

# UD 3. OBJECT/RELATIONAL MAPPING (ORM). JPA E HIBERNATE

- [Introducción ORM](#)
- [Persistencia en Sistemas de Bases de Datos](#)
- [Técnicas de persistencia](#)

## Introducción ORM

La **persistencia** consiste en almacenar los datos de forma permanente.

La persistencia se puede realizar mediante **ficheros** (planos, XML, JSON,...) o **sistemas de base de datos** (relacionales, orientados a objetos, JSON, XML, etc.).

En esta unidad vamos a estudiar el almacenamiento en bases de datos relacionales por medio de **mapeo objeto-relacional (ORM)** y su implementación en Java mediante **JPA con Hibernate o EclipseLink**, entre otros.

El **uso de ficheros** se recomienda en pocos casos, como por ejemplo, para almacenar datos de configuración de la aplicación.

Entre las **desventajas** están:

- **Redundancia de datos**: puede haber **datos duplicados** en diferentes ficheros.
- **Complejidad de acceso a datos**: un **cambio en los datos puede requerir cambios en la aplicación**.
- **Seguridad**: en un conjunto de ficheros es más **complicado establecer permisos**, en SGBD se implantan de forma nativa.
- **Concurrencia**: se precisa establecer un sistema de bloqueo de ficheros para evitar que dos usuarios accedan al mismo tiempo a un fichero.
- **Integridad** de datos: no se pueden establecer restricciones de integridad referencial. Que viene impuesta por la aplicación.
- **No se puede realizar consultas complejas ni por índices**.

## Persistencia en Sistemas de Bases de Datos

Pueden utilizarse diferentes tipos de bases de datos:

- **Bases de datos relacionales**: son las más utilizadas. Se basan en el modelo relacional de datos. Los datos se almacenan en tablas y se relacionan entre sí mediante claves primarias y foráneas. Ejemplos son:

- [MariaDB](#).
- [PostgreSQL](#).
- [Oracle](#).
- [MySQL](#).
- [H2 Database Engine](#).
- [HSQLDB](#).
- [SQLite](#).
- [SQL Server](#).
- [DB2](#).
- [Access](#).
- [Derby](#).

- [DB2](#).
- [Informix](#).
- [Firebird](#).
- Otros: [Sybase](#), [Teradata Database](#), [Ingres](#), [Adabas D](#), [Progress](#), [InterBase](#).

- **Bases de datos orientadas a documentos:** se basan en el modelo de datos orientado a documentos.

- [MongoDB](#).
- [Databricks](#). Databricks es el nombre de la plataforma analítica de datos basada en Apache Spark desarrollada por la compañía con el mismo nombre. La empresa se fundó en 2013 con los creadores y los desarrolladores principales de Spark. Permite hacer analítica Big Data e inteligencia artificial con Spark de una forma sencilla y colaborativa. Esta plataforma está disponible como servicio cloud en Microsoft Azure y Amazon Web Services (AWS)
- [Amazon DynamoDB](#).
- [Azure Cosmos DB](#).
- [Firebase Realtime Database](#).
- [Cloud Firestore](#).
- Otros SGBD documentales: [CouchDB](#), [RavenDB](#), [Couchbase](#), [MarkLogic](#), [OrientDB](#), [ArangoDB](#), [RethinkDB](#), [Cosmos DB](#), [Amazon DocumentDB](#), [Elasticsearch](#).

- **Bases de datos orientadas a objetos:** se basan en el modelo orientado a objetos.

Los datos se almacenan en objetos. Ejemplos son:

- [ObjectDB](#)
- [db4o](#)
- [Action NoSQL Databases](#)
- Otros: [Versant](#), [ZODB](#)...

## Técnicas de persistencia

1. **JDBC** (Java Database Connectivity) Nativa: es una API de Java que **permite ejecutar sentencias SQL y procedimientos almacenados en un SGBD**.
2. **DAO** (Data Access Object): es un **patrón de diseño que permite separar la lógica de negocio de la lógica de acceso a datos**.  
Cada clase del modelo de datos tiene su clase DAO asociada con método para realizar las operaciones CRUD (Create, Read, Update, Delete).
3. **Frameworks de persistencia/ORM (Object/Relational Mapping)**: son librerías que permiten realizar la persistencia de datos de forma transparente.
  1. **JPA** (Java Persistence API): es una especificación de una API de Java que permite mapear objetos Java a tablas de una base de datos relacional.
  2. **Implementaciones de JPA o nativas**: existen varias implementaciones de la especificación JPA, pero la más popular es [Hibernate](#), un **framework de persistencia que implementa la especificación JPA**. Otras:
    - [EclipseLink](#).
    - [OpenJPA](#).
    - [DataNucleus](#).
    - [TopLink](#).
    - [Batoo JPA](#).
    - Kundera..

## 03.01. JAKARTA PERSISTENCE (JPA).

---

- [Mapeo Objeto-Relacional \(ORM\)](#)
- [Referencias](#)

# Mapeo Objeto-Relacional (ORM)

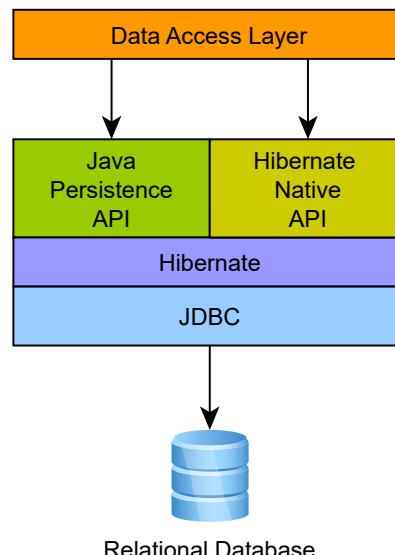
Mapeo Objeto-Relacional (ORM) es el proceso de **convertir objetos Java en tablas de bases de datos**. Esto permite interactuar con una base de datos relacional **sin necesidad de utilizar SQL**.

**Jakarta/Java Persistence API (JPA)** es una **especificación** que define cómo persistir datos en aplicaciones Java. El enfoque principal de JPA es la capa de ORM.

**Hibernate** es uno de los frameworks de ORM más populares en uso hoy en día y **una implementación estándar de la especificación JPA**, con algunas características adicionales específicas de Hibernate. Su primera versión se lanzó hace casi veinte años y aún cuenta con un excelente soporte de la comunidad y lanzamientos regulares.

En esta unidad **nos centraremos en Jakarta Persistence API (JPA) con Hibernate**, aunque también veremos alguna otra implementación de referencia de JPA, como [EclipseLink](#) o [DataNucleus](#).

**SpringBoot Data JPA utiliza Hibernate como implementación de JPA**, pero, en cuanto a rendimiento, no es la mejor opción.



## Referencias

- API de Jakarta Persistence (JPA): <https://jakarta.ee/specifications/persistence/>
- JPA 3.1 Javadoc: <https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/module-summary>
- JPA 3.2 Javadoc: <https://jakarta.ee/specifications/persistence/3.2/apidocs/jakarta.persistence/module-summary>
- Hibernate 6.6 documentación: <https://hibernate.org/orm/documentation/6.6/>
- Hibernate 6 Guide: [https://docs.jboss.org/hibernate/orm/6.6/introduction/html\\_single/Hibernate\\_Introduction.html](https://docs.jboss.org/hibernate/orm/6.6/introduction/html_single/Hibernate_Introduction.html)

---

Autor/a: Pepe Calo Última actualización: 13.02.2025

# 01. JAKARTA PERSISTENCE (JPA).

---

- [1. Jakarta Persistence](#)

- [1.1. Historia](#)
- [1.2. Las versiones de Jakarta Persistence](#)
  - [Java Persistence API 2.0 \(2009\)](#)
  - [Java Persistence 2.1 \(2013\)](#)
  - [Java Persistence 2.2 \(2017\)](#)
  - [Jakarta Persistence 3.0 \(2020\)](#)
  - [Jakarta Persistence 3.1 \(2021\)](#)
- [1.3. Referencias](#)
- [2. Jakarta Persistence \(JPA\)](#)
  - [2.2. Implementaciones JPA](#)
    - [Dependencias Maven](#)
  - [2.3 Fichero de configuración persistence.xml](#)
  - [2.3. Entidades/Entity](#)
  - [2.4. Relaciones](#)
  - [2.5. Tipos de relaciones](#)

## 1. Jakarta Persistence

Desde los primeros días de la plataforma Java, han existido **interfaces de programación para proporcionar pasarelas hacia la base de datos** y para **abstraer las necesidades de persistencia** específicas del dominio de las aplicaciones empresariales.

Jakarta Persistence define un **estándar para la gestión de persistencia y el mapeo objeto/relacional en entornos Java basado en POJO (Plain Old Java Object)** para la persistencia en Java.

**Jakarta Persistence es sólo una especificación** que no puede realizar la persistencia por sí misma. Por supuesto, Jakarta Persistence **requiere una base de datos para persistir**.

El API Jakarta para la gestión de persistencia y el mapeo objeto/relacional **puede emplearse en Jakarta EE o Java SE**.

En la actualidad, existen varias soluciones de persistencia en Java. Nos centraremos en la especificación JPA y soluciones propietarias como **Hibernate**, **EclipseLink**, **DataNucleus**, etc.

La API de Persistencia de Jakarta consta de cuatro áreas:

- La **Jakarta Persistence API (JPA)**
- La API de Criterios de Persistencia de Jakarta (**Jakarta Persistence Criteria API**)
- El Lenguaje de Consulta de Persistencia de Jakarta (**JPQL**,*Jakarta Persistence Query Language*)
- **Metadatos de mapeo objeto-relacional**

### 1.1. Historia

La API de Java Persistence (JPA) es una **especificación** de Java EE que describe cómo administrar datos relacionales en aplicaciones empresariales de Java. La API de JPA **se basa en la especificación de Java Data Objects (JDO)**, especificación de Java EE que describe cómo administrar datos en aplicaciones empresariales de Java.

- **JPA 1.0**: la fecha de lanzamiento final de la especificación JPA 1.0 fue el 11 de mayo de 2006 como parte del Java Community Process JSR 220.
- **JPA 2.0** se lanzó el 10 de diciembre de 2009 (la plataforma Java EE 6 requiere JPA 2.0).
- **JPA 2.1** se lanzó el 22 de abril de 2013 (la plataforma Java EE 7 requiere JPA 2.1).
- **JPA 2.2** se lanzó en el verano de 2017.
- **JPA 2.3** se lanzó en el verano de 2019.
- **JPA 3.0** se lanzó en el verano de 2020. Fue renombrada a **Jakarta Persistence 3.0** (requiere Java 8). Así, todos los paquetes se renombraron de `javax.persistence` a `jakarta.persistence`. Implementaciones:
  - **Hibernate** (desde versión 5.5)
  - **EclipseLink** (desde versión 3.0)

- **DataNucleus** (desde versión 6.0)
- **JPA 3.1** se lanzó en la primavera de 2022 como parte de Jakarta EE 10 (requiere Java 11). Implementaciones:
  - **Hibernate** (desde versión 6.0)
  - **EclipseLink** (desde versión 4.0)
  - **DataNucleus** (desde versión 6.0)
- **JPA 3.2** se lanzó el 30 de abril de 2024. Implementaciones:
  - **Hibernate** (desde versión 7.0), actualmente en desarrollo (<https://hibernate.org/orm/releases/7.0/>). Compatible con Java 17, 21 y 23. Jakarta EE 11. Esta versión de Hibernate está en desarrollo y no se recomienda para producción (Beta3):  
<https://mvnrepository.com/artifact/org.hibernate/hibernate-core>
  - **EclipseLink** (desde versión 5.0), actualmente en desarrollo (<https://projects.eclipse.org/projects/ee4j.eclipselink>). Esta versión de EclipseLink está en desarrollo y no se recomienda para producción (Beta 5):  
<https://mvnrepository.com/artifact/org.eclipse.persistence/eclipselink>

La novedades de **Jakarta Persistence 3.2** se pueden encontrar en [este enlace](#):  
<https://jakarta.ee/specifications/persistence/3.2/>

## 1.2. Las versiones de Jakarta Persistence

La especificación **Jakarta Persistence 3.1** es la primera versión con nuevas características y mejoras después de que la especificación se trasladara a la Eclipse Foundation ([jakarta.persistence](#)).

### Java Persistence API 2.0 (2009)

La segunda versión **Java Persistence 2.0 en 2009**. Incluyó varias características que no estaban presentes en la primera versión:

- Capacidades de **mapeo adicionales**.
- Formas flexibles de determinar la forma en que el proveedor accedía al estado de la entidad.
- **Extensiones al Lenguaje de Consulta de Persistencia de Java (JPQL)**.
- Nueva **API de Criterios de Java**, una forma programática de **crear consultas dinámicas**.

### Java Persistence 2.1 (2013)

**Java Persistence 2.1 en 2013** agregó algunas características:

- Soporte para **generación de esquemas**.
- Métodos de **conversión de tipos**.
- **Creación de gráficos de entidades y pasarlos a consultas**, lo que se conoce comúnmente como **restricciones de grupo** de recuperación en el conjunto de objetos devueltos.
- Contextos de persistencia no sincronizados para operaciones conversacionales mejoradas.
- **Soporte para procedimientos almacenados**.
- Inyección en clases de escuchadores de entidades.
- **Mejoras en el lenguaje de consulta de Java Persistence**, la API de criterios y en el mapeo de consultas nativas.

### Java Persistence 2.2 (2017)

Java Persistence 2.2 fue publicada por Oracle en **junio de 2017**:

- Métodos para recuperar los resultados de las consultas ([Query](#)) y consultas tipadas ([TypedQuery](#)) como flujos ([streams](#)).
- Soporte para **tipos básicos de Fecha y Hora de Java 8**: `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.LocalTime`, `java.time.OffsetTime` y `java.time.OffsetDateTime`.
- Permitir que los **convertidores de atributos admitan la inyección de CDI**.

- Actualización del mecanismo de **descubrimiento del proveedor de persistencia**.
- Permitir que todas las **anotaciones de Java Persistence se utilicen en metaanotaciones**.

Jakarta Persistence 2.2 se puede encontrar [aquí](#).

## Jakarta Persistence 3.0 (2020)

Jakarta Persistence 3.0, **lanzada en 2020**, fue el **cambio al espacio de nombres del paquete jakarta**.

**Trasladó las API existentes del paquete** `javax.persistence` **al paquete** `jakarta.persistence`. Todas las propiedades que contienen `javax` como parte del nombre se renombran de manera que `javax` se reemplace con `jakarta`.

Actualización de los espacios de nombres del esquema para un archivo de configuración de unidad de persistencia y un **archivo XML de mapeo objeto-relacional**.

## Jakarta Persistence 3.1 (2021)

El lanzamiento de **Jakarta Persistence 3.1** fue publicado por la Eclipse Foundation en **diciembre de 2021**.

En general, los cambios en Jakarta Persistence 3.1 incluyeron:

- Estandarización de la función **EXTRACT en el Lenguaje de Consulta de Persistencia de Jakarta**.
- Estandarización de la **Generación de UUID para claves primarias**.
- Definición del nombre del **módulo** `jakarta.persistence` para la API de Persistencia de Jakarta para el Sistema de Módulos de Plataforma Java.
- Permitir que las **interfaces** `EntityManagerFactory` y `EntityManager` **extiendan la interfaz** `java.lang.AutoCloseable`.
- Actualizaciones editoriales y aclaraciones en la especificación.

Para obtener una lista completa de cambios, consulta la sección de Historial de Revisiones del Documento de Especificaciones disponible en [este enlace](#).

### 1.3. Referencias

- Página **Jakarta Persistence**: <https://jakarta.ee/specifications/persistence/>.
- **Especificación** Jakarta Persistence 3.1: <https://jakarta.ee/specifications/persistence/3.1/>.
- (PDF) **Especificación** Jakarta Persistence 3.1: <https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.pdf>.
- **Javadoc de Jakarta Persistence 3.1**:  
<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/module-summary.html>.

Dependencias Maven, Gradle, Ivy, SBT para **Jakarta Persistence 3.1**:

- <https://mvnrepository.com/artifact/jakarta.persistence/jakarta.persistence-api/3.1.0>
- <https://search.maven.org/artifact/jakarta.persistence/jakarta.persistence-api/3.1.0/jar>.

```
<dependency>
  <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
  <version>3.1.0</version>
</dependency>
```

Versiones:

- [Jakarta Persistence 3.2](#). **En desarrollo las implementaciones**.
- [Jakarta Persistence 3.1](#)
- [Jakarta Persistence 3.0](#)

- [Jakarta Persistence 2.2](#)
- [Jakarta Persistence 1.0 \(JSR 220\)](#)

## 2. Jakarta Persistence (JPA)

Jakarta Persistence, anteriormente conocida como **Java Persistence API**, es una **especificación de interfaz de programación de aplicaciones de Jakarta EE que describe la gestión de datos relacionales en aplicaciones empresariales de Java**.

Como se ha comentado, JPA abarca cuatro áreas:

1. La API en sí, definida en el paquete `jakarta.persistence` (`javax.persistence` para Jakarta EE 8 y versiones anteriores).
2. La API de Criterios de Persistencia de Jakarta (***Jakarta Persistence Criteria API***)
3. El **Lenguaje de Consulta de Jakarta Persistence (JPQL**; anteriormente Lenguaje de Consulta de Java Persistence) que permite realizar consultas a una base de datos relacional obteniendo colecciones de objetos.
4. **Metadatos objeto/relacional**: la configuración puede hacerse con **anotaciones (@Id, @Entity,...)** o **mediante ficheros XML**.

### Características:

- JPA es una **especificación (no implementación)** que facilita el **mapeo objeto-relacional para gestionar datos relacionales en aplicaciones Java**.
- No se puede utilizar JPA directamente. **Deben emplearse implementaciones ORM** como [Hibernate](#), [EclipseLink](#), [MyBatis](#) (antes [IBatis](#)), [DataNucleus](#),... que emplean la especificación de JPA.
- La última versión con implementaciones estables es la 3.1, que se lanzó en la primavera de 2022 como parte de Jakarta EE 10 (requiere Java SE 11 o superior).

Algunas de las implementaciones compatibles con esta especificación son:

- [EclipseLink](#) 4.0.0-M3 (o superior)
- [Hibernate](#) 6.0.0.Final (o superior)
- [DataNucleus](#)
- La mayoría de las herramientas ORM como [Hibernate](#), [MyBatis](#) (antes [IBatis](#)) o [EclipseLink](#), que es la implementación de referencia, **implementan este estándar**.
- JPA proporciona soporte para **trabajar directamente con objetos en lugar de utilizar declaraciones SQL**.
- Dispone de un **fichero de configuración denominado `persistence.xml`**.

### Consejo

**JPA** define un proceso de inicio diferente, junto con un formato estándar de archivo de configuración llamado `persistence.xml`. En entornos de **Java™ SE**, se requiere que el proveedor de persistencia (Hibernate, EclipseLink,...) localice cada archivo de configuración de JPA en el classpath en la ruta `META-INF/persistence.xml`.

Los **XML Schemas de Jakarta Persistence** se pueden encontrar en <https://jakarta.ee/xml/ns/persistence/>.

## 2.2. Implementaciones JPA

La API de Jakarta Persistence proporciona métodos para administrar la persistencia de objetos a un almacén de datos relacional. La implementación de referencia para JPA es EclipseLink, pero existen otras que cubren las necesidades de los desarrolladores, como:

- Hibernate: es una solución de Mapeo Objeto/Relacional (ORM) para programas escritos en Java y otros lenguajes que admiten la JVM.
- EclipseLink: es una solución de persistencia de objetos para Java.
- Spring Data JPA: es una biblioteca de Spring que simplifica el acceso a los sistemas de almacenamiento de datos relacionales. Se basa en la tecnología de acceso a datos de Spring y utiliza las características de JPA para simplificar el acceso a los sistemas de almacenamiento de datos relacionales.
- Apache OpenJPA: es una implementación de JPA que puede utilizarse como un almacén de datos independiente o como una extensión de Apache Geronimo.
- Oracle TopLink: es una solución de persistencia de objetos para Java. Desarrollada por Oracle con licencia dual, tanto comercial como de código abierto que derivó en Eclipse Link. Podría decirse que **ya está descontinuado**.
- DataNucleus: es una solución de **persistencia de objetos para Java, OSGi y la plataforma de Google App Engine. Es una implementación de JDO y JPA**.

Por supuesto, también es posible desarrollar una implementación propia de JPA.

**Nos centraremos en Hibernate, que es una de las implementaciones más populares (y mejor) de JPA,** pero podremos utilizar cualquiera de las otras implementaciones.

## Dependencias Maven

Se precisa la especificación de Jakarta Persistence API y la implementación. Para **Hibernate**, la dependencia Maven sería:

```
<!-- Se precisa la dependencia de Hibernate y la de la API de Jakarta Persistence -->
<dependency>
    <groupId>jakarta.persistence</groupId>
    <artifactId>jakarta.persistence-api</artifactId>
    <version>3.1.0</version>
</dependency><!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.6.4.Final</version>
</dependency>
```

### Ejercicio 01.01. Creación de un proyecto con JPA

Para crear un proyecto con JPA y Hibernate, se puede utilizar el asistente de creación de proyectos de Eclipse o IntelliJ IDEA, sin embargo con la versión Community de IntelliJ IDEA no se puede crear un proyecto con JPA a través del asistente. **Crea un proyecto Java Maven y añade las dependencias de Hibernate y la API de Jakarta Persistence.**

## 2.3 Fichero de configuración `persistence.xml`

Los **artefactos (y elementos de configuración)** de la **unidad de persistencia** se suelen empaquetar en un **“archivo de persistencia”**. Un archivo con formato JAR que contiene el archivo `persistence.xml` en el directorio **META-INF** y los archivos de clase de entidad (clases de persistencia).

Para desplegar la aplicación se precisa situar el **archivo de persistencia**, las **clases de aplicación que utilizan las entidades** y los **archivos JAR del proveedor de persistencia** (JDBC) en el classpath cuando se ejecuta el programa.

La configuración que describe la unidad de persistencia se define en un **archivo XML llamado `META-INF/persistence.xml`**. Cada unidad de persistencia tiene un nombre, por lo que cuando una aplicación de referencia desea especificar la configuración para una entidad, solo necesita hacer referencia al nombre de la unidad de persistencia que define esa configuración. Un solo archivo `persistence.xml` puede contener una o más configuraciones de unidades de persistencia con nombres, pero cada unidad de persistencia es independiente y distinta de las demás.

Los ==únicos que necesitamos especificar para este ejemplo son **name**, **transaction-type**, **class** y **properties=0**.

```
<persistence>
    <persistence-unit name="ServicioEmpleado"
        transaction-type="RESOURCE_LOCAL">
        <class>com.pepinho.ad.modelo.Empleado</class>
        <properties>
            <property name="jakarta.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="jakarta.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/EmpServDB;
create=true"/>
            <property name="jakarta.persistence.jdbc.user"
                value="APP"/>
            <property name="jakarta.persistence.jdbc.password"
                value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

Ejemplo de configuración con **Hibernate y H2 en memoria** que indicar el proveedor de persistencia [org.hibernate.jpa.HibernatePersistenceProvider](#), Hibernate, y la base de datos que se va a utilizar, que en el ejemplo H2 en memoria:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="com.pepinho.ad.jpa.example" transaction-type="RESOURCE_LOCAL">
        <description>Ejemplo de unidad de persistencia con Hibernate y H2 en memoria</description>
        <!-- 1. El proveedor de persistencia ES OPCIONAL, pero se recomienda -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> <!-- Hibernate -->
        <!-- para EclipseLink sería:
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider> -->
        <!-- 2. Escanea las clases y las detecta automáticamente. En caso contrario
        habría que indicarlo con "class" -->
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <!-- propiedades de JPA: -->
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <!-- Si usamos la configuración de Hibernate: -->
            <!-- <property name="hibernate.connection.driver_class" value="org.h2.Driver"/> -->
            <property name="jakarta.persistence.jdbc.url"
                value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
            <!-- Si usamos la configuración de Hibernate: -->
            <!-- <property name="hibernate.connection.url" value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
-->
            <property name="jakarta.persistence.jdbc.user" value="sa"/>
            <!-- Si usamos la configuración de Hibernate: -->
            <!-- <property name="hibernate.connection.username" value="sa"/> -->
            <property name="jakarta.persistence.jdbc.password" value="" />
            <property name="jakarta.persistence.schema-generation.database.action" value="drop-and-
create"/> <!-- create, drop-and-create, none, drop -->
            <property name="jakarta.persistence.lock.timeout" value="100"/>
            <property name="jakarta.persistence.query.timeout" value="100"/>
            <property name="jakarta.persistence.validation.mode" value="NONE"/>
            <!-- propiedades de Específicas de Hibernate: -->
            <property name="hibernate.archive.autodetection" value="class, hbm"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.connection.pool_size" value="50"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
<!--
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/> &lt;!&ndash; create-drop,
            update, create, validate &ndash;>-->
            <property name="hibernate.max_fetch_depth" value="5"/>
            <property name="hibernate.cache.region_prefix" value="hibernate.test"/>
            <property name="hibernate.cache.region.factory_class"
                value="org.hibernate.testing.cache.CachingRegionFactory"/>
            <!--NOTE: hibernate.jdbc.batch_versioned_data debe ponerse como "false" en Oracle -->
            <property name="hibernate.jdbc.batch_versioned_data" value="true"/>
            <property name="hibernate.service.allow_crawling" value="false"/>
```

```

<property name="hibernate.session.events.log" value="true"/>
</properties>
</persistence-unit>
</persistence>

```

Para Hibernate con MySQL con JPA 3.1, por ejemplo, sería:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="jpa-hibernate-mysql">
        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="jakarta.persistence.jdbc.url"
            value="jdbc:mysql://localhost:3306/ejemploDBHibernate" />
            <property name="jakarta.persistence.jdbc.user" value="root" />
            <property name="jakarta.persistence.jdbc.password" value="" />
            <property name="jakarta.persistence.schema-generation.database.action" value="create" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>

```

Ejemplos de Dialectos de Hibernate, que se pueden utilizar en la propiedad `hibernate.dialect`, y no son más que clases que implementan la interfaz `Dialect`:

`AbstractHANADialect`, `AbstractTransactSQLDialect`, `CockroachDialect`, `DB2Dialect`, `DerbyDialect`,  
`DialectDelegateWrapper`, `HSQLDialect`, `MySQLDialect`, `OracleDialect`, `PostgreSQLDialect`, `SpannerDialect`.

- Para cambiar de implementación de EclipseLink a Hibernate en la aplicación Java, **sólo se precisa cambiar el fichero de configuración de la aplicación**, denominado `persistence.xml`. Sin embargo, **Hibernate y EclipseLink tienen algunas características específicas que no están incluidas en la especificación JPA**. Por lo tanto, si utilizas estas características específicas, no podrás cambiar de implementación y deberás utilizar la implementación específica, con los respectivos archivos de configuración:

- `hibernate.cfg.xml` para Hibernate y
- `eclipselink.xml` para EclipseLink.

Si se utiliza `persistence.xml`, la especificación sigue siendo la misma. Esa es la ventaja de utilizar JPA.

#### ☒ Ejercicio 01.02. Creación de un archivo de configuración de persistencia

Crea un directorio `META-INF` en el directorio `src/main/resources` y añade un archivo `persistence.xml` con la configuración de la unidad de persistencia con el nombre `com.sanclemente.ad.jpa.exemplo`.

El fichero de configuración `persistence.xml` **debe apuntar a una base de datos H2 en memoria**. Además, debes añadir los Drivers de H2 para que la aplicación pueda conectarse a la base de datos:

```

<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.3.232</version>
</dependency>

```

Ten en cuenta que necesitas crear la base de datos en memoria H2 y añadir las tablas necesarias, por lo que el parámetro `jakarta.persistence.schema-generation.database.action` debe ser "create".

## 2.3. Entidades/Entity



### Entidades, valores y tablas

Una entidad es una clase que **representa un objeto persistente almacenado** en una base de datos relacional.

Para que una clase sea una Entidad debe cumplir:

- Debe ser una **clase POJO (Plain Old Java Object)**: POJO es un objeto Java que no está sujeto a ninguna restricción de las impuestas por la Especificación del lenguaje Java (sin herencias, implementaciones, dependencias de bibliotecas, etc.). Sólo puede tener:
  - Atributos.
  - Constructores.
  - getters y setters (además de métodos de Object...)
- Debe tener un **constructor por defecto NO privado**.
- Puede tener **constructores adicionales** y declararse como abstracta.
- **No** debe ser una **clase interna** (aunque puede ser una clase anidada estática).
- **No puede ser final**.
- Suelen **implantar** `java.io.Serializable` (aunque no es obligatorio en entornos SE).
- Para convertirla en una entidad debe tener la anotación `@Entity`, declarada en `jakarta.persistence.Entity`.
- Debe tener un **identificador** (ID) que se puede definir con la anotación `@Id` (declarada en `jakarta.persistence.Id`). El identificador puede ser de cualquier tipo, aunque lo más habitual es que sea un tipo primitivo o un objeto de tipo `java.lang.Long` o `java.lang.Integer`. Dicho identificador debe ser **único para cada entidad** y está **asociado a la clave primaria de la tabla de la base de datos**.

Ejemplo de declaración de una Entity/clase `Persona`:

```
import jakarta.persistence.*;
import java.util.UUID;

@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // AUTO, SEQUENCE, TABLE, IDENTITY, UUID
    private Long id;

    // @Id
    // @GeneratedValue(strategy = GenerationType.UUID)
    // private UUID id;

    private String nome;

    public Persona() {
    }

    public Persona(String nome) {
        this.nome = nome;
    }

    public Long getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public String toString() {
```

```

        return "Persona{" +
            "id=" + id +
            ", nome='"+ nome + '\'' +
        '}';
    }
}

```

Anotaciones para la **clase** (lo veremos más adelante al detalle):

- **@Entity**: indica que la clase es una **entidad**. Elementos: **name** (String): el nombre de la entidad empleado en las consultas. Por defecto, el nombre de la clase (sin paquete). Por ejemplo: `@Entity(name = "Persoa")`.
- **@Table**: especifica el nombre de la tabla de la base de datos. Si no se indica, el nombre de la tabla es el nombre de la clase. Por ejemplo: `@Table(name = "persona")`. Elementos
  - **name** (String): el nombre de la tabla de la base de datos.
  - **catalog** (String): el nombre del catálogo de la base de datos.
  - **schema** (String): el nombre del esquema de la base de datos.
  - **uniqueConstraints** (de tipo `UniqueConstraint[]`): las restricciones de unicidad de la tabla de la base de datos.
  - **indexes** (Index[]): los índices de la tabla de la base de datos, para generación.

Anotaciones para los **atributos**: por defecto se mapean todos los atributos de la clase, pero se pueden excluir con la anotación **@Transient**.

- **@Id**: Indica que el atributo es la clave primaria de la entidad.
- **@GeneratedValue**: Indica que el valor del atributo es generado automáticamente por el sistema de persistencia. Posibles valores:
  - **AUTO**: El **sistema de persistencia elige la estrategia** de generación de claves primarias.
  - **IDENTITY**: El sistema de persistencia utiliza una columna de tipo **autoincremental**.
  - **SEQUENCE**: El sistema de persistencia utiliza una **secuencia de base de datos**.
  - **TABLE**: El sistema de persistencia utiliza una **tabla adicional de base de datos**.
  - **UUID**: El sistema de persistencia utiliza un **UUID** (JPA 3.1), identificador único universal, que es un número de 128 bits.
- **@Transient**: Indica que el atributo **no es persistente**, es decir, **no se almacena en la base de datos**.
- **@Column**: Indica que el atributo es una **columna de la tabla de la base de datos**. Permite definir el nombre de la columna, el tipo de datos, etc. Por ejemplo: `@Column(name = "nombre", nullable = false, length = 50)`.
  - **name**: Indica el nombre de la columna de la base de datos.
  - **nullable**: Indica si el atributo puede tener valores nulos (true) o no (false).
  - **length**: Indica la longitud máxima del atributo.
  - **unique**: Indica si el atributo debe ser único (true) o no (false).
  - **insertable**: Indica si el atributo se debe insertar en la base de datos (true) o no (false).
  - **updatable**: Indica si el atributo se debe actualizar en la base de datos (true) o no (false).
  - **precision**: Indica el número de dígitos de precisión de un atributo de tipo numérico.
  - **scale**: Indica el número de dígitos decimales de un atributo de tipo numérico.
  - ...

Se mapean automáticamente los atributos de la clase con los campos de la tabla de la base de datos con el mismo nombre. Por ejemplo, el atributo **nome** se mapea con el campo **nome** de la tabla de la base de datos. Los tipos admitidos son los siguientes:

- Tipos primitivos: `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`, `short`.
- Tipos envolventes de los tipos primitivos: `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`, `Byte`, `Short`.
- `String`.
- `java.util.Date`.
- `java.util.Calendar`.
- `java.sql.Date`.
- `java.sql.Time`.
- `java.sql.Timestamp`.

- `java.math.BigDecimal`.
- `java.math.BigInteger`.
- `byte[]`.
- `java.util.UUID`
- `java.time.LocalDate`
- `java.time.LocalDateTime`
- `java.time.LocalTime`
- `java.time.OffsetTime`
- ...

Los atributos que no se pueden mapear automáticamente con los campos de la tabla de la base de datos se deben excluir con la anotación `@Transient` y se deben mapear manualmente con la anotación `@Column`.

Datos temporales:

- `@Temporal`: Indica que el atributo es un dato temporal (`java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`). Por ejemplo: `@Temporal(TemporalType.DATE)`. Así como `@Temporal(TemporalType.TIME)` y `@Temporal(TemporalType.TIMESTAMP)`.

#### Ejercicio 01.03. Creación de una entidad

Crea una entidad Estudiante con `idEstudiante` (`Long`), `nombre`, `apellidos`, `fechaDeNacimiento` y `dirección`. Añade los atributos necesarios y las anotaciones para que sea una entidad. La clave primaria **será** `idEstudiante` de tipo **autoincremental**.

#### Ejercicio 01.04. Creación de una entidad

Crea una clase AppEstudiante que se conecte a la base de datos y añada un estudiante a la tabla de la base de datos.

Aunque lo veremos más adelante, lo que precisamos es crear un gestor de entidades e invocar al método `persist` para añadir un estudiante a la base de datos:

```
public class AppEstudiante {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("com.sanclemente.ad.jpa.exemplo");
        EntityManager em = emf.createEntityManager();

        Estudiante estudiante = new Estudiante("Juan", "Pérez", LocalDate.of(2000, 1, 1), "Calle
Mayor, 1");

        em.getTransaction().begin();
        em.persist(estudiante);
        em.getTransaction().commit();

        // IMPRIME EL ESTUDIANTE PARA VER SI SE HA AÑADIDO CORRECTAMENTE Y TIENE UN ID

        em.close();
        emf.close();
    }
}
```

Para recuperarlo precisamos invocar al método `find` del gestor de entidades:

```
Estudiante estudiante = em.find(Estudiante.class, 1L); // Recupera el estudiante con id 1
```

## 2.4. Relaciones

Una relación es una **"relación" entre dos entidades.**

Puede ser **unidireccional o bidireccional**.

- Una relación unidireccional tiene una entidad de origen y una entidad de destino.
- Una relación bidireccional tiene una entidad de origen y una entidad de destino, pero también tiene una entidad de destino y una entidad de origen. *Una relación bidireccional tiene dos lados: el lado propietario y el lado inverso. El lado propietario de una relación bidireccional determina qué entidad de la relación se actualizará en la base de datos cuando se actualice la relación en el código. El lado inverso de una relación bidireccional se actualiza automáticamente siempre que se actualice el lado propietario.*

## 2.5. Tipos de relaciones

Las relaciones entre entidades pueden ser de los siguientes tipos:

- **Uno a uno:** una entidad de origen se asocia con una entidad de destino. Una entidad de destino también se asocia con una entidad de origen. Por ejemplo, una persona tiene un pasaporte y un pasaporte pertenece a una persona.
- **Uno a muchos:** una entidad de origen se asocia con una colección de entidades de destino. Una entidad de destino se asocia con una entidad de origen. Por ejemplo, una persona tiene varias direcciones y cada dirección pertenece a una persona.
- **Muchos a uno:** una entidad de origen se asocia con una entidad de destino. Una entidad de destino se asocia con una colección de entidades de origen. Por ejemplo, una dirección tiene una persona y una persona pertenece a varias direcciones.
- **Muchos a muchos:** una entidad de origen se asocia con una colección de entidades de destino. Una entidad de destino se asocia con una colección de entidades de origen. Por ejemplo, una persona tiene varios teléfonos y un teléfono pertenece a varias personas.

---

 Autor/a: Pepinho  Última actualización: 13.02.2025

# 02. JPA VS HIBERNATE.

---

- JPA
- Hibernate

## JPA

- **JPA** significa **Java Persistence API** (Interfaz de Programación de Aplicaciones).
- Fue lanzado inicialmente el 11 de mayo de 2006.
- Es una **especificación de Java** que proporciona funcionalidad y estándares para herramientas de Mapeo Objeto-Relacional (ORM).
- Se utiliza para examinar, controlar y persistir datos entre objetos Java y bases de datos relacionales.
- Se considera como una **técnica estándar para el Mapeo Objeto-Relacional**.
- Se le considera como un **enlace entre un modelo orientado a objetos y un sistema de base de datos relacional**.
- Como es una especificación de Java, JPA **no realiza ninguna funcionalidad** por sí misma. Por lo tanto, necesita una implementación. De este modo, para la persistencia de datos, **herramientas ORM como Hibernate implementan las especificaciones de JPA**. Para la persistencia de datos, el paquete `jakarta.persistence` (antes `javax.persistence`) contiene las clases e interfaces de JPA.
- JPA **es solo una especificación**, no es una implementación.
- Es un **conjunto de reglas y pautas para establecer interfaces** para la implementación del mapeo objeto-relacional.
- Necesita algunas clases e interfaces.
- Admite un mapeo objeto-relacional **simple, limpio y asimilado**.
- Admite **polimorfismo e herencia**.
- Pueden incluirse **consultas dinámicas y con nombre en JPA**.

## Hibernate

- Es un **Framework de Java, de código abierto**, ligero y una herramienta de Mapeo Objeto-Relacional (ORM) para el lenguaje Java que simplifica la construcción de aplicaciones Java para interactuar con la base de datos.
- Se utiliza para **guardar objetos Java en el sistema de base de datos relacional**.
- Hibernate es una **implementación de que sigue el estándar de JPA**.
- Ayuda a **mapear los tipos de datos Java a los tipos de datos SQL**.
- Contribuye a JPA.

*Nota: El framework de Hibernate ORM fue inicialmente diseñado por Red Hat. Se lanzó el 23 de mayo de 2007. Es compatible con JVM multiplataforma y está escrito en Java.*

La característica principal de Hibernate es **mapear las clases Java a tablas de base de datos**.

JPA es una especificación. Proporciona funcionalidad y prototipo comunes para las herramientas ORM. Todas las herramientas ORM (como Hibernate) siguen los estándares comunes, ejecutando la misma especificación. Por lo tanto, si necesitamos cambiar nuestra aplicación de una herramienta ORM a otra, podemos hacerlo fácilmente.

Como sabemos, JPA es solo una especificación, lo que significa que no hay implementación. Podemos anotar clases en la medida que queramos con anotaciones de JPA, aunque, nada sucederá sin una implementación. Supongamos que JPA son las pautas que deben seguirse, sin embargo, Hibernate es un código de implementación de JPA que une la API según lo descrito por la especificación de JPA y proporciona la funcionalidad anónima.

Diferencias entre JPA e Hibernate:

JPA	Hibernate
Está descrito en el paquete <code>jakarta.persistence</code> (+3.0) <code>javax.persistence</code> (2.3 o inferior).	Está descrito en el paquete <code>org.hibernate</code> .
Describe el manejo de datos relacionales en aplicaciones Java.	Hibernate es una herramienta de Mapeo Objeto-Relacional (ORM) que se utiliza para guardar objetos Java en un sistema de base de datos relacional.
No es una implementación, es solo una especificación de Java.	Hibernate es una implementación de JPA. Por lo tanto, sigue el estándar común proporcionado por JPA.
Es una API estándar que permite realizar operaciones en la base de datos.	Se utiliza para mapear tipos de datos Java con tipos de datos SQL y tablas de base de datos.
Utiliza Java Persistence Query Language (JPQL) como lenguaje de consulta orientado a objetos.	Utiliza Hibernate Query Language (HQL) como lenguaje de consulta orientado a objetos.
Utiliza la <b>interfaz</b> <code>EntityManagerFactory</code> para interactuar con la fábrica del administrador de entidades para la unidad de persistencia.	Utiliza la <b>interfaz</b> <code>SessionFactory</code> para crear instancias de sesión.
Utiliza la <b>interfaz</b> <code>EntityManager</code> para realizar acciones de crear, leer y eliminar para instancias de clases de entidad mapeadas.	Utiliza la <b>interfaz</b> <code>Session</code> para realizar acciones de crear, leer y eliminar para instancias de clases de entidad mapeadas.
Actúa como una interfaz de tiempo de ejecución entre una aplicación Java y Hibernate.	Actúa como una interfaz de tiempo de ejecución entre una aplicación Java y Hibernate.

La principal diferencia entre Hibernate y JPA es que Hibernate es un framework mientras que JPA son especificaciones de API. Hibernate es la implementación de todas las pautas de JPA.

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 03. EJERCICIO BÁSICO DE JPA.

- [1. Añadir dependencias](#)
- [2. Creación del archivo de configuración persistence.xml](#)
  - [2.1. Para Hibernate](#)
  - [2.2. Para EclipseLink](#)
- [3. Creación de la clase de Entidad Usuario](#)
- [4. Creación del EntityManagerFactory y EntityManager](#)
- [5. Creación del ejemplo de persistencia](#)
- [6. Creación de la clase TestUsuario](#)
- [7. Creación de la clase UsuarioDAO](#)
- [8. Ejercicio. JPA de una biblioteca](#)
  - [8.1. Solución](#)

## 1. Añadir dependencias

Hibernate se divide en varios módulos/artefactos bajo el grupo `org.hibernate.orm`. El artefacto principal se llama `hibernate-core`.

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter
    Pruebas unitarias -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.1</version>
        <scope>test</scope>
    </dependency>

    <!-- Dependencias para conexiones a bases de datos.
        Sólo necesitamos la que vayamos a emplear. -->
    <!-- https://mvnrepository.com/artifact/com.h2database/h2
        Ojo con la versión. Si empleamos la versión 2.2.224 tendremos
        que gestionar la versión de Driver en DBeaver -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>2.3.232</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.4</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.1.0</version>
    </dependency>

    <!-- JPA -->
    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>3.1.0</version>
    </dependency>

    <!-- Implementaciones JPA. Usaremos una u otra-->
    <!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.6.4.Final</version>
    </dependency>
```

```
<!-- https://mvnrepository.com/artifact/org.eclipse.persistence/org.eclipse.persistence.jpa -->
<!--
<dependency>
<groupId>org.eclipse.persistence</groupId>
<artifactId>org.eclipse.persistence.jpa</artifactId>
<version>4.0.5</version>
</dependency>-->
</dependencies>
```

## 2. Creación del archivo de configuración persistence.xml

JPA define un proceso de arranque diferente al nativo de Hibernate, junto con un formato de **archivo de configuración estándar denominado** `persistence.xml`. En entornos Java™ SE, se requiere que el proveedor de persistencia (Hibernate, EclipseLink, etc.) ubique cada archivo de configuración JPA en la ruta de clases en la ruta `META-INF/persistence.xml`.

Añadidlo al **directorio maven**: `src/main/resources/META-INF/persistence.xml`.

