empleado, utilizando **parámetro por posición**. Se pueden concatenar condiciones utilizando operadores **and** y **or**:

```
Session session = HibernateUtil.getSessionFactory().openSession();
String hql = "from Employee where id = ?1 and age < ?2"; // position param
Query query = session.createQuery(hql);
query.setParameter(1, 1L);
query.setParameter(2, 30);
Employee employee = (Employee) query.getSingleResult();
```



## 4.2. Sentencias select

El objeto sesión proporciona un **método find** equivalente a una búsqueda HQL con filtro para encontrar un registro por su clave primaria:

```
Session session = HibernateUtil.getSessionFactory().openSession();  
Employee employee = session.find(Employee.class, 1L);
```



## 4.3. Sentencias insert

Para realizar operaciones que permitan **modificar datos** en base de datos como por ejemplo insertar, actualizar o borrar es necesario ejecutar la operación en el **contexto de una transacción**. Para ello se inicia una nueva transacción y se hace efectiva en base de datos con el método **commit()**.

```
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
// operaciones que modifican datos ...
session.getTransaction().commit();
session.close();
```



## 4.3. Sentencias insert

Ambos lenguajes JPQL y HQL admiten sentencias de tipo **SELECT**, **UPDATE** y **DELETE**. Las sentencias **INSERT** permiten insertar nuevos datos en la base de datos. Sólo **HQL admite sentencias de tipo INSERT**, permitiendo insertar únicamente resultados de otra consulta SELECT de HQL:

```
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
Query query = session.createQuery("insert into Employee (name, age, active) "+
    "select name, age, active from Employee where active=:active");
query.setParameter("active", false);
int rowsCopied=query.executeUpdate();
session.getTransaction().commit();
```



## 4.4. Sentencias update

Las **sentencias UPDATE** modifican datos en la base de datos, se utilizan para actualizar datos existentes.

```
session.beginTransaction();
int updatedEntities = session.createQuery(
    "update Employee " +
        "set name = :newName " +
        "where name = :oldName" )
    .setParameter( "oldName", "Mike" )
    .setParameter( "newName", "Mike updated" )
    .executeUpdate();
session.getTransaction().commit();
```





## 4.5. Sentencias delete

Las **sentencias DELETE** permiten borrar datos existentes de la base de datos.

```
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
Query query = session.createQuery("delete from Employee");
int rowsDeleted = query.executeUpdate();
System.out.println(rowsDeleted);
session.getTransaction().commit();
session.close();
```

**Nota:** si no se aplican filtros *WHERE* se borrarán todos los registros de la tabla.



## 5. API Criteria

Desde la versión 2.0 de JPA se incluye una API para la **creación de sentencias de forma programática**, desde el código Java. Mediante la **API Criteria** se puede crear una consulta de forma segura utilizando únicamente métodos java, sin necesidad de emplear directamente SQL, JPQL ni HQL.

Hibernate desarrolló su propia API Criteria bajo el paquete [org.hibernate.Criteria](http://org.hibernate.Criteria) que desde la versión 5.2 se considera deprecada en favor de la propia **API Criteria definida en JPA** bajo el paquete [javax.persistence.criterio.CriteriaQuery](http://javax.persistence.criterio.CriteriaQuery).



## 5.1. Introducción a Criteria API

La **API Criteria** permite construir sentencias aprovechando las características de la Programación Orientada a Objetos, creando objetos *CriteriaQuery* sobre los que aplicar todo tipo de filtros y condiciones lógicas.

Una ventaja clara del uso de API Criteria es que los errores se pueden detectar en tiempo de **compilación**, así como la **flexibilidad** que aporta el lenguaje Java. Se emplea normalmente para la construcción de consultas complejas de forma **dinámica**.



## 5.2. Estructura Criteria

**CriteriaBuilder** actúa como una factoría que permite construir las consultas **CriteriaQuery**. Dentro de **CriteriaQuery** el método *from* permite indicar el origen donde buscar los datos y diferentes métodos como *select* y *multiselect* permiten indicar los datos concretos que se quieren recuperar.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Employee> criteria = criteriaBuilder.createQuery(Employee.class);
Root<Employee> root = criteria.from(Employee.class);
criteria.select(root);
Query<Employee> query = session.createQuery(criteria);
List<Employee> results = query.getResultList();
```



## 5.3. Filtros

El **filtrado de datos** en las consultas es un mecanismo muy empleado a la hora de recuperar datos. Con el método *where* de **CriteriaQuery** se pueden asignar filtros. Ejemplo para filtrar aquellos empleados cuyo atributo *active* es igual a *true*:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Object[]> criteria = builder.createQuery(Object[].class);
Root<Employee> root = criteria.from(Employee.class);

criteria.multiselect(root.get("name"), root.get("age"));
criteria.where(builder.isTrue(root.get("active")));

Query<Object[]> query = session.createQuery(criteria);
List<Object[]> list = query.getResultList();
```



## 5.3. Filtros

Las condiciones de filtrado se pueden **concatenar** mediante métodos *and* y *or*.

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Employee> criteria = builder.createQuery(Employee.class);
Root<Employee> root = criteria.from(Employee.class);
criteria.select(root);
Predicate ageGreater20 = builder.gt(root.get("age"), 20); // greater than
Predicate ageLess30 = builder.lt(root.get("age"), 30); // less than
criteria.where(builder.and(ageGreater20, ageLess30));
List<Employee> employees = session.createQuery(criteria).list();
employees.forEach(employee -> System.out.println(employee.getName()));
```



## 5.3. Filtros

En las **condiciones de filtrado** también se pueden emplear filtros especiales como la cláusula *like* para obtener aquellos elementos que contienen un texto en un atributo:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Employee> criteria = builder.createQuery(Employee.class);
Root<Employee> root = criteria.from(Employee.class);
criteria.select(root);

criteria.where(builder.like(root.get("name"), "A%"));

List<Employee> employees = session.createQuery(criteria).list();
```



## 5.4. Operaciones agregadas

Criteria API también permite utilizar las **funciones agregadas de SQL** como por ejemplo *AVG()*, *COUNT()*, *MAX()*, *MIN()*, *SUM()* por medio de métodos específicos:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Double> criteria = builder.createQuery(Double.class);
Root<Employee> root = criteria.from(Employee.class);

criteria.select(builder.sum(root.get("salary")));

Double totalCosts = session.createQuery(criteria).getSingleResult();
System.out.println(totalCosts);
```





## 5.5. Actualizar datos

Al igual que ocurre con SQL/JPQL/HQL, Criteria API también permite ejecutar sentencias que puedan **modificar datos**. Ejemplo de actualización de datos, cambia CriteriaQuery por **CriteriaUpdate**:

```
CriteriaUpdate<Employee> criteria = builder.createCriteriaUpdate(Employee.class);
Root<Employee> root = criteria.from(Employee.class);
criteria.set("age", 199);
criteria.set("salary", 100000D);
// criteria.where(...) // aplicar filtros si es necesario
Query<Employee> query = session.createQuery(criteria);
session.beginTransaction();
query.executeUpdate(); // executeUpdate para efectuar los cambios
session.getTransaction().commit();
```



## 5.6. Borrar datos

De forma similar a la actualización de datos, también se **pueden borrar**. Cambia CriteriaQuery por **CriteriaDelete**:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaDelete<Employee> criteria = builder.createCriteriaDelete(Employee.class);
Root<Employee> root = criteria.from(Employee.class);
criteria.where(builder.lt(root.get("age"), 30L));
session.beginTransaction();
session.createQuery(criteria).executeUpdate();
session.getTransaction().commit();
```

# JPA - Hibernate



***Everything data***