### 2.1. Para Hibernate

Por ejemplo, para hibernate y h2:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">

    <!-- nombre único de la unidad de persistencia-->
    <persistence-unit name="ejemplopersistenciaJPA">
        <description>
            Ejemplo de unidad de persistencia para Jakarta Persistence
        </description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <!--
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>-->
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <!-- Clases que se van a persistir -->
        <class>com.pepinho.ad.orm.Usuario</class>      -->

        <!-- Propiedades de la unidad de persistencia -->
        <properties>
            <!-- Configuración de conexión a base de datos. H2 en memoria. -->
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:E:/ruta/baseDatos;DATABASE_TO_UPPER=FALSE;FILE_LOCK=NO;DB_CLOSE_DELAY=-1" />
            <!--
                <property name="jakarta.persistence.jdbc.url" value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1"
/>-->
            <property name="jakarta.persistence.jdbc.user" value="" />
            <property name="jakarta.persistence.jdbc.password" value="" />
            <!--
                crete: automáticamente, genera el esquema de la base de datos.
                none: no hace nada (la base de datos debe existir)
                create: crea las tablas (si no existen)
                drop-and-create: borra las tablas y las vuelve a crear.
                drop: borra las tablas cuando se cierra la factoría de persistencia, pero no las vuelve
                a crear.
            -->
            <property name="jakarta.persistence.schema-generation.database.action" value="create" /> <!--
                - none, create, drop-and-create, drop -->

            <!-- Muestra por pantalla las sentencias SQL -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.highlight_sql" value="true" />
            <property name="hibernate.globally_quoted_identifiers" value="true"/>
            <!--
                <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />-->
        </properties>
    </persistence-unit>
```

&lt;/persistence&gt;

El archivo `persistence.xml` se definen las propiedades de la base de datos, como el driver, la URL, el usuario y la contraseña. En el ejemplo anterior las propiedades y las etiquetas principales son:

- **provider**: el proveedor de persistencia. En este caso, Hibernate con la clase: `org.hibernate.jpa.HibernatePersistenceProvider`.
- `jakarta.persistence.jdbc.driver`: el **driver** de la base de datos.
- `jakarta.persistence.jdbc.url`: la **URL** de la base de datos.
- `jakarta.persistence.jdbc.user`: el **usuario** de la base de datos.
- `jakarta.persistence.jdbc.password`: la **contraseña** del usuario de la base de datos.
- `jakarta.persistence.schema-generation.database.action`: la acción a realizar sobre la base de datos. En este caso, se **crean (create)** las tablas de la base de datos.
- `hibernate.show_sql`: muestra las sentencias SQL (propio de hibernate).
- `hibernate.format_sql`: formatea las sentencias SQL (propio de hibernate).
- `hibernate.highlight_sql`: resalta las sentencias SQL (propio de hibernate).
- `hibernate.globally_quoted_identifiers`: permite el **uso de comillas dobles en las sentencias SQL** (pone los nombres de las tablas y columnas entre comillas dobles de manera automática) (propio de hibernate).

## 2.2. Para EclipseLink

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">

    <persistence-unit name="default">
        <!--          <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>-->
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="jakarta.persistence.jdbc.url"
                value="jdbc:h2:E:/98 - Bases de
datos/h2/juego/xogos;DATABASE_TO_UPPER=FALSE;FILE_LOCK=NO;DB_CLOSE_DELAY=-1"/>
            <property name="jakarta.persistence.jdbc.user" value="root"/>
            <property name="jakarta.persistence.jdbc.password" value="admin"/>
            <property name="jakarta.persistence.schema-generation.database.action" value="drop-and-
create"/>

            <property name="eclipselink.logging.level" value="INFO"/>
            <property name="eclipselink.logging.level.sql" value="FINE"/>
            <property name="eclipselink.logging.parameters" value="true"/>

            <!-- JPA 3.x -->
            <!--          <property name="jakarta.persistence.lock.timeout" value="100"/>-->
            <!--          <property name="jakarta.persistence.query.timeout" value="100"/>-->

            <!-- JPA 2.x -->
            <!--          <property name="javax.persistence.lock.timeout" value="100"/>-->
            <!--          <property name="javax.persistence.query.timeout" value="100"/>-->

        </properties>
    </persistence-unit>
</persistence>
```

## 3. Creación de la clase de Entidad `Usuario`

1. Precisamos la anotación `@Entity` para indicar que la **clase Usuario es una entidad** (obligatoria)
2. Precisamos la anotación `@Id` para indicar que el atributo **id es la clave primaria** (obligatoria)
3. Usamos la **anotación @GeneratedValue** para indicar que el valor de la clave primaria **se genera automáticamente**. Solo **cuando las clave primarias son autogeneradas**.

4. Se usa la **anotación** `@Table` para indicar que la **tabla se llama** `usuarios` (si no deseamos que se llame `Usuario`).

```
package com.pepinho.ad.orm;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
@jakarta.persistence.Table(name = "User")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private String apellidos;
    private String email;
    private String password;

    public Usuario() {
    }

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    public Usuario(String nombre, String apellidos, String email, String password) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.email = email;
        this.password = password;
    }

    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 getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "id: " + id +
            ", " + nombre +
            " " + apellidos +
            " (" + email +
            "
```

```
    }  
    + password + ) ,
```

## 4. Creación del EntityManagerFactory y EntityManager

### Implementad una clase de utilidad:

```
package com.pepinho.ad.orm;

import jakarta.persistence.*;

public class JPAUtil {

    // Equivalente a SessionFactory
    private static final EntityManagerFactory ENTITY_MANAGER_FACTORY =
        Persistence.createEntityManagerFactory("default"); // Nombre de la unidad de persistencia

    // Equivalente a Session
    public static EntityManager getEntityManager() {
        return ENTITY_MANAGER_FACTORY.createEntityManager();
    }

    public static void shutdown() {
        ENTITY_MANAGER_FACTORY.close();
    }
}
```

#### Otros ejemplos de creación del EntityManagerFactory:

#### Ejemplo 1 dentro de la clase Main:

```
package com.pepinho.ad.orm;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("ejemplopersistenciaJPA");
        EntityManager em = emf.createEntityManager();
    }
}
```

Podemos emplear un método `main` para probar la conexión a la base de datos:

```
package com.pepinho.ad.orm;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("ejemplopersistenciaJPA");
        EntityManager em = emf.createEntityManager();

        Usuario usuario = new Usuario("Pepe", "Pérez", " ", "1234");
        em.getTransaction().begin();
        em.persist(usuario);
        em.getTransaction().commit();
        System.out.println(usuario);
    }
}
```

Ejemplo de método setUp():

```
package com.pepinho.ad.orm;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;

public class TestUsuario {

    private static EntityManagerFactory emf;
    private static EntityManager em;

    @BeforeAll
    static void setUp() {
        emf = Persistence.createEntityManagerFactory("ejemplopersistenciaJPA");
        em = emf.createEntityManager();
    }

    @AfterAll
    static void tearDown() {
        em.close();
        emf.close();
    }

    // ...
}
```

## 5. Creación del ejemplo de persistencia

```
package com.pepinho.ad.orm.test;

import com.pepinho.ad.orm.Usuario;
import jakarta.persistence.EntityManager;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Root;
import org.junit.jupiter.api.Test;

public class JPATest {

    @Test
    void jpql() {
        insertData();
        var em = JpaUtil.getEntityManager();

        em.createQuery("select a from Usuario a", Usuario.class)
            .getResultList()
            .forEach(System.out::println);
    }

    @Test
    void criteria() {
        insertData();
        var em = JpaUtil.getEntityManager();

        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Usuario> query = cb.createQuery(Usuario.class);
        Root<Usuario> root = query.from(Usuario.class);
        query.select(root);
        em.createQuery(query).getResultList().forEach(System.out::println);
    }

    void insertData(){
        EntityManager em = JpaUtil.getEntityManager();
        em.beginTransaction();

        var a1 = new Usuario("a1");
        var a2 = new Usuario("a2");

        em.persist(a1);
        em.persist(a2);
    }
}
```

```

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

```

Hasta aquí todo correcto.

Si ejecutamos el método `insertData()` varias veces, se crean nuevos registros en la base de datos.

Podríamos seguir avanzando y mejorando la arquitectura de la aplicación, pero, por ahora, nos quedaremos aquí.

Podéis echarle un vistazo a los siguientes apartados para ver cómo mejorar la arquitectura de la aplicación.

## 6. Creación de la clase TestUsuario

```

package com.pepinho.ad.orm;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import org.junit.jupiter.api.*;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class TestUsuario {

    private static EntityManagerFactory emf;
    private static EntityManager em;

    @BeforeAll
    static void setUp() {
        emf = Persistence.createEntityManagerFactory("ejemplopersistenciaJPA");
        em = emf.createEntityManager();
    }

    @AfterAll
    static void tearDown() {
        em.close();
        emf.close();
    }

    @Test
    void testInsertar() {
        Usuario usuario = new Usuario("Pepe", "Pérez", " ", "1234");
        em.getTransaction().begin();
        em.persist(usuario);
        em.getTransaction().commit();
        assertNotNull(usuario.getId());
    }

    @Test
    void testBuscarPorId() {
        Usuario usuario = em.find(Usuario.class, 1L);
        assertNotNull(usuario);
        assertEquals("Pepe", usuario.getNombre());
    }

    @Test
    void testBuscarTodos() {
        List<Usuario> usuarios = em.createQuery("SELECT u FROM Usuario u",
            Usuario.class).getResultList();
        assertEquals(1, usuarios.size());
    }

    @Test
    void testActualizar() {
        Usuario usuario = em.find(Usuario.class, 1L);
        usuario.setNombre("Juan");
        em.getTransaction().begin();
        em.merge(usuario);
        em.getTransaction().commit();
        assertEquals("Juan", usuario.getNombre());
    }

    @Test
    void testBorrar() {

```

```
    Usuario usuario = em.find(Usuario.class, 1L);
    em.getTransaction().begin();
    em.remove(usuario);
    em.getTransaction().commit();
    Usuario usuarioBorrado = em.find(Usuario.class, 1L);
    assertNull(usuarioBorrado);
}
```

## 7. Creación de la clase UsuarioDAO

Un ejemplo un modo más sencillo de implementar el patrón DAO por medio de la clase UsuarioDAO:

```
package com.pepinho.ad.orm;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.TypedQuery;

import java.util.List;

public class UsuarioDAO {

    private static EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ejemplopersistenciaJPA");

    private EntityManager em;

    public UsuarioDAO(EntityManager em) {
        this.em = em;
    }

    public static void insert(Usuario usuario) {
        em.getTransaction().begin();
        em.persist(usuario);
        em.getTransaction().commit();
    }

    public static void delete(Usuario usuario) {
        em.getTransaction().begin();
        em.remove(usuario);
        em.getTransaction().commit();
    }

    public static void update(Usuario usuario) {
        em.getTransaction().begin();
        em.merge(usuario);
        em.getTransaction().commit();
    }

    public static Usuario getById(Long id) {
        return em.find(Usuario.class, id);
    }

    public static List<Usuario> getAll() {
        TypedQuery<Usuario> consulta = em.createQuery("SELECT u FROM Usuario u", Usuario.class);
        List<Usuario> usuarios = consulta.getResultList();
        // También podría ser con CriteriaQuery:
        // CriteriaQuery<Usuario> query = em.getCriteriaBuilder().createQuery(Usuario.class);
        // query.select(query.from(Usuario.class));
        // List<Usuario> usuarios = em.createQuery(query).getResultList();

        return usuarios;
    }
}
```

## 8. Ejercicio. JPA de una biblioteca

💡 Ejercicio 03.01. Creación de una aplicación de persistencia de una biblioteca

Queremos desarrollar una **aplicación para una biblioteca** y necesitamos interactuar con una base de datos que contiene información sobre los libros que tenemos en nuestra colección.

Para ello, vamos a crear una clase `Book` que **represente la entidad libro**, la clase `Contido` y otra clase `BookDAO` que nos permita realizar **operaciones básicas CRUD (Create, Read, Update y Delete) sobre la tabla Book en la base de datos.**

Además, precisamos **una clase `BibliotecaJpaManager`** para la gestión y obtención de los objetos de tipo `EntityManagerFactory` de una manera eficiente. Emplearemos el **patrón Singleton para el gestor `BibliotecaJpaManager`**, que tenga un único objeto de tipo `EntityManagerFactory` y que nos permita obtener un objeto de tipo `EntityManager` para realizar las operaciones sobre la base de datos (**queremos que el objeto de tipo EntityManagerFactory sea único para cada unidad de persistencia**, para cada unidad de persistencia, no así el EntityManager, que podrá hacer varios para cada unidad de persistencia).

A) **BASE DE DATOS** (es la misma base de datos que hemos empleado en la unidad de bases de datos con JDBC):

Está formada por una **tabla `Book` y una tabla `Contido`**. La tabla `Book` tiene una estructura SIMILAR a la siguiente:

Columna	Tipo de dato	Descripción
<code>idBook</code>	<code>int</code>	Identificador único del ejemplar del libro
<code>isbn</code>	<code>varchar(13)</code>	Identificador del libro
<code>titulo</code>	<code>varchar(100)</code>	Título del libro
<code>autor</code>	<code>varchar(100)</code>	Autor del libro
<code>anho</code>	<code>int</code>	Año de publicación del libro
<code>disponible</code>	<code>boolean</code>	Indica si el libro está disponible
<code>portada</code>	<code>Blob</code>	Portada del libro en formato binario
<code>dataPublicacion</code>	<code>Date</code>	Fecha de publicación del libro

```
-- PUBLIC.Book definition
-- Drop table
-- DROP TABLE PUBLIC.Book;
CREATE TABLE PUBLIC.Book (
    idBook INTEGER NOT NULL AUTO_INCREMENT,
    isbn CHARACTER VARYING(13) NOT NULL,
    titulo CHARACTER VARYING(255) NOT NULL,
    autor CHARACTER VARYING(255),
    anho INTEGER,
    disponible BOOLEAN DEFAULT TRUE,
    portada BINARY LARGE OBJECT,
    dataPublicacion DATE,
    CONSTRAINT BOOK_PK PRIMARY KEY (idBook)
);
CREATE UNIQUE INDEX IdBookPK ON PUBLIC.Book (idBook);
CREATE INDEX IdxBookISBN ON PUBLIC.Book (isbn);
CREATE INDEX IdxBookTitle ON PUBLIC.Book (titulo);
CREATE UNIQUE INDEX PRIMARY_KEY_93 ON PUBLIC.Book (idBook);
```

La tabla `Contido` tiene una estructura SIMILAR a la siguiente:

Columna	Tipo de dato	Descripción
<code>idContido</code>	<code>int</code>	Identificador único del contenido del libro
<code>idBook</code>	<code>int</code>	Identificador del libro
<code>contido</code>	<code>Blob</code>	Contenido del libro en formato binario

\* `idBook` es una clave foránea+ que referencia a la tabla `Book`.

```
-- PUBLIC.Contido definition
-- Drop table
-- DROP TABLE PUBLIC.Contido;

CREATE TABLE PUBLIC.Contido (
    idContido INTEGER NOT NULL AUTO_INCREMENT,
    idBook INTEGER NOT NULL,
    contido CHARACTER LARGE OBJECT,
    CONSTRAINT Contido_PK PRIMARY KEY (idContido)
);
CREATE INDEX FK_ID_BOOK_INDEX_9 ON PUBLIC.Contido (idBook);
CREATE UNIQUE INDEX PRIMARY_KEY_9 ON PUBLIC.Contido (idContido);

-- PUBLIC.Contido foreign keys
ALTER TABLE PUBLIC.Contido ADD CONSTRAINT FK_ID_BOOK FOREIGN KEY (idBook) REFERENCES
PUBLIC.Book(idBook) ON DELETE CASCADE ON UPDATE CASCADE;
```

Parámetros de la base de datos:

```
DRIVER: "org.h2.Driver"
URL: "jdbc:h2:rutaBaseDatosSinExtensión;DB_CLOSE_ON_EXIT=TRUE;FILE_LOCK=NO;DATABASE_TO_UPPER=FALSE"
```

El fichero `persistencia.xml` debe tener la siguiente configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="bibliotecaH2" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>-->
        <exclude-unlisted-classes>false</exclude-unlisted-classes> <-- false si no se listan las
        clases en el archivo de configuración -->
        <properties>
            <!-- <property name="jakarta.persistence.jdbc.url"
            value="jdbc:mariadb://localhost:3306/peliculas"/>-->
            <property name="jakarta.persistence.jdbc.url"
            value="jdbc:h2:rutaALaBaseDeDatos;DB_CLOSE_ON_EXIT=TRUE;DATABASE_TO_UPPER=FALSE;FILE_LOCK=NO"/>
            <!-- Ejemplo con Access -->
            <!--<property name="jakarta.persistence.jdbc.url"
            value="jdbc:ucanaccess://rutabase_base_datos.mdb"/>-->
            <!-- <property name="jakarta.persistence.jdbc.user" value="root"/>-->
            <!-- <property name="jakarta.persistence.jdbc.password" value="" />-->
            <property name="jakarta.persistence.jdbc.user" value="/" />
            <property name="jakarta.persistence.jdbc.password" value="/" />
            <!-- <property name="jakarta.persistence.jdbc.driver"
            value="net.ucanaccess.jdbc.UcanaccessDriver"/>-->
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <!-- Automáticamente, genera el esquema de la base de datos -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <!-- Muestra por pantalla las sentencias SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
            <!-- <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"
            />--> <!-- para HSQLDB y Ucanaccess -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
        </properties>
    </persistence-unit>
</persistence>
```

B) Clase `BibliotecaJpaManager`:

Mediante el **patrón Singleton crea una clase `BibliotecaJpaManager`**, mediante el patrón Singleton de manera que tenga un **atributo `emFactory` de tipo `EntityManagerFactory` y que nos permita obtener un objeto de tipo `EntityManager` para realizar las operaciones sobre la base de datos.**

Además, debe tener un **método estático `getEntityManager`** que devuelva un objeto de tipo `EntityManager` y que se encargue de crear el objeto `EntityManager`.

Hazlo con Thread-Safe y doble comprobación.

**Reto: haz que la clase `BibliotecaJpaManager` tenga un singleton para cada factory, guardándolos en un mapa con el nombre de la unidad de persistencia como clave:**

```
private static Map<String, EntityManagerFactory> instances = new HashMap<>();
```

C) Clase `Book` implementa `Serializable`:

Haz que sea una **entidad JPA** y que implemente la interfaz `Serializable`.

La clase `Book` debe tener los siguientes atributos:

- `idBook: Long` (autonumérico)
- `isbn: String` (tamaño 13)
- `title: String`
- `author: String`
- `ano: Integer`
- `available: Boolean`
- `portada: byte[]`
- `dataPublicacion: LocalDate` (**Nuevo** campo)
- `List<Contido> contenido;` (**Nuevo**, lista de contenidos del libro, de momento, mientras no tengamos relaciones, hazlo `transient`)

(Fíjate que ya **no existe el campo contido**] que habíamos definido en la clase `Book` de la unidad de bases de datos con JDBC).

La clase debe tener, al menos, los siguientes constructores:

- `Book()`
- `Book(String isbn, String title, String author, Short year, Boolean available, byte[] portada)`
- `Book(Long idBook, String isbn, String title, String author, Short year, Boolean available, byte[] portada)`
- Aquellos que consideres necesarios.

La **lista de Contido** es una lista de objetos de tipo `Contido` que representan los contenidos del libro. La clase `Contido` tiene los siguientes atributos: `idContido` y `contido`. Ten en cuenta que existe en la base de datos **una tabla** `Contido` con los campos `idContido` y `contido` y **una referencia al libro mediante una clave foránea** `idBook`. De momento, no incluyas la `List` de contenidos en la clase `Book`, hazlos `transient` (**bien con la anotación `@Transient` o con la palabra reservada `transient`**), hasta que veamos las relaciones, que será `@OneToMany`.

Los métodos “set” de las propiedades deben devolver una referencia al propio objeto para poder encadenarlos.

**IMPORTANTE: ten en cuenta que los atributos de la clase `Book` no coinciden con los campos de tabla por lo que debes refactorizar: `author -> autor`, `ano -> anho`, `available -> disponible`, ... o emplear la anotación `@Column` para mapear los atributos de la clase con los campos de la tabla.**

Métodos de la clase Book (ya implantados):

- Get y set para cada atributo.
- `setPortada` (sin implantar): recibe File y lo asigna al atributo `portada`.
- `setPortada` (sin implantar): recibe un array de bytes y lo asigna al atributo `portada`.
- `setPortada` (Sin implantar): recibe un String con el nombre del fichero y lo asigna al atributo `portada`.
- 
- `getImage`: devuelve un objeto de tipo `Image` con la portada del libro.

```

public Image getImage() {
    if (portada != null) {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(portada)) {
            return ImageIO.read(bis);
        } catch (IOException e) {
        }
    }
    return null;
}

```

- `equals` y `hashCode`: considerando que son iguales cuando tienen el mismo `isbn`. Además, el método `hashCode` debe devolver un valor coherente con el método `equals` (todos los objetos iguales deben tener, al menos el mismo `hashCode`).
- `toString`: devuelve el título, el autor y el año. Si no está disponible escribe un asterisco.

D) Clase `Contido` implementa `Serializable`:

A diferencia de la clase empleada en la unidad de bases de datos con JDBC, **la clase `Contido` no debe tener referencia al `idBook`**, pues no es la mejor práctica (está hecho sólo a modo de ejemplo), **debe tener, si queremos la relación bidireccional, una referencia a Book**.

- `idContido: Long` (autonumérico)
- `contido: String` (contenido del libro en formato texto). Puedes hacer un atributo de tipo `String` o `byte[]` (para almacenar el contenido en formato binario), en cualquier caso, deberías modificar la tabla `Contido` en la base de datos.
- `Book book` (relación con la clase `Book`)

Si has implantado la clase `ContidoDao`, debes **modificar los métodos que obtienen el `idBook` del book**:

```
contido.getBook().getIdBook();
```

E) Clase `BookJPADao`:

Esta clase, al igual que la clase `BookDao`, la clase `BookJPADao` **debe implantar la interface `Dao<T>`**, de modo que tenga **un objeto de tipo `EntityManager` como atributo**. En sistemas empresariales, como la gestión de transacciones no se suele hacer por método, se guarda una referencia a la clase `EntityManagerFactory` y se gestiona por medio de try-with-resources para manejar los cierres de los `EntityManager`.

`Dao<T>`:

```

import java.util.List;

/**
 *
 * @author pepecalo
 * @param <T> Tipo de dato del objeto
 */
public interface DAO<T> {

    T get(long id);

    List<T> getAll();

    void save(T t);

    void update(T t);

    void delete(T t);

    public boolean deleteById(long id);

    public List<Integer> getAllIds();

    public void updateLOB(T book, String f); // en BookJPADao recibe un objeto de tipo Book y un
                                             // String con el nombre del fichero

    public void updateLOBById(long id, String f);
}

```

```
    void deleteAll();
}
```

**Clase BookJPADao :**

Implementa la interfaz `DAO<Book>` y gestiona las operaciones CRUD sobre la tabla `Book` de la base de datos. Tiene como atributo un objeto de tipo `EntityManager` que recoge en el constructor.

**Clase BookDAOFactory :**

Factory de clases que implanten la interfaz `DAO<Book>`.

```
import jakarta.persistence.EntityManager;

/**
 * Factory de clases que implanten la interfaz DAO<Book>.
 *
 * @version 1.0
 * @since 1.0
 * @see BookJpaDAO
 * @see TipoDAO
 */

public class BookDaoFactory {

    public enum TipoDao {
        JDBC_H2, JPA_H2, JPA_POSTGRES, HIBERNATE, JSON, JDBC_POSTGRES;
    }

    public static Dao<Book> getBookDAO(TipoDao tipo) {
        switch (tipo) {
            // ..
        }
        return null;
    }
}
```

Implementa un método estático `getBookDAO` que recoge el tipo de DAO que se va a emplear y devuelve el objeto de tipo `BookJPADAO`. Sería interesante hacer cambios para que `getBookDAO` recoja los parámetros necesarios como propiedades de la base de datos, nombre del archivo JSON, nombre de la unidad de persistencia, etc.

```
public static Dao<Book> getBookDAO(TipoDao tipo, Map<String, String> propiedades) {
    switch (tipo) {
        case JPA_H2:
            return new
BookJPADao(BibliotecaJpaManager.getEntityManager(propiedades.get("unidadPersistencia")));
            // ...
        default:
            return null;
    }
}
```

**AppBiblioteca:**

Ejecuta la aplicación para que haga uso del `BookDaoFactory` para obtener un objeto de tipo `DAO<Book>` para asignarlo al controlador de la aplicación. La aplicación debe funcionar igual que con JDBC, pero ahora con JPA.

Con JDBC\_H2:

```
Dao<Book> bookDao = BookDaoFactory.getBookDAO(BookDaoFactory.TipoDAO.JDBC_H2);
```

Con JPA\_H2:

```
Dao<Book> bookDao = BookDaoFactory.getBookDAO(BookDaoFactory.TipoDAO.JPA_H2);
```

Haz pruebas con los dos tipos de DAO. ¿Has notado alguna diferencia? Haz mejoras sobre el funcionamiento de la aplicación.

Puedes hacer pruebas de persistencia de libros en la base de datos:

```
Book libro = new Book("9788424937744", "Tractatus logico-philosophicus-investigaciones filosóficas",
    "Ludwig Wittgenstein", 2017, false);
libro = new Book("9788499088150", "Verano", "J. M. Coetzee", 2011, true);
```

## 8.1. Solución

› Solución: BibliotecaJpaManager

› Solución: Book

› Solución: Clase Contido

› Solución: BookJPADao

› Solución: BookDaoFactory

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 04. GESTIÓN DE ENTIDADES CON ENTITYMANAGER.

---

- [1. EntityManager](#)
- [2. Creación de un EntityManager](#)
- [3. Operaciones CRUD](#)
  - [3.1. Persistir una entidad](#)
  - [3.2. Obtención de una entidad](#)
  - [3.3. Eliminación de una entidad](#)
  - [3.4. Actualización de una entidad](#)
- [4. Transacciones](#)
- [5. Consultas](#)
  - [5.1. Consultas dinámicas](#)
  - [5.2. Consultas con nombre \(estáticas\)](#)
  - [5.3. Ejecución de consultas](#)
- [5. Consultas \(ampliado\)](#)
  - [Tipos de consultas en Jakarta Persistence](#)
- [Ejercicios](#)
  - [Ejercicio 04.01. Descarga y creación de la base de datos de JokeAPI](#)
    - [Enumeraciones](#)
    - [Clases](#)
    - [Ejercicio](#)

## 1. EntityManager

El **gestor de entidades** (`EntityManager`) es el encargado de **gestionar el ciclo de vida de las entidades**. Con él podemos **persistir (persist)**, **actualizar**, **eliminar (remove)** y **recuperar (find)** entidades, así como **realizar consultas (createQuery)**.

- **Contexto de persistencia (Persistence Context)**: es **conjunto de instancias de entidad gestionadas dentro de un gestor de entidades** (`EntityManager`) en un momento dado.
- Es necesario **invocar una llamada de API específica** antes de que una entidad se persista realmente en la base de datos.
- Las llamadas de API para las **operaciones en entidades**, implementada por el gestor de entidades, se **encapsula casi por completo dentro de una única interfaz** `jakarta.persistence.EntityManager`, gestor de entidades al que **se le delega el trabajo real de la persistencia**.
- **Hasta que se utilice un gestor de entidades para crear, leer o escribir realmente una entidad**, la entidad no es más que un objeto Java regular (no persistente). Se dice que **ese objeto está gestionado por el EntityManager (gestor de entidades)**<sup>9</sup>.
- **Sólo puede existir una instancia Java con la misma identidad persistente en un contexto de persistencia** en cualquier momento (con un único ID).

### 💡 Consejo

Las implementaciones concretas de la interface `EntityManager` **permiten leer y escribir en una base de datos específica**, y ser implementadas por un proveedor de persistencia particular (o simplemente proveedor).

**Es el proveedor es el que suministra el motor de implementación de respaldo para toda la API de Persistencia de Jakarta, desde el EntityManager hasta la implementación de las clases de consulta y la generación de SQL.**

Para obtener un gestor de entidades, se debe crear una instancia de la fábrica de gestores de entidades, del tipo `jakarta.persistence.EntityManagerFactory`.

Cada **EntityManager gestiona una unidad de persistencia**. Una unidad de persistencia dicta de manera implícita o explícita la configuración y las clases de entidad utilizadas por todos los gestores de entidades obtenidos de la única instancia de `EntityManagerFactory` vinculada a esa unidad de persistencia. Por lo tanto, hay una **correspondencia uno a uno entre una unidad de persistencia y su instancia concreta de EntityManagerFactory**.

### Objetos, Clases y Conceptos de la API

Objeto	API	Descripción del Objeto
Persistence	<code>Persistence</code>	Clase de inicio utilizada para obtener una fábrica de gestores de entidades ( <code>EntityManagerFactory</code> )
Entity Manager Factory	<code>EntityManagerFactory</code>	Objeto Factory configurado utilizado para obtener gestores de entidades ( <code>EntityManager</code> )
Persistence Unit	–	<b>Configuración con nombre</b> que declara las <b>clases de entidad y la información de la base de datos</b>
Entity Manager	<code>EntityManager</code>	Objeto principal de la API utilizado para realizar operaciones y consultas en entidades
Persistence Context	–	Conjunto de todas las instancias de entidad gestionadas por un gestor de entidades específico

## 2. Creación de un EntityManager

Un gestor de entidades siempre se obtiene de una `EntityManagerFactory`.

En el entorno de Java SE, podemos utilizar una clase de llamada `Persistence` invocando al **método estático `createEntityManagerFactory()`** de la clase `Persistence` que devuelve el EntityManagerFactory para el nombre de la unidad de persistencia especificado. Por ejemplo, para una unidad de persistencia llamada **ServicioEmpleado**:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("ServicioEmpleado");
```

El nombre de la unidad de persistencia especificada, “**ServicioEmpleado**”, pasado al método `createEntityManagerFactory()`, **identifica la configuración de la unidad de persistencia** dada que determina cosas como los **parámetros de conexión** que los gestores de entidades creados a partir de ese objeto Factory utilizarán al conectarse a la base de datos.

Se puede obtener fácilmente un gestor de entidades de ella:

```
EntityManager em = emf.createEntityManager();
```

Un modo muy usual de crear un gestor de entidades es por medio de una clase Singleton:

```
public class EMF {
    private static final EntityManagerFactory emfInstance =
        Persistence.createEntityManagerFactory("ServicioEmpleado");
```

```

private EMF() {}

public static EntityManagerFactory get() {
    return emfInstance;
}
}

```

Que se puede utilizar de la siguiente manera:

```
EntityManager em = EMF.get().createEntityManager();
```

Más interesante es el uso de **patrón Singleton con Thread-Save y Lazy-Initialization** para obtener un `EntityManagerFactory`:

```

public class EMF {
    private static volatile EntityManagerFactory emfInstance;

    private EMF() {}

    public static EntityManagerFactory get() {
        if (emfInstance == null) {
            synchronized (EMF.class) {
                if (emfInstance == null) {
                    emfInstance = Persistence.createEntityManagerFactory("ServicioEmpleado");
                }
            }
        }
        return emfInstance;
    }
    //...
}

```

## 3. Operaciones CRUD

Veremos ejemplos básicos de cómo realizar las operaciones CRUD (Create, Read, Update, Delete) con JPA. y una clase Empleado.

Ejemplo de Entidad Empleado:

```

@Entity
public class Empleado {
    @Id private int id;
    private String nome;
    private long salario;
    public Empleado() {}
    public Empleado(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public long getSalario() { return salario; }
    public void setSalario (long salario) { this.salario = salario; }
}

```

### 3.1. Persistir una entidad

Persistir una entidad es la operación de tomar una entidad transitoria, o una que aún no tiene ninguna representación persistente en la base de datos, y **almacenar su estado para que pueda ser recuperado más tarde**.

```
Empleado emp = new Empleado(158); // Crea una instancia de la entidad Empleado
em.persist(emp);
```

- **Creamos un objeto de tipo Empleado** configurando el ID, no el nombre ni el salario del **Empleado**.
- Llamamos a **persist()** para iniciar la persistencia en la base de datos.

Si el gestor de entidades encuentra un **error lanzará una excepción no verificada de tipo PersistenceException**.

Cuando se completa la llamada a **persist()**, **emp** se convertirá en una **entidad gestionada dentro del contexto de persistencia del gestor de entidades**.

Ejemplo de un método sencillo que crea un nuevo empleado y lo persiste en la base de datos.

```
public Empleado createEmpleado(int id, String nome, long salario) {
    Empleado emp = new Empleado(id);
    emp.setNome(nome);
    emp.setSalario(salario);
    em.persist(emp);
    return emp;
}
```

## 3.2. Obtención de una entidad

Una vez que una entidad está en la base de datos, lo siguiente que normalmente se quiere hacer es obtenerla de nuevo:

```
Empleado emp = em.find(Empleado.class, 158);
```

El método find():

```
<T> T find (Class<T> entityClass, Object primaryKey)
```

Recoge la **clase de la entidad que se está buscando** (**Empleado**), permite que el método **find** sea parametrizado y devuelva un objeto del mismo tipo, y el objeto con **ID o clave primaria** que identifica la entidad en particular (con id 158).

Con esta información el gestor de entidades encuentra la instancia en la base de datos y el **empleado que se devuelve será una entidad gestionada**, lo que significa que **existirá en el contexto de persistencia actual** asociado con el gestor de entidades.

**En el caso de que el objeto no se encuentre la llamada a `find()` simplemente devuelve `null`.** Debe realizarse una comprobación de nulos antes de la próxima vez que se utilice la variable **emp**.

Método de búsqueda:

```
public Empleado findEmpleado(int id) {
    return em.find(Empleado.class, id);
}
```

## 3.3. Eliminación de una entidad

Aunque podría parecer lo contrario, el **borrado (DELETE) de entidad de la base de datos no demasiado común**.

Muchas aplicaciones nunca eliminan objetos, o si lo hacen, simplemente marcan los datos como obsoletos o ya no válidos y los mantienen fuera de la vista de los clientes.

Para **eliminar una entidad debe estar gestionada**, debe estar **presente en el contexto de persistencia**.

La aplicación que realiza la llamada **ya debería haber cargado o accedido a la entidad** y ahora está emitiendo una sentencia para eliminarla.

Puede hacerse por **medio del método remove**:

```
void remove (Object entity)
```

```
Empleado emp = em.find(Empleado.class, 158);
em.remove(emp);
```

El método `find()` devuelve una instancia gestionada de `Empleado`, y luego se elimina la entidad usando la llamada `remove()` en el gestor de entidades.

Si la entidad **no se encuentra, entonces el método `find()` devolverá `null`, resultando una `java.lang.IllegalArgumentException`**.

Se debe incluir una verificación de nulidad antes de llamar a `remove()`:

```
public void removeEmpleado(int id) {
    Empleado emp = em.find(Empleado.class, id);
    if (emp != null) {
        em.remove(emp);
    }
}
```

## 3.4. Actualización de una entidad

La **actualización de una entidad es la operación de tomar una entidad gestionada y modificar su estado** para que se refleje en la base de datos.

```
Empleado emp = em.find(Empleado.class, 158);
emp.setSalario(1000000);
```

Existen **varias formas de actualizar una entidad**, pero por ahora veremos el caso más simple y común, cuando **se dispone de una entidad gestionada y se desea realizar cambios en ella**.

Si no tenemos una referencia a la entidad gestionada:

1. Debemos **obtener la entidad** una usando `find()`.
2. Realizar **operaciones de modificación en la entidad gestionada**.

El siguiente código agrega 1000 euros al salario del empleado con un ID de 158 (yo ;-)):

```
Empleado emp = em.find(Empleado.class, 158);
emp.setSalario(emp.getSalario() + 1000);
```

No se llama al gestor de entidades para modificar el objeto, sino **accediendo al objeto en sí**.

Por esta razón, **es importante que la entidad sea una instancia gestionada**; de lo contrario, el proveedor de persistencia no tendrá medios para detectar el cambio y no se realizarán cambios en la representación persistente del empleado.

```
public Empleado raiseSalarioEmpleado(int id, long cantidad) {
    Empleado emp = em.find(Empleado.class, id);
    if (emp != null) {
        emp.setSalario(emp.getSalario() + cantidad);
    }
    return emp;
}
```

Si no pudimos encontrar al empleado, devolvemos `null` para que el llamador sepa que no se pudo realizar ningún cambio. Indicamos el éxito devolviendo al empleado actualizado.

## 4. Transacciones

En los ejemplos anteriores, no se ha hecho referencia a las transacciones, aunque **los cambios en las entidades deben hacerse persistentes mediante una transacción**.

Excepto `find()`, asumimos que cada método estaba envuelto en una transacción.

La llamada a `find()` **no es una operación de mutación, por lo que puede llamarse en cualquier momento**, con o sin una transacción.

En estos ejemplos estamos empleando un **entorno de Java SE**, y **el servicio de transacciones que debe usarse en Java SE es jakarta.persistence.EntityTransaction** necesitamos comenzar y confirmar la transacción en los métodos operativos, o necesitamos comenzar y confirmar la transacción antes y después de llamar a un método operativo.

### Inicio de la transacción:

En ambos casos, **se inicia una transacción llamando a `getTransaction()`** en el entity manager para obtener la `EntityTransaction` e **invocando `begin()` en ella**:

```
EntityTransaction tx = em.getTransaction();
tx.begin();
```

Para confirmar la transacción, se invoca a `commit()` en el objeto `EntityTransaction` obtenido del entity manager.

Ejemplo completo:

```
em.getTransaction().begin();
createEmpleado(158, "John Doe", 45000);
em.getTransaction().commit();
```

### 💡 Jakarta EE vs Java SE

La clave del uso de transacciones es el **entorno en el que se ejecuta el código**.

La situación típica al ejecutarse **dentro del entorno del contenedor Jakarta EE** utiliza el API estándar de Transacciones de Jakarta. El modelo de transacción cuando se ejecuta en el contenedor es asumir que la aplicación se encargará de que exista un contexto transaccional cuando sea necesario.

Si no hay una transacción presente, entonces la operación de modificación lanzará una excepción o el cambio simplemente no se persistirá en el almacén de datos.

## 5. Consultas

Una consulta es una **solicitud de datos**. En el contexto de JPA, una consulta es una **solicitud de entidades**.

Las consultas se pueden realizar de dos maneras:

- **Consultas dinámicas**: se construyen en tiempo de ejecución como cadenas de consulta.
- **Consultas con nombre**: se definen en tiempo de compilación como consultas con nombre.

### 5.1. Consultas dinámicas

Las consultas dinámicas se construyen en tiempo de ejecución como cadenas de consulta. Las cadenas de consulta son **sentencias de consulta en lenguaje de consulta de entidades (JPQL)**.

El lenguaje de consulta de entidades (JPQL) es un lenguaje de consulta orientado a objetos que se utiliza para definir consultas de entidades y sus resultados.

Las consultas dinámicas se crean utilizando el método `createQuery()` en el gestor de entidades:

```
Query q = em.createQuery("SELECT e FROM Empleado e WHERE e.salario > 100000");
```

El método `createQuery()` toma una **cadena de consulta JPQL** y devuelve un objeto `Query` que se puede utilizar para ejecutar la consulta y recuperar los resultados.

## 5.2. Consultas con nombre (estáticas)

Las consultas con nombre se definen en tiempo de compilación como consultas con nombre. Las consultas con nombre se definen en un archivo de metadatos de la entidad o en un archivo de metadatos de consulta.

Las consultas con nombre se crean utilizando el método `createNamedQuery()` en el gestor de entidades:

```
Query q = em.createNamedQuery("findEmpleadoPorSalario");
```

El método `createNamedQuery()` toma el **nombre de la consulta** y devuelve un objeto `Query` que se puede utilizar para ejecutar la consulta y recuperar los resultados.

Ejemplo de creación de una consulta con nombre:

```
@Entity
@NamedQuery(name="findEmpleadoPorSalario", query="SELECT e FROM Empleado e WHERE e.salario > 100000")
public class Empleado {
    ...
}
```

## 5.3. Ejecución de consultas

Una vez que se ha creado una consulta, se puede ejecutar utilizando el método `getResultSet()` o `getSingleResult()`:

```
TypedQuery<Empleado> q = em.createQuery("SELECT e FROM Empleado e WHERE e.salario > 100000",
Empleado.class);
List<Empleado> results = q.getResultSet();
```

Existen consultas tipadas y no tipadas. Las consultas tipadas devuelven un tipo específico de entidad, mientras que las consultas no tipadas devuelven un tipo de entidad genérico.

Con un consulta no tipada sería:

```
Query q = em.createQuery("SELECT e FROM Empleado e WHERE e.salario > 100000");
List results = q.getResultSet();
```

- El `método getResultSet()` devuelve una **lista de resultados**.
- El `método getSingleResult()` devuelve un **único resultado**.

Si la consulta no devuelve ningún resultado, `getResultSet()` devuelve una lista vacía y `getSingleResult()` **lanza una excepción NoResultException**. Si el resultado no es único y devuelve más de un resultado, `getSingleResult()` **lanza una excepción NonUniqueResultException**.

## 5. Consultas (ampliado)

En Jakarta Persistence, una consulta es similar a una consulta de base de datos, excepto que en lugar de utilizar Structured Query Language (SQL) para especificar los criterios de la consulta, estamos consultando sobre entidades y utilizando un lenguaje llamado Jakarta Persistence Query Language (Jakarta Persistence QL).

Una consulta se implementa en código como un objeto `Query` o `TypedQuery<X>`.

Se construye utilizando el `EntityManager` como fábrica.

La interfaz `EntityManager` incluye una variedad de llamadas a la API que devuelven un nuevo objeto `Query` o `TypedQuery<X>`.

### Tipos de consultas en Jakarta Persistence

Una consulta puede definirse de forma **estática o dinámica**.

1. Una **consulta estática** (con nombre) se define típicamente en metadatos de anotación o XML, y debe incluir los criterios de la consulta, así como un nombre asignado por el usuario. Este tipo de consulta también se llama **consulta nombrada** y se busca posteriormente por su nombre en el momento de su ejecución.
2. Una **consulta dinámica** puede emitirse en tiempo de ejecución proporcionando los criterios de consulta de Jakarta Persistence QL o un objeto de criterios. Pueden ser un poco más costosas de ejecutar porque el proveedor de persistencia no puede realizar ninguna preparación de consulta de antemano, pero las consultas de Jakarta Persistence QL son, no obstante, muy simples de usar y pueden emitirse en respuesta a la lógica del programa o incluso la lógica del usuario.

El siguiente ejemplo muestra cómo crear una consulta dinámica:

(Nota: por supuesto, esta puede no ser una consulta muy buena para ejecutar si la base de datos es grande y contiene cientos de miles de empleados, pero sigue siendo un ejemplo adecuado):

#### Ejemplo usando `getResultSet`:

```
TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);
List<Empleado> emps = query.getResultList();
```

#### Ejemplo usando `getResultSetStream`:

```
TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);
Stream<Empleado> employee = query.getResultSetStream();
```

Creamos un objeto `TypedQuery<Empleado>` emitiendo la llamada `createQuery()` en el EntityManager y pasando la cadena de Jakarta Persistence QL que especifica los criterios de la consulta, así como la clase que debería ser parametrizada en la consulta.

La cadena de **Jakarta Persistence QL no se refiere a una tabla de base de datos EMPLEADO, sino a la entidad Empleado**, por lo que esta consulta **selecciona todos los objetos Empleado sin filtrarlos más**.

Para ejecutar la consulta, simplemente invocamos el método `getResultSet()` o el método `getResultSetStream()` en ella.

- El **método `getResultSet()` devuelve un `List<Empleado>`** que contiene los objetos Empleado que coincidieron con los criterios de la consulta. Observa que el **List** está parametrizado por Empleado, ya que el tipo parametrizado se propaga desde el argumento de clase inicial pasado al método `createQuery()`. Podemos crear fácilmente un método que devuelva todos los empleados.

- El método `getResultSet()` devuelve un flujo del resultado de la consulta, por lo que, en este caso, devuelve el flujo del resultado de la consulta Empleado. Por defecto, delega en `getResultSet().stream()`.

El método `getResultSet()` proporciona una mejor manera de moverse a través del conjunto de resultados de la consulta, ya que, para conjuntos de datos grandes, evita leer todo el "conjunto de resultados" en memoria antes de que pueda usarse en la aplicación.

### Método para Emitir una Consulta

```
public List<Empleado> findAllEmpleados() {
    TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);
    return query.getResultList();
}
```

Con Stream:

```
public Stream<Empleado> findAllEmpleadosStream() {
    TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);
    return query.getResultSet();
}
```

que podríamos usar de la siguiente manera:

```
Stream<Empleado> empleadosStream = findAllEmpleadosStream();
empleadosStream.forEach(System.out::println);
```

## Ejercicios

### Ejercicio 04.01. Descarga y creación de la base de datos de JokeAPI

Dado el modelo de la aplicación de JokeAPI, en la que tenemos las enumeraciones `Categoría` y `TipoChiste`, y la clase `Chiste`, vamos a crear una base de datos con JPA y los chistes de la API.

### Enumeraciones

A) La enumeración `Categoría` tiene los siguientes valores:

```
public enum Categoría {
    ANY("Any"),
    MISC("Misc"),
    PROGRAMMING("Programming"),
    DARK("Dark"),
    PUN("Pun"),
    SPOOKY("Spooky"),
    CHRISTMAS("Christmas");
    //...
}
```

[> Detalle de implementación de la enumeración Categoría](#)

B) La enumeración `TipoChiste` contiene los siguientes valores:

```
public enum TipoChiste {
    SINGLE("single"),
    TWOPART("twopart");
    //...
}
```

› Detalle de implementación de la enumeración TipoChiste

C) La enumeración **Flag** contiene los siguientes valores:

Flag es una enumeración con los siguientes valores:

```
```java
public enum Flag {
    EXPLICIT("Explicit"),
    NSFW("NSFW"),
    RELIGION("Religion"),
    POLITICAL("Political"),
    RACIST("Racist"),
    SEXIST("Sexist");
    //...
}
```

› Detalle de implementación de la enumeración Flag

D) **Lenguaje** es una enumeración con los siguientes valores:

```
public enum Lenguaje {
    CS("cs"),
    DE("de"),
    EN("en"),
    ES("es"),
    FR("fr"),
    PT("pt");
    //...
}
```

› Detalle de implementación de la enumeración Lenguaje

## Clases

A) La clase **Chiste** tiene los siguientes atributos:

```
public class Chiste {
    private int id;
    private Categoria categoria;
    private TipoChiste tipo;
    private final List<Flag> banderas;
    private String chiste;
    private String respuesta;

    private Lenguaje lenguaje;
    //...
}
```

› Detalle de implementación de la clase Chiste

B) El adapter **ChisteDeserializer**:

› Detalle de implementación de la clase ChisteDeserializer

C) La clase **ChisteTypeAdapter**:

› Detalle de implementación de la clase ChisteTypeAdapter

D) La interface **IChisteDAO** y clase **ChisteDAO** se usa para obtener los chistes de la API:

› Detalle de implementación de la interfaz IChisteDAO

Podrías realizar mejoras en el código, como la gestión de excepciones, la comprobación de valores nulos, la simplificación de código, etc.

### [» Detalle de implementación de la clase ChisteDAO](#)

## Ejercicio

**Crear una base de datos con JPA** y Hibernate para la aplicación JokeAPI y transfiere todos los datos de JSON a la base de datos.

Añade las dependencias necesarias y el fichero de configuración `persistence.xml` en el directorio `META-INF` de `src/main/resources`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="chistesH2" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>-->
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="jakarta.persistence.jdbc.url"
                value="jdbc:h2:RutaABaseDatos;DB_CLOSE_ON_EXIT=TRUE;DATABASE_TO_UPPER=FALSE;FILE_LOCK=NO"/>
            <property name="jakarta.persistence.jdbc.user" value="" />
            <property name="jakarta.persistence.jdbc.password" value="" />
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver" />
            <!-- Automáticamente, genera el esquema de la base de datos -->
            <property name="jakarta.persistence.schema-generation.database.action" value="drop-and-
create" />
            <!-- Muestra por pantalla las sentencias SQL -->
            <property name="hibernate.show_sql" value="false" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.highlight_sql" value="true" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
        </properties>
    </persistence-unit>
</persistence>
```

Para ello, crea las siguientes clases:

A) **ChisteJpaManager** que empleando el patrón Singleton, gestione la creación de la factoría de entidades y el EntityManager.

### [» Solución de ChisteJpaManager](#)

B) **Chiste** que emplea JPA para mapear la clase Chiste con la tabla Chiste de la base de datos.

### [» Solución de Chiste](#)

C) **ChisteDownloader** que descarga los chistes de la API y los guarda en la base de datos.

Ten en cuenta que ChisteDownloader es un Singleton y que se puede configurar el número de chistes a descargar, además de un tiempo de espera entre chiste y chiste (la API sólo permite 120 peticiones por minuto).

Por ello, haz que sea un hilo que se ejecute cada cierto tiempo ( implements Runnable ) y tenga los siguientes atributos:

- **tiempoEspera** que es el tiempo de espera entre chiste y chiste.
- **instance** que es la instancia de ChisteDownloader.
- **MAX\_CHISTES** que es el número máximo de chistes a descargar.
- **chisteDAO** que es el DAO de Chiste.
- **numeroChistes** que es el número de chistes a descargar (si no se indica debe ser MAX\_CHISTES).

[› Solución de ChisteDownloader](#)

**D) Main** que descarga los chistes y los guarda en la base de datos.

[› Solución de Main](#)

---

 Autor/a: Pepinho  Última actualización: 13.02.2025

# 05. MAPEO DE ENTIDADES.

---

- [1. Mapeo Objeto-Relacional](#)
- [2. Anotaciones de Persistencia](#)
- [1. Modo de acceso a una Entidad](#)
  - [1.2. Acceso por atributo](#)
  - [1.2. Acceso por propiedad](#)
  - [1.3. Acceso Mixto: @Access\(AccessType.FIELD|AccessType.PROPERTY\)\)](#)
- [2. Mapeo a una Tabla concreta: @Table](#)
- [2.1. Nombres sensibles a mayúsculas de tablas y columnas](#)
- [3. Mapeo de Tipos Simples](#)
- [4. Mapeo de columnas: @Column](#)
- [5. Carga perezosa \(Lazy Fetching\): @Basic\(fetch=FetchType.LAZY\)](#)
- [6. Objetos Grandes \(LOBs\): @Lob](#)
- [7. Tipos Enumerados \(enum\): @Enumerated](#)
  - [7.1. Mapeo enumeraciones como cadenas](#)
  - [7.2. Mapeo de enumeraciones con @PostLoad y @PrePersist](#)
  - [7.3 Mapeo de enumeraciones con @Converter](#)
  - [7.4. Uso de Enums en JPQL](#)
- [8. Tipos temporales: @Temporal](#)
- [9. Atributos transitorios](#)
- [10. Mapeo de clave primaria: @Id](#)
  - [10.1. Sobrescritura de la clave primaria](#)
  - [10.2. Tipos de claves primarias](#)
  - [10.3. Generación de claves primarias: @GeneratedValue](#)
    - [10.3.1. Generación Automática de ID: GenerationType.AUTO](#)
    - [10.3.2. Generación de ID utilizando una tabla: GenerationType.TABLE](#)
    - [10.3.3. Generación de ID Utilizando una Secuencia de Base de Datos: GenerationType.SEQUENCE](#)
    - [10.3.4. Generación de ID utilizando una Identidad de Base de Datos](#)
    - [10.3.5. Generación de ID Utilizando un UUID: GenerationType.UUID](#)
- [11. Ejercicio. Persistencia de una biblioteca](#)

## 1. Mapeo Objeto-Relacional

El componente de mapeo objeto-relacional (ORM) incluye:

- **Correspondencia del estado del objeto con las columnas de la base de datos.**
- Cómo enviar **consultas entre los objetos**.

En este apartado veremos **cómo mapear entidades y atributos con la base da datos** y generar automáticamente identificadores de entidad.

## 2. Anotaciones de Persistencia

- Las especificaciones de **Jakarta Persistence** (y de Enterprise Beans) **emplea principalmente anotaciones**.
- Las anotaciones pueden **aplicarse a ==clases, métodos y atributos=0**.
- La anotación debe colocarse principio a la definición de código del artefacto que se está anotando: bien en la misma línea justo **antes de la clase, método o atributo o en la línea superior**.



Consejo

*La elección se basa completamente en las preferencias de la persona que aplica las anotaciones, y creo que tiene sentido hacer una cosa en algunos casos y la otra en otros casos. Depende de cuán extensa sea la anotación y cuál sea el formato más legible.*

Las anotaciones de Jakarta Persistence fueron diseñadas para ser **legibles, fáciles de especificar y lo suficientemente flexibles** como para permitir diferentes combinaciones de metadatos. La mayoría de las anotaciones se **especifican como hermanas en lugar de estar anidadas entre sí**, lo que significa que múltiples anotaciones pueden anotar la misma clase, atributo o propiedad en lugar de tener anotaciones incrustadas dentro de otras anotaciones.

Las anotaciones de mapeo se pueden clasificar en **dos categorías**:

- **Anotaciones lógicas:** describen el **modelo de entidad desde una perspectiva de modelado de objetos**. Están fuertemente vinculadas al modelo de dominio y son el tipo de metadatos que podría querer especificar en UML o cualquier otro lenguaje o marco de modelado de objetos. Ejemplos: `@Entity`, `@Id`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`, `@OneToOne`.
- **Anotaciones ==físicas=0:** se relacionan con el **modelo de datos concreto de la base de datos**. Tratan con **tablas, columnas, restricciones y otros artefactos en base de datos** de los que el modelo de objetos podría no estar al tanto de otra manera. Ejemplos: `@Table`, `@Column`, `@JoinColumn`, `@JoinTable`.

**Existen equivalentes XML para todas las anotaciones de mapeo** lo que permite utilizar el enfoque que mejor se adapte a las necesidades de desarrollo. **Nosotros nos centraremos en anotaciones**, que es la forma más común de especificar metadatos en aplicaciones modernas.

#### ! Nota

**Consejo:** **Las anotaciones de mapeo de JPA se pueden aplicar a atributos o métodos.** Si se aplican a un atributo, el proveedor de persistencia accederá al atributo directamente. Si se aplican a un método, el proveedor de persistencia accederá al atributo a través del método getter y setter. Lo veremos ahora.

## 1. Modo de acceso a una Entidad

La **forma en que se accede al estado en la entidad desde el proveedor de persistencia** se llama **modo de acceso**.

El mecanismo que se usa para designar el estado persistente es el mismo que el modo de acceso que el proveedor utiliza para acceder a ese estado, y hay dos modos de acceso: **acceso por atributo** (atributos de la entidad) y **acceso por propiedad** (métodos *getter* y *setter* de la entidad).

- **Acceso por atributo:** a partir de los **atributos/atributos de la entidad utilizando reflexión (Java reflection)** (se precisa un `@Id` sobre el atributo).
- **Acceso por propiedad:** las **anotaciones se colocan en los métodos getter de las propiedades**, esos métodos *getter* y *setter* serán invocados por el proveedor para acceder y establecer el estado. En este caso se indica la *anotación* `@Id` en el método *getter*.

### 1.2. Acceso por atributo

**Anotar los atributos de la entidad hará que el proveedor use el acceso por atributo** para obtener y establecer el estado de la entidad. Los métodos *getter* y *setter* pueden estar presentes o no, pero si están presentes, el proveedor los ignora.

Todos los **atributos deben declararse como `protected`, de paquete (sin modificador) o `private`**. Se **prohiben los atributos `public`**.

El ejemplo de entidad Employee mapeada usando el acceso por atributo:

- La anotación `@Id` indica que `id` es el identificador persistente o clave primaria de la entidad y que se debe asumir el acceso por atributo.
- Los atributos `name` y `salary` se configuran por defecto como persistentes y se mapean a columnas del mismo nombre.

```
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private long salary;

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary = salary; }
}
```

## 1.2. Acceso por propiedad

Cuando se utiliza el modo de acceso por propiedad **debe haber métodos *getter* y *setter* para las propiedades persistentes.**

- El tipo de propiedad se determina por el tipo devuelto del método *getter* y debe ser el mismo que el tipo del único parámetro pasado al método *setter*.
- Ambos métodos deben tener visibilidad `public` o `protected`.
- Las anotaciones de mapeo para una propiedad deben estar en el *método getter*.

Ejemplo de la clase Employee tiene una anotación `@Id` en el método `getId()`, por lo que el proveedor utilizará el acceso por propiedad para obtener y establecer el estado de la entidad. Las propiedades `name` y `salary` se harán persistentes gracias a los métodos *getter* y *setter* y se mapearán a las columnas `NAME` y `SALARY`, respectivamente.

Observa que la propiedad `salary` está respaldada por el atributo `wage`, que no comparte el mismo nombre. Esto pasa desapercibido para el proveedor porque al especificar el acceso por propiedad, le estamos diciendo al proveedor que ignore los atributos de la entidad y utilice solo los métodos *getter* y *setter* para la nomenclatura.

```
@Entity
public class Employee {

    private long id;
    private String name;
    private long wage;

    @Id
    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return wage; }
    public void setSalary(long salary) { this.wage = salary; }
}
```

## 1.3. Acceso Mixto:

`@Access(AccessType.FIELD|AccessType.PROPERTY)`

Por lo general se accede a los datos a través del acceso por atributo, pero posible combinar el acceso por atributo con el acceso por propiedad dentro de la misma jerarquía de entidades o dentro de la misma entidad. Puede ser útil cuando se agrega una subclase de entidad a una jerarquía existente que utiliza un tipo de acceso diferente.

Agregar una anotación `@Access` con un modo de acceso hace que el tipo de acceso predeterminado se anule para esa subclase de entidad.

`@Access` también es útil cuando es necesario realizar una simple transformación de los datos al leer o escribir en la base de datos.

Por ejemplo, la entidad `Employee` que tiene un modo de acceso predeterminado de `AccessType.FIELD`, pero la columna de la base de datos almacena el código de área como parte del número de teléfono, y solo queremos almacenar el código de área en el atributo `phoneNum` de la entidad si no es un número local. Podemos **agregar una propiedad persistente que realice la transformación correspondiente en lecturas y escrituras**.

1. Se debe hacer es **marcar explícitamente el modo de acceso predeterminado para la clase** mediante la anotación `@Access` e indicar el tipo de acceso. A menos que se haga esto, será indefinido si ambos atributos y propiedades están anotados:

```
@Entity
@Access(AccessType.FIELD)
public class Employee {
    // ...
}
```

2. Se anota el **atributo o propiedad adicional con la anotación @Access**, pero esta vez especificando el tipo de acceso opuesto al especificado a nivel de clase. *No es redundante especificar el tipo de acceso de AccessType.PROPERTY en una propiedad persistente porque es obvio al verlo que es una propiedad, pero al hacerlo se indica que es una excepción al caso predeterminado:*

```
@Access(AccessType.PROPERTY)
@Column(name="PHONE")
protected String getPhoneNumberForDb() {
    // ...
}
```

3. El **atributo o propiedad correspondiente al que se está haciendo persistente debe marcarse como \* transient \*** para que las reglas de acceso predeterminadas no provoquen que el mismo estado se persista dos veces. El atributo en el cual se está almacenando el estado de la propiedad persistente en la entidad debe estar anotado con `@Transient`:

```
@Transient
private String phoneNum; // no persiste este atributo, pues se persiste el atributo por la propiedad
getPhoneNumberForDb()
```

**Ejemplo completo** de la clase `Employee` con un atributo `phoneNum` que se mapea a la columna `PHONE` de la base de datos, pero que realiza una transformación simple en la lectura y escritura:

```
@Entity
@Access(AccessType.FIELD)
public class Employee {

    public static final String LOCAL_AREA_CODE = "613";
    @Id private long id;
    @Transient private String phoneNum;
    // ...

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getPhoneNumber() { return phoneNum; }
    public void setPhoneNumber(String num) {
        this.phoneNum = num;
    }

    @Access(AccessType.PROPERTY)
    @Column(name="PHONE") // Si no se indica la columna, se mapearía a PhoneNumberForDb
    protected String getPhoneNumberForDb() {
        if (phoneNum.length() == 10)
            return phoneNum;
    }
}
```

```

        else
            return LOCAL_AREA_CODE + phoneNum;
    }

    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3);
        else
            phoneNum = num;
    }
    // ...
}

```

#### 💡 Ejercicio 05.01. Acceso combinado a la entidad Chiste.

Mofifica la entidad Chiste para que guarde el chiste y la respuesta en un solo campo en la base de datos, pero que se muestren por separado en la aplicación.

```

@Entity
public class Chiste {
    @Id
    private int id;
    private Categoria categoria;
    private TipoChiste tipo;
    private List<Flag> banderas;
    private String chiste;
    private String respuesta;

    private Lenguaje lenguaje;
    // ...
}

```

## 2. Mapeo a una Tabla concreta: `@Table`

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/table>

Para mapear entidad a una tabla de la base de datos una entidad **sólo se necesitan las anotaciones `@Entity` y `@Id`**.

El **nombre de la tabla predeterminado es el nombre no calificado de la clase de entidad**.

Para **cambiar el nombre predeterminado de la tabla se anota la clase de entidad con la anotación `@Table` e incluyendo el nombre de la tabla mediante el elemento `name`**. Por ejemplo:

```

@Entity
@Table(name="EMP")
public class Employee {
    // ...
}

```

#### 💡 Consejo

**Consejo** Los nombres predeterminados **no se especifican en mayúsculas o minúsculas. La mayoría de las bases de datos no distinguen entre mayúsculas y minúsculas**, por lo que generalmente no importará si un proveedor usa el caso del nombre de la entidad o lo convierte a mayúsculas.

"\_In MySQL text columns are case insensitive by default, while in H2 they are case sensitive. However H2 supports case insensitive columns as well. To create the tables with case insensitive texts, append IGNORECASE=TRUE to the database URL (example: jdbc:h2:~/test;IGNORECASE=TRUE)."\_\_

#### Esquemas:

La anotación `@Table` proporciona la capacidad también de especificar un esquema o catálogo de la base de datos. El nombre del esquema se usa comúnmente para **diferenciar un conjunto de tablas de otro y se indica mediante el uso del elemento `schema`**. Por ejemplo: la entidad *Employee* que se asigna a la tabla *EMP* en el esquema *HR*.

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }
```

El **nombre del esquema se antepondrá al nombre de la tabla cuando el proveedor de persistencia vaya a la base de datos para acceder a la tabla** (*HR.EMP* en el ejemplo).

### 💡 Consejo

Algunos proveedores pueden permitir que el esquema se incluya en el elemento `name` de la tabla sin tener que especificar el elemento `schema`, como en `@Table(name="HR.EMP")`, pero no es un estándar.

### Catálogos:

Algunas bases de datos admiten la noción de un **catálogo**. Para estas bases de datos, se puede especificar el elemento `catalog` de la anotación `@Table`. Por ejemplo, para la tabla *EMP*.

```
@Entity
@Table(name="EMP", catalog="HR")
public class Employee {
    // ...
}
```

## 2.1. Nombres sensibles a mayúsculas de tablas y columnas

Los nombres de las tablas y columnas como identificadores en mayúsculas ayudan a diferenciarlos de los identificadores en Java y es el **estándar SQL establece que los identificadores de base de datos no delimitados no sean sensibles a mayúsculas y la mayoría tiende a mostrarlos en mayúsculas**.

`@Table` y `@Column` la cadena de identificador se pasa al controlador JDBC exactamente como se **especifica** o establece por defecto. Por ejemplo, cuando no se especifica un nombre de tabla para la entidad Autor, entonces el nombre de la tabla asumido y utilizado por el proveedor será Autor, que por definición SQL no es diferente de AUTOR.

Las siguientes anotaciones deberían ser equivalentes, ya que se refieren a la misma tabla en una base de datos compatible con el estándar SQL:

```
@Table(name="autor")
@Table(name="Autor")
@Table(name="AUTOR")
```

Aunque no es común ni una buena práctica, en teoría, una base de datos podría tener una tabla `AUTOR` y otra `Autor`. Estas necesitarían ser **envueltas en comillas dobles para distinguirlas**, que deben ser escapadas, alrededor del identificador. El mecanismo de escape es la barra invertida (el carácter `\`):

```
@Table(name="\\"Autor\\\"")
@Table(name="\\"AUTOR\\\"") // Son tablas diferentes
```

## 3. Mapeo de Tipos Simples

Los tipos simples de Java se asignan de manera inmediata en campos o propiedades de una entidad. Incluyen:

- **Tipos primitivos de Java:** `byte`, `int`, `short`, `long`, `boolean`, `char`, `float` y `double`.
- **Clases envolventes** de tipos primitivos de Java: `Byte`, `Integer`, `Short`, `Long`, `Boolean`, `Character`, `Float` y `Double`
- Tipos de **arrays de byte y carácter**: `byte[]`, `Byte[]`, `char[]` y `Character[]`
- Tipos **numéricos grandes**: `java.math.BigInteger` y `java.math.BigDecimal`
- **Cadenas**: `java.lang.String`
- **Tipos temporales** de Java: `java.util.Date` y `java.util.Calendar`, además de todos los subtipos y las Java 8 `java.time` API:
  - `java.time.LocalDate`
  - `java.time.LocalDateTime`
  - `java.time.LocalTime`
  - `java.time.OffsetTime`
  - `java.time.OffsetDateTime`
  - Para tipos como `java.time.Instant`, se necesita un `AttributeConverter`, que veremos más adelante.
- **Tipos temporales JDBC:** `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`
- Tipos **enumerados**: cualquier tipo enumerado definido por el sistema o el usuario
- Objetos **serializables**: cualquier tipo serializable definido por el sistema o el usuario.

Si el tipo de la capa JDBC **no se puede convertir al tipo de Java del campo o propiedad, normalmente se lanzará una excepción**, aunque no está garantizado.

### Consejo

*Cuando el tipo persistente no coincide con el tipo JDBC, algunos proveedores pueden optar por tomar medidas propietarias o hacer una suposición para convertir entre los dos. En otros casos, el controlador JDBC podría realizar la conversión por sí mismo.*

Opcionalmente, se puede colocar una anotación `@Basic` en un campo o propiedad para **marcarlo explícitamente como persistente**. Esta anotación es principalmente con fines de documentación y **no es necesaria para que el campo o propiedad sea persistente**.

## 4. Mapeo de columnas: `@Column`

La **anotación `@Basic`** (o el mapeo básico asumido en su ausencia) puede considerarse como una **indicación lógica de que un atributo dado es persistente**.

La **anotación física** que acompaña al mapeo básico es la anotación `@Column`:

`@Column`

Con `@Column` en el atributo indica características específicas de la columna física de la base de datos. El nombre de la columna y los metadatos de asignación física pueden estar en un archivo XML separado.

Elementos de la anotación `@Column`:

- `name`: **nombre de la columna** de la base de datos. String predeterminado es el nombre del atributo o propiedad.
- `length`: longitud de la columna de la base de datos. Solo se aplica si el tipo de **columna es una cadena o un tipo de array de caracteres**. Por defecto 255.
- `unique`: si es una **clave primaria** (valor único). Booleano con un valor predeterminado de `false`.
- `nullable`: **si puede ser nulo**. Booleano con un valor predeterminado de `true`.

- `insertable`: si el valor de la columna se incluye en las declaraciones de SQL INSERT generadas. Booleano con un valor predeterminado de `true`.
- `updatable`: si el valor de la columna se incluye en las declaraciones de SQL UPDATE generadas por el proveedor. Este es un atributo booleano con un valor predeterminado de `true`.
- `precision` y `scale`: se aplican a los tipos numéricos y se utilizan para especificar la **precisión y la escala de la columna de la base de datos**. Si se omite, se utilizarán los valores predeterminados 0.
- `table`: El nombre de la tabla de la base de datos que contiene la columna. Este nombre se refiere a la tabla que contiene la columna, que puede ser la tabla de la entidad o una tabla secundaria. Esta anotación es útil para especificar una columna que se mapea a una tabla secundaria.
- `columnDefinition`: definición de columna SQL, que es una cadena que se pasará directamente al DDL de la base de datos. Esta característica puede hacer que la aplicación sea menos portátil. Si se omite, se utilizará la definición de columna predeterminada del proveedor de persistencia.

El **principal elemento que es relevante es el elemento `name`**, que es simplemente una cadena que especifica el nombre de la columna a la que se ha asignado el atributo.

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    // ...
}
```

## 5. Carga perezosa (Lazy Fetching): `@Basic(fetch=FetchType.LAZY)`

A veces, **algunas partes de la entidad se accede pocas veces (imagen, etc.)**. En estas situaciones, **se puede optimizar el rendimiento al recuperar sólo los datos que se espera que se accedan con frecuencia**. Es lo que se denomina: **carga perezosa, carga diferida, carga lenta, carga bajo pedido, lectura justo a tiempo, indirección y otros**.

En este caso, los datos del objeto **no se leen inicialmente desde la base de datos, sino que se recuperarán solo cuando se hagan referencia o se lean**.

Se especifica con el elemento `fetch` de la anotación `@Basic`, que se corresponde con un valor de la enumeración `FetchType`:

- `EAGER` (por defecto): carga ansiosa.
- `LAZY`: el proveedor puede posponer la carga del estado para ese atributo hasta que se haga referencia.

**Casi nunca es una buena idea cargar perezosamente los tipos simples.** Las únicas veces en las que debería considerarse la carga perezosa de un mapeo básico son cuando **hay muchas columnas en una tabla** (por ejemplo, docenas o cientos) o cuando las **columnas son grandes** (por ejemplo, cadenas de caracteres o cadenas de bytes muy grandes).

```
@Entity
public class Employee {
    // ...
    @Basic(fetch=FetchType.LAZY)
    @Column(name="COMM")
    private String comments;
    // ...
}
```

La aplicación no tiene que hacer nada especial para obtenerlo. Al acceder al campo de comentarios, se leerá y completará automáticamente por el proveedor si aún no se había cargado.

### 💡 Consejo

La directiva `LAZY` solo pretende ser una **sugerencia para el proveedor de persistencia para ayudar a la aplicación a lograr un mejor rendimiento**. No se requiere que el proveedor respete la solicitud porque el comportamiento de la entidad no se ve comprometido si el proveedor procede y carga el atributo. La situación contraria no es cierta, ya que especificar que un atributo se cargue ansiosamente podría ser fundamental para poder acceder al estado de la entidad una vez que la entidad se ha desvinculado del contexto de persistencia.

## 6. Objetos Grandes (LOBs): `@Lob`

Un **LOB** es un campo de caracteres o bytes que puede ser muy grande (hasta el rango de gigabytes). Típicamente, CLOB se utiliza para almacenar texto y BLOB para almacenar datos binarios. Los **LOB se almacenan en la base de datos, pero se accede a ellos de manera diferente a los tipos simples**.

La anotación `@Lob` se puede usar para los LOB y puede aparecer junto con la anotación `@Basic`, o puede aparecer cuando `@Basic` está ausente y se asume implícitamente en el mapeo.

Dado que la anotación `@Lob` realmente **sólo califica el mapeo básico**, también **puede ir acompañada de una anotación `@Column`**.

Existen (básicamente) dos tipos de LOB en las BD:

- **CLOB** contiene una **secuencia de caracteres grande**. Los tipos de datos Java son `char[]`, `Character[]` y objetos `String`.
- **BLOB** puede almacenar una **secuencia de bytes grande**. Los tipos de Java asignados a columnas BLOB son `byte[]`, `Byte[]` y tipos `Serializable`.

Un ejemplo, en el que se marca `LAZY`, algo útil en los LOB poco empleados:

```
@Entity
public class Employee {
    @Id
    private long id;
    @Basic(fetch=FetchType.LAZY)
    @Lob
    @Column(name="PIC")
    private byte[] picture;
    // ...
}
```

### 💡 Ejercicio 05.02. CLOB y BLOB de una entidad Documento

Crea una entidad Documento que tenga un campo de texto grande (CLOB) para el contenido del documento y un campo de bytes grande (BLOB) para la imagen del documento. Haz pruebas con tres gestores de bases de datos: H2, SQLite y PostgreSQL y comprueba el resultado creando la tabla en cada uno de ellos, con y sin declaración de tipo de LOB.

```
@Entity
public class Documento {
    @Id
    private long id;
    @Lob
    private String contenido;
    @Lob
    private byte[] imagen;
    // ...
}
```

### 💡 Consejo

**Consejo:** Los LOB son útiles para almacenar datos grandes, pero no se deben abusar de ellos. Los LOB pueden ser inefficientes para recuperar y almacenar. Siempre que sea posible, se deben evitar los LOB. Si se necesita almacenar datos grandes, se debe considerar el uso de un sistema de archivos o un sistema de almacenamiento de objetos.

## 7. Tipos Enumerados (enum): `@Enumerated`

Los valores de un tipo enumerado en Java tienen una **asignación ordinal** implícita que se **determina por el orden en que se declararon**.

El **ordinal** se usa de modo predeterminado para representar y almacenar los valores del tipo enumerado en la base de datos.

El proveedor asumirá que la **columna de la base de datos es de tipo entero**.

Por ejemplo, el tipo enumerado `EmployeeType`:

```
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,
    PART_TIME_EMPLOYEE,
    CONTRACT_EMPLOYEE
}
```

Los ordinales en tiempo de compilación serían **0 para FULL\_TIME\_EMPLOYEE, 1 para PART\_TIME\_EMPLOYEE y 2 para CONTRACT\_EMPLOYEE**.

```
@Entity
public class Employee {
    @Id
    private long id;
    private EmployeeType type;
    // ...
}
```

### 7.1. Mapeo enumeraciones como cadenas

`EmployeeType`, en ejemplo anterior, el **atributo type se asignará a una columna TYPE de tipo entero**.

**Si se cambia el tipo (el orden) hay una inconsistencia y problemas.**

En este ejemplo, si la política de beneficios de la empresa cambia y comenzamos a dar beneficios adicionales a los empleados a tiempo parcial que trabajan más de 20 horas por semana, queríamos diferenciar entre los dos tipos de empleados a tiempo parcial. Al agregar un valor `PART_TIME_BENEFITS_EMPLOYEE` después de `PART_TIME_EMPLOYEE`, estaríamos provocando una nueva asignación de ordinal, donde nuestro nuevo valor recibiría el ordinal 2 y `CONTRACT_EMPLOYEE` obtendría 3. Esto tendría el efecto de hacer que todos los empleados contratados previamente como empleados a tiempo parcial se conviertan repentinamente en empleados a tiempo parcial con beneficios, claramente no el resultado que esperábamos.

Una solución es **almacenar el nombre de la enumeración como una cadena en lugar de almacenar el ordinal**. Para ello existe la anotación `@Enumerated`:

`@Enumerated`

Para modificar cómo guardar los enumerados se puede realizar con la **anotación `@Enumerated` en el atributo y especificando un valor de `EnumType.STRING`** (la otra posibilidad es `EnumType.ORDINAL`):

`EnumType`

La anotación `@Enumerated` permite especificar un `EnumType`, que a su vez es un tipo enumerado que define el valor de `value` de `EnumType.ORDINAL` y `EnumType.STRING`.

El valor predeterminado de `@Enumerated` es `ORDINAL`, especificar `@Enumerated(ORDINAL)` solo es útil cuando se desea hacer explícito este mapeo.

Por ejemplo:

```
@Entity
public class Employee {
    @Id
    private long id;

    @Enumerated(EnumType.STRING)
    private EmployeeType type;

    // ...
}
```

El uso de cadenas resuelve el problema de insertar valores adicionales en medio del tipo enumerado, pero dejará los **datos vulnerables a cambios en los nombres de los valores**.

Por ejemplo, si quisieramos cambiar `PART_TIME_EMPLOYEE` a `PT_EMPLOYEE`, tendríamos problemas. Aunque este es un problema menos probable, cambiar los nombres de un tipo enumerado obligaría a cambiar todo el código que utiliza ese tipo enumerado, lo cual sería más engorroso que reasignar valores en una columna de base de datos.

Almacenar el **ordinal es la mejor y más eficiente manera de manejar los tipos enumerados, siempre y cuando la probabilidad de agregar nuevos valores en el medio no sea alta**. Se podrían agregar nuevos valores al final del tipo sin consecuencias negativas.

#### 💡 Consejo

Es posible tener valores enumerados que contengan estado. Actualmente, **no hay soporte en Jakarta Persistence para mapear el estado contenido dentro de los valores enumerados**, pero hay alguna estrategia que veremos más adelante o en ejercicios.

## 7.2. Mapeo de enumeraciones con `@PostLoad` y `@PrePersist`

Otra opción para la persistencia de enumeraciones es **utilizar los métodos del estándar de JPA**. Podemos mapear enumeraciones de ida (preescritura) y vuelta (después de la carga) en los **eventos** `@PostLoad` y `@PrePersist`:

- `@PostLoad`: se invoca después de que se cargue una entidad de la base de datos. `PostLoad`.
- `@PrePersist`: se invoca antes de que se persista una entidad en la base de datos. `PrePersist`.

La idea es tener dos atributos en la entidad:

- El primero se **mapea a un valor de base de datos**.
- El segundo es un **campo** `@Transient` que **almacena un valor real de la enumeración**, que es utilizado por el código de lógica de negocio.

Por ejemplo:

```
public enum Prioridad {
    BAJA(100), MEDIA(200), ALTA(300);

    private int prioridad;

    private Prioridad(int prioridad) {
        this.prioridad = prioridad;
    }

    public int getPrioridad() {
        return prioridad;
    }
}
```

```

    }

    public static Prioridad of(int prioridad) {
        return Stream.of(Prioridad.values())
            .filter(p -> p.getPrioridad() == prioridad)
            .findFirst()
            .orElseThrow(IllegalArgumentException::new);
    }
}

```

El método `Prioridad.of()` que hemos empleado muchas veces, facilita la obtención de una instancia de `Prioridad` basada en su valor entero.

En la entidad `Articulo`, añadimos los dos atributos e implantamos los métodos para lectura y escritura:

```

@Entity
public class Articulo {

    @Id
    private int id;

    private String titulo;

    @Enumerated(EnumType.ORDINAL) // Ejemplo con ORDINAL
    private Status estado;

    @Enumerated(EnumType.STRING) // Ejemplo con STRING
    private Tipo tipo;

    @Basic
    private int valorPrioridad; // Propiedad de base de datos

    @Transient
    private Prioridad prioridad; // Propiedad de negocio

    @PostLoad
    void completarTransient() {
        if (valorPrioridad > 0) {
            this.prioridad = Prioridad.of(valorPrioridad);
        }
    }

    @PrePersist
    void completarPersistente() {
        if (prioridad != null) {
            this.valorPrioridad = prioridad.getPrioridad();
        }
    }
}

```

Ahora, al persistir una entidad `Articulo`:

```

Articulo articulo = new Articulo();
articulo.setId(3);
articulo.setTitulo("Título ejemplo");
articulo.setPrioridad(Prioridad.ALTA);

```

JPA desencadenará la siguiente consulta SQL:

```

INSERT INTO Articulo (valorPrioridad, estado, titulo, tipo, id)
VALUES (?, ?, ?, ?, ?)

binding parameter [1] as [INTEGER] - [300]
binding parameter [2] as [INTEGER] - [null]
binding parameter [3] as [VARCHAR] - [Título ejemplo]
binding parameter [4] as [VARCHAR] - [null]
binding parameter [5] as [INTEGER] - [3]

```

No es ideal tener dos atributos que representan una sola enumeración de la entidad. Además, si usamos este tipo de mapeo, **no podemos utilizar el valor del enum en consultas JPQL**.

## 7.3 Mapeo de enumeraciones con `@Converter`

La versión 2.1 de JPA introdujo una nueva API estandarizada que puede ser utilizada para convertir un atributo de entidad a un valor de base de datos y viceversa. Todo lo que necesitamos hacer es crear una nueva clase que implemente `jakarta.persistence.AttributeConverter` y anotarla con `@Converter`.

Una **tercera opción es utilizar un `@Converter`**. Un `@Converter` es una clase que implementa la interfaz `AttributeConverter<X, Y>`, donde `X` es el tipo de atributo de la entidad y `Y` es el tipo de columna de la base de datos. La interfaz `AttributeConverter` tiene dos métodos:

- `Y convertToDatabaseColumn(X attribute)`: convierte el atributo de la entidad en un tipo de columna de la base de datos.
- `X convertToEntityAttribute(Y dbData)`: convierte el tipo de columna de la base de datos en un atributo de la entidad.
- `Class<X> getJavaType()`: devuelve el tipo de atributo de la entidad.
- `Class<Y> getDatabaseType()`: devuelve el tipo de columna de la base de datos.
- `@Converter(autoApply = true)`: indica que el convertidor debe aplicarse a todos los atributos de la entidad que tengan el tipo de atributo `X` y el tipo de columna de la base de datos `Y`.

Primero, crearemos un nuevo enumerado:

```
public enum Categoria {
    DEPORTE("D"), MUSICA("M"), TECNOLOGIA("T");

    private String codigo;

    private Categoria(String codigo) {
        this.codigo = codigo;
    }

    public String getCodigo() {
        return codigo;
    }
}
```

También necesitamos agregarlo a la clase `Articulo`:

```
@Entity
public class Articulo {

    @Id
    private int id;

    private String titulo;

    @Enumerated(EnumType.ORDINAL)
    private Status estado;

    @Enumerated(EnumType.STRING)
    private Tipo tipo;

    @Basic
    private int valorPrioridad;

    @Transient
    private Prioridad prioridad;

    private Categoria categoria;
}
```

Ahora creamos un nuevo convertidor de categoría:

```
@Converter(autoApply = true)
public class ConvertidorCategoria implements AttributeConverter<Categoria, String> {

    @Override
    public String convertToDatabaseColumn(Categoria categoria) {
        if (categoria == null) {
```

```

        return null;
    }
    return categoria.getCodigo();
}

@Override
public Categoria convertToEntityAttribute(String codigo) {
    if (codigo == null) {
        return null;
    }

    return Stream.of(Categoria.values())
        .filter(c -> c.getCodigo().equals(codigo))
        .findFirst()
        .orElseThrow(IllegalArgumentException::new);
}
}
}

```

Hemos configurado el valor `autoApply` de `@Converter` en `true` para que JPA aplique automáticamente la lógica de conversión a todos los atributos mapeados de tipo `Categoría`. De lo contrario, tendríamos que poner la anotación `@Convert` directamente en el campo de la entidad.

## Convert

<https://jakarta.ee/specifications/persistence/3.2/apidocs/jakarta.persistence/jakarta/persistence/convert>

La anotación `@Convert` se puede utilizar para aplicar un convertidor a un atributo específico de una entidad. Tiene como parámetros `attributeName`, `converter` y `disableConversion`.

- `attributeName`: nombre del atributo de la entidad a la que será aplicado el convertidor
- `converter`: clase del convertidor.
- `disableConversion`: booleano que indica si se debe deshabilitar la conversión automática (por defecto `false`). Si está como `true`, no se aplicará el converter, y no debería estar indicado.

```

@Entity
public class Articulo {

    // ...

    @Convert(converter = ConvertidorCategoria.class)
    private Categoria categoria;
}

```

Ahora persistamos una entidad `Articulo`:

```

Articulo articulo = new Articulo();
articulo.setId(4);
articulo.setTitulo("título convertido");
articulo.setCategoria(Categoría.MUSICA);

```

Entonces JPA ejecutará la siguiente instrucción SQL:

```

insert
into
    Articulo
    (categoria, valorPrioridad, estado, titulo, tipo, id)
values
    (?, ?, ?, ?, ?, ?)
Valor convertido al enlazar : MUSICA -> M
binding parameter [1] as [VARCHAR] - [M]
binding parameter [2] as [INTEGER] - [0]
binding parameter [3] as [INTEGER] - [null]
binding parameter [4] as [VARCHAR] - [título convertido]
binding parameter [5] as [VARCHAR] - [null]
binding parameter [6] as [INTEGER] - [4]

```

Podemos establecer reglas para convertir enums a un valor de base de datos correspondiente si usamos la interfaz `AttributeConverter`. Además, podemos agregar nuevos valores de enum o cambiar los existentes sin romper los datos ya persistidos.

Es sencillo de implementar y supera las desventajas de las opciones presentadas en las secciones anteriores.

## 7.4. Uso de Enums en JPQL

Ahora veamos lo sencillo que es usar enums en las consultas JPQL.

Para encontrar todas las entidades `Articulo` con la categoría `Categoría.DEPORTE`:

```
String jpql = "select a from Articulo a where a.categoría = com.pepinho.ad.jpa.Categoría.DEPORTE";
List<Articulo> artículos = em.createQuery(jpql, Articulo.class).getResultList();
```

Es importante destacar que en este caso **necesitamos utilizar el nombre completo de la enumeración**.

Para consultas dinámicas, podemos utilizar parámetros con nombres:

```
String jpql = "select a from Articulo a where a.categoría = :categoría";
TypedQuery<Articulo> query = em.createQuery(jpql, Articulo.class);
query.setParameter("categoría", Categoría.TECNOLOGIA);
List<Article> artículos = query.getResultList();
```

Es la forma más adecuada, pues no se necesita utilizar nombres completamente calificados.

### Ejercicio 05.03. Conversores personalizados y enumeraciones

Declara una entidad `Persona` con atributos:

- `idPersona`.
- `nombre`.
- `apellidos`.
- `fechaNacimiento` de tipo `LocalDate`.
- `sexo` de tipo enumerado `Sexo` que puede ser `HOMBRE` o `MUJER`.
- `estadoCivil` de tipo enumerado `EstadoCivil` que puede ser `SOLTERO`, `CASADO`, `DIVORCIADO` o `VIUDO`.
- `foto` de tipo `byte[]`.

Realiza las conversiones para que:

- El **nombre** y **apellidos** se guardan en la base de datos como “**apellidos, nombre**”, con la primera letra de cada palabra en mayúsculas (empleando acceso por campo y por propiedad).
- La **fecha de nacimiento** como un **entero** que representa la edad de la persona en años (obviamente no es la mejor forma de almacenar la edad, pero quiero que practiquéis con los convertidores), usando anotaciones `@PostLoad` y `@PrePersist`. Haz pruebas de comportamiento haciendo consultas, inserciones y actualizaciones.
- Las **enumeraciones** se guardarán como **cadenas en el caso de estado civil** y como un **carácter de 'H' o 'M' en el caso del sexo**. Hazlo con conversores personalizados.
- La **fotografía** se guardará en un campo de tipo BLOB.

Debes completar la entidad `Persona` y los convertidores necesarios para que funcione correctamente.

```
public class Persona {
    private long idPersona;
    private String nombre;
    private String apellidos;
    private LocalDate fechaNacimiento;
    private Sexo sexo;
```

```

    private EstadoCivil estadoCivil;
    private byte[] foto;
    // ...
}

```

Hazlo contra la base de datos H2 y comprueba que los datos se guardan correctamente, creando varios registros y recuperándolos.

## 8. Tipos temporales: `@Temporal`

Los tipos temporales se refieren al **conjunto de tipos basados en el tiempo que se pueden utilizar en mapeos de estado persistentes**.

La lista de tipos temporales admitidos **incluye los tres tipos** `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`, así como los dos tipos `java.util.Date` y `java.util.Calendar`, así como los tipos de `java.time` de Java 8.

Funcionan como cualquier otro tipo de mapeo simple, sin necesidad de consideraciones especiales.

Sin embargo, **los dos tipos** `java.util.Date` y `java.util.Calendar` **necesitan metadatos adicionales para indicar cuál de los tipos** `java.sql` **de JDBC usar al comunicarse con el controlador JDBC** y sólo pueden ser especificados en campos propiedades de estos dos tipo (o subclases). Esto se hace **anotándolos con la anotación** `@Temporal` **y especificando el tipo JDBC como un valor del tipo enumerado** `TemporalType`.

Tiene un único **elemento value que es un valor de la enumeración** `TemporalType`. Hay tres valores enumerados, que representan los tres tipos de la base de datos `java.sql`:

- `DATE`.
- `TIME`.
- `TIMESTAMP`.

Por ejemplo, con `java.util.Date` y `java.util.Calendar` se pueden asignar a columnas de fecha en la base de datos:

```

@Entity
public class Employee {
    @Id
    private long id;

    @Temporal(TemporalType.DATE)
    private Calendar dob;

    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;

    // ...
}

```

⚡ JPA 3.2 y superior

En JPA 3.2 y superior, esta **anotación está desaprobada (deprecated)**. Se recomienda utilizar los tipos de la API de fecha y hora de Java 8 (`java.time`). Si se necesita persistir un tipo de fecha, se debe utilizar la anotación `@Convert` con un convertidor de atributos.

## 9. Atributos transitorios

Los atributos que no se pretende que sean persistentes pueden **modificarse con el modificador** `transient` en **Java o con la anotación** `@Transient`. El tiempo de ejecución del proveedor no aplicará sus reglas de mapeo predeterminadas al atributo en el que se especificó.

Los campos transitorios se utilizan por diversas razones:

Podría ser el caso anteriormente mencionado **cuando mezclamos el modo de acceso y no queríamos persistir el mismo estado dos veces**. Otra razón podría ser cuando **se desea almacenar en caché algún estado en memoria que no se desea volver a calcular, redescubrir o reinicializar**. Por ejemplo siguiente, se usa un campo `transient` para guardar el nombre específico del idioma para “`Employee`” de modo que lo imprimamos correctamente donde sea que se muestre. Hemos utilizado el modificador `transient` en lugar de la anotación `@Transient` para que si el `Employee` se serializa de una VM a otra, entonces el nombre traducido se reinicializará para corresponder al idioma de la nueva VM. En **casos en los que el valor no persistente debe conservarse durante la serialización**, se debe **utilizar la anotación en lugar del modificador**.

A continuación se muestra un ejemplo de cómo se utilizaría un campo transitorio:

```
@Entity
public class Employee {
    @Id
    private long id;

    private String name;
    private long salary;

    transient private String translatedName;

    // ...

    public String toString() {
        if (translatedName == null) {
            translatedName = ResourceBundle.getBundle("EmpResources").getString("Employee");
        }
        return translatedName + ":" + id + " " + name;
    }
}
```

## 10. Mapeo de clave primaria: `@Id`

- Cualquier entidad **debe tener un mapeo a una clave primaria en la tabla**.
- `@Id` indica el identificador de la entidad.

*Nota: Cuando el identificador de una entidad está compuesto solo por un atributo, se llama un identificador simple.*

### 10.1. Sobrescritura de la clave primaria

Se **puede usar la anotación `@Column`** para sobrescribir el nombre de la columna al que se asigna el atributo `ID`.

Las **claves primarias son insertables**, pero **no nulas ni actualizables**.

Con la anotación `@Column`, **los elementos `nullable` y `updatable` no deben ser anulados**. Sólo al asignar la misma columna a varios campos/relaciones, se debe establecer el elemento `insertable` en `false`.

### 10.2. Tipos de claves primarias

Los mapeos de `@Id` generalmente están restringidos a los siguientes tipos:

- **Tipos primitivos** de Java: `byte`, `int`, `short`, `long` y `char`.
- **Clases envolventes** de tipos primitivos de Java: `Byte`, `Integer`, `Short`, `Long` y `Character`.
- **Cadena**: `java.lang.String`
- Tipo **numérico grande**: `java.math.BigInteger`
- Tipos **temporales**: `java.util.Date` y `java.sql.Date`, `java.util.Calendar` y `java.sql.Timestamp`, además de todos los subtipos y las Java 8 `java.time` API:
  - `java.time.LocalDate`
  - `java.time.LocalDateTime`

- `java.time.LocalDateTime`
- `java.time.OffsetTime`
- `java.time.OffsetDateTime`

### 💡 Float/Double para claves primarias

Se permiten tipos de punto flotante como float y double, así como las clases envolventes `Float` y `Double` y `java.math.BigDecimal`, pero **se desaconsejan debido a la naturaleza del error de redondeo y la poca confiabilidad del operador `equals()` cuando se aplica a ellos**. Utilizar tipos flotantes para claves primarias es arriesgado y definitivamente no se recomienda.

## 10.3. Generación de claves primarias: `@GeneratedValue`

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/generatedvalue>

La **generación de ID y se especifica mediante la anotación `@GeneratedValue`**.

El proveedor de persistencia **generará un valor de identificador para cada instancia de ese tipo de entidad**.

Dependiendo de cómo se genere, es posible que en realidad **no esté presente en el objeto hasta que la entidad se haya insertado en la base de datos, hasta después de que se haya producido un `flush` o la transacción haya finalizado**.

Existen **5 tipos estrategias de generación de ID**, especificando en el elemento `strategy` a alguno de los valores de la enumeración `GenerationType`:

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/generationtype>

- `AUTO`: el proveedor de persistencia debería seleccionar una estrategia apropiada para la base de datos particular.
- `IDENTITY`: asigna claves primarias para la entidad utilizando una **columna de identidad de base de datos**.
- `SEQUENCE`: asigna las claves primarias para la entidad utilizando una **secuencia de base de datos**.
- `TABLE`: asigna claves primarias para la entidad **utilizando una tabla de base de datos subyacente para garantizar la unicidad**.
- `UUID`: asigna las claves primarias para la entidad mediante la **generación de un Identificador Único Universal según la norma RFC 4122**. El tipo de atributo debe ser `java.util.UUID`;

Para obtener más detalles, puedes consultar la documentación oficial en la [API de JPA 3.1](#).

Por ejemplo:

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

Los generadores de **tabla y secuencia** pueden definirse específicamente y luego **reutilizarse por múltiples clases de entidad**. Estos generadores **tienen un nombre y son globalmente accesibles para todas las entidades en la unidad de persistencia**.

### 10.3.1. Generación Automática de ID: `GenerationType.AUTO`

La estrategia de AUTO el proveedor utilizará cualquier estrategia que deseé para generar identificadores.

Se crea un valor de identificador por el proveedor e insertado en el campo `id` de cada entidad `Employee` que se persista.

*Consejo: no se requiere explícitamente que el campo del identificador de la entidad sea de tipo entero, pero generalmente es el único tipo que genera AUTO. Se recomienda emplear `Long` para abarcar toda la extensión del dominio del identificador generado.*

### Ejemplo. Uso de la Generación Automática de ID

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    // ...
}
```

Un **inconveniente** al usar AUTO es que **el proveedor elige su propia estrategia para almacenar los identificadores**, pero si elige una estrategia basada en tabla, necesita crear una tabla, por lo que **necesita permisos para crear una tabla** en la base de datos.

AUTO es realmente una estrategia de generación para desarrollo o prototipado. En cualquier otra situación, sería mejor usar una de las otras estrategias de generación.

### 10.3.2. Generación de ID utilizando una tabla: `GenerationType.TABLE`

La forma **más flexible y portátil de generar identificadores** es utilizar una tabla de base de datos. Se puede adaptar a diferentes bases de datos y **permite almacenar múltiples secuencias de identificadores diferentes para diferentes entidades dentro de la misma tabla**.

Una **tabla de generación de ID debe tener dos columnas**:

- La primera columna es de **tipo cadena y se utiliza para identificar la secuencia del generador en particular**. Es la **clave primaria** para todos los generadores en la tabla.
- La segunda columna es **de tipo entero y almacena la secuencia de ID real que se está generando**. El valor almacenado en esta columna es el **último identificador que se asignó en la secuencia**.
- Cada **generador definido representa una fila en la tabla**.

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE)
private long id;
```

Existen varios enfoques para definir un generador de tabla:

- El enfoque más sencillo es **no definir ningún generador y dejar que el proveedor cree la tabla**. Si se **utiliza la generación (`create`) de esquema, se creará; si no, la tabla predeterminada asumida por el proveedor debe ser conocida y debe existir en la base de datos**.
- Un enfoque más preciso es **especificar la tabla que se utilizará para almacenar el ID**. Esto se hace **definiendo un generador de tabla** que no crea tablas en realidad, pues **es un generador de identificadores que utiliza una tabla para almacenar los valores del identificador**

Podemos definir uno usando la anotación `@TableGenerator` y luego hacer referencia a él por nombre en la anotación `@GeneratedValue`:

```
@TableGenerator(name ="Emp_Gen")
@Id
@GeneratedValue(generator="Emp_Gen")
private long id;
```

```
@Entity
public class Empleado {
    ...
    @TableGenerator(
        name = "generadorEmpleado",
        table = "ID_GEN",
        pkColumnName = "GEN_KEY",
        valueColumnName = "GEN_VALUE",
```

```

    pkColumnValue = "EMP_ID",
    allocationSize = 1)
@Id
@GeneratedValue(strategy = TABLE, generator = "generadorEmpleado")
int id;
...
}

```

Aunque en el ejemplo se indica la anotación `@TableGenerator` anotando el atributo del identificador **se puede definir en cualquier atributo o clase**. El elemento `name` **nombre globalmente al generador**, lo que nos permite **hacer referencia a él en el elemento generator de la anotación @GeneratedValue**. Pero no aprovechamos la flexibilidad de la tabla de generación de ID, pues **no hemos definido ninguna de las propiedades opcionales**.

Independientemente de dónde se defina, **estará disponible para toda la unidad de persistencia**.

*Es una buena práctica definirla localmente en el atributo de ID si solo una clase la está utilizando, y definirla en XML, si se va a utilizar para varias clases.*

Elementos de la anotación `@TableGenerator`:

- `name`: **nombre del generador (opcional)**. El valor por predeterminado es el nombre de la entidad cuando la anotación se produce en una entidad o en una clave primaria.
- `table`: **nombre de la tabla** que almacena los valores de la secuencia de ID (*opcional*). El valor por defecto **lo elige el proveedor de persistencia**.
- `catalog`: catálogo de la tabla (*opcional*). Catálogo por defecto.
- `schema`: esquema de la tabla (*opcional*). Esquema por defecto para el usuario actual.
- `pkColumnName`: **nombre de la columna de clave primaria** en la tabla que identifica de manera única al generador (*opcional*). Por defecto lo elige el proveedor de persistencia.
- `valueColumnName`: **nombre de la columna que almacena el valor real de la secuencia de ID** que se está generando (*opcional*). Por defecto lo elige el proveedor de persistencia.
- `pkColumnValue`: **valor de clave primaria en la tabla generadora** que distingue este conjunto de valores generados de otros que pueden almacenarse en la tabla. El valor predeterminado es un valor elegido por el proveedor para almacenar en la columna de clave principal de la tabla del generador (*opcional*).
- `initialValue`: **valor inicial** de la secuencia de ID (*opcional*).
- `allocationSize`: tamaño de asignación de la secuencia de ID (*opcional*).
- `uniqueConstraints`: restricciones de unicidad de la tabla (*opcional*).
- `indexes`: índices de la tabla (*opcional*).

Un enfoque más cualificado sería **especificar los detalles de la tabla::**

```
@TableGenerator(name="Emp_Gen", table="ID_GEN", pkColumnName="GEN_NAME", valueColumnName="GEN_VAL")
```

Se ha incluido algunos elementos adicionales después del nombre del generador. **Después del nombre, hay tres elementos: table, pkColumnName y valueColumnName, que definen la tabla real que almacena los identificadores para Emp\_Gen**. En el ejemplo:

La tabla se llama `ID_GEN`, el nombre de la columna de clave primaria (la columna que almacena los nombres de los generadores) se llama `GEN_NAME`, y la columna que almacena los valores de la secuencia de ID se llama `GEN_VAL`.

El nombre del generador se convierte en el valor almacenado en la columna `pkColumnName` para esa fila y es utilizado por el proveedor para buscar el generador y obtener su último valor asignado.

El elemento `initialValue` que representa el último identificador asignado puede especificarse como parte de la definición del generador, pero la configuración predeterminada de 0 será suficiente en casi todos los casos. Esta configuración solo se utiliza durante la generación de esquemas cuando se crea la tabla. Durante ejecuciones posteriores, el proveedor leerá el contenido de la columna de valores para determinar el próximo identificador a asignar.

Para evitar actualizar la fila cada vez que se solicita un identificador, se utiliza un tamaño de asignación. Esto hará que el proveedor preasigne un bloque de identificadores y luego asignará identificadores desde la memoria según sea necesario hasta que se agote el bloque. Una vez que se agota este bloque, la próxima solicitud de un identificador activará otro bloque de identificadores para preasignar y el valor del identificador se incrementará por el tamaño de asignación. De forma predeterminada, el tamaño de asignación está configurado en 50. Este valor puede anularse para ser más grande o más pequeño mediante el uso del elemento `allocationSize` al definir el generador.

Ejemplo de cómo definir un segundo generador que se utilizará para entidades de dirección pero que utiliza la misma tabla `ID_GEN` para almacenar la secuencia de identificadores.

Precisamos indicar el valor que estamos almacenando en la columna de clave en elemento `pkColumnName`. Este elemento permite que el nombre del generador sea diferente del valor de la columna:

Especifica un generador de ID de dirección llamado `Address_Gen`, pero luego define el valor almacenado en la tabla para la generación de ID de dirección como `Addr_Gen`. El generador también establece el valor inicial en 10000 y el tamaño de asignación en 100.

```
@TableGenerator(name="Address_Gen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    pkColumnValue="Addr_Gen",
    initialValue=10000,
    allocationSize=100)
@Id @GeneratedValue(generator="Address_Gen")
private long id;
```

Si no se ha indicado “`create`” o “`drop-and-create`”, la tabla debe existir o crearse en la base de datos a través de algún otro medio y configurarse para estar en este estado cuando la aplicación se inicie por primera vez:

```
CREATE TABLE id_gen (
    gen_name VARCHAR(80),
    gen_val INTEGER,
    CONSTRAINT pk_id_gen
    PRIMARY KEY (gen_name)
);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Emp_Gen', 0);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Addr_Gen', 10000);
```

#### Ejercicio 05.04. Generación de ids con tabla

A partir del ejercicio anterior con Persona, haz que el campo `idPersona` de tipo `Long` y genera el identificador con una tabla. La tabla debe ser compartida con otras entidades que tengan un campo `id` de tipo `Long`.

- Nombre de la tabla: `LONG_ID_GEN`
- Columnas:
  - `nomePK`.
  - `valorPK`.
- El valor de la columna `nomePK` para la entidad `Persona` debe ser `PERSONA_ID`.
- Dale un valor inicial de 1000 y un tamaño de asignación de 100.

Crea otro generador para esa tabla que se utilizará para la entidad `Direccion` con un valor inicial de 2000 y un tamaño de asignación de 50.

Haz pruebas de inserción de datos.

### 10.3.3. Generación de ID Utilizando una Secuencia de Base de Datos:

`GenerationType.SEQUENCE`

Muchas **bases de datos admiten un mecanismo interno de generación de ID llamado secuencias**.

Una secuencia de base de datos **se puede utilizar para generar identificadores cuando la base de datos subyacente las admite**.

#### ! Secuencia de una base de datos

Una secuencia de base de datos es un **objeto de base de datos que genera una secuencia de números únicos**. Cada vez que se solicita un número de secuencia, se genera el siguiente número de secuencia. Las secuencias de base de datos son **muy eficientes y se pueden asignar en bloques**. Esto significa que el proveedor puede asignar un bloque de identificadores de la base de datos a la memoria y luego asignar identificadores desde la memoria hasta que se agote el bloque. Una vez que se agota el bloque, el proveedor solicitará otro bloque de identificadores de la base de datos. Esto **reduce la cantidad de comunicación necesaria con la base de datos y mejora el rendimiento**.

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private long id;
```

La **única diferencia entre usar una secuencia para varios tipos de entidad y usar una para cada entidad sería el orden de los números de secuencia y la posible competencia en la secuencia**. La opción más segura es **definir un generador de secuencias con nombre y hacer referencia a él en la anotación @GeneratedValue**:

```
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id
@GeneratedValue(generator="Emp_Gen")
private long id;
```

Se **requeriría que la secuencia esté definida y ya exista**:

```
CREATE SEQUENCE Emp_Seq
MINVALUE 1
START WITH 1
INCREMENT BY 50
```

Si no se utiliza la generación de esquema y la secuencia se crea manualmente, la cláusula `INCREMENT BY` debería configurarse para que coincida con el elemento `allocationSize` o el tamaño de asignación predeterminado de la anotación `@SequenceGenerator` correspondiente.

#### Ejercicio 05.05. Generación de ids con una secuencia

Repite el ejercicio anterior con `Persona`, pero esta vez **utiliza una secuencia para generar el identificador en una base de datos H2**. Haz pruebas compartiendo la secuencia y sin compartir la. Si puedes, haz lo mismo con una base de datos **PostgreSQL**.

### 10.3.4. Generación de ID utilizando una Identidad de Base de Datos

**Muchas bases de datos admiten una columna de identidad de clave primaria, a veces denominada columna autonumérica.**

La identidad **se usa a menudo cuando las secuencias de bases de datos no son compatibles con la base de datos o porque un esquema heredado ya ha definido que la tabla utilice columnas de identidad.**

Generalmente, **son menos eficientes para la generación de identificadores objeto-relacional porque no se pueden asignar en bloques y porque el identificador no está disponible hasta después del tiempo de commit.**

Para indicar que la generación de IDENTITY debe ocurrir, la anotación `@GeneratedValue` **debe especificar una estrategia de generación de IDENTITY:**

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

**No hay una anotación de generador para IDENTITY** porque debe definirse como parte de la definición del esquema de la base de datos para la columna de clave primaria de la entidad.

La generación de **IDENTITY no se puede compartir entre varios tipos de entidades.**

El identificador **no será accesible hasta después de que se haya realizado la inserción.** Es la acción de la inserción la que hace que se genere el identificador. Esto significa que **no se puede utilizar el identificador en una relación bidireccional hasta después de que se haya realizado la inserción.**

Al usar IDENTITY, **algunos proveedores insertan entidades (cuando se invoca el método persist) que están configuradas para usar la generación de IDENTITY, en lugar de esperar hasta el tiempo de commit.**

### 10.3.5. Generación de ID Utilizando un UUID: `GenerationType.UUID`

Incorporado en JPA 3.1, el proveedor de persistencia generará un identificador único universal (UUID) para cada instancia de esa entidad.

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

El tipo de atributo debe ser `java.util.UUID`.

## 11. Ejercicio. Persistencia de una biblioteca

### Ejercicio 05.06. Ampliación de la aplicación de persistencia de una biblioteca

Amplía el ejercicio de la biblioteca para que la entidad `Book` tenga un identificador generado automáticamente por medio de una tabla.

Además:

- Crea una enumeración llamada Categoría con los siguientes valores: `NOVELA`, `POESIA`, `ENSAYO`, `TEATRO` y `OTROS`.
- Haz que la entidad `Book` tenga un atributo de tipo `Categoría` y que se persista en la base de datos como una cadena. Realiza una conversión de la enumeración a cadena y viceversa de modo que guarde la categoría con el nombre en mayúsculas sólo la primera letra y con acentos.
- Haz que la columna ISBN sea única, de un tamaño de 13 caracteres y que no pueda ser nula.
- Crea un atributo de tipo `Calendar` para la fecha de publicación del libro y haz que se persista en la base de datos como un tipo `DATE`.

- Crea un atributo transitorio que sea el número de días que han pasado desde la fecha de publicación hasta la fecha actual. Utiliza la clase `java.time.LocalDate` para obtener la fecha actual.
- Crea otro atributo **transitorio con el ISBN en versión de 10 dígitos**, teniendo en cuenta que el ISBN es un número de 13 dígitos. Para ello, puedes utilizar la clase `java.math.BigInteger` para realizar la conversión y el siguiente algoritmo:
  1. Elimina los primeros tres dígitos (normalmente 978)
  2. Elimina el último dígito. Ahora tienes nueve dígitos
  3. Ahora necesitas calcular el 'dígito de control', que será el décimo dígito de tu ISBN. El objetivo del dígito de control es asegurarse de no haber cometido un error tipográfico: transponer dos dígitos, por ejemplo, o escribir mal uno. Esto es bastante complicado:
  4. Multiplica el primer dígito por 10, el segundo por 9, el tercero por 8 y así sucesivamente, hasta llegar al último dígito (multiplicado por 2).
  5. Ahora tienes una cadena de 9 números nuevos. Agrégalos todos juntos.
  6. Divide esta suma por once. Ahora estás interesado en el resto. Por ejemplo, si la suma fuera 242, que es exactamente  $11 \times 22$ , entonces el resto es cero. Si la suma fuera 243, entonces sobraría 1. Tendrás un resto que está entre 0 y 10.
  7. Resta ese resto de 11 para obtener el dígito de control.
  8. Si el resultado es 10, entonces el dígito de control es 'X'.

Código Java:

```
public class ISBN {
    public static void main(String[] args) {
        String isbn = "978-3-16-148410-0";
        String isbn10 = isbn.substring(3, isbn.length() - 1);
        System.out.println(isbn10);
        BigInteger sum = BigInteger.ZERO;
        for (int i = 0; i < isbn10.length(); i++) {
            int digit = Character.getNumericValue(isbn10.charAt(i));
            sum = sum.add(BigInteger.valueOf(digit).multiply(BigInteger.valueOf(10 - i)));
        }
        System.out.println(sum);
        BigInteger remainder = sum.mod(BigInteger.valueOf(11));
        System.out.println(remainder);
        BigInteger controlDigit = BigInteger.valueOf(11).subtract(remainder);
        System.out.println(controlDigit);
        if (controlDigit.intValue() == 10) {
            System.out.println("X");
        } else {
            System.out.println(controlDigit);
        }
    }
}
```

Un ejemplo más completo:

```
public class ISBNConverter {

    public static void main(String[] args) {
        String isbn13 = "9780123456789"; // ISBN-13
        String isbn10 = convertirISBN13aISBN10(isbn13);
        System.out.println("ISBN-10: " + isbn10);
    }

    public static String convertirISBN13aISBN10(String isbn13) {
        // Verifica si el ISBN-13 proporcionado es válido
        if (!esISBN13Valido(isbn13)) {
            return "ISBN-13 no válido";
        }

        // Elimina los primeros 3 dígitos (978 o 979) del ISBN-13
        String isbn10Parcial = isbn13.substring(3);

        // Calcula el dígito de verificación para el ISBN-10 parcial
        int suma = 0;
        for (int i = 0; i < 9; i++) {
            int digito = Character.getNumericValue(isbn10Parcial.charAt(i));
            suma += (i + 1) * digito;
        }
    }
}
```

```
int digitoVerificador = suma % 11;
char digitoVerificadorChar;

if (digitoVerificador == 10) {
    digitoVerificadorChar = 'X';
} else {
    digitoVerificadorChar = (char) ('0' + digitoVerificador);
}

// Combina el ISBN-10 parcial con el dígito de verificación calculado
return isbn10Parcial + digitoVerificadorChar;
}

public static boolean esISBN13Valido(String isbn13) {
    // Verifica que el ISBN-13 tenga 13 dígitos y comience con "978" o "979"
    return isbn13.matches("^97[89]\\d{10}$");
}
}
```

Crea varios libros y pérsistelos en la base de datos (una nueva). Recupéralos y muestra los valores de los datos, incluyendo transitorios.

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 06. RELACIONES JPA.

---

- 1. Relaciones entre entidades
  - 1.1. Roles de las entidades en las relaciones
  - 1.2. Direccionalidad de las relaciones
  - 1.3. Cardinalidad de las relaciones
  - 1.4. Ordinalidad de las relaciones (opcionalidad)
- 2. Relaciones entre Entidades
- 3. Relaciones mono-valuadas: OneToOne y ManyToOne
  - 3.1. @OneToOne unidireccionales
  - 3.2. @OneToOne bidireccionales
  - Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador
  - 3.3. @ManyToOne unidireccional
    - Empleando @JoinColumn
  - Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo
- 4. Relaciones multi-valuadas
  - 4.1. @OneToMany
    - 4.1.1. @OneToMany bidireccional
    - 4.1.2. @OneToMany unidireccional
  - 4.2. @ManyToMany
- 5. Nombre de la columna de Clave foránea
  - Ejercicio 06.03. Relación ManyToMany unidireccional Jugador-Posición
- Ejercicio 6.4. Mapeo de una base de datos de juegos
- 6. Claves compartidas en relaciones uno a uno
  - 6.1. Claves compartidas con @MapsId
  - 6.2. PrimaryKeyJoinColumn y PrimaryKeyJoinColumns
  - Ejercicio 6.5. Claves compartidas en relaciones uno a uno

## 1. Relaciones entre entidades

La mayoría de las entidades necesitan **referenciar o tener relaciones con otras entidades**. Es lo que produce un modelo gráfico de entidades y relaciones común en las aplicaciones de negocio.

En JPA, las relaciones entre entidades **se definen mediante anotaciones en los atributos/propiedades de las entidades**.

En este apartado vamos a ver cómo se pueden definir relaciones entre entidades en JPA.

Las anotaciones que se utilizan son las siguientes:

- @OneToOne : relación uno a uno.
- @OneToMany : relación uno a muchos.
- @ManyToOne : relación muchos a uno.
- @ManyToMany : relación muchos a muchos.

Además, también se emplean otras anotaciones que permiten concretar y especificar los cuatro tipos de relaciones:

- @Embedded: define una **relación de tipo embebida** (una entidad embebida en otra).
- @ElementCollection: definir una **relación de tipo colección** (una relación uno a muchos en la que hay una dependencia entre las entidades).
- @JoinColumn: define el **nombre de la columna que se utilizará para la relación**.
- @JoinTable: permite definir el **nombre de la tabla que se utilizará para la relación**.
- @MapKey: permite **definir el nombre de la columna que se utilizará como clave en una relación de tipo mapa**.

- `@OrderBy`: nombre de la **columna** que se utilizará para **ordenar los elementos de una relación**.
- `@OrderColumn`: nombre de la **columna que se utilizará para ordenar los elementos de una relación**.
- `@Index`: Permite definir el nombre de la columna que se utilizará para crear un índice en una relación.
- `@ForeignKey`: nombre de la columna que se utilizará para crear una clave foránea en una relación.
- `@AssociationOverride`: nombre de la columna que se utilizará para crear una clave foránea en una relación.
- `@AttributeOverride`: Permite definir el nombre de la columna que se utilizará para crear una clave foránea en una relación.
- `@EmbeddedId`: **definir una clave primaria compuesta**.
- `@IdClass`: **definir una clave primaria compuesta**.

## 1.1. Roles de las entidades en las relaciones

En cada relación hay **dos entidades que están relacionadas**, y **cada entidad se dice que tiene un rol en la relación**.

Los dos roles son:

- Una entidad tiene un rol de **propietario**.
- otra entidad tiene un rol de **inversor**.

El **rol de propietario determina cómo se actualiza la relación** en la base de datos.

El **elemento mappedBy** en la anotación de la relación designa la propiedad o campo en la entidad que es el **propietario de la relación**.

## 1.2. Direccionalidad de las relaciones

El modo más sencillo de implantar relaciones es que **una entidad tenga un atributo que referencia a otra entidad**, que identifica el papel que juega en la relación.

Además, es usual que la **otra entidad tenga un atributo que apunte a la entidad original (relación bidireccional)**.

Las relaciones entre entidades pueden ser:

- **Unidireccionales**: cuando sólo un atributo apunta a la otra entidad (es el lado propietario).
- **Bidireccionales**: cuando cada entidad tiene un/os atributo/s que referencian a la otra entidad.

Más concretamente:

- Una relación **bidireccional tiene un lado propietario (owning) y un lado inverso (non-owning)**.
- Una **relación unidireccional tiene solo un lado propietario**. El lado propietario de una relación determina las actualizaciones de la relación en la base de datos.

### Relación unidireccional:

La **otra entidad no tiene referencia a la primera entidad**. Por ejemplo, en una relación unidireccional **uno a muchos**, la entidad que representa el lado “uno” de la relación tiene una referencia a la entidad que representa el lado “muchos” de la relación, pero la entidad que representa el lado “muchos” de la relación no tiene referencia a la entidad que representa el lado “uno” de la relación.

Por ejemplo, una **relación unidireccional uno a uno** entre las entidades `Empleado` y `Dirección` se puede definir de la siguiente manera:



Empleado-Direccion

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    @OneToOne
```

```
    private Direccion direccion; // Empleado tiene una referencia a Direccion
    // ...
}
```

Se **creará una tabla** `Empleado` **con una columna** `direccion_idDireccion` **que será la clave foránea que referencia a la tabla** `Direccion`. Se dice que Empleado es el propietario de la relación.

```
@Entity
public class Direccion {
    @Id
    private int idDireccion;
    private String calle;
    private String ciudad;
    private String provincia;
    private String pais;
    private String codigoPostal;
    // ...
}
```

La tabla `Direccion` no tiene referencia a la tabla `Empleado`.

A veces, las relaciones unidireccionales en el modelo de objetos son un problema en el modelo de la base.

### Relación bidireccional:

En una **relación bidireccional**, **cada entidad tiene una referencia a la otra entidad**. Por ejemplo, en una relación bidireccional uno a muchos, la entidad que representa el lado “uno” de la relación tiene una referencia a la entidad que representa el lado “muchos” de la relación, y la entidad que representa el lado “muchos” de la relación tiene una referencia a la entidad que representa el lado “uno”

Por ejemplo, `Empleado` y `Proyecto` podría ser una relación bidireccional si el empleado tiene referencia de los proyectos en los que trabaja y el Proyecto tiene referencia de los objetos de tipo Empleado que trabajan en el Proyecto. Ejemplo de **relación bidireccional** entre las entidades `Empleado` y `Proyecto`:



Empleado-Proyecto

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    @ManyToMany
    private List<Proyecto> proyectos;
    // ...
}
```

La tabla `Empleado` no tiene referencia a la tabla `Proyecto`. Sin embargo, se creará una tabla de unión `Empleado_Proyecto` que contendrá las claves primarias de ambas tablas.

```
@Entity
public class Proyecto {
    @Id
    private int idProyecto;
    private String nombre;
    @ManyToMany(mappedBy="proyectos")
    private List<Empleado> empleados;
    // ...
}
```

## 1.3. Cardinalidad de las relaciones

La **cardinalidad de una relación** es el número de instancias de una entidad que pueden estar asociadas con una instancia de la otra entidad.

La cardinalidad de una relación **se especifica mediante el uso de las anotaciones** `@OneToOne`, `@OneToMany`,  
`@ManyToOne` o `@ManyToMany`.

Por ejemplo, **muchos a uno entre Empleado y Departamento**:



Empleado-Departamento



Empleado-Departamento

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    @ManyToOne
    private Departamento departamento;
    // ...
}
```

```
@Entity
public class Departamento {
    @Id
    private int idDepartamento;
    private String nombre;
    @OneToMany(mappedBy="departamento")
    private List<Empleado> empleados;
    // ...
}
```

**Empleado con Proyecto, muchos a muchos:**



Empleado-Proyecto

## 1.4. Ordinalidad de las relaciones (opcionalidad)

La **ordinalidad** indica la **necesidad de que exista** una entidad destino cuando se crea una entidad.

Sirve para mostrar **si la entidad de destino necesita ser especificada cuando se crea la entidad de origen**. Dado que la ordinalidad es realmente **sólo un valor booleano**, también se le conoce como la **opcionalidad de la relación**.

En términos de cardinalidad, la **ordinalidad se indica mediante la cardinalidad siendo un rango en lugar de un valor simple, y el rango comenzaría con 0 o 1 dependiendo de la ordinalidad**.

Es más sencillo simplemente indicar que **la relación es opcional o obligatoria**. Si es **opcional**, el destino **puede no estar presente**; si es **obligatoria**, una entidad de origen sin una referencia a su entidad de destino asociada se encuentra en un estado no válido.

## 2. Relaciones entre Entidades

Las relaciones entre entidades pueden ser:

- **Many-to-One** (Muchos a Uno): `@ManyToOne`
- **One-to-One** (Uno a Uno): `@OneToOne`
- **One-to-Many** (Uno a Muchos): `@OneToMany`
- **Many-to-Many** (Muchos a Muchos): `@ManyToMany`

Si existe una **asociación entre dos entidades**, se debe aplicar una de las siguientes anotaciones de modelado de relaciones **a la propiedad persistente correspondiente o al campo de la entidad referenciadora**:

`@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`. Para asociaciones que no especifican el tipo de destino (por

ejemplo, cuando no se utilizan tipos genéricos de Java para colecciones), es necesario especificar la entidad que es el destino de la relación.

## 3. Relaciones mono-valuadas: OneToOne y ManyToOne

Son aquellas **relaciones en las que la cardinalidad del destino es 1**:

- **OneToOne:**

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetooner>

- **ManyToOne:**

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytoone>

Las relaciones mono-valuadas son las que se establecen entre dos entidades y que se pueden **representar mediante una única columna en la tabla de la entidad que representa el lado “muchos” de la relación** (la entidad tiene un **atributo simple que referencia a la otra entidad**).

La entidad origen referencia a una entidad destino.

### 3.1. `@OneToOne` unidireccionales

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetooner>

Un ejemplo de una asociación uno a uno sería un **Empleado** que tiene **un Aparcamiento**. Suponiendo que cada empleado tenga asignada su propia plaza de aparcamiento, crearíamos una **relación uno a uno desde Empleado hasta Aparcamiento**:

 [Empleado-Aparcamiento](#)

**Entidad propietaria de la relación:**

La entidad **Empleado** tendría un atributo **aparcamiento** que referencia a la entidad **Aparcamiento** y se dice que es la **entidad propietaria de la relación** (tendrá una clave foránea relacionada con Aparcamiento).

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @OneToOne
    private Aparcamiento aparcamiento;
    // ...
}
```

```
@Entity
public class Aparcamiento {
    @Id
    private int idAparcamiento;
    private int numero;
    private String direccion;
    // ...
}
```

Las tablas resultantes serían:



[Empleado-Aparcamiento](#)

Puede verse que la tabla **Empleado** tiene una columna **aparcamiento\_idAparcamiento** que es la clave foránea que referencia a la tabla **Aparcamiento**.

### Anotación @JoinColumn:

También es posible indicar la columna de la relación por medio de la **anotación @JoinColumn**, que permite sustituir el nombre de la columna predeterminada, `aparcamiento_idAparcamiento`, por una nueva:

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @OneToOne
    @JoinColumn(name="idAparcamiento")
    private Aparcamiento aparcamiento;
    // ...
}
```

Tablas resultantes:



Empleado-Aparcamiento

## 3.2. `@OneToOne` bidireccionales

La entidad objetivo de la relación **uno a uno a menudo tiene una relación de vuelta a la entidad fuente**; por ejemplo, `Aparcamiento` tiene una referencia de vuelta al `Empleado` que lo utiliza. Es lo que se llama a **una relación bidireccional uno a uno**.

Sólo se necesita **añadir un atributo `Aparcamiento` para que apunte a `Empleado`**:



Empleado-Aparcamiento

La tabla de **entidad que contiene la columna de unión determina la entidad que es propietaria de la relación**.

En una relación bidireccional uno a uno, **ambas asignaciones son asignaciones de uno a uno, y cualquiera de los lados puede ser el propietario**, por lo que la columna de unión podría terminar en uno u otro lado. *Es una decisión de modelado de datos, no una decisión de programación Java.*

Ahora tenemos que agregar una referencia de vuelta de `Aparcamiento` a `Empleado`. Esto se logra añadiendo la anotación de relación `@OneToOne` en un atributo `empleado`. Como parte de la anotación, **debemos agregar un elemento `mappedBy` para indicar que el lado propietario es `Empleado`, no `Aparcamiento`**.

Dado `Aparcamiento` es el lado inverso de la relación, no se puede suministrar la información de la columna de unión.

```
@Entity
public class Aparcamiento {
    @Id
    private int idAparcamiento;
    private int numero;
    private String direccion;

    @OneToOne(mappedBy="aparcamiento")
    private Empleado empleado;
    // ...
}
```



Empleado-Aparcamiento

Las dos reglas para asociaciones bidireccionales uno a uno son las siguientes:

- La **anotación `@JoinColumn` va en el mapeo de la entidad que está mapeada a la tabla que contiene la columna de unión, o el lado propietario de la relación**. Esto podría estar en cualquiera de los lados de la asociación.

- El **elemento mappedBy** debe especificarse en la anotación `@OneToOne` en la entidad que no define una columna de unión, o el lado inverso de la relación.

 Aviso

No es legal tener una asociación bidireccional que tuviera `mappedBy` en ambos lados, al igual que tampoco incorrecto no tenerlo en ninguno de los lados. La diferencia es que si estuviera ausente en ambos lados de la relación, el proveedor trataría cada lado como una relación unidireccional independiente. Esto estaría bien, excepto que asumiría que cada lado era el propietario y que cada uno tenía una columna de unión.

Si le hubiésemos puesto `@JoinColumn` a la entidad `Aparcamiento`, la tabla resultante sería:

```
@Entity
public class Aparcamiento {
    @Id
    private int idAparcamiento;
    private int numero;
    private String direccion;

    @OneToOne
    @JoinColumn(name="idEmpleado")
    private Empleado empleado;
    // ...
}
```



### Empleado-Aparcamiento

A continuación, pondremos un ejercicio de ejemplo de **relación uno a uno bidireccional**.

## Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador

Vamos a crear una aplicación de equipos de la NBA. Cada equipo tiene un entrenador y cada entrenador tiene un equipo, por lo que la **relación es uno a uno bidireccional**.

Crea las siguientes entidades:

- **Equipo**: con los atributos `idEquipo`, `nombre`, `ciudad`, `conferencia`, `division`, `nombreCompleto` y `abreviatura`.
  - Crea una enumeración `Conferencia` con los valores `ESTE` y `OESTE`.
  - Crea una enumeración `Division` con los valores `ATLANTICO`, `CENTRAL`, `SURESTE`, `NOROESTE`, `PACIFICO` y `SUROESTE`.
  - En la base de datos, **la conferencia y la división se guardarán como cadenas**:
    - `EAST`, `WEST`
    - `ATLANTIC`, `CENTRAL`, `SOUTHEAST`, `NORTHWEST`, `PACIFIC`, `SOUTHWEST`
  - La abreviatura debe ser única, así como el `idEquipo`.

Los equipos puedes cargarlos del siguiente archivo JSON:

[Ver datos de ejemplo](#)

- **Entrenador**: con los atributos `idEntrenador`, `nombre`, `fechaNacimiento`, `salario` y `equipo`.

Mediante JPA e Hibernate, crea una aplicación que permita:

- Añadir un equipo.
- Insertar un entrenador.
- Asignar un entrenador a un equipo.
- Asignar un equipo a un entrenador.

- Mostrar los datos de un equipo y su entrenador.

Para ello, debes crear las clases de utilidad necesarias para realizar las operaciones anteriores. `JpaNbaManager`, `EquipoDAO`, `EntrenadorDAO`, etc.

### 3.3. `@ManyToOne` unidireccional

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytoone>

Se trata de **una de las relaciones más comunes entre entidades**.

Por tratarse de una relación unidireccional, **sólo una entidad tiene una referencia a la otra entidad**, la parte de "muchos" de la relación.

Por ejemplo, una **relación muchos-a-uno entre Empleado y Departamento**:



Relación Many-to-one de Empleado a Departamento

Para ello se usa la **anotación `@ManyToOne`**.

Así la entidad Empleado queda del siguiente modo:

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @ManyToOne
    private Departamento departamento;
    // ...
}
```

La **entidad Departamento no tiene referencia a la parte muchos**, por lo que no se necesita ninguna anotación adicional:

```
@Entity
public class Departamento {
    @Id
    private int idDepartamento;
    private String nombre;
    // ...
}
```

### Empleando `@JoinColumn`

En una base de datos, las relaciones significan que una tabla referencia a otra. Cuando **una columna referencia un clave (primaria) de otra tabla es lo que se denomina “Clave foránea”**.

En JPA las claves foráneas se denominan **“Join Columns”** y, para ello, se emplea la **anotación `@JoinColumn`**.

#### ! `@JoinColumn` y `@JoinTable`

La anotación `@JoinColumn` se utiliza para **especificar una columna de clave foránea en una relación**, usualmente, el **nombre de la relación** (`name`). Si la anotación `@JoinColumn` no se indica, el nombre de la columna de clave foránea se forma como el **nombre de la propiedad o campo de relación de referencia de la entidad o clase embeddable “\_”; el nombre de la columna de clave primaria referenciada**. Por ejemplo, si la relación es `departamento` en la entidad `Empleado`, la columna de clave foránea se llamará `departamento_idDepartamento`.

A veces, las `@JoinColumn` están dentro de otras tablas llamadas **tablas de unión**. En estos casos, se utiliza la anotación `@JoinTable` para especificar el **nombre de la tabla de unión**. Lo veremos en las

relaciones multi-valuadas, como muchos-a-muchos.

Por ejemplo, si quisieramos que la columna de la relación se llamara `idDepartamento` en lugar de `departamento_idDepartamento`, podríamos hacerlo de la siguiente manera:

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @ManyToOne
    @JoinColumn(name = "idDepartamento")
    private Departamento departamento;
    // ...
}
```

La columna `idDepartamento` se añadiría a la tabla `Empleado` y se referenciaría a la columna `idDepartamento` de la tabla `Departamento`.



[Relación tablas Many-to-one de Empleado a Departamento con JoinColumn](#)

**En la mayoría de las relaciones**, independientemente de los lados fuente u origen, **uno de los dos lados tiene una columna de clave foránea que referencia la clave primaria de la otra tabla. El lado que tiene la columna de clave foránea es el lado propietario de la relación.**

La anotación `@JoinColumn` dispone de varios elementos:

- `name`: nombre de la columna de clave foránea.
- `referencedColumnName`: nombre de la columna referenciada por la columna de la clave foránea. Por ejemplo: `@JoinColumn(name="idDepartamento", referencedColumnName="idDepartamento")`. En dónde `idDepartamento` es el nombre de la columna de clave foránea y `idDepartamento` es el nombre de la columna referenciada.
- `nullable`: indica si la columna de clave foránea puede ser nula.
- `unique`: indica si la columna de clave foránea debe ser única.
- `insertable`: indica si la columna de clave foránea debe incluirse en las operaciones de inserción.
- `updatable`: indica si la columna de clave foránea debe incluirse en las operaciones de actualización.
- `columnDefinition`: fragmento SQL que se usa para la generación del DDL de la columna de clave foránea.

#### ! IMPORTANTE: elemento mappedBy

La ausencia del elemento `mappedBy` en la anotación `@ManyToOne` indica que la relación es unidireccional. Si se especifica el elemento `mappedBy` en la entidad no propietaria (inversa, la que no tiene clave foránea), la relación es bidireccional. Además, su ausencia indica que es el propietario de la relación, mientras que la presencia de `mappedBy` indica que no es el propietario de la relación.

## Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo

Siguiendo el ejemplo anterior, vamos a crear una relación muchos a uno unidireccional entre `Jugador` y `Equipo`.

Para ello debe crear una nueva entidad `Jugador` con los siguientes atributos:

- `idJugador`: identificador del jugador.
- `nombre`: nombre del jugador.
- `apellidos`: apellidos del jugador.
- `equipo`: equipo al que pertenece el jugador.
- `altura`: altura del jugador (Double).
- `peso`: peso del jugador (Double).

- `numero`: número de camiseta del jugador (SmallInt).
- `anoDraft`: año de elección en el draft (entero).-
- `numeroDraft`: número de elección en el draft (SmallInt).
- `rondaDraft`: ronda de elección en el draft (SmallInt).
- `posicion`: posición en la que juega (base, escolta, alero, ala-pívot, pivot, como enumeración, que debe guardarse como 'G', 'C', 'F', 'F-C', 'C-F').
- `país`: país de origen del jugador.
- `colegio`: universidad o equipo en el que jugó.
- `foto`: foto del jugador.

Haz que la relación sea unidireccional, de modo que la entidad `Jugador` tenga una referencia al `Equipo` y el nombre de la clave foránea sea `idEquipo`.

Crea jugadores y añádelos a los equipos que has creado en el ejercicio anterior. Completa la aplicación para que puedas añadir jugadores a los equipos y mostrar los jugadores de un equipo.

Datos de ejemplo:

[Ver datos de ejemplo](#)

## 4. Relaciones multi-valuadas

Las relaciones multi-valuadas son aquellas en las que **la cardinalidad del destino es mayor que uno (muchos)**. Esto es, cuando una entidad puede estar asociada con más de una instancia de la otra entidad:

- `@OneToMany`: es la más frecuente.
- `@ManyToMany`:  
<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytomany>

En este caso, **se utilizan colecciones para representar las relaciones** y es importante anotar la parte de la colección con: `@OneToMany` o `@ManyToMany`.

### ⚡ IMPORTANTE: mappedBy en relaciones OneToMany (y ManyToMany)

En las relaciones `@OneToMany`, **la entidad que representa el lado "uno" de la relación suele estar indicada con el elemento `mappedBy`** en la anotación `@OneToMany` para indicar que el lado inverso de la relación es el propietario de la relación y el que tiene la columna de clave foránea.

Aunque en **una relación `@ManyToMany` no es preciso indicar quién es la entidad propietaria con `mappedBy`**, es **recomendable hacerlo** para evitar problemas, y, además, poder especificar el nombre de la tabla de unión (en la que se almacenan las claves foráneas de ambas entidades).

### 4.1. `@OneToMany`

#### 4.1.1. `@OneToMany` bidireccional

Cuando **una entidad** está asociada a una colección, `Collection`, (`java.util.Collection`) de otras entidades, se utiliza la anotación `@OneToMany`.

**Una relación bidireccional one-to-many se establece mediante la anotación `@OneToMany` e implica una relación `@ManyToOne` en el lado opuesto de la relación**, pues **siempre implica una relación many-to-one en el lado opuesto de la relación**.

En este tipo de relaciones, **son (casi) siempre bidireccionales** y el lado "UNO", normalmente, **NO es el propietario de la relación**.

Por ejemplo, entre `Departamento` y `Empleado`:



### Relación One-to-Many de Departamento a Empleado

En el siguiente ejemplo, la entidad `Departamento` tiene una colección de `Empleado` y delega la responsabilidad de la relación a la entidad `Empleado` por el elemento `mappedBy` (**la tabla Departamento no tendrá referencia a la tabla Empleado y la tabla Empleado tendrá una referencia a la tabla Departamento**):

```
@Entity
public class Departamento {
    @Id
    private int idDepartamento;
    private String nombre;
    @OneToMany(mappedBy="departamento") // Empleado debe tener un atributo "departamento"
    private Collection<Empleado> empleados;
    // ...
}
```

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @ManyToOne
    @JoinColumn(name="idDepartamento") // Nombre de la columna de clave foránea
    private Departamento departamento;
    // ...
}
```

El resultado es una **tabla Empleado con la clave foránea del Departamento que referencia a la tabla Departamento**:



### Relación One-to-Many de Departamento a Empleado

La **única diferencia** con la relación `@ManyToOne` es que **se añade el atributo `mappedBy` en la anotación `@OneToMany`**, que indica que el lado inverso de la relación es el propietario de la relación.

Es importante saber lo siguiente en las **relaciones one-to-many o many-to-one bidireccionales**:

- El lado **Many-To\_One**, que tiene la columna de clave foránea (`@JoinColumn`) es el lado propietario de la relación.
- El lado **One-To-Many**, que tiene la anotación `@OneToMany` debe tener el elemento `mappedBy`, pues es el lado inverso de la relación. Si no se especifica `mappedBy`, el proveedor puede tratarlo como una relación unidireccional uno a muchos, que se define con una tabla intermedia.

#### **! Omisión de mappedBy en relaciones OneToMany**

**IMPORTANTE:** si no se indica el elemento `mappedBy` en la anotación `@OneToMany` el proveedor puede tratarlo como una relación unidireccional uno a muchos, que se define con una tabla intermedia.

Es un **error común no especificar `mappedBy` en el lado inverso de una relación bidireccional uno a muchos**. Si no se especifica `mappedBy`, el proveedor puede crear una tabla intermedia para la relación, que no es lo que se desea:



### Relación One-to-Many de Departamento a Empleado con JoinColumn

**Sólo en el caso de relaciones one-to-many unidireccionales se puede omitir `mappedBy`.** Pues en ese caso la parte muchos no tiene referencia a la parte uno.

## 4.1.2 `@OneToMany` unidireccional

En algunos casos, la relación uno a muchos **no tiene elemento `mappedBy` en la anotación `@OneToMany`**, lo que **indica que la relación es unidireccional**. En ese caso **la entidad muchos no tiene referencia a la entidad**

**uno** y sólo existe una colección en la entidad uno que referencia a la entidad muchos.

Por ejemplo, entre **Empleado** y **Telefono** podría verse como una relación unidireccional:

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @OneToMany
    @JoinTable(
        name="EmpleadoTelefono",
        joinColumns=@JoinColumn(name="idEmpleado"),
        inverseJoinColumns=@JoinColumn(name="idTelefono"))
    private Collection<Telefono> telefonos;
    // ...
}
```

Y la parte de la colección de la entidad **Telefono** no tendría referencia a la entidad **Empleado**:

```
@Entity
public class Telefono {
    @Id
    private int idTelefono;
    private String numero;
    private String tipo;
    // ...
}
```

Las tablas resultantes serían:



[Relación One-to-Many de Empleado a Telefono con JoinTable](#)

**Genéricos en colecciones:**

JPA permite el uso de genéricos, por lo que se puede (y se debe) especificar el tipo de colección que se utilizará para la relación. Por ejemplo, **Collection**, **List**, **Set**, **Map**, etc. sin parametrizar.

En el caso de que se quiera emplear genéricos **sin parametrizar**, se debe especificar el tipo de la relación con la anotación **@OneToMany** y el elemento **targetEntity**:

```
@Entity
public class Departamento {
    @Id
    private int idDepartamento;
    private String nombre;
    @OneToMany(targetEntity=Empleado.class, mappedBy="departamento")
    private Collection empleados;
    // ...
}
```

Es **opcional si la colección usa genéricos**, en caso contrario debe especificarse el tipo de elemento con **targetEntity**.

## 4.2. **@ManyToMany**

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytomany>

Cuando **ambos lados** de una relación de entidades tienen una **asociación de una colección**, se trata de una **relación Muchos-a-muchos** y se utiliza la anotación **@ManyToMany**.

Por ejemplo, entre **Empleado** y **Proyecto**:



[Relación Many-to-Many de Empleado a Proyecto](#)

Ambos lados se mapean con la anotación `@ManyToMany`, especificando los parámetros de la **tabla de unión con la anotación `@JoinTable`** (evitamos los valores por defecto):

```
@Entity
public class Empleado {
    @Id
    private int idEmpleado;
    private String nombre;
    private long salario;
    @ManyToMany
    @JoinTable(name = "EmpleadoProyecto",
        joinColumns = @JoinColumn(name="idEmpleado"), // Si hubiera más de una columna, se especificaría
    con un array:
        // joinColumns = {@JoinColumn(name="idEmpleado"), @JoinColumn(name="idOtraColumna")}
        inverseJoinColumns = @JoinColumn(name="idProyecto"))
    private Collection<Proyecto> proyectos;
    // ...
}
```

`joinColumns` es un array (`JoinColumn[] joinColumns`) y podría tener más de un elemento, si la tabla de unión tiene más de una columna de clave foránea:

```
{
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
}
```

```
@Entity
public class Proyecto {
    @Id
    private int idProyecto;
    private String nombre;
    @ManyToMany
    @JoinTable(name = "EmpleadoProyecto",
        joinColumns = { @JoinColumn(name="idProyecto") },
        inverseJoinColumns = @JoinColumn(name="idEmpleado"))
    // ...
}
```

Queda pendiente que hagáis pruebas sin especificar `@JoinTable` para ver cómo se comporta con los valores por defecto. Del mismo modo, que sucede si no se especifica `mappedBy` en el lado inverso de la relación.

```
@Entity
public class Proyecto {
    @Id
    private int idProyecto;
    private String nombre;
    @ManyToMany(mappedBy="proyectos")
    private Collection<Empleado> empleados;
    // ...
}
```

Las tablas resultantes serían:



[Relación Many-to-Many de Proyecto a Empleado con JoinTable](#)

Si no se indica la tabla de unión, el proveedor de JPA creará una tabla de unión con los valores por defecto:



[Relación Many-to-Many de Proyecto a Empleado sin JoinTable](#)

`@JoinTable` permite declarar toda la información sobre las columnas de la tabla de unión, como el **nombre de la tabla, el nombre de las columnas de clave foránea**, etc.

Los nombres de las columnas son en plural porque **podría haber varias columnas por cada clave foránea** (en el caso de una clave primaria con varias columnas).

! IMPORTANTE: `mappedBy` en relaciones `ManyToMany`

Hay una importante diferencia entre las relaciones **many-to-many** y **one-to-many**:

- Cuando **muchos-a-muchos es bidireccional**, **ambos lados de la relación son muchos-a-muchos**.
- **NO HAY columnas @JoinColumn en ninguna de las entidades**, pues **no hay un lado propietario de la relación** y la **única forma de mapear una relación muchos-a-muchos es con una tabla de unión (@JoinTable)**.
- En una relación **many-to-many**, **no hay un lado propietario de la relación**. Ambos lados de la relación son iguales. Por ello, **hay que especificar el lado propietario de la relación con el elemento mappedBy en la anotación @ManyToMany en el lado inverso de la relación**.
- **Da igual cuál es el lado propietario**, pero **sólo se puede especificar mappedBy en uno de los lados de la relación**.

## 5. Nombre de la columna de Clave foránea

El nombre de la **columna de clave foránea** (o de las tablas) se especifica con el **elemento name de la anotación @JoinColumn**. Si no se especifica, el nombre de la columna de clave foránea se genera automáticamente.

```
@Entity
public class Empleado {

    @Id private int idEmpleado;
    private String nombre;
    private long salario;

    @ManyToOne
    @JoinColumn(name = "idDepartamento")
    private Departamento departamento;
    // ...
}
```

El **nombre de la tabla en la que se encuentra depende del contexto**.

Dónde se encuentra la columna de clave foránea depende del tipo de relación y de la estrategia de mapeo de clave foránea:

- Si la unión es para un **mapeo OneToOne o ManyToOne** utilizando una estrategia de mapeo de clave externa, la columna de clave externa está **en la tabla de la entidad fuente o embebible**.
- Si la unión es para un **mapeo unidireccional OneToMany** utilizando una estrategia de mapeo de clave externa, la clave externa está en la **tabla de la entidad objetivo**.
- Si la unión es para un **mapeo ManyToMany o para un mapeo OneToOne o ManyToOne/OneToMany bidireccional utilizando una tabla de unión**, la clave externa está **en una tabla de unión**.
- Si la unión es para una **colección de elementos**, la **clave foránea está en una tabla de colección**. *La colección de elementos lo veremos en otro apartado*.

### Ejercicio 06.03. Relación ManyToMany unidireccional

#### Jugador-Posición

Vamos a crear una relación **muchos a muchos unidireccional** entre **Jugador** y **Posicion**. Para eso debes crear una nueva entidad **Posicion** con los siguientes atributos:

- **idPosicion**: identificador de la posición (Long).
- **nombre**: nombre de la posición (String, tamaño máximo 50).

- **abreviatura**: abreviatura de la posición (String, tamaño máximo 3).
- **descripcion**: descripción de la posición (String, tamaño máximo 255).

Haz que la relación sea unidireccional, de modo que la entidad **Jugador** tenga una colección de **Posicion** y el nombre de la tabla de unión sea **JugadorPosicion**.

Crea posiciones y añádelas a los jugadores que has creado en el ejercicio anterior.

## Ejercicio 6.4. Mapeo de una base de datos de juegos

### Migración de base de datos H2 entre versiones

En la base de datos origen se ejecuta el siguiente script:

```
SCRIPT TO '<ruta-al-archivo-backup>/backup.sql';
```

En la base de datos destino se ejecuta el siguiente script:

```
RUNSCRIPT FROM '<ruta-al-archivo-backup>/backup.sql';
```

### Ejercicio 6.4. Mapeo de una base de datos de juegos.

Disponemos de una base de datos de juegos, que se compone de las siguientes tablas (la base de datos compartida está en el fichero anexo)

**Plataforma**: idPlataforma, nombre. (*Ya contiene datos*) **Genero**: idGenero, nombre. (*Ya contiene datos*)

**Juego**: idJuego, idGenero (FK), idPlataforma (FK), titulo, miniatura (varchar), estado, descripciónCorta, descripción, url, editor, desarrollador, fecha. **Imagen**: idImagen, idJuego (FK), url, imagen (tipo BLOB).

**RequisitosSistema**: idJuego (PK), almacenamiento, graficos, memoria, os, procesador.

Referencias: <https://www.freetogame.com/api-doc>

- Las plataformas pueden ser: pc, browser, all, etc. (Ya disponibles en la tabla Plataforma)
- Las categorías (géneros) pueden ser:
  - *mmorpg, shooter, strategy, moba, racing, sports, social, sandbox, open-world, survival, pvp, pve, pixel, voxel, zombie, turn-based, first-person, third-Person, top-down, tank, space, sailing, side-scroller, superhero, permadeath, card, battle-royale, mmo, mmofps, mmotps, 3d, 2d, anime, fantasy, sci-fi, fighting, action-rpg, action, military, martial-arts, flight, low-spec, tower-defense, horror, mmorts*, etc. (Ya incorporadas en la tabla Genero)

Cuyos datos se ajustan al formato del siguiente JSON (ejemplo). Debes tener en cuenta que no se ha creado la tabla de requerimientos mínimos, pero se puede hacer si se desea en una nueva tabla de la base de datos, relacionada, uno a uno:

```
{
  "id": 452,
  "title": "Call Of Duty: Warzone",
  "thumbnail": "https://www.freetogame.com/g/452/thumbnail.jpg",
  "status": "Live",
  "short_description": "A standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare.",
  "description": "Call of Duty: Warzone is both a standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare. Warzone features two modes \u2014 the general 150-player battle royale, and \u2014 Plunder\u2014. The latter mode is described as a \u2014crace to deposit the most Cash\u2014. In both modes players can both earn and loot cash to be used when purchasing in-match equipment, field upgrades, and more. Both cash and XP are earned in a variety of ways, including completing contracts.\r\n\r\nAn interesting feature of the game is one that allows"
}
```

players who have been killed in a match to rejoin it by winning a 1v1 match against other felled players in the Gulag.\r\n\r\nOf course, being a battle royale, the game does offer a battle pass. The pass offers players new weapons, playable characters, Call of Duty points, blueprints, and more. Players can also earn plenty of new items by completing objectives offered with the pass.",

```

    "game_url": "https://www.freetogame.com/open/call-of-duty-warzone",
    "genre": "Shooter",
    "platform": "Windows",
    "publisher": "Activision",
    "developer": "Infinity Ward",
    "release_date": "2020-03-10",
    "freetogame_profile_url": "https://www.freetogame.com/call-of-duty-warzone",
    "minimum_system_requirements": {
        "os": "Windows 7 64-Bit (SP1) or Windows 10 64-Bit",
        "processor": "Intel Core i3-4340 or AMD FX-6300",
        "memory": "8GB RAM",
        "graphics": "NVIDIA GeForce GTX 670 / GeForce GTX 1650 or Radeon HD 7950",
        "storage": "175GB HD space"
    },
    "screenshots": [
        {
            "id": 1124,
            "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-1.jpg"
        },
        {
            "id": 1125,
            "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-2.jpg"
        },
        {
            "id": 1126,
            "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-3.jpg"
        },
        {
            "id": 1127,
            "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-4.jpg"
        }
    ]
}
```

a) Crea entidades JPA en Java para las tablas de la base de datos, con las siguientes características:

- **Genero**: con los atributos `idGenero`, `nombre`. La clave es autonumérica.
- **Plataforma**: con los atributos `idPlataforma` y `nombre`. La clave es autonumérica. *Nota: si se hubiese declarado como enumeración, para poder mapear una enumeración en una tabla independiente, obligaría a crear una entidad independiente con el idPlataforma y el nombre. Sin embargo, en este caso, se podría mapear la enumeración directamente en la tabla Juego o declararla como una clase y no como una enumeración.*
- **Juego**: con todos los atributos de la tabla Juego, incluyendo la relación con Genero y Plataforma. La clave primaria, **`idJuego, no es autogenerada`**, es asignada. Ten en cuenta que la relación con la tabla Imagen se trata de una relación uno a muchos, por lo que se deberá declarar una colección de imágenes. Además, el **idGenero y el idPlataforma son claves foráneas de las entidades y no deben declararse como atributos de la entidad Juego, sino como objetos del tipo de las entidades Genero y Plataforma**.
- **Imagen**: con los atributos `idImagen` (**no autogenerada**), `Juego` (relacionada con la entidad Juego @OneToOne), `url`, `imagen` (tipo byte[]).
- **RequisitosSistema**: relacionada con la tabla Juego. Atributos: `idJuego` (PK), `sistemaOperativo` (su nombre no coincide con la columna de la tabla), `almacenamiento`, `graficos`, `memoria`, `procesador` y su relación `juego`. Debe emplearse una clave comparta con el idJuego. Para ello debe emplearse la anotación: `@MapsId : @MapsId("idJuego")`.

```

@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@MapsId
@JoinColumn(name = "idJuego")
private Juego juego;
```

b) Haz una sencilla aplicación que cree un juego y lo persista en la base de datos. Ten en cuenta que **las claves no son autonuméricas**

- Ejemplo de juegos: <https://www.freetogame.com/api/game?id=X>, pasándole el id del Juego, desde 1 al número de juegos que consideres. Ten en cuenta que el juego podría no existir devolviendo:

```
{"status":0,"status_message":"No game found with that id"}
```

## 🔗 Bases de datos

# 6. Claves compartidas en relaciones uno a uno

Una **clave compartida** es una clave primaria que se comparte entre dos o más entidades. Una clave compartida se puede mapear con la anotación `@MapsId` y se emplea para **relaciones uno a uno**.

En este caso el identificador de un solo atributo es la **clave foránea de la relación**.

Por ejemplo, en una **relación bidireccional uno a uno entre las entidades** `Empleado` y `HistorialEmpleado`. Dado que solo hay un `HistorialEmpleado` por `Empleado`, podríamos decidir **compartir la clave primaria** (la clave primaria de `HistorialEmplado` sería la misma que `Empleado`).

Si `HistorialEmpleado` es la entidad dependiente, indicamos que la clave foránea de la relación es el identificador anotando la relación con `@Id` y `@OneToOne`. (En realidad suele escribirse la anotación `@MapsId` se coloca en el atributo de relación para indicar que también está mapeando el atributo de ID).

```
@Entity
public class HistorialEmpleado {
    // ...
    @Id
    @OneToOne
    @JoinColumn(name="idEmpleado")
    private Empleado empleado;
    // ...
}
```

El **tipo de clave primaria de `HistorialEmpleado`** va a ser del mismo tipo que `Empleado`, por lo que si `Empleado` tiene un identificador simple de tipo entero, entonces el identificador de `HistorialEmpleado` también será un entero.

*Si `Empleado` tiene una clave primaria compuesta, ya sea con una clase ID o una clase ID incrustada, entonces `HistorialEmpleado` compartirá la misma clase ID (y también debería estar anotada con la anotación `@IdClass`).*

*El problema es que esto choca con la regla de la clase ID que dice que debe haber un atributo coincidente en la entidad por cada atributo en su clase ID. Esta es la excepción a la regla, debido al hecho mismo de que la clase ID se comparte entre ambas entidades, principal y dependiente.*

## 6.1. Claves compartidas con `@MapsId`

<https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/mapsid>

Generalmente, también se podría desear que la **entidad contenga un atributo de clave primaria además del atributo de relación**, con ambos atributos mapeados a la misma columna de clave foránea en la tabla.

Aunque el **atributo de clave primaria es innecesario en la entidad** podría querer definirse por separado para un **acceso más fácil**. A pesar de que los dos atributos se mapean a la misma columna de clave foránea (que también es la columna de clave primaria), el **mapeo no tiene que duplicarse en ambos lugares**. La anotación `@Id` se coloca en el atributo de identificación, y `@MapsId` anota el atributo de relación para indicar que también está mapeando el atributo de ID:

```
@Entity
public class HistorialEmpleado {
    // ...
    @Id
    int idEmpleado;

    @MapsId // Indica que el atributo de relación también mapea el atributo de ID
}
```

```

@OneToOne
@JoinColumn(name="idEmpleado")
private Empleado empleado;
// ...
}

```

Hay un par de puntos adicionales que vale la pena mencionar sobre @MapsId:

- La relación anotada con `@MapsId` **define el mapeo para el atributo de identificación también**. Si no hay una anotación `@JoinColumn` que anule en el atributo de relación, entonces la columna de unión se asignará por defecto (`nombreEntidad_idEntidad`). En el ejemplo anterior, si se eliminara la anotación `@JoinColumn`, tanto el atributo `empleado` como el `idEmpleado` se mapearían a la columna de clave foránea predeterminada `Empleado_idEmpleado` (suponiendo que la columna de clave primaria en la tabla Empleado fuera `idEmpleado`).
- Aunque **el atributo de identificación** comparte el mapeo de la base de datos definido en el atributo de relación, desde la perspectiva del atributo de identificación, **es realmente un mapeo de solo lectura**. **Las actualizaciones o inserciones en la columna de clave foránea de la base de datos solo ocurrirán a través del atributo de relación**. Esta es una de las razones por las que **siempre se debe establecer las relaciones padre antes de intentar persistir una entidad dependiente** (Debes persistir, por ejemplo, primero Empleado y luego HistorialEmpleado).

#### ! IMPORTANTE: Claves compartidas

**Nota:** No intentes establecer solo el atributo de identificación (y no el atributo de relación) como un medio para atajar la persistencia de una entidad dependiente. Algunos proveedores pueden tener soporte especial para hacer esto, pero no garantizará de manera portátil que la clave foránea se escriba en la base de datos.

**El atributo de identificación se completará automáticamente por el proveedor cuando se lea una instancia de entidad de la base de datos o cuando se realiza un flush/commit.** Sin embargo, **no se puede asumir que esté presente al llamar primero a persist() en una instancia a menos que el usuario lo establezca explícitamente**.

## 6.2. PrimaryKeyJoinColumn y PrimaryKeyJoinColumns

La anotación `@PrimaryKeyJoinColumn` se utiliza para **especificar una columna de clave primaria de una tabla de unión**. Esto es, cuando la **clave foránea es la clave primaria de la tabla de unión**. Se usa en **relaciones uno a uno y solo se puede usar en la entidad propietaria de la relación**.

`@PrimaryKeyJoinColumns` se utiliza para **especificar varias columnas de clave primaria de una tabla de unión**.

Por ejemplo:

```

@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idEmpleado;
    private String nombre;
    private long salario;

    @OneToOne
    @PrimaryKeyJoinColumn // Indica que la columna de clave primaria de la tabla de unión es la misma que la clave primaria de la tabla de la entidad
    private HistorialEmpleado historial;
    // ...
}

```

En este caso, la clave primaria de la tabla `HistorialEmpleado` sería la clave foránea de la tabla Empleado.

```

@Entity
public class HistorialEmpleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)

```

```
private int id;  
private String nombre;  
private long salario;  
// ...  
}
```

En el segundo código, la entidad `HistorialEmpleado` tiene un campo `@Id` que actúa como clave primaria de la tabla, por lo que no tiene sentido usar `@PrimaryKeyJoinColumn` en este caso. La entidad `HistorialEmpleado` tiene un campo `@Id` que actúa como clave primaria de la tabla.

## Ejercicio 6.5. Claves compartidas en relaciones uno a uno

Comprueba el funcionamiento de la anotación `@PrimaryKeyJoinColumn` en una relación uno a uno entre **Persona** y **Departamento**. Crea las entidades y realiza pruebas de persistencia.

**Persona:** `idPersona` (IDENTITY), `nombre`, `departamento` (uno a uno con anotación de `@PrimaryKeyJoinColumn`)

**Departamento:** `idDepartamento` (IDENTITY), `nombre`.

Modifica el ejercicio para que sea bidireccional con `@OneToOne` y `@MapsId` en la entidad Departamento y como propietaria de la relación.

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 07. OBJETOS EMBEBIDOS.

- 1. Objetos embebidos: `@Embeddable`
- 2. Sustitución de atributos embebidos: `@AttributeOverride`
  - 2.1. Sustitución de múltiples atributos embebidos: `@AttributeOverrides`
  - Ejercicio 7.1. Elementos embebidos.
- 4. ManyToMany usando una clave compuesta
  - 4.1. Modelando Atributos de Relación
  - 4.2. Creando una Clave Compuesta en JPA: `@Embeddable`
  - 4.3. Utilizando una Clave Compuesta en JPA: `@EmbeddedId`
  - 4.4. Características Adicionales
    - Ejercicio 7.2. Clave compuesta en una relación de muchos a muchos.
  - 4.5. La anotación `@IdClass`
  - 4.6. La anotación EmbeddedId (repaso)
  - 4.7. @IdClass vs @EmbeddedId
- Ejercicio 7.3. Entidades principales de base de datos de películas.

## 1. Objetos embebidos: `@Embeddable`

Un **objeto embebido** es un objeto que **no tiene identidad propia** y que **es parte de una entidad**.

Los objetos embebidos se utilizan para **modelar datos compuestos**.

Es parte del estado de una entidad que **ha sido extraído y modelado como un objeto independiente**.

En Java los objetos embebidos se modelan como **clases normales** y se **anotan con** `@Embeddable`. Sin embargo, en la base de datos, **los objetos embebidos se almacenan en la misma tabla que la entidad que los contiene**, como cualquier otro de sus atributos.

! `@Embeddable` y `@Embedded`

La anotación `@Embeddable` se utiliza para **indicar que una clase es un objeto embebido**. La anotación `@Embedded` se utiliza para **indicar que un atributo de una entidad es un objeto embebido**.

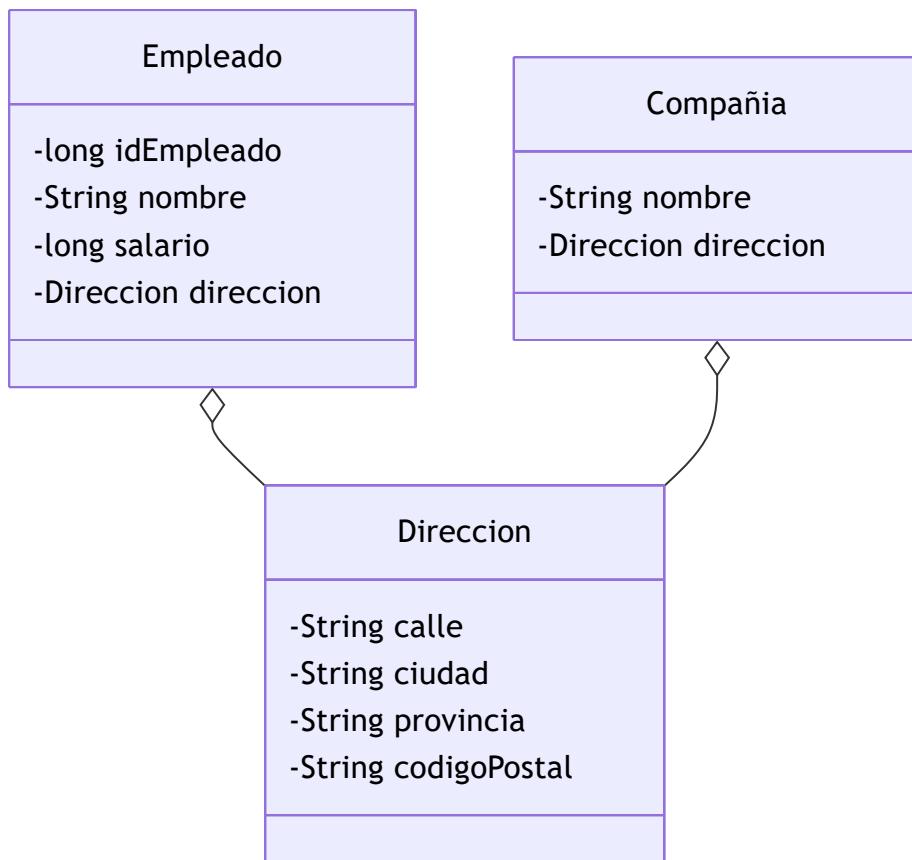
Aunque los objeto embebidos son referenciados por entidades, **no se consideran relaciones entre entidades**.

Aunque pudiera parecer contradictorio separar un objeto de una entidad y luego volver a unirlo, hay varias razones por las que esto es útil:

- **Los objetos embebidos pueden ser reutilizados.** Si tienes un objeto embebido que representa una dirección, por ejemplo, puedes reutilizarlo en cualquier entidad que necesite una dirección.

Aunque los tipos embebidos pueden ser compartidos o reutilizados, las instancias no. **Una instancia de objeto embebido pertenece a la entidad que la referencia; y ninguna otra instancia de entidad, de ese tipo de entidad o de cualquier otro, puede hacer referencia a la misma instancia embebida.**

Un ejemplo de reutilización es la información de la dirección postal:



La tabla Empleado:

Empleado	
idEmpleado	int
nombre	varchar
salario	int
calle	varchar
ciudad	varchar
provincia	varchar
codigoPostal	varchar

idEmpleado	nombre	salario	calle	ciudad	provincia	codigoPostal
1	Pepe	1500	C/1	Santiago	A Coruña	15706
2	Xan	2200	C/2	Vigo	Pontevedra	36201

Una tabla **Empleado** que contiene una mezcla de información básica del empleado, así como columnas que corresponden a la dirección postal del empleado. Las **columnas calle, ciudad, provincia y codigoPostal se combinan lógicamente para formar la dirección (clase Direccion)**.

En el modelo de objetos, es una excelente candidata para ser **“abstracta” en un tipo embebido Direccion en lugar de incorporar cada atributo en la clase de la entidad**. La clase de entidad solo tendría un atributo de

dirección que apunta a un objeto embebido de tipo Address. La figura muestra cómo `Empleado` y `Direccion` se relacionan entre sí.

```

@Embeddable
@Access(AccessType.FIELD)
public class Direccion {
    private String calle;
    private String ciudad;
    private String provincia;
    @Column(name="codigoPostal")
    private String codigo;
    // ...
}

@Entity
public class Empleado {
    @Id private long idEmpleado;
    private String nombre;
    private long salario;
    @Embedded
    private Direccion direccion;
    // ...
}

```

Al persistir una instancia de `Empleado`, se accede a los atributos del objeto `Direccion` como si estuvieran presentes en `Empleado`.

Las asignaciones de columnas en el tipo `Direccion` realmente se refieren a las columnas de la tabla `Empleado`, aunque estén en una clase separada.

### ! Objetos embebidos o entidades

Es una **decisión de diseño** el uso de objetos embebidos. **Si se precisa crear relaciones con ellos o desde ellos, no los uses.** Los objetos embebidos **no están destinados a ser entidades** y tan pronto como comiences a tratarlos como entidades, probablemente deberías convertirlos en entidades de primera clase si el modelo de datos lo permite.

**No es portátil definir objetos embebidos como parte de jerarquías de herencia.** Una vez que comienzan a heredarse entre sí, la complejidad de su incorporación aumenta y la relación costo-beneficio disminuye.

Una **clase `Direccion` podría ser reutilizada** tanto en las entidades `Empleado` como `Compañía` (como hemos indicado en la imagen anterior).



Empleado-Dirección-Compañía

Aunque tanto las clases `Empleado` como `Compañía` contiene la clase `Direccion`, **cada instancia de `Direccion` será utilizada solo por una única instancia de `Empleado` o `Compañía`.**

## 2. Sustitución de atributos embebidos: `@AttributeOverride`

`@AttributeOverride`

Como las asignaciones de columnas del tipo embebido `Direccion` se aplican a las columnas de la entidad contenedora (en tablas diferentes), **las tablas de entidades podrían tener nombres de columna diferentes para los mismos campos.**

Empleado		Compania	
idEmpleado	int	nombre	varchar
nombre	varchar	calle	varchar
salario	int	ciudad	varchar
calle	varchar	prov	varchar
ciudad	varchar	codPostal	varchar
provincia	varchar		
codigoPostal	varchar		

La tabla `Empleado` coincide con los atributos predeterminados y mapeados del tipo `Direccion`, pero la tabla `Compania` se ha modificado con otros nombres de provincia y código postal:

```

@Entity
public class Empleado {
    @Id private long idEmpleado;
    private String nombre;
    private long salario;
    @Embedded
    private Direccion direccion;
    // ...
}

@Entity
public class Compania {
    @Id private String name;
    @Embedded
    @AttributeOverride(name = "provincia", column = @Column(name = "prov")),
    @AttributeOverride(name = "codigoPostal", column = @Column(name = "codPostal"))
    private Direccion direccion;
    // ...
}

```

Como se muestra en el ejemplo, para los cambios de nombres en los atributos se puede emplear la anotación `@AttributeOverride`.

En la declaración de la entidad se emplea la anotación `@AttributeOverride` para cada atributo del objeto embebido que queremos renombrar en la entidad. Elementos requeridos:

- `name`: el nombre del campo o propiedad embebido en la entidad.
- `column`: la columna a la que se está asignando el atributo en la tabla de la entidad. Se especifica en forma de una anotación `@Column` anidada.

## 2.1. Sustitución de múltiples atributos embebidos:

`@AttributeOverrides`

Si sobre escribimos múltiples campos o propiedades, se puede usar la anotación plural `@AttributeOverrides` y anidar múltiples anotaciones `@AttributeOverride` dentro de ella:

```

@Entity
public class Compania {
    @Id private String name;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "provincia", column = @Column(name = "prov")),
        @AttributeOverride(name = "codigoPostal", column = @Column(name = "codPostal"))
    })
    private Direccion direccion;
    // ...
}

```

}

Dado que la anotación `@AttributeOverride` puede repetirse, **no es obligatorio el uso de la anotación `@AttributeOverrides`**. El siguiente ejemplo muestra el uso de `Direccion` tanto en `Empleado` como en `Compañía`. La entidad `Empleado` utiliza el tipo `Direccion` sin cambios, pero la entidad `Compañía` sobrescribe para asignar los atributos `provincia` y `codigoPostal` de `Direccion` a las columnas `prov` y `codPostal` de la tabla `Compañía`.

```

@Entity
public class Empleado {
    @Id private long idEmpleado;
    private String nombre;
    private long salario;
    @Embedded
    private Direccion direccion;
    // ...
}

@Entity
public class Compañía {
    @Id private String name;
    @Embedded
    @AttributeOverride(name = "provincia", column = @Column(name = "prov"))
    @AttributeOverride(name = "codigoPostal", column=@Column(name = "codPostal"))
    private Direccion direccion;
    // ...
}

```

## Ejercicio 7.1. Elementos embebidos.

Crea una **aplicación con JPA para la gestión de películas y series**.

1. Crea una clase `InfoContenido` con los siguientes atributos:

- `titulo` (String): de tamaño 100.
- `genero` (String): de tamaño 50.
- `pais` (String): de tamaño 2.
- `duracion` (int): duración en minutos.
- `año` (int): año.
- `sinopsis` (String): de tamaño clob.

2. Crea una entidad `Serie` con los siguientes atributos:

- `idSerie` (long): identificador de la serie. Secuencia.
- `informacion` (de tipo `InfoContenido`)
- `fechaEstreno` (LocalDate).
- `temporadas` (int): número de temporadas.
- `capitulos` (int)
- `directores` (lista de String).

3. Crea una entidad `Pelicula` con los siguientes atributos:

- `idPelicula` (long): identificador de la película. Secuencia.
- `informacion` (de tipo `InfoContenido`)
- La entidad `Serie` y `Pelicula` deben tener el atributo `informacion` como un objeto embebido.
- La entidad `Pelicula` el atributo `pais` debe ser renombrado a `paisPelicula`.
- El atributo `directores` debe guardarse en una nueva tabla, como una colección con la anotación `@ElementCollection` (busca información sobre esta anotación).
- La fecha de estreno, `fechaEstreno`, de la serie debe guardarse en formato numérico (YYYYMMDD).

## 4. ManyToMany usando una clave compuesta

A modo de ejemplo, tenemos una **relación de muchos a muchos entre dos entidades**, `Estudiante` y `Curso`. Un estudiante puede inscribirse en varios cursos, y un curso puede tener varios estudiantes inscritos. Además, queremos que los estudiantes califiquen los cursos, que sería **un atributo de la relación**. Un estudiante puede calificar cualquier número de cursos, y cualquier número de estudiantes puede calificar el mismo curso.

Una **clave primaria compuesta**, también llamada **clave compuesta**, es una combinación de dos o más columnas para formar una clave primaria para una tabla.

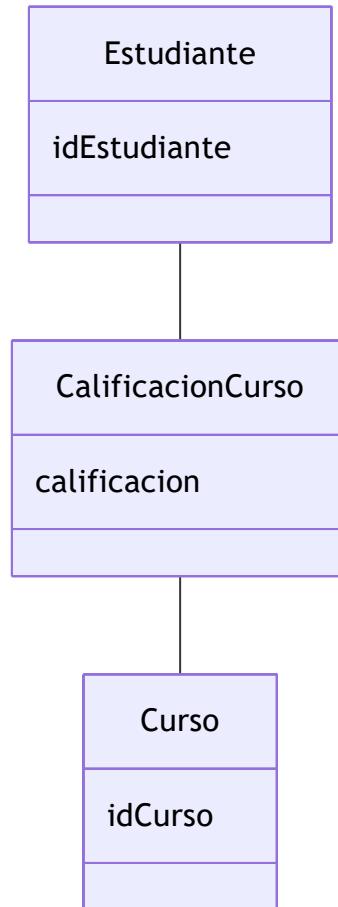
En JPA, tenemos **dos opciones para definir las claves compuestas: las anotaciones** `@IdClass` y `@EmbeddedId`

Para definir las claves primarias compuestas, debemos seguir algunas reglas:

- La clase de **clave primaria compuesta debe ser pública**.
- Debe tener un **constructor sin argumentos**.
- Debe **definir los métodos** `equals()` y `hashCode()`.
- Debe ser **Serializable**.

## 4.1. Modelando Atributos de Relación

Queremos permitir que **los estudiantes califiquen los cursos**. Un estudiante puede calificar cualquier número de cursos, y **cualquier número de estudiantes puede calificar el mismo curso**. Por lo tanto, también es una **relación de muchos a muchos**.

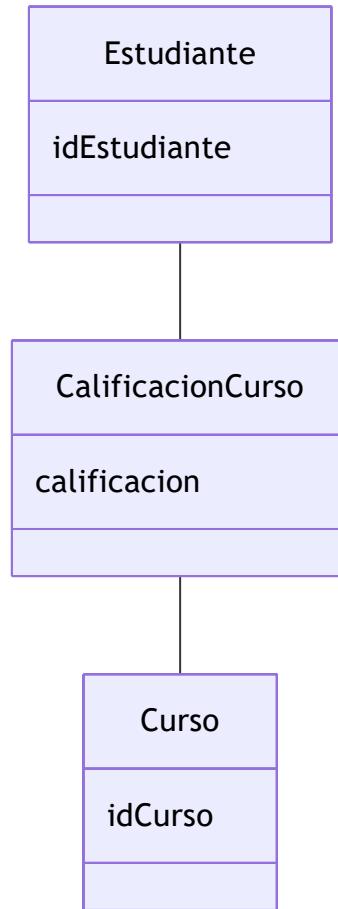


A diferencia de las otras relaciones muchos a muchos, necesitamos almacenar la puntuación de calificación que el estudiante dio al curso, en la tabla intermedia.

¿Dónde podemos almacenar esta información? No podemos ponerla en la entidad `Estudiante` ya que un estudiante puede dar diferentes calificaciones a diferentes cursos. De manera similar, almacenarlo en la entidad `Curso` tampoco sería una buena solución.

Esta es una situación en la que **la relación en sí tiene un atributo**.

Usando este ejemplo, adjuntar un atributo a una relación se ve así en un diagrama ER:



Podemos modelarlo casi de la misma manera que la relación de muchos a muchos sencilla. La única diferencia es que **añadimos un nuevo atributo a la tabla de unión**:



[Estudiante - CalificacionCurso - Curso](#)

## 4.2. Creando una Clave Compuesta en JPA: `@Embeddable`

La implementación de una relación de **muchos a muchos sencilla** fue bastante directa, pero **no podemos agregar una propiedad a una relación de esa manera porque conectamos las entidades directamente**. Por lo tanto, **no teníamos forma de añadir una propiedad a la relación en sí**.

Dado que mapeamos los atributos de la base de datos a campos de clase en JPA, **necesitamos crear una nueva clase de entidad para la relación**:



[Estudiante - CalificacionCurso - Curso](#)

Cada entidad JPA **necesita una clave primaria**. Dado que **la clave primaria es una clave compuesta**, tenemos que crear una **nueva clase para la clave**, `ClaveCalificacionCurso`, que contendrá las diferentes **partes de la clave**:

```

@Embeddable
class ClaveCalificacionCurso implements Serializable {

    @Column(name = "idEstudiante")
    private Long idEstudiante;

    @Column(name = "idCurso")
    private Long idCurso;
}

```

```
// constructores estándar, getters y setters
// implementación de hashCode y equals
}
```

Ten en cuenta que una clase de clave compuesta debe cumplir con algunos requisitos clave:

- **Debe marcarse con `@Embeddable`.**
- **Debe implementar `java.io.Serializable`.**
- Necesitamos proporcionar una **implementación de los métodos `hashCode()` e `equals()`**.

## 4.3. Utilizando una Clave Compuesta en JPA. `@EmbeddedId`

Usando esta clase de clave compuesta, podemos crear la clase de entidad que modela la tabla de unión:

```
@Entity
class CalificacionCurso {

    @EmbeddedId
    private ClaveCalificacionCurso id;

    @ManyToOne
    @MapsId("idEstudiante")
    @JoinColumn(name = "idEstudiante")
    private Estudiante estudiante;

    @ManyToOne
    @MapsId("idCurso")
    @JoinColumn(name = "idCurso")
    private Curso curso;

    int calificacion;

    // constructores estándar, getters y setters
}
```

Este código es muy similar a una implementación usual de entidad. Sin embargo, tenemos algunas diferencias clave:

- **Usamos `@EmbeddedId` para marcar la clave primaria**, que es una instancia de la clase `ClaveCalificacionCurso` (recuerda que antes añadimos la anotación `@Embeddable` a esta clase).
- **Marcamos los campos `estudiante` y `curso` con `@MapsId`.**
- **`@MapsId` significa que vinculamos esos campos a una parte de la clave** y son las claves externas de una relación de muchos a uno.

Después de esto, podemos **configurar las referencias inversas en las entidades `Estudiante` y `Curso` como antes**:

```
class Estudiante {
    // ...
    @OneToMany(mappedBy = "estudiante")
    Set<CalificacionCurso> calificaciones;
    // ...
}

class Curso {
    // ...
    @OneToMany(mappedBy = "curso")
    Set<CalificacionCurso> calificaciones;
    // ...
}
```

Ten en cuenta que **hay una forma alternativa de usar claves compuestas: la anotación `@IdClass`**.

## 4.4. Características Adicionales

Configuramos las relaciones con las **clases Estudiante** y **Curso como @ManyToOne**, porque **con la nueva entidad descompusimos estructuralmente la relación de muchos a muchos en dos relaciones de muchos a uno.**

Ahora tenemos **dos relaciones de muchos a uno**. En otras palabras, no hay ninguna relación de muchos a muchos en un RDBMS. Llamamos a las estructuras que creamos con tablas de unión relaciones de muchos a muchos porque eso es lo que modelamos.

Además, es más claro si hablamos de relaciones de muchos a muchos porque esa es nuestra intención. Mientras tanto, **una tabla de unión es solo un detalle de implementación**; realmente no nos importa.

Esta solución tiene una característica adicional que aún no hemos mencionado. **La solución simple de muchos a muchos crea una relación entre dos entidades**. Por lo tanto, no podemos expandir la relación a más entidades. Pero no tenemos este límite en esta solución: **podemos modelar relaciones entre cualquier número de tipos de entidades**.

Cuando varios profesores pueden enseñar un curso, los estudiantes pueden calificar cómo un profesor específico enseña un curso específico. De esa manera, una calificación sería una relación entre tres entidades: un estudiante, un curso y un profesor.

### Ejercicio 7.2. Clave compuesta en una relación de muchos a muchos.

Data la aplicación de gestión de películas y series, añade dos nuevas entidades: **Usuario** y **Calificacion** que permita a los usuarios calificar las películas.

1. Crea una clase **Usuario** con los siguientes atributos:

- **idUsuario** (long): identificador del usuario. Secuencia.
- **nombre** (String): nombre del usuario.
- **email** (String): email del usuario.
- **password** (String): contraseña del usuario.
- **fechaRegistro** (LocalDate): fecha de registro.

2. Crea una clase **Calificacion** con los siguientes atributos:

- **calificacion** (int): calificación del contenido, con valores de 10 a 0.
- **fechaCalificacion** (LocalDate): fecha de la calificación.
- **comentario** (String): comentario de la calificación.
- Además, debe estar relacionado con las entidades **Usuario**, **Pelicula** y **Serie**. Como un usuario puede calificar varias películas y series, y una película o serie puede ser calificada por varios usuarios, es una relación de muchos a muchos. *No es preciso que califique series, pues el caso de uso es similar al de las películas.*

La clave primaria de la tabla **Calificacion** debe ser compuesta por los atributos **idUsuario**, **idPelicula**.

## 4.5. La anotación `@IdClass`

Digamos que tenemos una tabla llamada **Cuenta** y tiene dos columnas, **numeroCuenta** y **tipoCuenta**, que forman la clave compuesta. Ahora tenemos que mapearlo en JPA.

Según la especificación de JPA, creamos una clase **IdCuenta** con estos campos de clave primaria:

```
public class IdCuenta implements Serializable {
    private String numeroCuenta;
    private String tipoCuenta;
```

```
// constructor por defecto

public IdCuenta(String numeroCuenta, String tipoCuenta) {
    this.numeroCuenta = numeroCuenta;
    this.tipoCuenta = tipoCuenta;
}

// métodos equals() y hashCode()
```

A continuación, asociemos la clase `IdCuenta` con la entidad `Cuenta`.

Para hacer eso, necesitamos anotar la entidad con la anotación `@IdClass`. También debemos declarar los campos de la clase `IdCuenta` en la entidad `Cuenta` y anotarlos con `@Id`:

```
@Entity
@IdClass(IdCuenta.class)
public class Cuenta {
    @Id
    private String numeroCuenta;

    @Id
    private String tipoCuenta;

    // otros campos, getters y setters
}
```

## 4.6 La anotación `EmbeddedId` (repaso)

`@EmbeddedId` es una alternativa a la anotación `@IdClass`.

Consideremos otro ejemplo en el que tenemos que persistir información de un `Book`, con `titulo` y `idioma` como los campos de clave primaria.

En este caso, la clase de clave primaria, `IdLibro`, debe estar anotada con `@Embeddable`:

```
@Embeddable
public class IdLibro implements Serializable {
    private String titulo;
    private String idioma;

    // constructor por defecto

    public IdLibro(String titulo, String idioma) {
        this.titulo = titulo;
        this.idioma = idioma;
    }

    // getters, métodos equals() y hashCode()
}
```

Luego, necesitamos incrustar esta clase en la entidad `Libro` usando `@EmbeddedId`:

```
@Entity
public class Libro {
    @EmbeddedId
    private IdLibro bookId;

    // constructores, otros campos, getters y setters
}
```

## 4.7. `@IdClass` vs `@EmbeddedId`

Como podemos ver, la diferencia superficial entre estos dos es que con `@IdClass` tuvimos que especificar las columnas dos veces, una vez en `IdCuenta` y nuevamente en `Cuenta`; sin embargo, con `@EmbeddedId` no lo hicimos.

Sin embargo, hay algunas otras compensaciones.

Por ejemplo, estas estructuras diferentes **afectan las consultas JPQL que escribimos**.

Con `@IdClass`, la consulta es un poco más sencilla:

```
SELECT cuenta.numeroCuenta FROM Cuenta cuenta
```

Con `@EmbeddedId`, tenemos que hacer un recorrido extra:

```
SELECT libro.idLibro.titulo FROM Libro book
```

Además, `@IdClass` puede ser bastante útil en lugares donde estamos utilizando una clase de **clave compuesta que no podemos modificar**.

Si vamos a acceder a partes de la clave compuesta individualmente, podemos hacer uso de `@IdClass`, pero **en lugares donde usamos frecuentemente el identificador completo como un objeto, se prefiere @EmbeddedId**

## Ejercicio 7.3. Entidades principales de base de datos de películas.

```
<property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas"/>
<property name="jakarta.persistence.jdbc.user" value="accesodatos"/>
<property name="jakarta.persistence.jdbc.password" value="ad123.."/>
<property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver"/>
<property name="jakarta.persistence.schema-generation.database.action" value="none"/>
```

Sea la siguiente estructura de la base de datos:



Estructura de la base de datos

› SQL de las tablas de la base de datos

En el que:

- El **título de la película** se guarda en el campo `castelan`.
- El **identificador de la película es entero** (no autoincremento).
- Los participantes de la película están relacionados por medio de la tabla `peliculapersonaxe`, en la que el campo `ocupacion` identifica o tipo de ocupación de la película ('Actor', ...):

```
CREATE TABLE IF NOT EXISTS `ocupacion` (
  `ocupacion` varchar(50) COLLATE utf8_spanish_ci DEFAULT NULL,
  `orde` int(11) NOT NULL,
  UNIQUE KEY `Ocupación#PX` (`ocupacion`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```



Ocupación



Personaxeocupación

Para empezar, **crea las entidades** `Pelicula`, `Personaxe` y `Ocupacion`. A continuación, **crea la entidad** `PeliculaPersonaxe` **que relaciona las entidades** `Pelicula` y `Personaxe` y `Ocupacion`. Ten en cuenta que tiene un nuevo atributo `personaxeInterpretado`, que **es el nombre del personaje interpretado por el actor en la película**.

Mejora:

Crea las entidades asociadas a la base de datos. De modo que las columnas `anoInicio`, `otrasDuraciones`, `video`, `laserDisc` pertenezca a una entidad `DetallePelicula` de tipo embebido. Hay que tener en cuenta que estos elementos no siempre están presentes, por lo que deben ser opcionales.

---

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 08. MAPEO DE COLECCIONES.

- 1. Relaciones y colecciones de elementos (`@ElementCollection`).
- 2. Tabla de colección: `@CollectionTable`
  - 2.1 Columnas de colección: `@Column`
- 3. Ordenación de colecciones
  - 3.1. `@OrderBy`
  - 3.2. `@OrderColumn`
- 4. Generación de claves primarias para colecciones de elementos: `@CollectionId` (Hibernate) (\*)
- 5. Ejemplo de mapeo de colecciones
- 6. One-to-many vs `@ElementCollection`
  - Anotaciones One to Many
  - Anotación ElementCollection
- Resumen

## 1. Relaciones y colecciones de elementos (

`@ElementCollection`)

- <https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/elementcollection>

Cuando hablamos de mapear colecciones hay **tres tipos de objetos que pueden contener colecciones**:

- **Entidades** (`@OneToMany` o `@ManyToMany`).
- **Elementos embebidos** (`@ElementCollection`)
- **Elementos básicos** (`@ElementCollection`)

Sin embargo, las **colecciones de elementos embebidos y básicos no se consideran relaciones**. Son colecciones de elementos que se denominan **colecciones de elementos**, pues, a diferencia de las relaciones, que se relacionan con entidades independientes, **las colecciones de elementos contienen objetos que dependen de la entidad que los referencia**.

Sólo pueden ser obtenidos a través de la entidad que los contiene.

La anotación `@ElementCollection` se utiliza para **mapear colecciones de elementos embebidos o básicos**. Dispone de dos elementos opcionales:

- `targetClass`: de tipo `Class` indica la clase básica o embebida del elemento de la colección. Es opcional si el tipo de elementos de la colección se indica con genéricos, obligatorio en caso contrario.
- `fetch`: de tipo `FetchType` tipo de carga de los elementos de la colección, perezosa o ansiosa. **Por defecto es perezosa**.

En el siguiente ejemplo se requiere indicar la clase de los elementos de la colección con el atributo `targetClass`, pues es una colección sin genéricos (perezosa es el valor por defecto):

```

@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection(targetClass=Vacaciones.class, fetch=FetchType.LAZY)
    private Collection vacaciones;
    // ...
}

```

Por ejemplo con lista de cadenas de texto con genéricos (ambos atributos son opcionales):

```
@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection
    private List<String> telefonos;
    // ...
}
```

En este caso, la colección de teléfonos es una colección de elementos básicos. No es una relación, pues los **teléfonos no son entidades independientes, sino que dependen de la persona que los referencia.**

También podría ser un **conjunto (Set) de elementos**:

```
@Entity public class Persona {
    @Id
    protected String numeroSeguridadSocial;
    protected String nombre;
    // ...
    @ElementCollection
    protected Set<String> apodos = new HashSet();
    // ...
}
```

O una **colección de elementos embebidos**:

```
@Embeddable
public class Vacaciones {
    @Temporal(TemporalType.DATE)
    private Calendar fechaInicio;

    @Column(name="duracionDias")
    private int duracion;
    // ...
}

@Entity
public class Persona {
    @Id
    private long idPersona;
    private String nombre;
    @ElementCollection(targetClass=Vacaciones.class, fetch=FetchType.LAZY)
    private Collection vacaciones;
    @ElementCollection
    protected Set<String> apodos = new HashSet();
    // ...
}
```

En el ejemplo anterior, la **anotación** `@ElementCollection` **incluye el atributo** `targetClass` **que indica la clase de los elementos de la colección** porque no se ha indicado el tipo de elementos que contiene la colección.

## 2. Tabla de colección: `@CollectionTable`

La anotación `@CollectionTable` se utiliza para **especificar la tabla de la colección**.

En el caso de las colecciones de elementos embebidos, los elementos no son entidades, por lo que no tendrá asociada una tabla, sin embargo, al no ser posible almacenar varios elementos en la misma columna, **las colecciones de elementos embebidos y tipos básicos (algunos sistemas gestores de base de datos disponen de tipo de datos para ARRAY) requieren una tabla independiente denominada "Tabla de colección" que almacena los elementos de la colección.**

Es posible **especificar la tabla de la colección con la anotación** `@CollectionTable`:

```

@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection(targetClass=Vacaciones.class, fetch=FetchType.LAZY)
    @CollectionTable(name="Vacaciones", joinColumns=@JoinColumn(name="idPersona"))
    private Collection vacaciones;
    // ...
}

```

En este caso, la tabla `Vacaciones` almacenará los elementos de la colección `vacaciones` de la entidad `Persona`. La tabla `Vacaciones` tendrá una columna `idPersona` que será la clave foránea que relacionará los elementos de la colección con la entidad `Persona`.

Si no se especifica la tabla de la colección, se utilizará una **tabla por defecto con el nombre de la entidad y el nombre de la propiedad de la colección separados por un guion bajo**. Por ejemplo, si no se especifica la tabla de la colección, la tabla por defecto de la colección `vacaciones` de la entidad `Persona` se llamará `Persona_Vacaciones`.

```

@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection(targetClass=Vacaciones.class, fetch=FetchType.LAZY)
    private Collection vacaciones;
    // ...
}

```

## 2.1 Columnas de colección: `@Column`

Por defecto, las columnas de la tabla de la colección tendrán el mismo nombre que la propiedad de la colección. Sin embargo, **se pueden especificar las columnas de la tabla de colección con la anotación `@Column`**.

- Con **colecciones de tipo básico**, el nombre de la columna se obtiene de nombre de campo o propiedad de la colección.
  - Se puede sobrescribir con la anotación `@Column` indicando el nombre de la columna.

```

@ElementCollection // usa la tabla por defecto: Persona_telefonos
@Column(name="telefono")
private List<String> telefonos;

```

- Con **colecciones de clases embebidas** los nombres de las columnas se corresponden con los de las propiedades de la clase embebida.
  - Se puedes sobrescribir con la anotación `@AttributeOverride` o `@AttributeOverrides` indicando el nombre de la columna (si tiene referencias a otras entidades puede sobrescribirse con `@AssociationOverride` o `@AssociationOverrides`).

```

@ElementCollection
@CollectionTable(name="Residencia")
@AttributeOverrides({
    @AttributeOverride(name="calle", column=@Column(name="calleCasa")),
    @AttributeOverride(name="ciudad", column=@Column(name="ciudadCasa")),
    @AttributeOverride(name="provincia", column=@Column(name="provinciaCasa"))
})
private Set<Direccion> direcciones = new HashSet();

```

En el siguiente ejemplo completo se detalla cómo se pueden **especificar las columnas de la tabla de colección con la anotación `@Column`**:

```

@Embeddable public class Direccion {
    protected String calle;
    protected String ciudad;
    protected String provincia;
    // ...
}

@Entity public class Persona {
    @Id protected String numeroSeguridadSocial;
    protected String nombre;
    protected Direccion casa;
    // ...
    @ElementCollection // usa la tabla por defecto: Persona_Alias
    @Column(name="nombre", length=50)
    protected Set<String> alias = new HashSet();
    // ...
}

@Entity public class Medico extends Persona {
    @ElementCollection
    @CollectionTable(name="Casa") // usa el nombre por defecto de la clave foránea.
    @AttributeOverrides({
        @AttributeOverride(name="calle", column=@Column(name="calleCasa")),
        @AttributeOverride(name="ciudad", column=@Column(name="ciudadCasa")),
        @AttributeOverride(name="provincia", column=@Column(name="provinciaCasa"))
    })
    protected Set<Direccion> casas = new HashSet();
    // ...
}

```

En el ejemplo anterior hemos sobreescrito los nombres de las columnas de la tabla de colección `Casa` con la anotación `@AttributeOverride` y `@AttributeOverrides`.

## 3. Ordenación de colecciones

### 3.1. `@OrderBy`

La anotación `@OrderBy` se utiliza para **ordenar los elementos de una colección**. Se puede aplicar a **colecciones de elementos básicos o embebidos, así como cualquier relación multivaluada (`@OneToMany`, `@ManyToMany`) de tipo List**.

```

@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection
    @OrderBy("fechaInicio DESC")
    private List<Vacaciones> vacaciones;
    // ...
}

```

En este caso, la colección `vacaciones` se ordenará por la fecha de inicio de las vacaciones en orden descendente.

El **elemento de la anotación `@OrderBy` es una lista de nombres de propiedades separados por comas**, que se utilizan para ordenar los elementos de la colección.

La sintaxis es la siguiente:

```

@OrderBy("propiedad1 [ASC|DESC], propiedad2 [ASC|DESC], ...")

```

Si no se especifica `ASC` o `DESC`, el orden **por defecto es ascendente (`ASC`)**.

```

orderby_list ::= orderby_item [,orderby_item]*
orderby_item ::= [property_or_field_name] [ASC | DESC]

```

Si no se especifica `@OrderBy` en una asociación de una entidad con una colección de elementos, **el orden por defecto es el orden de la clave primaria**.

Para **referir a una propiedad de un elemento embebido, se puede utilizar la notación de punto**. Por ejemplo:

```
@Entity
public class Persona {
    @Id
    private long idPersona;
    private String nome;
    @ElementCollection
    @OrderBy("rua ASC")
    private List<Direccion> direccions;
    // ...
}
```

En este caso, la colección `direccions` se ordenará por la `rua` de las direcciones en orden ascendente.

Si se quiere ordenar por una propiedad de un elemento embebido, **se puede utilizar la notación de punto**. Por ejemplo:

```
@Entity
public class Persona {

    @ElementCollection
    @OrderBy("codigoPostal.codigoProvincia, codigoPostal.codigoConcello")
    public Set<Direccion> getResidencias() {
        // ...
    }
    //...

    @Embeddable
    public class Direccion {
        protected String rua;
        protected String cidade;
        protected String provincia;
        @Embedded protected CodigoPostal codigoPostal;
    }

    @Embeddable
    public class CodigoPostal {
        protected String codigoProvincia;
        protected String codigoConcello;
    }
}
```

Para ordenar por varias propiedades, se pueden especificar varias propiedades separadas por comas. Por ejemplo:

```
@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection
    @OrderBy("fechaInicio DESC, duracion ASC")
    private List<Vacaciones> vacaciones;
    // ...
}
```

La ordenación puede realizarse para cualquier **relación multivaluada de tipo List**:

```
@Entity
public class Curso {
    // ...
    @ManyToMany
    @OrderBy("apellidos ASC")
    public List<Estudiante> getEstudiantes() {
        //...
    }
}
```

```

    }
    // ...
}

@Entity
public class Estudiante {
    // ...
    @ManyToMany(mappedBy="estudiantes")
    @OrderBy // ordena por clave primaria
    public List<Curso> getCurso() {
        //...
    }
    // ...
}

```

### 3.2. `@OrderColumn`

Otro tipo de ordenación es por medio de la anotación `@OrderColumn` que se utiliza para **ordenar los elementos de una colección**. Se puede aplicar a colecciones de elementos básicos o embebidos, así como cualquier relación multivaluada (`@OneToMany`, `@ManyToMany`) de tipo `List`.

El uso de `@OrderColumn` es incompatible con `@OrderBy` (uno u otro, no ambos).

```

@Entity
public class Estudiante {
    @Id
    private long idEstudiante;
    private String nombre;
    @OneToMany(mappedBy="estudiante")
    @OrderColumn(name="nombre")
    private List<Materia> materias;
    // ...
}

```

Si no se especifica el nombre de la columna, **se utilizará el nombre de la propiedad de la colección seguido de “\_ORDER”**.

```

@Entity
public class Estudiante {
    @Id
    private long idEstudiante;
    private String nombre;
    @OneToMany(mappedBy="estudiante")
    @OrderColumn
    private List<Materia> materias;
    // ...
}

```

## 4. Generación de claves primarias para colecciones de elementos: `@CollectionId` (Hibernate) (\*)

A modo de curiosidad, se puede mencionar que existe una anotación `@CollectionId` que se utiliza para **generar claves primarias para las colecciones de elementos** en Hibernate (no en JPA, por lo que **este apartado no es relevante para el examen**).

La anotación `@CollectionId` es exclusiva de Hibernate se utiliza para **generar claves primarias para las colecciones de elementos**. Se puede aplicar a colecciones de elementos básicos o embebidos.

```

@Entity
public class Persona {
    @Id
    private long id;
    private String nombre;
    @ElementCollection

```

```

@CollectionId(columns=@Column(name="idVacaciones"), type=@Type(type="long"),
generator="sequence")
private Collection vacaciones;
// ...
}

```

En este caso, la colección `vacaciones` tendrá una clave primaria generada por una secuencia.

## 5. Ejemplo de mapeo de colecciones

En el siguiente ejemplo se muestra cómo mapear una colección de elementos básicos y una colección de elementos embebidos:

```

@Entity
public class Persona {
    @Id
    private long idPersona;
    private String nombre;
    @ElementCollection
    @CollectionTable(name="Vacaciones", joinColumns=@JoinColumn(name="idPersona"))
    @Column(name="fechaInicio")
    private Collection<Date> vacaciones;
    @ElementCollection
    @CollectionTable(name="Direccion", joinColumns=@JoinColumn(name="idPersona"))
    @AttributeOverrides({
        @AttributeOverride(name="calle", column=@Column(name="calleCasa")),
        @AttributeOverride(name="ciudad", column=@Column(name="ciudadCasa")),
        @AttributeOverride(name="provincia", column=@Column(name="provinciaCasa"))
    })
    private Collection<Direccion> direcciones;
    // ...
}

```

En este caso, la entidad `Persona` tiene una colección de fechas de vacaciones y una colección de direcciones. La tabla `Vacaciones` almacenará las fechas de vacaciones y la tabla `Direccion` almacenará las direcciones de la entidad `Persona`.

```

@Entity
public class Curso {
    // ...
    @ManyToMany
    @OrderBy("apellidos ASC")
    public List<Estudiante> getEstudiantes() {
        // ...
    };
    // ...
}

```

En este caso, la colección `estudiantes` se ordenará por el apellido de los estudiantes en orden ascendente.

```

@Entity
public class Estudiante {
    // ...
    @ManyToMany(mappedBy="estudiantes")
    @OrderBy // ordena por clave primaria
    public List<Curso> getCurso() {
        //...
        //
    }
    // ...
}

```

## 6. One-to-many vs `@ElementCollection`

### Anotaciones One to Many

Necesitamos usar anotaciones **One to Many** si creamos una **relación entre dos entidades (TABLAS)**.

Podemos representar **Tienda**, la tienda tiene muchas sucursales, y ambas son una entidad, lo que significa que ambas tienen una tabla dentro de nuestra base de datos.

### Tienda.java

```
@Entity
@Table(name = "tienda")
public class Tienda {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idTienda;

    @Column(name = "nombre")
    private String nombre;

    @Column(name = "url")
    private String url;

    @OneToMany(mappedBy = "tienda", cascade = CascadeType.ALL)
    private Set<Sucursal> sucursales = new HashSet<>();
}
```

### Y Sucursal.java

```
@Entity
@Table(name = "sucursal")
public class Sucursal {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idSucursal;

    @Column(name = "nombre")
    private String nombre;

    @Column(name = "url")
    private String url;

    @ManyToOne
    private Tienda tienda;
}
```

Después de ejecutar el código, el proveedor de persistencia (Hibernate...) creará dos tablas dentro de tu base de datos, la primera llamada **tienda** y la segunda llamada **sucursal**.

Ahora necesitamos crear un repositorio o una clase DAO para ambas entidades para insertar y obtener los datos de la base de datos.

Si usamos Spring Boot, podemos usar **@Repository** para crear un componente Repositorio (lo veremos más adelante). Podemos usar **@Repository** para crear un componente Repositorio.

### TiendaRepository.java

```
@Repository
public interface TiendaRepository extends JpaRepository<Tienda, Long> { }
```

Y ahora necesitamos crear un Repositorio para la entidad **Sucursal**.

### SucursalRepository.java

```
@Repository
public interface SucursalRepository extends JpaRepository<Sucursal, Long> { }
```

Y después de eso, tenemos una nueva relación entre Tienda y Sucursal.

## Anotación ElementCollection

`@ElementCollection` es una anotación JPA estándar (anteriormente con Hibernate se empleaba la anotación propietaria `CollectionOfElements`).

Significa que la colección no es una colección de entidades, sino **una colección de tipos simples** (cadenas, etc.) o **una colección de elementos integrables/embedidos** (clase anotada con `@Embeddable`).

También significa que **los elementos son propiedad completamente de las entidades que los contienen**: se modifican cuando se modifica la entidad, se eliminan cuando se elimina la entidad, etc. **No pueden tener su propio ciclo de vida**.

Ahora veremos cómo podemos implementar con colección de elementos.

La anotación más simple, `@ElementCollection`, le dice al compilador que estamos asignando una colección, en la que `@CollectionTable` proporciona el nombre de la tabla objetivo, y luego `@JoinColumn` especifica la columna real que unimos como se muestra a continuación:

Del último ejemplo, ahora tenemos dos entidades `Tienda` y `Sucursal`, y la relación entre ellas, y añadiremos una nueva clase llamada `Producto`.

**Producto.java:**

```
@Embeddable
public class Producto {
    // no necesitamos usar id porque no es una entidad
    @Column(name="nombre")
    private String nombre;

    @Column(name="precio")
    private Double precio;
}
```

Y ahora añadimos una nueva anotación `Elementcollection` en la clase `Tienda` y un objeto `producto` de la clase `Producto`.

**Tienda.java**

```
@Entity
@Table(name = "tienda")
public class Tienda {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idTienda;

    @Column(name = "nombre")
    private String nombre;

    @Column(name = "url")
    private String url;

    @OneToMany(mappedBy = "tienda" , cascade = CascadeType.ALL)
    private Set<Sucursal> sucursales = new HashSet<>();

    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "ProductoTienda", joinColumns = @JoinColumn(name = "idTienda", nullable = false), uniqueConstraints = {@UniqueConstraint(columnNames = {"idTienda"})})
    private Set<Product> products = new HashSet<>();
}
```

Y ahora después de crearlo **la tabla de `Producto` tiene `idTienda` como clave primaria**. Pero **es una clave foránea de la tabla de tienda y no una clave primaria para la tabla de productos**.

El **objeto `Embeddable` no tiene un Repository/DAO porque depende de la entidad**, por ejemplo, **el producto embebido depende de la entidad de la Tienda**. Y podemos insertarlo sin usar el **Repository de la Tienda**.

### La diferencia entre ambos enfoques:

#### Entidad:

Cuando creamos una entidad, podemos relacionarla con el repositorio/DAO JPA para escribir nuestra consulta o usar la consulta integrada de JPA.

#### Embeddable:

Cuando creamos una clase integrable, debe relacionarse con cualquier clase de entidad, porque **depende de la clase de entidad. y no podemos crear un repositorio.**

## Resumen

- `@ElementCollection`, por otro lado, es muy similar a `@OneToMany` excepto que el **objeto objetivo no es una entidad**.
- Con `@ElementCollection`, **no podemos consultar, persistir o fusionar (merge) objetos objetivo de forma independiente de su objeto principal**.
- **No admite operaciones de cascada**. Esto significa que **los objetos objetivo siempre se persisten, se fusionan o se eliminan junto con su objeto principal**.
- `@ElementCollection` **es una forma fácil de definir una colección con objetos simples/básicos**.
- `@OneToMany` **es el mejor para casos de uso complejos en los que se requiere un control detallado**.

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# 09. CONSULTAS.

- [1. Introducción a consultas JPA](#)
  - [1.1. Métodos de EntityManager para crear consultas](#)
  - [1.2. Métodos de Query](#)
  - [Ejemplo completo de consulta JPA](#)
- [2. Jakarta Persistence Query Language \(JPQL\)](#)
  - [2.1. Historia de JPQL](#)
  - [2.2. Sintaxis de JPQL](#)
    - [2.2.1. Consultas SELECT](#)
    - [2.2.2. Filtrado de resultados](#)
    - [2.2.3. Proyección de resultados](#)
      - [Referencias de constructores NEW](#)
      - [Resultados de consulta distintos](#)
      - [Expresiones condicionales con CASE](#)
    - [2.2.4. Joins entre entidades](#)
      - [Inner Joins \(Joins relacionados\)](#)
      - [Left Outer Joins](#)
      - [Fetch Joins](#)
    - [2.2.5. Consultas Agregadas](#)
    - [2.2.6. Parámetros en las consultas](#)
  - [3. Definición de Consultas](#)
    - [3.1. Definición Dinámica de Consultas](#)
    - [3.2. Consultas con nombre](#)

## 1. Introducción a consultas JPA

### Especificación de JPA para consultas

Las consultas JPA son consultas que se realizan sobre las entidades de la base de datos. Estas consultas se pueden realizar de varias formas:

#### a. Lenguajes de consulta:

- **Consultas JPQL: (Jakarta Persistence Query Language)**: lenguaje de consulta independiente de bases de datos, orientado a objetos que opera sobre el modelo de entidades lógico (o sobre el modelo físico de la base de datos). Las consultas JPQL se realizan **sobre las entidades de la base de datos y no sobre las tablas de la base de datos**. Ejemplo:

```
TypedQuery<Empleado> q = em.createQuery("SELECT e FROM Empleado e WHERE e.nome = :nome",
Empleado.class);
q.setParameter("nome", "Otto");
List<Empleado> resultado = q.getResultList();
```

- **Consultas nativas SQL**: consultas que se realizan sobre la base de datos, utilizando el lenguaje de consulta de la base de datos (SQL). Ejemplo:

```
Query q = em.createNativeQuery("SELECT * FROM EMPLEADO WHERE NOME = ?1", Empleado.class);
q.setParameter(1, "Otto");
List<Empleado> resultado = q.getResultList();
```

#### b. API Criteria: API que se utiliza para construir consultas de forma programática. Esta API se usa para **construir consultas basadas en objetos Java y no en cadenas/String de consulta**. Ejemplo:

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Empleado> q = cb.createQuery(Empleado.class);
Root<Empleado> c = q.from(Empleado.class);
q.select(c)
    .where(cb.equal(c.get("nome"), "Otto"));
List<Empleado> resultado = em.createQuery(q).getResultList();

```

## 1.1. Métodos de EntityManager para crear consultas

Los **métodos de EntityManager** que se utilizan para realizar consultas son:

Método	Descripción
<code>Query createQuery(String qlString)</code>	Crea una instancia de <u>Query</u> para ejecutar una consulta JPQL.
<code>&lt;T&gt; TypedQuery&lt;T&gt; createQuery(String qlString, Class&lt;T&gt; claseResultado)</code>	Crea una instancia de <u>TypedQuery</u> de JPQL. La <b>lista de elementos</b> del SELECT <b>debe tener un único elemento</b> , que debe poder asignarse al <b>tipo pasado como argumento</b> , <code>claseResultado</code> .
<code>Query createQuery (CriteriaUpdate updateQuery)</code>	Crea una instancia de <u>Query</u> para ejecutar una <b>consulta de un Criteria de actualización</b> .
<code>Query createQuery(CriteriaDelete deleteQuery)</code>	Crea una instancia de <u>Query</u> para ejecutar una consulta de <b>eliminación</b> de criterios.
<code>&lt;T&gt; TypedQuery&lt;T&gt; createQuery(CriteriaQuery&lt;T&gt; criteriaQuery)</code>	Crea una instancia de <u>TypedQuery</u> para ejecutar una <b>consulta</b> de criterios.
<code>Query createNativeQuery(String sqlString)</code>	Crea una instancia de <u>Query</u> para <u>ejecutar una declaración SQL nativa</u> , por ejemplo, para actualizar o eliminar. Si la consulta no es de borrado/actualización, devuelve un array de objetos ( <code>Object[]</code> ).
<code>Query createNativeQuery(String sqlString, Class claseResultado)</code>	Crea una instancia de <u>Query</u> para ejecutar una consulta SQL nativa, indicando el tipo de datos revuelto.
<code>Query createNativeQuery(String sqlString, String resultSetMapping)</code>	Crea una instancia de <u>Query</u> para <u>ejecutar una consulta SQL nativa</u> . Recoge el nombre del mapeo del conjunto de resultados.

También se pueden crear consultas con nombre, **NamedQuery**, que se definen en la entidad con la anotación

`@NamedQuery :`

```

@Entity
@NamedQuery(name="Empleado.findByName", query="SELECT e FROM Empleado e WHERE e.nome = :nome")
public class Empleado {

```

```
// ...
}
```

## 1.2. Métodos de Query

Los principales de la interfaz `Query` que se utilizan para ejecutar consultas son:

Método	Descripción
<code>int executeUpdate()</code>	Ejecuta la consulta de actualización o eliminación y devuelve el <b>número de entidades afectadas</b> .
<code>List getResultList()</code>	Ejecuta la consulta y devuelve una lista de resultados. Las lista de resultados <b>no es tipada</b> .
<code>default Stream getResultStream()</code>	Ejecuta la consulta y devuelve un <code>java.util.stream.Stream</code> <b>no tipado</b> de resultados. El Stream de resultados <b>no es tipada</b> . Por defecto delega al método <code>getResultList().stream()</code> , sin embargo el proveedor de persistencia, Hibernate..., puede sobrescribirlo para <b>mejorar rendimiento y funcionalidades</b> .
<code>Object getSingleResult()</code>	Ejecuta la consulta y devuelve el <b>resultado único</b> . Si la consulta devuelve <b>más de un resultado</b> , se lanza una excepción de tipo <code>NonUniqueResultException</code> .
<code>Query setFirstResult(int startPosition)</code>	Asigna la posición del primer resultado a recuperar. <i>Útil para paginación.</i>
<code>Query setMaxResults(int maxResult)</code>	Establece el número máximo de resultados a recuperar. <i>Útil para paginación.</i>
<code>int getFirstResult()</code>	Ejecuta la consulta y <u>devuelve la posición del primer resultado</u> . Devuelve 0 si no se ha aplicado el método <code>setFirstResult</code> . <i>Útil para paginación.</i>
<code>Query setParameter(int position, Object value)</code>	Asigna un valor a un parámetro de la consulta.
<code>Query setParameter(String name, Object value)</code>	Asigna un valor a un parámetro con nombre de la consulta.
<code>Query setParameter(int position, Date value, TemporalType temporalType)</code>	Asigna un valor a un parámetro de tipo temporal ( <code>java.util.Date</code> ) a la consulta.

### ! Consultas con tipo y excepciones.

- La interfaz `jakarta.persistence.TypedQuery<X>` sobrescribe los métodos `List<X> getResultList()`, `default Stream<X> getResultStream()` y `X getSingleResult()` para devolver una lista de elementos de tipo `X`, un Stream de tipo `X` y un elemento de tipo `X`, respectivamente.
- El método `getSingleResult()` lanza una excepción de tipo `NoResultException` si no se encuentran resultados.

## Ejemplo completo de consulta JPA

```

import java.util.Scanner;

import jakarta.persistence.EntityManager;
import jakarta.persistence.TypedQuery;

import java.util.List;

public class JPAQuery {

    public static Scanner SCAN = new Scanner(System.in);

    public static void main(String[] args) {

        EntityManager em;
        if (args.length != 1) {
            // em = JPAUtil.getEntityManager();
            em = Persistence.createEntityManagerFactory("unidaddepersistencia").createEntityManager();
        } else {
            // em = JPAUtil.getEntityManager(args[0]);
            em = Persistence.createEntityManagerFactory(args[0]).createEntityManager();
        }

        System.out.println("Escribe la orden \"salir;\" para salir.");
        boolean salir = false;

        while (!salir) {

            System.out.print("Jakarta Persistence QL> ");
            StringBuilder sb = new StringBuilder(SCAN.nextLine().trim());
            while (!sb.toString().endsWith(";")) {
                sb.append(" ").append(SCAN.nextLine().trim());
            }
            String consulta = sb.substring(0, sb.length() - 1);
            if (!consulta.equalsIgnoreCase("salir")) { //
                if (consulta.isEmpty()) {
                    continue;
                }
                try {
                    if ("select".equalsIgnoreCase(consulta.trim().substring(0, 6))) {
                        // Consulta JPQL. La interfaz TypedQuery hereda de Query
                        // y permite la ejecución de consultas JPQL con la devolución de
                        // resultados tipados.
                        // TypedQuery<?> q = em.createQuery(consulta, Object.class);
                        List<?> resultado = em.createQuery(consulta).getResultList(); // Las wildcard
permitten devolver cualquier tipo de objeto
                        if (!resultado.isEmpty()) {
                            int count = 0;
                            for (Object o : resultado) {
                                System.out.print(++count + " ");
                                mostrarResultados(o);
                            }
                        } else {
                            System.out.println("0 resultados de la consulta");
                        }
                    } else {
                        int i = em.createQuery(consulta).executeUpdate();
                        System.out.println(i + " elementos modificados");
                    }
                } catch (Exception e) {
                    System.out.println("Error al procesar la consulta: " + e.getMessage());
                }
            } else {
                salir = true;
            }
        }
    }
}

```

```

private static void mostrarResultados(Object resultado) {
    if (resultado == null) {
        System.out.print("NULL");
    } else if (resultado instanceof Object[]) {
        System.out.print("[");
        for (Object o : fila) {
            mostrarResultados(o);
        }
        System.out.print("]");
    } else if (resultado instanceof Long || resultado instanceof Double || resultado instanceof String) {
        System.out.print(resultado.getClass().getName() + ":" + resultado);
    } else {
        // ReflectionToStringBuilder es una clase de Apache Commons Lang que
        // permite la conversión de objetos a cadenas de texto.
        // System.out.print(ReflectionToStringBuilder.toString(resultado,
        // ToStringStyle.SHORT_PREFIX_STYLE));
        System.out.print(resultado);
    }
    System.out.println();
}
}

```

## 🔗 Bases de datos películas

- [bdpelicuasmariadb.sql](#) (52 MB)

## 🔗 Modelo de base de datos películas

- [modelopeliculas.zip](#) (7 KB)

### ☒ Ejercicio. Consultas.

Implementa el ejemplo anterior contra una base de datos de películas proporcionada.

URL de la base de datos mysql: `jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas` URL con usuario y contraseña:

`jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas?user=accesoadatos&password=abc123..&useSSL=false`

Pese a todo lo mejor es que el usuario y contraseña se guarden en un archivo de propiedades, `persistence.xml`, como se ha visto en el tema de configuración.

Realiza las siguientes consultas:

Las películas que no tienen año de inicio definido:

```

SELECT p.castelan, p.anoFin, p.anoInicio
FROM Pelicula p
WHERE p.anoInicio IS NOT NULL;

```

Las películas con una duración superior a 120 minutos:

```

SELECT p.castelan, p.anoFin, p.duracion
FROM Pelicula p
WHERE p.duracion > 120;

```

Las películas con Antonio Banderas:

```

SELECT p FROM Pelicula p JOIN p.personaxes pp JOIN pp.personaxe per WHERE per.nomeOrdenado LIKE
'Antonio Banderas';

```

Las tablas de la base de datos sobre las que hay que implantar las entidades y realizar las consultas son las del ejercicio anterior.



[Estructura de la base de datos](#)



[Personaxeocupación](#)

## 2. Jakarta Persistence Query Language (JPQL)

### 2.1. Historia de JPQL

El **origen** de JPQL es **Enterprise JavaBeans Query Language (EJB QL)**, que se introdujo en la especificación EJB 2.0 para permitir a los desarrolladores escribir métodos portables de búsqueda y selección para beans de entidad gestionados por contenedores. Se basaba en un pequeño subconjunto de SQL que introdujo una forma de **navegar a través de las relaciones de entidad** tanto para seleccionar datos como para filtrar los resultados. Sin embargo, imponía **limitaciones estrictas en la estructura de la consulta**, limitando los resultados a una única entidad o a un campo persistente de una entidad. Aunque eran posibles las uniones internas entre entidades, se utilizaba una notación extraña. La versión inicial ni siquiera admitía la ordenación.

La **especificación EJB 2.1 ajustó EJB QL**, añadiendo soporte para la **ordenación** e introduciendo funciones agregadas básicas; pero nuevamente, la limitación de un **único tipo de resultado** obstaculizaba el uso de agregados. Se podía filtrar los datos, pero no había equivalente a las expresiones GROUP BY y HAVING de SQL.

**Jakarta Persistence QL extiende significativamente EJB QL**, eliminando muchas debilidades de las versiones anteriores mientras preserva la compatibilidad hacia atrás. Algunas de las características disponibles añadidas:

- Tipos de **resultados** de valor **único y múltiple**: las consultas JPQL pueden devolver una única entidad, un campo persistente de una entidad, una lista de entidades o una lista de campos persistentes.
- **Funciones agregadas**, con cláusulas de **ordenación y agrupación**: `GROUP BY` y `HAVING`.
- Una sintaxis de unión más natural, incluyendo soporte para **inner joins y outer joins**: `LEFT JOIN` y `RIGHT JOIN`.
- Expresiones condicionales que involucran **subconsultas**: `EXISTS`, `ALL`, `ANY` y `SOME`.
- Consultas de **actualización y eliminación para cambios masivos de datos**: `UPDATE` y `DELETE`.
- Proyección de **resultados en clases no persistentes**: `SELECT NEW`.

### 2.2. Sintaxis de JPQL

La sintaxis de JPQL es similar a la de SQL, pero **opera sobre entidades y sus atributos** en lugar de tablas y columnas.

Las consultas JPQL se definen como cadenas de texto y se pueden **incrustar en el código Java**.

Las consultas JPQL se pueden **ejecutar de forma dinámica** en tiempo de ejecución, lo que permite a las aplicaciones **adaptar las consultas a las condiciones cambiantes**.

#### 2.2.1. Consultas SELECT

Las consultas SELECT de JPQL se utilizan para **recuperar datos de la base de datos**. La sintaxis básica de una consulta SELECT es:

```
SELECT [DISTINCT] select_expression
FROM identification_variable_declaracion
[WHERE conditional_expression]
[GROUP BY grouping_expression]
[HAVING conditional_expression]
[ORDER BY ordering_expression [ASC | DESC]]
```

Una declaración SELECT es una cadena que consta de las siguientes cláusulas:

- Una **cláusula** `SELECT`, que determina el tipo de objetos o valores que se seleccionarán.
- Una **cláusula** `FROM`, que proporciona declaraciones que designan el dominio al cual se aplican las expresiones especificadas en las otras cláusulas de la consulta.
- Una **cláusula** `WHERE` **opcional**, que se puede utilizar para restringir los resultados que devuelve la consulta.
- Una **cláusula** `GROUP BY` **opcional**, que permite agregar los resultados de la consulta en términos de grupos.
- Una **cláusula** `HAVING` **opcional**, que permite filtrar sobre grupos agregados.
- Una **cláusula** `ORDER BY` **opcional**, que se puede utilizar para ordenar los resultados que devuelve la consulta.

En la sintaxis de BNF, una declaración SELECT se define como:

```
select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause]
[orderby_clause]
```

Una declaración `SELECT` **siempre debe tener una cláusula** `SELECT` **y una cláusula** `FROM`. Los corchetes **cuadrados []** **indican que las otras cláusulas son opcionales**.

La consulta más simple en Jakarta Persistence QL selecciona todas las instancias de un solo tipo de entidad:

```
SELECT e
FROM Empleado e
```

Jakarta Persistence QL **utiliza la sintaxis de SQL**.

La **principal diferencia entre SQL y Jakarta Persistence QL para esta consulta es que, en lugar de seleccionar de una tabla, se ha especificado una entidad del modelo de dominio de la aplicación**. La cláusula `SELECT` de la consulta también es ligeramente diferente, **enumerando solo el alias de** `Empleado e`. Esto indica que el tipo de resultado de la consulta es la entidad `Empleado`, por lo que ejecutar esta instrucción dará como resultado una lista de cero o más instancias de `Empleado`.

Con un **alias se puede navegar a través de las relaciones de entidad utilizando el operador punto (.)**:

```
SELECT e.nombre
FROM Empleado e
```

El campo persistente (`e.nombre`, en este caso) de la entidad puede ser de tipo simple o incurable, o a una asociación que conduce a otra entidad o colección de entidades. Dado que la entidad `Empleado` tiene un campo persistente llamado `nombre` de tipo String, esta consulta **dará como resultado una lista de cero o más objetos String**.

También se puede seleccionar una entidad que ni siquiera mencionamos en la cláusula `FROM`. Consideremos el siguiente ejemplo:

```
SELECT e.departamento
FROM Empleado e
```

Un `Empleado` **tiene una relación de muchos a uno con su** `Departamento` llamada `departamento`, por lo que el tipo de **resultado de la consulta es la entidad** `Departamento`.

## 2.2.2. Filtrado de resultados

Para filtrar resultados **se utiliza la cláusula WHERE**. La mayoría de operaciones disponibles en SQL están disponibles en JPQL, como:

- Operadores básicos de comparación: `=, >, <, >=, <=, <>`.
- Expresiones: `BETWEEN, LIKE, IN, IS NULL, IS NOT NULL`.
- Funciones de cadena: `CONCAT, SUBSTRING, TRIM, LOWER, UPPER, LENGTH, LOCATE`.
- Funciones aritméticas: `ABS, CEILING, EXP, FLOOR, LN, MOD, POWER, ROUND, SIGN, SQRT, SIZE, INDEX`.
- Funciones de fecha: `CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, EXTRACT, ...`.
- Funciones de agregado: `AVG, COUNT, MAX, MIN, SUM`.

Por ejemplo:

```
SELECT e
FROM Empleado e
WHERE e.salario > 1000
```

En este caso, la cláusula **WHERE** se utiliza para **filtrar los resultados de la consulta**. La condición `e.salario > 1000` se evalúa para cada instancia de `Empleado` en la base de datos, y solo las instancias que satisfacen la condición se incluyen en el resultado de la consulta.

```
SELECT e
FROM Empleado e
WHERE e.departamento.nombre = 'Ventas'
AND e.direccion.ciudad = 'Santiago'
```

## 2.2.3. Proyección de resultados

En aquellos casos en los que se encuentre interesado en **recuperar solo algunos campos de una entidad**, se puede utilizar la cláusula **SELECT** para **proyectar los resultados**. Por ejemplo:

```
SELECT e.nombre, e.salario
FROM Empleado e
WHERE e.salario > 1000
```

Dependiendo de cómo una entidad está mapeada en la base de datos, puede ser **más eficiente recuperar solo algunos campos de una entidad que recuperar la entidad completa**. En este caso, la consulta **devuelve una lista de objetos Object[]**, donde cada objeto es un array de dos elementos que contiene el nombre y el salario de un empleado.

### Referencias de constructores NEW

Las referencias de constructores son una buena opción **para casos de uso de solo lectura**. Son más cómodos de usar que las proyecciones de valores escalares y **evitan los gastos generales de las entidades administradas**.

**JPQL le permite definir una llamada al constructor en la cláusula SELECT**. Sólo se necesita **proporcionar el nombre de clase completo y especificar los parámetros del constructor de un constructor existente**. De manera similar a la proyección de entidad, genera una consulta SQL que **devuelve las columnas de la base de datos requeridas y utiliza la referencia del constructor para crear una instancia de un nuevo objeto** para cada registro en el conjunto de resultados.

```
SELECT new com.pepинho.ad.jpa.AutorDTO(a.idAutor, a.nome, a.apelidos) FROM Autor a
```

### Resultados de consulta distintos

El operador DISTINCT de SQL que elimina duplicados de una proyección también lo admite JPQL:

```
SELECT DISTINCT e.departamento
FROM Empleado e
```

## Expresiones condicionales con CASE

Las expresiones condicionales se pueden utilizar en la cláusula WHERE para **filtrar los resultados de la consulta**. Por ejemplo:

```
SELECT e
FROM Empleado e
WHERE e.salario > 1000 AND e.salario < 2000
```

Sin embargo, JPQL admite expresiones CASE: case general, case simple y case de búsqueda.

```
SELECT e.nombre,
CASE WHEN e.salario > 2000 THEN 'Alto' ELSE 'Bajo' END
FROM Empleado e
```

O también para valores concretos:

```
SELECT e.nombre,
CASE e.salario
WHEN 1000 THEN 'Bajo'
WHEN 2000 THEN 'Medio'
ELSE 'Alto'
END
FROM Empleado e
```

Ejemplos:

```
UPDATE Empleado e
SET e.salario =
CASE WHEN e.clasificacion = 1 THEN e.salario * 1.1
      WHEN e.clasificacion = 2 THEN e.salario * 1.05
      ELSE e.salario * 1.01
END

UPDATE Empleado e
SET e.salario =
CASE e.clasificacion WHEN 1 THEN e.salario * 1.1
                      WHEN 2 THEN e.salario * 1.05
                      ELSE e.salario * 1.01
END

SELECT e.nombre,
CASE TYPE(e) WHEN Desarrollador THEN 'Desarrollador'
              WHEN Administrador THEN 'Administrador'
              WHEN Profesor THEN 'Profesor'
              ELSE 'Empleado'
END
FROM Empleado e
WHERE e.departamento.nombre = 'Sistemas'

SELECT e.nombre,
f.nombre,
CONCAT(CASE WHEN f.kmAnuales > 50000 THEN 'Platinum '
            WHEN f.kmAnuales > 25000 THEN 'Dorada '
            WHEN f.kmAnuales > 10000 THEN 'Plateada '
            ELSE ''
END,
' Frecuencia')
FROM Empleado e JOIN e.planDeViaje f
```

### 2.2.4. Joins entre entidades

El tipo de resultado de una consulta SELECT no puede ser una colección; debe ser un objeto de valor único, como una instancia de entidad o un tipo de campo persistente.

Expresiones como `e.telefonos` son ilegales en la cláusula SELECT porque darían como resultado instancias de Collection (cada ocurrencia de `e.telefonos` es una colección, no una instancia). Por lo tanto, al igual que con SQL y tablas, si queremos navegar a lo largo de una asociación de colección y devolver elementos de esa colección, debemos hacer un join las dos entidades.

Por ejemplo:

```
SELECT p.numero
FROM Empleado e, Telefono t
WHERE e = t.empleado AND
    e.departamento.nombre = 'Desarrollo' AND
    t.tipo = 'Móvil'
```

También se pueden expresar en la cláusula FROM utilizando el operador JOIN. La ventaja de este operador es que el join se puede expresar en términos de la propia asociación, y el motor de consulta suministrará automáticamente los criterios de unión necesarios cuando genere el SQL. La consulta anterior se puede reescribir para usar el operador JOIN. Recuerda que el alias `p` es de tipo `Telefono`:

```
SELECT p.numero
FROM Empleado e JOIN e.telefonos p
WHERE e.departamento.nombre = 'Desarrollo' AND
    p.tipo = 'Móvil'
```

Jakarta Persistence QL admite varios tipos de uniones, incluidas uniones internas y externas, así como una técnica llamada `fetch joins` para cargar de manera proactiva datos asociados con el tipo de resultado de una consulta pero que no se devuelven directamente.

## Inner Joins (Joins relacionados)

Un Inner Join devuelve solo las filas que tienen correspondencias en ambas tablas. En Jakarta Persistence QL, un join interno se expresa con la palabra clave [INNER] JOIN:

```
[INNER] JOIN join_association_path_expression [AS] identification_variable [join_condition]
```

Por ejemplo:

```
SELECT a, p FROM Autor a JOIN a.libros p
```

La definición de la entidad Autor proporciona toda la información que se necesita para unirla a la entidad Libro, y no es necesario proporcionar una declaración ON adicional.

La palabra INNER es opcional:

```
SELECT c FROM Cliente c INNER JOIN c.pedidos p WHERE c.estado = 1
```

Es equivalente a la consulta con el constructor IN:

```
SELECT OBJECT(c) FROM Cliente c, IN(c.pedidos) p WHERE c.estado = 1
```

La siguiente consulta se une a `Empleado`, `Información de contacto` y `Teléfono`. `ContactoInfo` es una clase embedded que consta de una dirección y un conjunto de teléfonos. El teléfono es una entidad.

```
SELECT t.compañia
FROM Empleado e JOIN e.contactoInfo c JOIN c.telefonos t
WHERE c.direccion.codigoPostal = '15705'
```

Se puede especificar una condición de unión para una unión interna. **Esto equivale a la especificación de la misma condición en la cláusula WHERE.**

## Left Outer Joins

**LEFT JOIN** y **LEFT OUTER JOIN** son sinónimos.

A veces sólo quieras **unirte a las entidades relacionadas que cumplen condiciones adicionales.**

Este Join permite recibir un conjunto de entidades cuyos **valores asociados a la condición de unión pueden ser nulos.**

Sintaxis:

```
LEFT [OUTER] JOIN join_association_path_expression [AS] identification_variable [join_condition]
```

Si no se especifica la condición de unión, **el join se realiza en función de la relación de asociación.**

```
SELECT s.nombre, COUNT(p)
FROM Proveedor s LEFT JOIN s.productos p
GROUP BY s.nombre
```

Equivalente a SQL:

```
SELECT s.nombre, COUNT(p.id)
FROM Proveedor s LEFT JOIN Producto p
    ON s.idProveedor = p.idProducto
GROUP By s.nombre
```

Puede añadirse otra condición a la unión explícita:

```
SELECT s.nombre, COUNT(p)
FROM Proveedor s LEFT JOIN s.productos p
    ON p.estado = 'stock'
GROUP BY s.nombre
```

Equivalente a SQL:

```
SELECT s.nombre, COUNT(p.id)
FROM Proveedor s LEFT JOIN Producto p
    ON s.idProveedor = p.idProducto
        AND p.estado = 'stock'
GROUP By s.nombre
```

## Fetch Joins

### Ejemplo de fetch join

Un **FETCH JOIN** permite la obtención de una asociación o colección de elementos como un **efecto secundario de la ejecución de una consulta.**

La sintaxis para un fetch join es

```
fetch_join ::= [LEFT [OUTER] | INNER] JOIN FETCH join_association_path_expression
```

La asociación referenciada por el **lado derecho de la cláusula `FETCH JOIN` debe ser una asociación o colección de elementos** que se referencia desde una entidad o integrable que se devuelve como resultado de la consulta.

No se permite especificar una variable de identificación para los objetos referenciados por el lado derecho de la cláusula `FETCH JOIN` y, por lo tanto, las referencias a las entidades o elementos obtenidos de manera implícita no pueden aparecer en ninguna otra parte de la consulta.

La siguiente consulta **devuelve un conjunto de departamentos**. Como efecto secundario, también se recuperan los empleados asociados a esos departamentos, aunque no formen parte del resultado explícito de la consulta. **La inicialización del estado persistente o de los campos o propiedades de relación de los objetos que se recuperan como resultado de un `fetch join`** está determinada por los metadatos de esa clase, en este ejemplo, la clase de entidad `Empleado`.

```
SELECT d
FROM Departamento d LEFT JOIN FETCH d.empleados
WHERE d.numeroDepartamento = 1
```

Un **fetch join tiene las mismas semánticas de unión que la unión interna u externa correspondiente**, excepto que los objetos relacionados especificados en el lado derecho de la operación de unión no se devuelven en el resultado de la consulta ni se hacen referencia de ninguna otra manera en la consulta. Por lo tanto, por ejemplo, si el departamento 1 tiene cinco empleados, la consulta anterior devuelve cinco referencias a la entidad del departamento 1.

## 2.2.5. Consultas Agregadas

La sintaxis para consultas agregadas en Jakarta Persistence QL es **muy similar a la de SQL**. Hay cinco funciones agregadas admitidas:

- AVG.
- COUNT.
- MIN.
- MAX.
- SUM.

Los resultados **pueden agruparse en la cláusula `GROUP BY` y filtrarse mediante la cláusula `HAVING`**.

Nuevamente, la diferencia está en el uso de expresiones de entidad al especificar los datos que se van a agregar. Por ejemplo:

```
SELECT d, COUNT(e), MAX(e.salario), AVG(e.salario)
FROM Departamento d JOIN d.empleados e
GROUP BY d
HAVING COUNT(e) >= 5
```

## 2.2.6. Parámetros en las consultas

Jakarta Persistence QL admite **dos tipos de sintaxis para la vinculación de parámetros**:

- Vinculación **posicional**, donde los parámetros se indican en la cadena de consulta mediante un **signo de interrogación seguido del número de parámetro** (similar a JDBC):

```
SELECT e
FROM Empleado e
WHERE e.departamento = ?1 AND
e.salario > ?2
```

- **Parámetros con nombre**, que se indican en la **cadena de consulta mediante dos puntos seguidos del nombre del parámetro**:

```
SELECT e
FROM Empleado e
WHERE e.departamento = :dept AND
e.salario > :base
```

## Ejercicio. Consultas sobre la base de datos de películas

Amplía el ejercicio anterior la base de datos de películas proporcionada.

URL de la base de datos mysql: `jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas` URL con usuario y contraseña:

`jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas?user=accesoadatos&password=abc123..&useSSL=false`

Pese a todo lo mejor es que el usuario y contraseña se guarden en un archivo de propiedades,

`persistence.xml`, como se ha visto en el tema de configuración.

Las tablas de la base de datos sobre las que hay que implantar las entidades y realizar las consultas son las del ejercicio anterior.



[Estructura de la base de datos](#)



[Personaxeocupación](#)

Realiza las siguientes consultas:

1. Muestra la película solicitando el id:

```
SELECT castelan, orixinal, anoFin, poster IS NOT NULL as tenPoster
FROM pelicula WHERE idPelicula = :identificador
```

2. Muestra las películas que tienen algún personaje (IS EMPTY) o no tienen personajes (IS NOT EMPTY).

3. Muestra las películas que tienen personajes con una ocupación concreta:

```
SELECT P.nome FROM peliculapersonaxe PP, personaxe P
WHERE P.idPersonaxe=PP.idPersonaxe AND
PP.ocupacion='OCUPACIÓNCONCRETA' AND PP.idPelicula=IDENTIFICADOR_PELICULA
```

5. Muestra los títulos de las películas en las que ha trabajado un actor concreto.

6. Listar el número de películas de acuerdo con el nombre proporcionado: (Crea una clase PeliculaDTO con los campos idPelicula, castelan, orixinal, anoFin, tenPoster (booleano) y realiza la consulta)

```
SELECT idPelicula, castelan, orixinal, anoFin, poster IS NOT NULL as tenPoster
FROM pelicula WHERE castelan LIKE '%:nombre%' ORDER BY 5 DESC, castelan ASC
```

7. Consulta los datos de las ocupaciones de los personajes de una película:

```
SELECT O.ocupacion FROM ocupacion O WHERE EXISTS (
SELECT idPelicula FROM peliculapersonaxe PP WHERE
O.ocupacion=PP.ocupacion
AND PP.idPelicula=IDENTIFICADOR_DE_PELICULA)
AND O.orde<>0 ORDER BY O.orde
```

y los nombres sde los personajes que tienen esa ocupación:

```
SELECT P.nombre FROM peliculapersonaxe PP, personaxe P
WHERE P.idPersonaxe=PP.idPersonaxe AND
PP.ocupacion='OCUPACIÓNCONCRETA' AND PP.idPelicula=IDENTIFICADOR_PELICULA
```

## 3. Definición de Consultas

Jakarta Persistence proporciona las interfaces `Query` y `TypedQuery` para configurar y ejecutar consultas.

- La `interfaz Query` se utiliza en casos en los que el tipo de resultado es `Object` o en consultas dinámicas cuando el tipo de resultado puede no ser conocido de antemano.
- La `interfaz TypedQuery` es la preferida y se puede utilizar siempre que se conozca el tipo de resultado.

`TypedQuery` hereda de `Query`, por lo que una consulta fuertemente tipada siempre se puede tratar como una versión no tipada, aunque no al revés.

Una implementación de la interfaz adecuada para una consulta dada se obtiene a través de uno de los métodos Factory ("`createQuery`, `createNamedQuery`, `createNativeQuery`, `createStoredProcedureQuery`, `getCriteriaBuilder().createQuery()`) en la interfaz `EntityManager` (pueden verse en la tabla anterior). La elección del método Factory depende del tipo de consulta (Jakarta Persistence QL, SQL u objeto de criterios), si la consulta ha sido predefinida y si se desean resultados fuertemente tipados (en la tabla anterior pueden verse ejemplos).

Hay tres enfoques para definir una consulta Jakarta Persistence QL:

- Una consulta puede ser **creada dinámicamente en tiempo de ejecución**: Las **consultas dinámicas de Jakarta Persistence QL más que cadenas** y, por lo tanto, pueden **definirse sobre la marcha según sea necesario**.

```
Query q = em.createQuery("SELECT e FROM Empleado e WHERE e.departamento = :dept AND e.salario > :base");
q.setParameter("dept", "Desarrollo");
q.setParameter("base", 1000);
List<Empleado> resultado = q.getResultList()
```

- **Configurada en los metadatos de la unidad de persistencia** (por medio de una anotación o XML) y posteriormente ser **referenciada por nombre**: Las **consultas con nombre son estáticas e inalterables**, pero son **más eficientes de ejecutar** porque el proveedor de persistencia puede traducir la cadena de Jakarta Persistence QL a SQL una vez cuando la aplicación comienza, en lugar de cada vez que se ejecuta la consulta.

```
@NamedQuery(name="findByDeptAndSalario", query="SELECT e FROM Empleado e WHERE e.departamento = :dept
AND e.salario > :base")
```

```
TypedQuery<Empleado> q = em.createNamedQuery("Empleado.findByDeptAndSalario", Empleado.class);
q.setParameter("dept", "Desarrollo");
q.setParameter("base", 1000);
List<Empleado> resultado = q.getResultList();
```

- Especificada **dinámicamente y** guardada para ser referenciada más adelante **por nombre**. Definir dinámicamente una consulta y luego darle un nombre permite que una consulta dinámica se reutilice varias veces a lo largo de la vida de la aplicación, pero **incurre en el costo de procesamiento dinámico solo una vez**.

### 3.1. Definición Dinámica de Consultas

Una consulta puede definirse **dinámicamente pasando la cadena de consulta Jakarta Persistence QL** y el tipo de resultado esperado al método `createQuery()` de la interfaz `EntityManager` (el tipo de resultado puede

omitirse para crear una consulta no tipada).

Se admiten **todos los tipos de consultas Jakarta Persistence QL y uso de parámetros**. La capacidad de construir una cadena en tiempo de ejecución y usarla para definir una consulta es útil, especialmente para aplicaciones donde el usuario puede especificar criterios complejos y la forma exacta de la consulta no puede conocerse de antemano.

Jakarta Persistence también admite una API Criteria para crear consultas dinámicas mediante objetos Java.

Un problema a considerar con las consultas de cadena dinámicas es el **costo de traducir la cadena de Jakarta Persistence QL a SQL para su ejecución**. Un motor de consulta típico tendrá que analizar la cadena de Jakarta Persistence QL en un árbol de sintaxis, obtener los metadatos de asignación objeto-relacional para cada entidad en cada expresión y luego generar el SQL equivalente. **Para aplicaciones que emiten muchas consultas, el costo de rendimiento del procesamiento dinámico de consultas puede convertirse en un problema**.

Muchos motores de consultas almacenarán en caché el SQL traducido para su uso posterior, pero esto se puede vencer fácilmente si la aplicación no utiliza la vinculación de parámetros y concatena los valores de parámetros directamente en las cadenas de consulta. Esto tiene el efecto de generar una consulta nueva y única cada vez que se construye una consulta que requiere parámetros.

Ejemplo, que busca información salarial dado el nombre de un departamento y el nombre de un empleado, **nada recomendable**:

```
public class ServicioConsulta {
    @PersistenceContext(unitName="ConsultasDinamicas") /* Se usa cuando se inyecta el EntityManager de la unidad de persistencia "UnidadConsultas" y es gestionado por el contenedor (no para aplicaciones Java SE). */
    EntityManager em;

    public long consultarSalarioEmpleado(String nombreDepartamento, String nombreEmpleado) {
        String consulta = "SELECT e.salario " +
            "FROM Empleado e " +
            "WHERE e.departamento.nombre = '" + nombreDepartamento +
            "' AND " +
            "e.nombre = '" + nombreEmpleado + "'";
        return em.createQuery(consulta, Long.class).getSingleResult();
    }
}
```

. Hay dos problemas: uno relacionado con el **rendimiento** y otro relacionado con la **seguridad**:

- Como los **nombres se concatenan en la cadena** (en lugar de usar la vinculación de parámetros), efectivamente **se crea una consulta nueva y única cada vez**. Cien llamadas a este método podrían generar potencialmente cien cadenas de consulta diferentes.
- En este ejemplo es que es **vulnerable a ataques de inyección SQL**, donde un usuario malintencionado podría pasar un valor que altera la consulta (por ejemplo, el gerente del departamento está consultando los salarios de sus empleados). Si el nombre fuera el texto "*CALQUERA' OR 'Pepe' = 'Pepe*", la consulta real analizada por el motor de consultas sería la siguiente:

```
SELECT e.salario
FROM Empleado e
WHERE e.departamento.nombre = 'Desarrollo' AND
      e.nombre = 'CALQUERA' OR
      'Pepe' = 'Pepe'
```

Al introducir la condición OR, el usuario se ha dado acceso efectivo al valor salarial de cualquier empleado, porque la condición AND original tiene una precedencia más alta que OR.

El uso de **parámetros con nombre reduce la cantidad de consultas únicas analizadas por el motor de consultas y elimina la posibilidad de inyección SQL**:

```
public class ServicioConsulta {
```

```

private static final String CONSULTA =
    "SELECT e.salario " +
    "FROM Empleado e " +
    "WHERE e.departamento.nombre = :nombreDepartamento AND " +
    " e.nombre = :nombreEmpleado ";

@PersistenceContext(unitName="UnidadConsultas") // Se usa cuando se inyecta el EntityManager de la
  unidad de persistencia "UnidadConsultas" y es gestionado por el contenedor (no para aplicaciones Java
  SE).
EntityManager em; // Cuando se inyecta el EntityManager, se inyecta el EntityManager de la unidad de
                  persistencia "UnidadConsultas".

public long consultarSalarioEmpleado(String nombreDepartamento, String nombreEmpleado) {
    return em.createQuery(CONSULTA, Long.class)
        .setParameter("nombreDepartamento", nombreDepartamento)
        .setParameter("nombreEmpleado", nombreEmpleado)
        .getSingleResult();
}
}

```

Los **parámetros se empaquetan utilizando la API de JDBC y son manejados directamente por la base de datos**. El texto de una cadena de parámetros se cita efectivamente por la base de datos, por lo que el ataque malicioso realmente produciría la siguiente consulta:

```

SELECT e.salario
FROM Empleado e
WHERE e.departamento.nombre = 'Desarrollo' AND
      e.nombre = 'CALQUERA' OR
      ''Pepe'' = ''Pepe'

```

Las **comillas simples se escapan añadiéndoles una comilla simple adicional**. Esto elimina cualquier significado especial de ellas y toda la secuencia se trata como un único valor de cadena.

#### Recomendación

Se recomienda el uso de consultas con nombre definidas estéticamente, especialmente para consultas que se ejecutan con frecuencia (siguiente sección). Si las consultas dinámicas son necesarias debe usarse la vinculación de parámetros en lugar de concatenar valores de parámetros en cadenas de consulta para minimizar la cantidad de cadenas de consulta distintas analizadas por el motor de consultas.

## 3.2. Consultas con nombre

Las consultas nombradas **permiten organizar las definiciones de consultas y mejorar el rendimiento de la aplicación**.

Una consulta nombrada se define utilizando la anotación `@NamedQuery`, que puede colocarse en la definición de clase para cualquier entidad. La anotación define el nombre de la consulta, así como el texto de la consulta:

Declaración de una consulta nombrada:

```

@NamedQuery(name="encontrarSalarioPorNombreYDepartamento",
query="SELECT e.salario FROM Empleado e " +
"WHERE e.departamento.nombre = :nombreDepartamento AND " +
" e.nombre = :nombreEmpleado")

```

Las consultas nombradas **se colocan normalmente en la clase de entidad que más corresponde directamente al resultado de la consulta**, por lo que la entidad Empleado sería un buen lugar para esta consulta nombrada.

La anotación `@NamedQuery` **puede aparecer varias veces en una clase de entidad**. Esto es útil si la entidad tiene varias consultas que se utilizan con frecuencia. Las consultas nombradas **se pueden referenciar por nombre en cualquier lugar donde se pueda usar una consulta dinámica**. Tienen dos elementos obligatorios:

- `name`: `String` con el nombre de la consulta.
- `query`: `String` con el texto de la consulta.

Uso de una consulta nombrada:

```
public class ServicioConsulta {  
    @PersistenceContext(unitName="UnidadConsultas") // Se usa cuando se inyecta el EntityManager de la  
    unidad de persistencia "UnidadConsultas" y es gestionado por el contenedor (no para aplicaciones Java  
    SE).  
    EntityManager em;  
  
    public long consultarSalarioEmpleado(String nombreDepartamento, String nombreEmpleado) {  
        return em.createNamedQuery("encontrarSalarioPorNombreYDepartamento", Long.class)  
            .setParameter("nombreDepartamento", nombreDepartamento)  
            .setParameter("nombreEmpleado", nombreEmpleado)  
            .getSingleResult();  
    }  
}
```

<https://thorben-janssen.com/jpa/>

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# EJERCICIOS DE LA UNIDAD DE JPA

---

- [Ejercicios de apuntes](#)
  - [Ejercicio 01.01. Creación de un proyecto con JPA](#)
  - [Ejercicio 01.02. Creación de un archivo de configuración de persistencia](#)
  - [Ejercicio 01.03. Creación de una entidad](#)
  - [Ejercicio 01.04. Creación de una entidad](#)
  - [Ejercicio 03.01. Creación de una aplicación de persistencia de una biblioteca](#)
    - [03.01. Solución](#)
  - [Ejercicio 04.01. Descarga y creación de la base de datos de JokeAPI](#)
    - [Enumeraciones](#)
    - [Clases](#)
    - [Ejercicio](#)
  - [Ejercicio 05.01. Acceso combinado a la entidad Chiste.](#)
  - [Ejercicio 05.02. CLOB y BLOB de una entidad Documento](#)
  - [Ejercicio 05.03. Conversores personalizados y enumeraciones](#)
  - [Ejercicio 05.04. Generación de ids con tabla](#)
  - [Ejercicio 05.05. Generación de ids con una secuencia](#)
  - [Ejercicio 05.06. Ampliación de la aplicación de persistencia de una biblioteca](#)
  - [Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador](#)
  - [Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo](#)
  - [Ejercicio 06.03. Relación muchos a muchos unidireccional Jugador-Posición](#)
  - [Ejercicio 06.04. Mapeo de una base de datos de juegos](#)
  - [Ejercicio 06.05. Claves compartidas en relaciones uno a uno](#)
  - [Ejercicio 07.01. Elementos embebidos.](#)
  - [Ejercicio 07.02. Clave compuesta en una relación de muchos a muchos.](#)
  - [Ejercicio 07.03. Entidades principales de base de datos de películas.](#)

## Ejercicios de apuntes

### Ejercicio 01.01. Creación de un proyecto con JPA

Para crear un proyecto con JPA y Hibernate, se puede utilizar el asistente de creación de proyectos de Eclipse o IntelliJ IDEA; sin embargo, con la versión Community de IntelliJ IDEA no se puede crear un proyecto con JPA a través del asistente. **Crea un proyecto Java Maven y añade las dependencias de Hibernate y la API de Jakarta Persistence.**

### Ejercicio 01.02. Creación de un archivo de configuración de persistencia

Crea un directorio `META-INF` en el directorio `src/main/resources` y añade un archivo `persistence.xml` con la configuración de la unidad de persistencia con el nombre `com.sanclemente.ad.jpa.exemplo`.

El fichero de configuración `persistence.xml` **debe apuntar a una base de datos H2 en memoria**. Además, debes añadir los Drivers de H2 para que la aplicación pueda conectarse a la base de datos:

```
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.3.232</version>
</dependency>
```

Ten en cuenta que precisas crear la base de datos en memoria H2 y añadir las tablas necesarias, por lo que el parámetro `jakarta.persistence.schema-generation.database.action` debe ser "create".

## Ejercicio 01.03. Creación de una entidad

Crea una entidad Estudiante con `idEstudiante` (`Long`), `nombre`, `apellidos`, `fechaDeNacimiento` y `dirección`. Añade los atributos necesarios y las anotaciones para que sea una entidad. La clave primaria será `idEstudiante` de tipo autoincremental.

## Ejercicio 01.04. Creación de una entidad

Crea una clase AppEstudiante que se conecte a la base de datos y añada un estudiante a la tabla de la base de datos.

Aunque lo veremos más adelante, lo que precisamos es crear un gestor de entidades e invocar al método `persist` para añadir un estudiante a la base de datos:

```
public class AppEstudiante {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("com.sanclemente.ad.jpa.exemplo");
        EntityManager em = emf.createEntityManager();

        Estudiante estudiante = new Estudiante("Juan", "Pérez", LocalDate.of(2000, 1, 1), "Calle Mayor,
1");

        em.getTransaction().begin();
        em.persist(estudiante);
        em.getTransaction().commit();

        // Imprime el estudiante para ver si se ha añadido correctamente y tiene un id
        em.close();
        emf.close();
    }
}
```

Para recuperarlo precisamos invocar al método `find` del gestor de entidades:

```
Estudiante estudiante = em.find(Estudiante.class, 1L); // Recupera el estudiante con id 1
```

## Ejercicio 03.01. Creación de una aplicación de persistencia de una biblioteca

Queremos desarrollar una **aplicación para una biblioteca** y necesitamos interactuar con una base de datos que contiene información sobre los libros que tenemos en nuestra colección.

Para ello, vamos a crear una clase `Book` que **represente la entidad libro**, la clase `Contido` y otra clase `BookDAO` que nos permita realizar **operaciones básicas CRUD (Create, Read, Update y Delete)** sobre la tabla `Book` en la base de datos.

Además, precisamos **una clase `BibliotecaJpaManager`** para la gestión y obtención de los objetos de tipo `EntityManagerFactory` de una manera eficiente. Emplearemos el **patrón Singleton para el gestor `BibliotecaJpaManager`**, que tenga un único objeto de tipo `EntityManagerFactory` y que nos permita obtener un objeto de tipo `EntityManager` para realizar las operaciones sobre la base de datos (**queremos que el objeto de tipo `EntityManagerFactory` sea único para cada unidad de persistencia**, para cada unidad de persistencia, no así el `EntityManager`, que podrá hacer varios para cada unidad de persistencia).

A) **BASE DE DATOS** (es la misma base de datos que hemos empleado en la unidad de bases de datos con JDBC):

Está formada por una **tabla Book** y una **tabla Contido**. La tabla **Book** tiene una estructura SIMILAR a la siguiente:

Columna	Tipo de dato	Descripción
idBook	int	Identificador único del ejemplar del libro
isbn	varchar(13)	Identificador del libro
titulo	varchar(100)	Título del libro
autor	varchar(100)	Autor del libro
anho	int	Año de publicación del libro
disponible	boolean	Indica si el libro está disponible
portada	Blob	Portada del libro en formato binario
dataPublicacion	Date	Fecha de publicación del libro

```
-- PUBLIC.Book definition
-- Drop table
-- DROP TABLE PUBLIC.Book;
CREATE TABLE PUBLIC.Book (
    idBook INTEGER NOT NULL AUTO_INCREMENT,
    isbn CHARACTER VARYING(13) NOT NULL,
    titulo CHARACTER VARYING(255) NOT NULL,
    autor CHARACTER VARYING(255),
    anho INTEGER,
    disponible BOOLEAN DEFAULT TRUE,
    portada BINARY LARGE OBJECT,
    dataPublicacion DATE,
    CONSTRAINT BOOK_PK PRIMARY KEY (idBook)
);
CREATE UNIQUE INDEX IdBookPK ON PUBLIC.Book (idBook);
CREATE INDEX IdxBookISBN ON PUBLIC.Book (isbn);
CREATE INDEX IdxBookTitle ON PUBLIC.Book (titulo);
CREATE UNIQUE INDEX PRIMARY_KEY_93 ON PUBLIC.Book (idBook);
```

La tabla **Contido** tiene una estructura SIMILAR a la siguiente:

Columna	Tipo de dato	Descripción
idContido	int	Identificador único del contenido del libro
idBook	int	Identificador del libro
contido	Blob	Contenido del libro en formato binario

\* **idBook** es una clave foránea+ que referencia a la tabla **Book**.

```
-- PUBLIC.Contido definition
-- Drop table
-- DROP TABLE PUBLIC.Contido;

CREATE TABLE PUBLIC.Contido (
    idContido INTEGER NOT NULL AUTO_INCREMENT,
    idBook INTEGER NOT NULL,
    contido CHARACTER LARGE OBJECT,
    CONSTRAINT Contido_PK PRIMARY KEY (idContido)
);
CREATE INDEX FK_ID_BOOK_INDEX_9 ON PUBLIC.Contido (idBook);
CREATE UNIQUE INDEX PRIMARY_KEY_9 ON PUBLIC.Contido (idContido);

-- PUBLIC.Contido foreign keys
ALTER TABLE PUBLIC.Contido ADD CONSTRAINT FK_ID_BOOK FOREIGN KEY (idBook) REFERENCES PUBLIC.Book(idBook)
ON DELETE CASCADE ON UPDATE CASCADE;
```

Parámetros de la base de datos:

```
DRIVER: "org.h2.Driver"
URL: "jdbc:h2:rutaBaseDatosSinExtensión;DB_CLOSE_ON_EXIT=TRUE;FILE_LOCK=NO;DATABASE_TO_UPPER=FALSE"
```

El fichero `persistencia.xml` debe tener la siguiente configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="bibliotecaH2" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!--      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>-->
        <exclude-unlisted-classes>false</exclude-unlisted-classes> <-- false si no se listan las clases
en el archivo de configuración -->
        <properties>
            <!--      <property name="jakarta.persistence.jdbc.url"
value="jdbc:mariadb://localhost:3306/peliculas"/>-->
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:ucanaccess://rutabase_base_datos.mdb"/>-->
            <!--      <property name="jakarta.persistence.jdbc.user" value="root"/>-->
            <!--      <property name="jakarta.persistence.jdbc.password" value="" />-->
            <property name="jakarta.persistence.jdbc.user" value="" />
            <property name="jakarta.persistence.jdbc.password" value="" />
            <!--      <property name="jakarta.persistence.jdbc.driver"
value="net.ucanaccess.jdbc.UcanaccessDriver"/>-->
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <!-- Automáticamente, genera el esquema de la base de datos -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>

            <!-- Muestra por pantalla las sentencias SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
            <!--      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />-->
        <!-- para HSQLDB y Ucanaccess -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
        </properties>
    </persistence-unit>
</persistence>
```

#### B) Clase `BibliotecaJpaManager`:

Mediante el **patrón Singleton crea una clase `BibliotecaJpaManager`**, mediante el patrón Singleton de manera que tenga un **atributo `emFactory` de tipo `EntityManagerFactory`** y **que nos permita obtener un objeto de tipo `EntityManager` para realizar las operaciones sobre la base de datos**.

Además, debe tener un **método estático `getEntityManager`** que devuelva un objeto de tipo `EntityManager` y que se encargue de crear el objeto `EntityManager`.

Hazlo con Thread-Safe y doble comprobación.

**Reto: haz que la clase `BibliotecaJpaManager` tenga un singleton para cada factory, guardándolos en un mapa con el nombre de la unidad de persistencia como clave:**

```
private static Map<String, EntityManagerFactory> instances = new HashMap<>();
```

#### C) Clase `Book` implementa `Serializable`:

Haz que sea una **entidad JPA** y que implemente la interfaz `Serializable`.

La clase `Book` debe tener los siguientes atributos:

- `idBook: Long` (autonumérico)
- `isbn: String` (tamaño 13)
- `title: String`
- `author: String`
- `ano: Integer`
- `available: Boolean`
- `portada: byte[]`
- `dataPublicacion: LocalDate` (**Nuevo** campo)
- `List<Contido> contenido;` (**Nuevo**, lista de contenidos del libro, de momento, mientras no tengamos relaciones, hazlo `transient`)

(Fíjate que ya **no existe el campo contenido** que habíamos definido en la clase `Book` de la unidad de bases de datos con JDBC).

La clase debe tener, al menos, los siguientes constructores:

- `Book()`
- `Book(String isbn, String title, String author, Short year, Boolean available, byte[] portada)`
- `Book(Long idBook, String isbn, String title, String author, Short year, Boolean available, byte[] portada)`
- Aquellos que consideres necesarios.

La **lista de Contido** es una lista de objetos de tipo `Contido` que representan los contenidos del libro. La **clase Contido tiene los siguientes atributos: idContido y contido**. Ten en cuenta que existe en la base de datos **una tabla Contido con los campos idContido y contido y una referencia al libro mediante una clave foránea idBook**. De momento, no incluyas la `List` de contenidos en la clase `Book`, hazlos `transient` (**bien con la anotación @Transient o con la palabra reservada transient**), hasta que veamos las relaciones, que será `@OneToMany`.

Los métodos "set" de las propiedades deben devolver una referencia al propio objeto para poder encadenarlos.

**IMPORTANTE: ten en cuenta que los atributos de la clase Book no coinciden con los campos de tabla por lo que debes refactorizar: author -> autor, ano -> anho, available -> disponible, ... o emplear la anotación @Column para mapear los atributos de la clase con los campos de la tabla.**

Métodos de la clase Book (ya implantados):

- Get y set para cada atributo.
- `setPortada` (sin implantar): recibe File y lo asigna al atributo `portada`.
- `setPortada` (sin implantar): recibe un array de bytes y lo asigna al atributo `portada`.
- `setPortada` (Sin implantar): recibe un String con el nombre del fichero y lo asigna al atributo `portada`.
- 
- `getImage`: devuelve un objeto de tipo `Image` con la portada del libro.

```
public Image getImage() {
    if (portada != null) {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(portada)) {
            return ImageIO.read(bis);
        } catch (IOException e) {
        }
    }
    return null;
}
```

- `equals` y `hashCode`: considerando que son iguales cuando tienen el mismo `isbn`. Además, el método `hashCode` debe devolver un valor coherente con el método `equals` (todos los objetos iguales deben tener, al menos el mismo `hashCode`).
- `toString`: devuelve el título, el autor y el año. Si no está disponible escribe un asterisco.

**D) Clase Contido implementa Serializable:**

A diferencia de la clase empleada en la unidad de bases de datos con JDBC, **la clase Contido no debe tener referencia al idBook**, pues no es la mejor práctica (está hecho sólo a modo de ejemplo), **debe tener, si queremos la relación bidireccional, una referencia a Book**.

- `idContido: Long` (autonumérico)
- `contido: String` (contenido del libro en formato texto). Puedes hacer un atributo de tipo `String` o `byte[]` (para almacenar el contenido en formato binario), en cualquier caso, deberías modificar la tabla `Contido` en la base de datos.
- `Book book` (relación con la clase `Book`)

Si has implantado la clase `ContidoDao`, debes **modificar los métodos que obtienen el idBook del book**:

```
contido.getBook().getIdBook();
```

**E) Clase BookJPADao:**

Esta clase, al igual que la clase `BookDao`, la clase `BookJPADao` **debe implantar la interface Dao<T>**, de modo que tenga **un objeto de tipo EntityManager como atributo**. En sistemas empresariales, como la gestión de transacciones no se suele hacer por método, se guarda una referencia a la clase `EntityManagerFactory` y se gestiona por medio de try-with-resources para manejar los cierres de los `EntityManager`.

`Dao<T>:`

```
import java.util.List;

/**
 *
 * @author pepecalo
 * @param <T> Tipo de dato del objeto
 */
public interface DAO<T> {

    T get(long id);

    List<T> getAll();

    void save(T t);

    void update(T t);

    void delete(T t);

    public boolean deleteById(long id);

    public List<Integer> getAllIds();

    public void updateLOB(T book, String f); // en BookJPADao recibe un objeto de tipo Book y un String
    con el nombre del fichero

    public void updateLOBById(long id, String f);

    void deleteAll();
}
```

**Clase BookJPADao:**

Implementa la interfaz `DAO<Book>` y gestiona las operaciones CRUD sobre la tabla `Book` de la base de datos. Tiene como atributo un objeto de tipo `EntityManager` que recoge en el constructor.

**Clase BookDAOFactory:**

Factory de clases que implanten la interfaz `DAO<Book>`.

```
import jakarta.persistence.EntityManager;
/**
```

```

* Factory de clases que implanten la interfaz DAO<Book>.
*
* @version 1.0
* @since 1.0
* @see BookJpaDAO
* @see TipoDAO
*/

```

```

public class BookDaoFactory {

    public enum TipoDao {
        JDBC_H2, JPA_H2, JPA_POSTGRES, HIBERNATE, JSON, JDBC_POSTGRES;
    }

    public static Dao<Book> getBookDAO(TipoDao tipo) {
        switch (tipo) {
            // ..
        }
        return null;
    }
}

```

Implementa un método estático `getBookDAO` que recoge el tipo de DAO que se va a emplear y devuelve el objeto de tipo `BookJPADAO`. Sería interesante hacer cambios para que `getBookDAO` recoja los parámetros necesarios como propiedades de la base de datos, nombre del archivo JSON, nombre de la unidad de persistencia, etc.

```

public static Dao<Book> getBookDAO(TipoDao tipo, Map<String, String> propiedades) {
    switch (tipo) {
        case JPA_H2:
            return new BookJPADao(BibliotecaJpaManager.getEntityManager(propiedades.get("unidadPersistencia")));
        // ...
        default:
            return null;
    }
}

```

### AppBiblioteca:

Ejecuta la aplicación para que haga uso del `BookDaoFactory` para obtener un objeto de tipo `DAO<Book>` para asignarlo al controlador de la aplicación. La aplicación debe funcionar igual que con JDBC, pero ahora con JPA.

Con JDBC\_H2:

```
Dao<Book> bookDao = BookDaoFactory.getBookDAO(BookDaoFactory.TipoDAO.JDBC_H2);
```

Con JPA\_H2:

```
Dao<Book> bookDao = BookDaoFactory.getBookDAO(BookDaoFactory.TipoDAO.JPA_H2);
```

**Haz pruebas con los dos tipos de DAO. ¿Has notado alguna diferencia? Haz mejoras sobre el funcionamiento de la aplicación.**

Puedes hacer pruebas de persistencia de libros en la base de datos:

```

Book libro = new Book("9788424937744", "Tractatus logico-philosophicus-investigaciones filosóficas",
    "Ludwig Wittgenstein", 2017, false);
libro = new Book("9788499088150", "Verano", "J. M. Coetzee", 2011, true);

```

## 03.01. Solución

› Solución: BibliotecaJpaManager

› Solución: Book

› Solución: Clase Contido

› Solución: BookJPADao

› Solución: BookDaoFactory

## Ejercicio 04.01. Descarga y creación de la base de datos de JokeAPI

Dado el **modelo de la aplicación de JokeAPI**, en la que tenemos las enumeraciones `Categoría` y `TipoChiste`, `Flag` y la clase `Chiste`, **vamos a crear una base de datos con JPA** y los chistes de la API.

### Enumeraciones

A) La **enumeración** `Categoría` tiene los siguientes valores:

```
public enum Categoría {
    ANY("Any"),
    MISC("Misc"),
    PROGRAMMING("Programming"),
    DARK("Dark"),
    PUN("Pun"),
    SPOOKY("Spooky"),
    CHRISTMAS("Christmas");
    //...
}
```

› Detalle de implementación de la enumeración Categoría

B) La **enumeración** `TipoChiste` contiene los siguientes valores:

```
public enum TipoChiste {
    SINGLE("single"),
    TWOPART("twopart");
    //...
}
```

› Detalle de implementación de la enumeración TipoChiste

C) La **enumeración** `Flag` contiene los siguientes valores:

`Flag` es una enumeración con los siguientes valores:

```
'''java
public enum Flag {
    EXPLICIT("Explicit"),
    NSFW("NSFW"),
    RELIGION("Religion"),
    POLITICAL("Political"),
    RACIST("Racist"),
    SEXIST("Sexist");
    //...
}
```

› Detalle de implementación de la enumeración Flag

D) **Lenguaje** es una enumeración con los siguientes valores:

```
public enum Lenguaje {
    CS("cs"),
    DE("de"),
    EN("en"),
    ES("es"),
    FR("fr"),
    PT("pt");
    //...
}
```

› Detalle de implementación de la enumeración Lenguaje

## Clases

A) La clase `Chiste` tiene los siguientes atributos:

```
public class Chiste {
    private int id;
    private Categoria categoria;
    private TipoChiste tipo;
    private final List<Flag> banderas;
    private String chiste;
    private String respuesta;

    private Lenguaje lenguaje;
    //...
}
```

› Detalle de implementación de la clase Chiste

B) El adapter `ChisteDeserializer`:

› Detalle de implementación de la clase ChisteDeserializer

C) La clase `ChisteTypeAdapter`:

› Detalle de implementación de la clase ChisteTypeAdapter

D) La interface `IChisteDAO` y clase `ChisteDAO` se usa para obtener los chistes de la API:

› Detalle de implementación de la interfaz IChisteDAO

*Podrías realizar mejoras en el código, como la gestión de excepciones, la comprobación de valores nulos, la simplificación de código, etc.*

› Detalle de implementación de la clase ChisteDAO

## Ejercicio

**Crear una base de datos con JPA** y Hibernate para la aplicación JokeAPI y transfiere todos los datos de JSON a la base de datos.

Añade las dependencias necesarias y el fichero de configuración `persistence.xml` en el directorio `META-INF` de `src/main/resources`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <persistence-unit name="chistesH2" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

```

<!-- <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>-->
<exclude-unlisted-classes>false</exclude-unlisted-classes>
<properties>
    <property name="jakarta.persistence.jdbc.url"
        value="jdbc:h2:RutaABaseDatos;DB_CLOSE_ON_EXIT=TRUE;DATABASE_TO_UPPER=FALSE;FILE_LOCK=NO"/>
    <property name="jakarta.persistence.jdbc.user" value="" />
    <property name="jakarta.persistence.jdbc.password" value="" />
    <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
    <!-- Automáticamente, genera el esquema de la base de datos -->
    <property name="jakarta.persistence.schema-generation.database.action" value="drop-and-
create"/>

    <!-- Muestra por pantalla las sentencias SQL -->
    <property name="hibernate.show_sql" value="false"/>
    <property name="hibernate.format_sql" value="true"/>
    <property name="hibernate.highlight_sql" value="true"/>
    <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
</properties>
</persistence-unit>
</persistence>

```

Para ello, crea las siguientes clases:

A) **ChisteJpaManager** que empleando el patrón Singleton, gestione la creación de la factoría de entidades y el EntityManager.

[› Solución de ChisteJpaManager](#)

B) **Chiste** que emplea JPA para mapear la clase Chiste con la tabla Chiste de la base de datos.

[› Solución de Chiste](#)

C) **ChisteDownloader** que descarga los chistes de la API y los guarda en la base de datos.

Ten el cuenta que ChisteDownloader es un Singleton y que se puede configurar el número de chistes a descargar, además de un tiempo de espera entre chiste y chiste (la API sólo permite 120 peticiones por minuto).

Por ello, haz que sea un hilo que se ejecute cada cierto tiempo ( implements Runnable ) y tenga los siguientes atributos:

- **tiempoEspera** que es el tiempo de espera entre chiste y chiste.
- **instance** que es la instancia de ChisteDownloader.
- **MAX\_CHISTES** que es el número máximo de chistes a descargar.
- **chisteDAO** que es el DAO de Chiste.
- **numeroChistes** que es el número de chistes a descargar (si no se indica debe ser MAX\_CHISTES).

[› Solución de ChisteDownloader](#)

D) **Main** que descarga los chistes y los guarda en la base de datos.

[› Solución de Main](#)

## Ejercicio 05.01. Acceso combinado a la entidad Chiste.

Mofifica la entidad Chiste para que guarde el chiste y la respuesta en un solo campo en la base de datos, pero que se muestren por separado en la aplicación.

```

@Entity
public class Chiste {
    @Id
    private int id;
    private Categoría categoria;
    private TipoChiste tipo;
    private List<Flag> banderas;
    private String chiste;
}

```

```

private String respuesta;
private Lenguaje lenguaje;
// ...
}

```

## Ejercicio 05.02. CLOB y BLOB de una entidad Documento

Crea una entidad Documento que tenga un campo de texto grande (CLOB) para el contenido del documento y un campo de bytes grande (BLOB) para la imagen del documento. Haz pruebas con tres gestores de bases de datos: H2, SQLite y PostgreSQL y comprueba el resultado creando la tabla en cada uno de ellos, con y sin declaración de tipo de LOB.

```

@Entity
public class Documento {
    @Id
    private long id;
    @Lob
    private String contenido;
    @Lob
    private byte[] imagen;
    // ...
}

```

## Ejercicio 05.03. Conversores personalizados y enumeraciones

Declara una entidad `Persona` con atributos:

- `idPersona`.
- `nombre`.
- `apellidos`.
- `fechaNacimiento` de tipo `LocalDate`.
- `sexo` de tipo enumerado `Sexo` que puede ser `HOMBRE` o `MUJER`.
- `estadoCivil` de tipo enumerado `EstadoCivil` que puede ser `SOLTERO`, `CASADO`, `DIVORCIADO` o `VIUDO`.
- `foto` de tipo `byte[]`.

Realiza las conversiones para que:

- El **nombre** y **apellidos** se guardan en la base de datos como “**apellido1, nombre**”, con la primera letra de cada palabra en mayúsculas (empleando acceso por campo y por propiedad).
- La **fecha de nacimiento** como un **entero** que representa la edad de la persona en años (obviamente no es la mejor forma de almacenar la edad, pero quiero que practiquéis con los convertidores), usando anotaciones `@PostLoad` y `@PrePersist`. Haz pruebas de comportamiento haciendo consultas, inserciones y actualizaciones.
- Las **enumeraciones** se guardarán como **cadenas en el caso de estado civil** y como un **carácter de 'H' o 'M' en el caso del sexo**. Hazlo con conversores personalizados.
- La **fotografía** se guardará en un campo de tipo BLOB.

Debes completar la entidad `Persona` y los convertidores necesarios para que funcione correctamente.

```

public class Persona {
    private long idPersona;
    private String nombre;
    private String apellidos;
    private LocalDate fechaNacimiento;
    private Sexo sexo;
    private EstadoCivil estadoCivil;
    private byte[] foto;
    // ...
}

```

Hazlo contra la base de datos H2 y comprueba que los datos se guardan correctamente, creando varios registros y recuperándolos.

## Ejercicio 05.04. Generación de ids con tabla

A partir del ejercicio anterior con Persona, haz que el campo `idPersona` de tipo `Long` y genera el identificador con una tabla. La tabla debe ser compartida con otras entidades que tengan un campo `id` de tipo `Long`.

- Nombre de la tabla: `LONG_ID_GEN`
- Columnas:
  - `nomePK`.
  - `valorPK`.
  - El valor de la columna `nomePK` para la entidad `Persona` debe ser `PERSONA_ID`.
  - Dale un valor inicial de 1000 y un tamaño de asignación de 100.

Crea otro generador para esa tabla que se utilizará para la entidad `Direccion` con un valor inicial de 2000 y un tamaño de asignación de 50.

Haz pruebas de inserción de datos.

## Ejercicio 05.05. Generación de ids con una secuencia

Repite el ejercicio anterior con `Persona`, pero esta vez **utiliza una secuencia para generar el identificador en una base de datos H2**. Haz pruebas compartiendo la secuencia y sin compartir la. Si puedes, haz lo mismo con una base de datos **PostgreSQL**.

## Ejercicio 05.06. Ampliación de la aplicación de persistencia de una biblioteca

Amplía el ejercicio de la biblioteca para que la entidad `Book` tenga un identificador generado automáticamente por medio de una tabla.

Además:

- Crea una enumeración llamada Categoría con los siguientes valores: `NOVELA`, `POESIA`, `ENSAYO`, `TEATRO` y `OTROS`.
- Haz que la entidad `Book` tenga un atributo de tipo `Categoría` y que se persista en la base de datos como una cadena. Realiza una conversión de la enumeración a cadena y viceversa de modo que guarde la categoría con el nombre en mayúsculas sólo la primera letra y con acentos.
- Haz que la columna ISBN sea única, de un tamaño de 13 caracteres y que no pueda ser nula.
- Crea un atributo de tipo `Calendar` para la fecha de publicación del libro y haz que se persista en la base de datos como un tipo `DATE`.
- Crea un atributo transitorio que sea el número de días que han pasado desde la fecha de publicación hasta la fecha actual. Utiliza la clase `java.time.LocalDate` para obtener la fecha actual.
- Crea otro atributo **transitorio con el ISBN en versión de 10 dígitos**, teniendo en cuenta que el ISBN es un número de 13 dígitos. Para ello, puedes utilizar la clase `java.math.BigInteger` para realizar la conversión y el siguiente algoritmo:
  1. Elimina los primeros tres dígitos (normalmente 978)
  2. Elimina el último dígito. Ahora tienes nueve dígitos
  3. Ahora necesitas calcular el 'dígito de control', que será el décimo dígito de tu ISBN. El objetivo del dígito de control es asegurarse de no haber cometido un error tipográfico: transponer dos dígitos, por ejemplo, o escribir mal uno. Esto es bastante complicado:

4. Multiplica el primer dígito por 10, el segundo por 9, el tercero por 8 y así sucesivamente, hasta llegar al último dígito (multiplicado por 2).
5. Ahora tienes una cadena de 9 números nuevos. Agrégalos todos juntos.
6. Divide esta suma por once. Ahora estás interesado en el resto. Por ejemplo, si la suma fuera 242, que es exactamente  $11 \times 22$ , entonces el resto es cero. Si la suma fuera 243, entonces sobraría 1. Tendrás un resto que está entre 0 y 10.
7. Resta ese resto de 11 para obtener el dígito de control.
8. Si el resultado es 10, entonces el dígito de control es 'X'.

Código Java:

```
public class ISBN {
    public static void main(String[] args) {
        String isbn = "978-3-16-148410-0";
        String isbn10 = isbn.substring(3, isbn.length() - 1);
        System.out.println(isbn10);
        BigInteger sum = BigInteger.ZERO;
        for (int i = 0; i < isbn10.length(); i++) {
            int digit = Character.getNumericValue(isbn10.charAt(i));
            sum = sum.add(BigInteger.valueOf(digit).multiply(BigInteger.valueOf(10 - i)));
        }
        System.out.println(sum);
        BigInteger remainder = sum.mod(BigInteger.valueOf(11));
        System.out.println(remainder);
        BigInteger controlDigit = BigInteger.valueOf(11).subtract(remainder);
        System.out.println(controlDigit);
        if (controlDigit.intValue() == 10) {
            System.out.println("X");
        } else {
            System.out.println(controlDigit);
        }
    }
}
```

Un ejemplo más completo:

```
public class ISBNConverter {

    public static void main(String[] args) {
        String isbn13 = "9780123456789"; // ISBN-13
        String isbn10 = convertirISBN13aISBN10(isbn13);
        System.out.println("ISBN-10: " + isbn10);
    }

    public static String convertirISBN13aISBN10(String isbn13) {
        // Verifica si el ISBN-13 proporcionado es válido
        if (!esISBN13Valido(isbn13)) {
            return "ISBN-13 no válido";
        }

        // Elimina los primeros 3 dígitos (978 o 979) del ISBN-13
        String isbn10Parcial = isbn13.substring(3);

        // Calcula el dígito de verificación para el ISBN-10 parcial
        int suma = 0;
        for (int i = 0; i < 9; i++) {
            int digito = Character.getNumericValue(isbn10Parcial.charAt(i));
            suma += (i + 1) * digito;
        }

        int digitoVerificador = suma % 11;
        char digitoVerificadorChar;

        if (digitoVerificador == 10) {
            digitoVerificadorChar = 'X';
        } else {
            digitoVerificadorChar = (char) ('0' + digitoVerificador);
        }

        // Combina el ISBN-10 parcial con el dígito de verificación calculado
        return isbn10Parcial + digitoVerificadorChar;
    }

    public static boolean esISBN13Valido(String isbn13) {
        // Verifica que el ISBN-13 tenga 13 dígitos y comience con "978" o "979"
    }
}
```

```
    return ISBN13.matches( "978[0-9]{13}" );  
}
```

Crea varios libros y pérsistelos en la base de datos (una nueva). Recupéralos y muestra los valores de los datos, incluyendo transitorios.

## Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador

Vamos a crear una aplicación de equipos de la NBA. Cada equipo tiene un entrenador y cada entrenador tiene un equipo, por lo que la **relación es uno a uno bidireccional**.

Crea las siguientes entidades:

- **Equipo**: con los atributos `idEquipo`, `nombre`, `ciudad`, `conferencia`, `division`, `nombreCompleto` y `abreviatura`.
    - Crea una enumeración `Conferencia` con los valores `ESTE` y `OESTE`.
    - Crea una enumeración `Division` con los valores `ATLANTICO`, `CENTRAL`, `SURESTE`, `NOROESTE`, `PACIFICO` y `SUROESTE`.
    - En la base de datos, **la conferencia y la división se guardarán como cadenas**:
      - `EAST`, `WEST`
      - `ATLANTIC`, `CENTRAL`, `SOUTHEAST`, `NORTHWEST`, `PACIFIC`, `SOUTHWEST`
    - La abreviatura debe ser única, así como el `idEquipo`.

Los equipos puedes cargarlos del siguiente archivo JSON:

› Ver datos de ejemplo

- **Entrenador**: con los atributos `idEntrenador`, `nombre`, `fechaNacimiento`, `salario` y `equipo`.

Mediante JPA e Hibernate, crea una aplicación que permita:

- Añadir un equipo.
  - Insertar un entrenador.
  - Asignar un entrenador a un equipo.
  - Asignar un equipo a un entrenador.
  - Mostrar los datos de un equipo y su entrenador.

Para ello, debes crear las clases de utilidad necesarias para realizar las operaciones anteriores. `JpaNbaManager`, `EquipoDAO`, `EntrenadorDAO`, etc.

## Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo

Siguiendo el ejemplo anterior, vamos a crear una relación muchos a uno unidireccional entre Jugador y Equipo .

Para ello debe crear una nueva entidad **Jugador** con los siguientes atributos:

- **idJugador**: identificador del jugador.
  - **nombre**: nombre del jugador.
  - **apellidos**: apellidos del jugador.
  - **equipo**: equipo al que pertenece el jugador.
  - **altura**: altura del jugador (Double).
  - **peso**: peso del jugador (Double).
  - **numero**: número de camiseta del jugador (SmallInt).
  - **anoDraft**: año de elección en el draft (entero).-
  - **numeroDraft**: número de elección en el draft (SmallInt).

- `rondaDraft`: ronda de elección en el draft (SmallInt).
- `posicion`: posición en la que juega (base, escolta, alero, ala-pívot, pivot, como enumeración, que debe guardarse como 'G', 'C', 'F', 'F-C', 'C-F').
- `país`: país de origen del jugador.
- `colegio`: universidad o equipo en el que jugó.
- `foto`: foto del jugador.

Haz que la relación sea unidireccional, de modo que la entidad `Jugador` tenga una referencia al `Equipo` y el nombre de la clave foránea sea `idEquipo`.

Crea jugadores y añádelos a los equipos que has creado en el ejercicio anterior. Completa la aplicación para que puedas añadir jugadores a los equipos y mostrar los jugadores de un equipo.

Datos de ejemplo:

[Ver datos de ejemplo](#)

## Ejercicio 06.03. Relación muchos a muchos unidireccional Jugador-Posición

Vamos a crear una relación muchos a muchos unidireccional entre `Jugador` y `Posicion`. Para eso debes crear una nueva entidad `Posicion` con los siguientes atributos:

- `idPosicion`: identificador de la posición (Long).
- `nombre`: nombre de la posición (String, tamaño máximo 50).
- `abreviatura`: abreviatura de la posición (String, tamaño máximo 3).
- `descripcion`: descripción de la posición (String, tamaño máximo 255).

Haz que la relación sea unidireccional, de modo que la entidad `Jugador` tenga una colección de `Posicion` y el nombre de la tabla de unión sea `JugadorPosicion`.

Crea posiciones y añádelas a los jugadores que has creado en el ejercicio anterior.

## Ejercicio 06.04. Mapeo de una base de datos de juegos

### Migración de base de datos H2 entre versiones

En la base de datos origen se ejecuta el siguiente script:

```
SCRIPT TO '<ruta-al-archivo-backup>/backup.sql';
```

En la base de datos destino se ejecuta el siguiente script:

```
RUNSCRIPT FROM '<ruta-al-archivo-backup>/backup.sql';
```

### Ejercicio 6.4. Mapeo de una base de datos de juegos.

Disponemos de una base de datos de juegos, que se compone de las siguientes tablas (la base de datos compartida está en el fichero anexo)

**Plataforma**: idPlataforma, nombre. (*Ya contiene datos*) **Genero**: idGenero, nombre. (*Ya contiene datos*)

**Juego**: idJuego, idGenero (FK), idPlataforma (FK), titulo, miniatura (varchar), estado, descripciónCorta, descripción, url, editor, desarrollador, fecha. **Imagen**: idImagen, idJuego (FK), url, imagen (tipo BLOB).

**RequisitosSistema**: idJuego (PK), almacenamiento, graficos, memoria, os, procesador.

Referencias: <https://www.freetogame.com/api-doc>

- Las plataformas pueden ser: pc, browser, all, etc. (Ya disponibles en la tabla Plataforma)
- Las categorías (géneros) pueden ser:
  - mmorpg, shooter, strategy, moba, racing, sports, social, sandbox, open-world, survival, pvp, pve, pixel, voxel, zombie, turn-based, first-person, third-person, top-down, tank, space, sailing, side-scroller, superhero, permadeath, card, battle-royale, mmo, mmofps, mmotps, 3d, 2d, anime, fantasy, sci-fi, fighting, action-rpg, action, military, martial-arts, flight, low-spec, tower-defense, horror, mmorts, etc. (Ya incorporadas en la tabla Genero)

Cuyos datos se ajustan al formato del siguiente JSON (ejemplo). Debes tener en cuenta que no se ha creado la tabla de requerimientos mínimos, pero se puede hacer si se desea en una nueva tabla de la base de datos, relacionada, uno a uno:

```
{
  "id": 452,
  "title": "Call Of Duty: Warzone",
  "thumbnail": "https://www.freetogame.com/g/452/thumbail.jpg",
  "status": "Live",
  "short_description": "A standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare.",
  "description": "Call of Duty: Warzone is both a standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare. Warzone features two modes \u2014 the general 150-player battle royale, and \u2014Plunder\u2014. The latter mode is described as a \u2014crace to deposit the most Cash\u2014. In both modes players can both earn and loot cash to be used when purchasing in-match equipment, field upgrades, and more. Both cash and XP are earned in a variety of ways, including completing contracts.\r\n\r\nAn interesting feature of the game is one that allows players who have been killed in a match to rejoin it by winning a 1v1 match against other felled players in the Gulag.\r\n\r\nOf course, being a battle royale, the game does offer a battle pass. The pass offers players new weapons, playable characters, Call of Duty points, blueprints, and more. Players can also earn plenty of new items by completing objectives offered with the pass.",
  "game_url": "https://www.freetogame.com/open/call-of-duty-warzone",
  "genre": "Shooter",
  "platform": "Windows",
  "publisher": "Activision",
  "developer": "Infinity Ward",
  "release_date": "2020-03-10",
  "freetogame_profile_url": "https://www.freetogame.com/call-of-duty-warzone",
  "minimum_system_requirements": {
    "os": "Windows 7 64-Bit (SP1) or Windows 10 64-Bit",
    "processor": "Intel Core i3-4340 or AMD FX-6300",
    "memory": "8GB RAM",
    "graphics": "NVIDIA GeForce GTX 670 / GeForce GTX 1650 or Radeon HD 7950",
    "storage": "175GB HD space"
  },
  "screenshots": [
    {
      "id": 1124,
      "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-1.jpg"
    },
    {
      "id": 1125,
      "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-2.jpg"
    },
    {
      "id": 1126,
      "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-3.jpg"
    },
    {
      "id": 1127,
      "image": "https://www.freetogame.com/g/452/Call-of-Duty-Warzone-4.jpg"
    }
  ]
}
```

a) Crea entidades JPA en Java para las tablas de la base de datos, con las siguientes características:

- **Genero**: con los atributos `idGenero`, `nombre`. La clave es autonumérica.
- **Plataforma**: con los atributos `idPlataforma` y `nombre`. La clave es autonumérica. *Nota: si se hubiese declarado como enumeración, para poder mapear una enumeración en una tabla independiente, obligaría a crear una entidad independiente con el idPlataforma y el nombre. Sin embargo, en este caso, se podría mapear la enumeración directamente en la tabla Juego o declararla como una clase y no como una enumeración.*

- **Juego**: con todos los atributos de la tabla Juego, incluyendo la relación con Genero y Plataforma. La clave primaria, **idJuego**, no es autogenerada, es asignada. Ten en cuenta que la relación con la tabla Imagen se trata de una relación uno a muchos, por lo que se deberá declarar una colección de imágenes. Además, el **idGenero** y el **idPlataforma** son claves foráneas de las entidades y no deben declararse como atributos de la entidad Juego, sino como objetos del tipo de las entidades **Genero** y **Plataforma**.
- **Imagen**: con los atributos **idImagen** (no autogenerada), **Juego** (relacionada con la entidad Juego @OneToOne), **url**, **imagen** (tipo byte[]).
- **RequisitosSistema**: relacionada con la tabla Juego. Atributos: **idJuego** (PK), **sistemaOperativo** (su nombre no coincide con la columna de la tabla), **almacenamiento**, **graficos**, **memoria**, **procesador** y su relación **juego**. Debe emplearse una clave comparta con el idJuego. Para ello debe emplearse la anotación: **@MapsId**: **@MapsId("idJuego")**.

```
@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@MapsId
@JoinColumn(name = "idJuego")
private Juego juego;
```

b) Haz una sencilla aplicación que cree un juego y lo persista en la base de datos. Ten en cuenta que **las claves no son autonuméricas**

- Ejemplo de juegos: <https://www.freetogame.com/api/game?id=X>, pasándole el id del Juego, desde 1 al número de juegos que consideres. Ten en cuenta que el juego podría no existir devolviendo:

```
{"status":0,"status_message":"No game found with that id"}
```

⌚ Bases de datos de videojuegos (H2)

## Ejercicio 06.05. Claves compartidas en relaciones uno a uno

Comprueba el funcionamiento de la anotación **@PrimaryKeyJoinColumn** en una relación uno a uno entre **Persona** y **Departamento**. Crea las entidades y realiza pruebas de persistencia.

**Persona**: **idPersona** (IDENTITY), **nombre**, **departamento** (uno a uno con anotación de **@PrimaryKeyJoinColumn**)  
**Departamento**: **idDepartamento** (IDENTITY), **nombre**.

Modifica el ejercicio para que sea bidireccional con **@OneToOne** y **@MapsId** en la entidad Departamento y como propietaria de la relación.

## Ejercicio 07.01. Elementos embebidos.

Crea una aplicación con JPA para la gestión de películas y series.

1. Crea una clase **InfoContenido** con los siguientes atributos:

- **titulo** (String): de tamaño 100.
- **genero** (String): de tamaño 50.
- **pais** (String): de tamaño 2.
- **duracion** (int): duración en minutos.
- **año** (int): año.
- **sinopsis** (String): de tamaño clob.

2. Crea una entidad **Serie** con los siguientes atributos:

- **idSerie** (long): identificador de la serie. Secuencia.
- **informacion** (de tipo **InfoContenido**)

- `fechaEstreno` (LocalDate).
- `temporadas` (int): número de temporadas.
- `capitulos` (int)
- `directores` (lista de String).

3. Crea una entidad `Pelicula` con los siguientes atributos:

- `idPelicula` (long): identificador de la película. Secuencia.
- `informacion` (de tipo `InfoContenido`)
- La entidad `Serie` y `Pelicula` deben tener el atributo `informacion` como un objeto embebido.
- La entidad `Pelicula` el atributo `pais` debe ser renombrado a `paisPelicula`.
- El atributo `directores` debe guardarse en una nueva tabla, como una colección con la anotación `@ElementCollection` (busca información sobre esta anotación).
- La fecha de estreno, `fechaEstreno`, de la serie debe guardarse en formato numérico (YYYYMMDD).

## Ejercicio 07.02. Clave compuesta en una relación de muchos a muchos.

Data la aplicación de gestión de películas y series, añade dos nuevas entidades: `Usuario` y `Calificacion` que permita a los usuarios calificar las películas.

1. Crea una clase `Usuario` con los siguientes atributos:

- `idUsuario` (long): identificador del usuario. Secuencia.
- `nombre` (String): nombre del usuario.
- `email` (String): email del usuario.
- `password` (String): contraseña del usuario.
- `fechaRegistro` (LocalDate): fecha de registro.

2. Crea una clase `Calificacion` con los siguientes atributos:

- `calificacion` (int): calificación del contenido, con valores de 10 a 0.
- `fechaCalificacion` (LocalDate): fecha de la calificación.
- `comentario` (String): comentario de la calificación.
- Además, debe estar relacionado con las entidades `Usuario`, `Pelicula` y `Serie`. Como un usuario puede calificar varias películas y series, y una película o serie puede ser calificada por varios usuarios, es una relación de muchos a muchos. *No es preciso que califique series, pues el caso de uso es similar al de las películas.*

La clave primaria de la tabla `Calificacion` debe ser compuesta por los atributos `idUsuario`, `idPelicula`.

## Ejercicio 07.03. Entidades principales de base de datos de películas.

```
<property name="jakarta.persistence.jdbc.url"
      value="jdbc:mariadb://dbalumnos.sanclemente.local:3312/Peliculas"/>
<property name="jakarta.persistence.jdbc.user" value="accesodatos"/>
<property name="jakarta.persistence.jdbc.password" value="ad123.."/>
<property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver"/>
<property name="jakarta.persistence.schema-generation.database.action" value="none"/>
```

Sea la siguiente estructura de la base de datos:



[Estructura de la base de datos](#)

## > SQL de las tablas de la base de datos

En el que:

- El **título de la película** se guarda en el campo `castelan`.
- El **identificador de la película es entero** (no autoincremento).
- Los participantes de la película están relacionados por medio da de la tabla `peliculapersonaxe`, en la que el campo `ocupacion` identifica o tipo de ocupación de la película ('Actor', ...):

```
CREATE TABLE IF NOT EXISTS `ocupacion` (
  `ocupacion` varchar(50) COLLATE utf8_spanish_ci DEFAULT NULL,
  `orde` int(11) NOT NULL,
  UNIQUE KEY `Ocupación#PX` (`ocupacion`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```



Ocupación



Personaxeocupación

Para empezar, **crea las entidades** `Pelicula`, `Personaxe` y `Ocupacion`. A continuación, **crea la entidad** `PeliculaPersonaxe` que **relaciona las entidades** `Pelicula` y `Personaxe` y `Ocupacion`. Ten en cuenta que tiene un nuevo atributo `personaxeInterpretado`, que **es el nombre del personaje interpretado por el actor en la película**.

Méjora:

Crea las entidades asociadas a la base de datos. De modo que las columnas `anoInicio`, `outrasDuracions`, `video`, `laserDisc` pertenezca a una entidad `DetallePelicula` de tipo embbebido. Hay que tener en cuenta que estos elementos no siempre están presentes, por lo que deben ser opcionales.

Autor/a: Pepinho Última actualización: 13.02.2025

# BOLETINES COMPLETOS.

---

- [Ejercicios JPA](#)
    - [0. Creación de EntityManagerFactory con patrón Singleton y Thread-Safe](#)
      - [0.1. EntityManagerFactory Singleton](#)
      - [0.2. EntityManagerFactory Singleton con propiedades](#)
      - [0.3. EntityManagerFactory Singleton para cada unidad de persistencia](#)
    - [1. Rango Legal y Organismo](#)
    - [2. Alquiler de películas](#)
    - [3. Pedidos PostgreSQL](#)
    - [4. Pedidos](#)
    - [5. Base de datos de legislación](#)
      - [Propiedades de la conexión a la base de datos](#)
      - [Esquema de la base de datos](#)
      - [1. JPAUtil](#)
      - [2. Clases del modelo](#)
        - [RangoLegal](#)
        - [Organismo](#)
        - [Publicacion \(enumeración\) y PublicacionConverter](#)
        - [Clasificacion](#)
        - [Norma](#)
        - [Documento](#)
        - [DocumentoNorma](#)
      - [3. DTO y Consultas](#)
      - [4. NormaDAO y NormaRepository](#)
        - [NormaDAO implements DAO<Norma, Integer>](#)
        - [NormaRepository](#)
      - [5. Servicio Rest con Spring Boot Data JPA](#)
      - [6. Servicio Rest con Spring Boot Data JPA y paginación](#)
    - [Ejecutores de código al inicio de la Aplicación](#)
      - [Diferencias entre ejecutores de código al inicio de la Aplicación](#)
- 

## Ejercicios JPA

### 0. Creación de EntityManagerFactory con patrón Singleton y Thread-Safe

#### 0.1. EntityManagerFactory Singleton

Crea un `EntityManagerFactory` con patrón Singleton y Thread-Safe. La clase debe tener las siguientes características:

- Un **método estático**, `getEmFactory`, que devuelva una instancia de `EntityManagerFactory`, recogiendo el nombre de la unidad de persistencia.
- Un **método estático**, `getEntityManager`, que devuelva una instancia de `EntityManager`, recogiendo el nombre de la unidad de persistencia.
- Un método, `isEntityManagerFactoryClosed`, que devuelva si la factoría es nula o está cerrada.
- Un **método para cerrar la factoría**.

## 0.2. EntityManagerFactory Singleton con propiedades

Añade a la clase anterior un método para que el `EntityManagerFactory` sea creado con un **mapa de propiedades que se le pasan al método `createEntityManagerFactory()`** de `Persistence`. El mapa de propiedades debe tener las siguientes propiedades:

- `jakarta.persistence.jdbc.url`: la URL de la base de datos.
- `jakarta.persistence.jdbc.user`: el usuario de la base de datos.
- `jakarta.persistence.jdbc.password`: la contraseña de la base de datos.
- `jakarta.persistence.jdbc.driver`: el driver de la base de datos.
- `jakarta.persistence.schema-generation.database.action`: la acción de la base de datos.
- `jakarta.persistence.schema-generation.create-source`: la fuente de creación de la base de datos.

## 0.3. EntityManagerFactory Singleton para cada unidad de persistencia

Mejora: el EntityManager debe ser creado con el método `createEntityManager()` de la factoría y debe ser único para cada unidad de persistencia. Para ello, en vez de tener una única instancia de `EntityManagerFactory`, **debes tener un Map de EntityManagerFactory**, una para cada unidad de persistencia, en el que la clave sea el nombre de la unidad de persistencia y el valor un objeto de tipo `EntityManagerFactory`.

## 1. Rango Legal y Organismo

Realizar **un proyecto JPA con EclipseLink que mapee las tablas de la base de datos** muestre todos los rangos legales y organismos de la base de datos.

### RangoLegal:

`idRangoLegal` (Integer), `nomeG` (String), `nomeC` (String), `descripcion` (texto largo). Los nombres de los atributos `nomeG` y `nomeC` no coinciden con los de la base de datos y tienen tamaño 128, además, son únicos. La clave primaria es auto numérica.

### Organismo:

`idOrganismo` (Integer), `nome` (String), `descripcion` (texto largo). El nombre es único. La clave primaria es autonumérica.

- URL: `jdbc:mariadb://dbalumnos.sanclemente.local:3312/Lexislacion`
- DRIVER: `org.mariadb.jdbc.Driver`
- USUARIO: `lexislacionuser`
- PASSWORD: `ABC123..`

Crea una base de datos en PostgreSQL con el nombre `Lexislacion` y las tablas `RangoLegal` y `Organismo`.

Haz que la aplicación migre los datos de la base de datos de MariaDB a la de PostgreSQL.

## 2. Alquiler de películas

Crea una base de datos en PostgreSQL con el nombre `videoclub` y restaura la base de datos `db-videoclub.tar`.

Dicha base de datos tiene 15 tablas:

- **actor**: almacena datos de actores, incluidos el nombre y el apellido.
- **film**: almacena datos de películas como título, año de lanzamiento, duración, clasificación, etc.
- **film\_actor**: almacena las relaciones entre películas y actores.
- **category**: almacena datos de las categorías de las películas.
- **film\_category**: almacena las relaciones entre películas y categorías.
- **store**: contiene los datos de la tienda, incluidos el personal gerencial y la dirección.

- **inventory**: almacena datos del inventario.
- **rental**: almacena datos de alquiler.
- **payment**: almacena los pagos de los clientes.
- **staff**: almacena datos del personal.
- **customer**: almacena datos de los clientes.
- **address**: almacena datos de dirección para el personal y los clientes.
- **city**: almacena los nombres de las ciudades.
- **country**: almacena los nombres de los países.

Ahora que conocemos todo sobre nuestra base de datos de videoclub de ejemplo, pasemos a cargar la misma base de datos en el servidor de la base de datos PostgreSQL. Los pasos para ello se enumeran a continuación:

**Paso 1:** Cree una base de datos de videoclub, abriendo la consola SQL. Una vez que abra la consola, deberás añadir las credenciales necesarias para la base de datos, que se verían algo así:

```
Servidor [localhost]:  
Base de datos [postgres]:  
Puerto [5432]:  
Nombre de usuario [postgres]:  
Contraseña para el usuario postgres:
```

Ahora, usando la declaración `CREATE DATABASE`, cree una nueva base de datos de la siguiente manera:

```
CREATE DATABASE videoclub;
```

**Paso 2:** Cargue el archivo de la base de datos creando una carpeta en la ubicación deseada (por ejemplo, `C:\users\sample_database\bd-videoclub.tar`). Ahora abra el símbolo del sistema y navegue hasta la carpeta `bin` de la carpeta de instalación de PostgreSQL como se muestra a continuación (en el caso de haber añadido la ruta de instalación de PostgreSQL al PATH no será necesario navegar hasta la carpeta `bin`):

```
cd C:\ruta\al\la\carpeta\bin
```

Use la herramienta `pg_restore` para cargar datos en la base de datos `videoclub` que acabamos de crear mediante el siguiente comando:

```
pg_restore -U postgres -d videoclub C:\users\ruta\db-videoclub.tar
```

Ahora introduce la contraseña de usuario de su base de datos y su base de datos se cargará.

#### Verificar la carga de la base de datos:

Ahora, si necesitas verificar si la base de datos, usa el siguiente comando para acceder a la base de datos en la consola SQL:

```
\c
```

Ahora, para listar todas las tablas en la base de datos, usa el siguiente comando:

```
\dt
```

1. Crea los siguientes tipos de entidad de acuerdo con los estándares Java:

- `País` que mapee la tabla `country`: `country_id` (de tipo serial4), `country` (varchar(50), `last_update` (timestamp)).

- **Categoría** (tabla `category`): `category_id` (serial4), `name` (varchar(25)), `last_update` (timestamp).
- **Idioma** (tabla `language`): `language_id` (serial4), `name` (varchar(20)), `last_update` (timestamp).
- **Actor** (tabla `actor`): `actor_id` (serial4), `first_name` (varchar(45)), `last_name` (varchar(45)), `last_update` (timestamp).

Importante: fíjate en los tipos de datos y en las claves primarias, cómo se generan y cómo se relacionan las tablas.

Dichas entidades no contienen relaciones entre sí.

2. Crea un tipo de entidad, **Película**, que mapee la tabla `film`. La creación de la tabla `film` es la siguiente:

```
CREATE TABLE IF NOT EXISTS public.film
(
    film_id integer NOT NULL DEFAULT nextval('film_film_id_seq'::regclass),
    title character varying(255) COLLATE pg_catalog."default" NOT NULL,
    description text COLLATE pg_catalog."default",
    release_year year,
    language_id smallint NOT NULL,
    rental_duration smallint NOT NULL DEFAULT 3,
    rental_rate numeric(4,2) NOT NULL DEFAULT 4.99,
    length smallint,
    replacement_cost numeric(5,2) NOT NULL DEFAULT 19.99,
    rating mpaa_rating DEFAULT 'G'::mpaa_rating,
    last_update timestamp without time zone NOT NULL DEFAULT now(),
    special_features text[] COLLATE pg_catalog."default",
    fulltext tsvector NOT NULL,
    CONSTRAINT film_pkey PRIMARY KEY (film_id),
    CONSTRAINT "FKbqsvlyhhs40rh7v7e6qpdt05i" FOREIGN KEY (language_id)
        REFERENCES public.language (language_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT film_language_id_fkey FOREIGN KEY (language_id)
        REFERENCES public.language (language_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
)
```

De momento, mapea el campo `fulltext` como un String:

3. Haz que la entidad **Película** tenga una con la entidad **Idioma** (un idioma puede tener muchas películas, pero una película sólo puede tener un idioma).
4. **CategoríaPelicula**: haz que la entidad **Película** tenga una relación con la entidad **Categoría** (una película puede tener muchas categorías y una categoría puede tener muchas películas), para ello, crea una entidad **CategoríaPelicula** que mapee la tabla `film_category`, que dispone de las siguientes columnas: `film_id` (int4), `category_id` (int4), `last_update` (timestamp). IMPORTANTE: la clave primaria de la tabla `film_category` es compuesta por `film_id` y `category_id`.
5. **PeliculaActor**: haz que la entidad **Película** tenga una relación con la entidad **Actor** (una película puede tener muchos actores y un actor pudo haber realizado muchas películas), para ello, crea una entidad **PeliculaActor** que mapee la tabla `film_actor`, que dispone de las siguientes columnas: `actor_id` (int4), `film_id` (int4), `last_update` (timestamp). La clave primaria de la tabla `film_actor` es compuesta por `actor_id` y `film_id`.
6. **Ciudad**: mapea la tabla `city` que dispone de las siguientes columnas: `city_id` (serial4), `city` (varchar(50)), `country_id` (int4), `last_update` (timestamp). Haz que la entidad **Ciudad** tenga una relación con la entidad **País** (una ciudad pertenece a un único país y un país puede tener muchas ciudades).
7. **Dirección**: mapea la tabla `address` que dispone de las siguientes columnas: `address_id` (serial4), `address` (varchar(50)), `address2` (varchar(50)), `district` (varchar(20)), `city_id` (int4), `postal_code` (varchar(10)), `phone` (varchar(20)), `last_update` (timestamp). Haz que la entidad **Dirección** tenga una relación con la entidad **Ciudad** (una dirección pertenece a una única ciudad y una ciudad puede tener muchas direcciones).
8. **Empleado**: mapea la tabla `staff` que dispone de las siguientes columnas: `staff_id` (serial4), `first_name` (varchar(45)), `last_name` (varchar(45)), `address_id` (int4), `email` (varchar(50)), `store_id` (int4), `active`

(boolean), `username` (varchar(16)), `password` (varchar(40)), `last_update` (timestamp). Haz que la entidad `Empleado` tenga una relación con la entidad `Direccion` (un empleado tiene una dirección y una dirección puede pertenecer a muchos empleados) y con la entidad `Tienda`.

9. `Tienda`: mapee la tabla `store` que dispone de las siguientes columnas: `store_id` (serial4), `manager_staff_id` (int4), `address_id` (int4), `last_update` (timestamp). Haz que la entidad `Tienda` tenga una relación con la entidad `Direccion` (una tienda tiene una dirección y una dirección puede pertenecer a muchas tiendas).
  10. `Inventario`: mapee la tabla `inventory` que dispone de las siguientes columnas: `inventory_id` (serial4), `film_id` (int4), `store_id` (int4), `last_update` (timestamp). Haz que la entidad `Inventario` tenga una relación con la entidad `Pelicula` (un inventario tiene una película y una película puede estar en muchos inventarios) y con la entidad `Tienda` (un inventario pertenece a una tienda y una tienda puede tener muchos inventarios).
  11. `Cliente`: mapee la tabla `customer` que dispone de las siguientes columnas: `customer_id` (serial4), `store_id` (int4), `first_name` (varchar(45)), `last_name` (varchar(45)), `email` (varchar(50)), `address_id` (int4), `activebool` (boolean), `create_date` (date), `last_update` (timestamp), `active` (int4). Haz que la entidad `Cliente` tenga una relación con la entidad `Tienda` (un cliente pertenece a una tienda y una tienda puede tener muchos clientes) y con la entidad `Direccion` (un cliente tiene una dirección y una dirección puede pertenecer a muchos clientes).
- `Alquiler`: mapee la tabla `rental` que dispone de las siguientes columnas: `rental_id` (serial4), `rental_date` (timestamp), `inventory_id` (int4), `customer_id` (int4), `return_date` (timestamp), `staff_id` (int4), `last_update` (timestamp). Haz que la entidad `Alquiler` tenga una relación con la entidad `Inventario` (un alquiler tiene un inventario y un inventario puede tener muchos alquileres), con la entidad `Cliente` (un alquiler tiene un cliente y un cliente puede tener muchos alquileres) y con la entidad `Staff` (un alquiler tiene un empleado y un empleado puede tener muchos pagos).
13. `Pago`: mapee la tabla `payment` que dispone de las siguientes columnas: `payment_id` (serial4), `customer_id` (int4), `staff_id` (int4), `rental_id` (int4), `amount` (numeric(5,2)), `payment_date` (timestamp). Haz que la entidad `Pago` tenga una relación con la entidad `Alquiler` (un pago tiene un alquiler y un alquiler puede tener muchos pagos), con la entidad `Cliente` (un pago tiene un cliente y un cliente puede tener muchos pagos) y con la entidad `Staff` (un pago tiene un empleado y un empleado puede tener muchos pagos).

Diagrama de la base de datos:



Diagrama de la base de datos

### 3. Pedidos PostgreSQL

Data la estructura de datos de MariaDB se define en el script `bd-pedidos.sql`, crea un proyecto JPA con Hibernate que mapee las tablas de la base de datos en PostgreSQL (no crees la base de datos en PostgreSQL, simplemente mapea las tablas, tampoco lo hagas en MariaDB):

```

CREATE TABLE IF NOT EXISTS public."Producto"
(
    "idProducto" integer NOT NULL DEFAULT nextval('"Producto_idProducto_seq"'::regclass),
    precio double precision,
    nombre character varying(125) COLLATE pg_catalog."default" NOT NULL,
    descripcion character varying(255) COLLATE pg_catalog."default",
    imagen oid,
    CONSTRAINT "Producto_pkey" PRIMARY KEY ("idProducto")
)

CREATE TABLE IF NOT EXISTS public."Cliente"
(
    "idCliente" integer NOT NULL DEFAULT nextval('"Cliente_idCliente_seq"'::regclass),
    dni character varying(12) COLLATE pg_catalog."default" NOT NULL,
    nombre character varying(128) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "Cliente_pkey" PRIMARY KEY ("idCliente")
)

CREATE TABLE IF NOT EXISTS public."Pedido"
(
    "idCliente" integer,

```

```

"idPedido" integer NOT NULL DEFAULT nextval('"Pedido_idPedido_seq"'::regclass),
fecha timestamp(6) without time zone NOT NULL,
CONSTRAINT "Pedido_pkey" PRIMARY KEY ("idPedido"),
CONSTRAINT "FKb7xr57df8semvktej7l1lo8se" FOREIGN KEY ("idCliente")
    REFERENCES public."Cliente" ("idCliente") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)

CREATE TABLE IF NOT EXISTS public."Comentario"
(
    "idPedido" integer NOT NULL,
comentario character varying(255) COLLATE pg_catalog."default",
CONSTRAINT "FKdne7p3hv47b016i5m2efvrpe4" FOREIGN KEY ("idPedido")
    REFERENCES public."Pedido" ("idPedido") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)

CREATE TABLE IF NOT EXISTS public."LineaPedido"
(
    cantidad smallint NOT NULL,
    "idLineaPedido" integer NOT NULL DEFAULT nextval('"LineaPedido_idLineaPedido_seq"'::regclass),
    "idPedido" integer,
    "idProducto" integer,
CONSTRAINT "LineaPedido_pkey" PRIMARY KEY ("idLineaPedido"),
CONSTRAINT "FK16r6q9njvef9fuecshutqo5ro" FOREIGN KEY ("idPedido")
    REFERENCES public."Pedido" ("idPedido") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION,
CONSTRAINT "FKjmo85q6spgveoxjmyjrvwhk1q" FOREIGN KEY ("idProducto")
    REFERENCES public."Producto" ("idProducto") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)

CREATE TABLE IF NOT EXISTS public."TagLineaPedido"
(
    "idLineaPedido" integer NOT NULL,
tag character varying(32) COLLATE pg_catalog."default",
CONSTRAINT "FKfh1px6cx035k4w4615810uxg6" FOREIGN KEY ("idLineaPedido")
    REFERENCES public."LineaPedido" ("idLineaPedido") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)

insert into producto(nombre, descripcion, precio, imagen)
values ('camiseta', 'Camiseta de manga corta.', 15.5, 'img/camiseta.jpg'),
       ('pantalón', 'Pantalon vaquero', 30, 'img/pantalon.jpg'),
       ('chaqueta', 'Chaqueta de cuero.', 47.75, 'img/chaqueta.jpg'),
       ('zapatos', 'Zapatos negros', 100, 'img/zapatos.jpg');

insert into cliente(dni, nombre)
values ('11111111A', 'Daniel'),
       ('2222222B', 'Lucia'),
       ('3333333C', 'Beatrix');

insert into pedido(idCliente, fecha)
values (1, '2020-11-05 12:24:37'),
       (2, '2022-10-20 08:34:11');

insert into lineaPedido(idPedido, idProducto, cantidad)
values (1, 1, 3),
       (1, 2, 6),
       (2, 2, 10),
       (2, 3, 5),
       (2, 4, 5);

```

Crea un proyecto con JPA y Hibernate tenga las siguientes entidades:

- **Producto**: nombre no nulo. La imagen como `bytea`.
- **Cliente**: dni y nombre no nulo.
- **Pedido**: fecha no nula.
- **Línea Pedido**: cantidad de tipo entero corto y no nula.

Haz que el producto tenga la imagen guardada en la base de datos, no como cadena, de tipo `bytea`. Los pedidos deben estar ordenados por fecha y las líneas de pedido por cantidad.

- **Producto** dispone de una colección de elementos con los comentarios del pedido.

- **LíneaPedido** debe mapearse como una colección de elementos. Comprueba el resultado y hazlo como entidad.
- **LíneaPedido** debe tener una colección de tags.

Las relaciones deben actualizarse y borrarse en cascada.

## 4. Pedidos

Para este ejercicio usaremos la **base de datos MariaDB definida en el script** `bd-pedidos.sql`. Deberás crear un proyecto JPA con EclipseLink que mapee las tablas de la base de datos, empleando las entidades del ejercicio anterior, pero mapeadas con EclipseLink.

Crea en el mismo archivo de persistencia, una nueva unidad de persistencia que se **conecte a la base de datos de MariaDB con EclipseLink**.

**Migra los datos de la base de datos de PostgreSQL a la de MariaDB**, si es que no lo has hecho en el ejercicio anterior.

La aplicación debe permitir hacer lo siguiente:

- Mostrar todos los productos de la base de datos.
- Mostrar todos los pedidos de un cliente.
- Añadir un pedido.
- Borrar un pedido.

Para ello, crea una **clase AppPedidos** con un menú que permita realizar las operaciones anteriores y una **clase DAO para cada entidad**.

La **clase genérica DAO<T, K >** recoge el tipo de objeto, el tipo de la clave primaria, que tenga los métodos comunes a todas las entidades. La clase DAO genérica debe tener como atributo un `EntityManager`. La clase DAO debe tener los métodos necesarios para realizar las operaciones anteriores, así como un **atributo de tipo EntityManager**.

## 5. Base de datos de legislación

Propiedades de la conexión a la base de datos

- **URL:** `jdbc:mariadb://dbalumnos.sanclemente.local:3312/Lexislacion`
- **DRIVER:** `org.mariadb.jdbc.Driver`
- **USUARIO:** `lexislacionuser`
- **PASSWORD:** `ABC123..`

Esquema de la base de datos

- Los **nombres de las tablas son en CamelCase, así como los nombres de los atributos**. Ten en cuenta que muchos atributos no coinciden con los de las entidades.
- Las entidades/enumeraciones que debes implantar están en amarillo.
- **Publicacion** no es una entidad, es una enumeración, de la aplicación, por lo que no debe ser implantada como entidad (el idPublicacion en Norma coindice con el índice de la enumeración más 1: DOG, BOE, DOCE).



Tabla de la base de datos

## 1. JPAUtil

Clase que **implanta el patrón Singleton con doble comprobación para obtener el objeto de tipo EntityManager**.

```
public static EntityManagerFactory getEmFactory(String unidadPersistencia)
public static EntityManager getEntityManager()
```

## 2. Clases del modelo

- Las **relaciones de la entidad** `Norma` con `RangoLegal` y `Organismo` **son unidireccionales** (`RangoLegal`, `Organismo` no tienen referencia en las normas, `Publicacion` es una enumeración).

### RangoLegal

`idRangoLegal` (`Integer`), `nomeG` (`String`), `nomeC` (`String`). Los nombres de los atributos `nomeG` y `nomeC` no coinciden con los de la base de datos y tienen tamaño 128, además, son únicos. La clave primaria es auto numérica. Sin relaciones directas.

### Organismo

`idOrganismo` (`Integer`), `nome` (`String`). El nombre es único. La clave primaria es autonumérica. Sin relaciones directas.

### Publicacion (enumeración) y PublicacionConverter

Enumeración con 3 valores: **DOG, BOE, DOCE**, en este orden y atributos llamados `idPublicacion`, `descripcion`.

Implanta una **clase `PublicacionConverter`** para que el mapeo correcto de los valores de la enumeración en la columna `idPublicacion`:

```
public class PublicacionConverter implements AttributeConverter<Publicacion, Integer> {
}
```

### Clasificacion

`idClasificacion` (`Integer`), `nomeG` (`String`), `nomeC` (`String`). La clave primaria es autonumérica y los nombres de los atributos no coinciden con los de la base de datos, además, son únicos. Relaciones:

- `Norma`, pues hay **una tabla intermedia** `ClasificacionNorma` (fíjate que dicha entidad, `ClasificacionNorma`, no se implanta, por lo que es una relación muchos a muchos). La instanciación debe ser "perezosa" con todas las operaciones en cascada.

### Norma

`idNorma` (`Integer`), `publicacion` (`Publicacion`, enumeración), `numeroPublicacion` (`Integer`), `numeroPaxina` (`Integer`), `titulo` (`String`), `dataNorma` (`LocalDate`), `dataPublicacion` (`LocalDate`), `derogada` (`boolean`). Ten en cuenta que el título es un texto largo (`Clob`) a la hora de mapear. Relaciones con:

- Organismo** (unidireccional)
- RangoLegal** (unidireccional)
- DocumentoNorma**: el propietario de la relación es DocumentoNorma.
- Clasificacion** hay una tabla intermedia `ClasificacionNorma` (fíjate que dicha entidad, `ClasificacionNorma`, no se implanta y que una `Norma` puede tener muchas clasificaciones y viceversa). La instanciación debe ser "perezosa" con todas las operaciones en cascada.

### Documento

`idDocumento` (`Integer`), `mimeType` (`String`), `extension` (`String`), `titulo` (`String`), `titulo` (`String`), `documento` (`byte[]`), `tamano` (`Integer`), `idioma` (`String`). Especifica tamaño de los atributos de la tabla. Además, el

documento es BLOB y con instanciación perezosa. Relaciones con:

- `DocumentoNorma`: el propietario de la relación es `DocumentoNorma`.

## DocumentoNorma

`idDocumentoNorma` (`IdDocumentoNorma`), `numero` (`Integer`). Ten en cuenta que la clave es compuesta y debes crear una clase `IdDocumentoNorma` que representa a la clave compuesta. Relaciones con:

- `Norma`. Mapea la clave.
- `Documento`. Mapea la clave.

## 3. DTO y Consultas

En la clase `AppConsultas` realiza las siguientes **consultas en JPQL**. Ten en cuenta que las **consultas JPQL son mucho más sencillas y sólo incorporan la condición de JOIN, el ON, para entidades no relacionadas**.

1. Liste las clasificaciones y la cantidad de normas que contienen (incluidos los que no tienen). Debe devolver en nombre de la clasificación (`nombreG`), el número de normas (puede ser 0) y el `idClasificacion`.

Ejemplo de resultado:

```
ACTIVIDADES CIENTÍFICAS E EDUCATIVAS [30 normas] idClasificacion: 1
ACTIVIDADES INDUSTRIALIS [2 normas] idClasificacion: 2
AGRICULTURA ECOLÓXICA [6 normas] idClasificacion: 3
```

```
SELECT C.nome_g, Count(N.idNorma), C.idClasificacion FROM Norma AS N RIGHT JOIN (Clasificacion AS C LEFT JOIN ClasificacionNorma AS CN ON C.idClasificacion = CN.idClasificacion) ON N.idNorma = CN.idNorma GROUP BY C.nome_g, C.idClasificacion ORDER BY 1;
```

2. Liste los rangos legales y la cantidad de normas que contienen. Debe devolver el nombre de `RangoLegal` (`nombreG`), la `cantidad` (puede ser 0) y el `idRangoLegal`.

```
SELECT R.nome_g, Count(N.idNorma), R.idRangoLegal FROM RangoLegal R LEFT JOIN Norma N ON R.idRangoLegal = N.idRangoLegal GROUP BY R.nome_g, R.idRangoLegal ORDER BY R.idRangoLegal ASC;
```

3. Dada la clase `NormaDTO` del proyecto, realiza una consulta que pida el `idRango` y muestre las normas, de tipo `NormaDTO`, con ese `idRangoLegal` (por ejemplo, `idRangoLegal` igual a 11). La clase `NormaDTO` tiene los campos `idNorma`, `titulo`, `dataNorma`, `dataPublicacion`, `derogada`.

4. En la Entidad `Norma`, crea dos consultas con nombre, llamadas `Norma.findByTitulo` y `Norma.countByTitulo`, que devuelvan las normas a partir de un título recogido por parámetro. Haz uso de ellas.

Crea una **interface** `DAO<T, K>`, que recoge el tipo de objeto, el tipo de la clave primaria:

```
import java.util.List;

public interface DAO <T, K>{

    void save(T t);
    void delete(T t);
    T get(K k);
    void update(T t);
    List<T> findAll();
    List<T> findByTituloContaining(String titulo, int offset, int limit);
    List<T> findByTituloContaining(String titulo);
    int countAll();
    int countByTitulo(String titulo);

}
```

## 5. Paginación de Normas

Se trata de realizar una aplicación que permita consultar las Normas de la base de datos por nombre (pide la introducción de un texto) y **muestre las normas de la base de datos de 10 en 10**. Se debe mostrar las NormasDTO.

Se debe **poder avanzar y retroceder en la paginación. Se debe mostrar el número de página actual y el número total de páginas.**

La aplicación debe ser una **aplicación de consola**, con un menú que permita avanzar y retroceder en la paginación.

- Crea una clase `NormaDTO` que tenga los campos `idNorma`, `titulo`, `dataNorma`, `dataPublicacion`, `derogada`.
- Crea una clase `NormaDAO` que tenga un método que **devuelva el número total de películas y otro que devuelva las lista de películas de una página concreta, ordenadas por año descendente.**

## 4. NormaDAO y NormaRepository

NormaDAO implements DAO<Norma, Integer>

Implementación mediante patrón DAO de las operaciones con la **entidad Norma**. Dispone de un **atributo privado y final, de tipo EntityManager, em**, para referenciar al gestor de entidades, y **un constructor que recoge la el objeto de este tipo.**

Implantación de los cuatro métodos de la interfaz:

- `T get(K k)`: devuelve la norma con esa clave.
- `List<T> findByCadenaContaining(String titulo)`: haciendo uso de la consulta con nombre `Norma.findByTitulo` consulta las normas que contienen ese título.
- `List<T> findByCadenaContaining(String titulo, int offset, int limit)`: haciendo uso de la consulta con nombre "Norma.findByTitulo" consulta las normas que contienen ese título y devuelve `limit elementos empezando en la posición offset`.
- `int countByTitulo(String titulo)`: haciendo uso de la consulta con nombre `Norma.countByTitulo`, devuelve el número de normas que contiene ese título.

Comprueba el funcionamiento en la clase `AppConsultas`.

## NormaRepository

Repositorio de String Data JPA, que, además, contiene dos métodos más de los del JpaRepository:

- Un método que **devuelve la lista de Normas que contiene un título** (como en el caso anterior).
- Un método que **devuelve el número de normas que contienen el título recogido**.

Comprueba el funcionamiento dentro del, creando un método `testData` dentro de la clase `LexislacionApplication`.

## 5. Servicio Rest con Spring Boot Data JPA

Crea una aplicación que accede a datos JPA relacionales a través de una interfaz frontal RESTful basada en Web contra la base de datos de legislación en PostgreSQL. Puedes consultar la documentación de Spring Boot Data JPA en <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference>, así como la configuración del archivo properties en <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#data-properties>.

La aplicación debe permitir realizar las siguientes operaciones:

- **Listar todas las normas.**
- **Listar todas las normas que contienen un título.**
- **Listar el número de normas que contienen un título.**

## 6. Servicio Rest con Spring Boot Data JPA y paginación

- Listar todas las normas que contienen un título, de 10 en 10.
- Listar todas las normas que contienen un título, de 10 en 10, a partir de una página concreta.

Realiza las pruebas con Postman ;-)

Modifica la aplicación para que la paginación se realice con el método `findAll` de la interfaz `PagingAndSortingRepository`.

Paginación de Normas: modifica la aplicación para que permita consultar las Normas de la base de datos por nombre (pide la introducción de un texto) y muestre las normas de la base de datos de 10 en 10. Se debe mostrar las NormasDTO.

Se debe poder avanzar y retroceder en la paginación. Se debe mostrar el número de página actual y el número total de páginas.

La aplicación debe ser una aplicación de consola, con un menú que permita avanzar y retroceder en la paginación.

- Crea una clase `NormaDTO` que tenga los campos `idNorma`, `titulo`, `dataNorma`, `dataPublicacion`, `derogada`.
- Crea una clase `NormaDAO` que tenga un método que devuelva el número total de películas y otro que devuelva las lista de películas de una página concreta, ordenadas por año descendente.

Nota: para la realización de una aplicación de consola en Spring Boot, puedes seguir el siguiente tutorial:  
<https://www.baeldung.com/spring-boot-console-app>.

Existen varias formas de hacerlo:

- Usando `CommandLineRunner`: implementa la interfaz `CommandLineRunner` y sobreescribe el método `run`.  
 Ejemplo:

```
@SpringBootApplication
public class MyApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

    @Override
    public void run(String... args) {
        // Aquí va el código de la aplicación
    }
}
```

- Usando `ApplicationRunner`: implementa la interfaz `ApplicationRunner` y sobreescribe el método `run`.

Ejemplo:

```
@SpringBootApplication
public class MyApplication implements ApplicationRunner {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments args) {
        // Aquí va el código de la aplicación
    }
}
```

- Usando un `@Component`: crea una clase anotada con `@Component` y un método anotado con `@PostConstruct`:

```
@Component
public class MiComponente {

    @PostConstruct
    public void init() {
        // Aquí va el código de la aplicación
    }
}
```

- **Usando un `@Bean`:** crea un método anotado con `@Bean` en una clase de configuración. El Bean debe devolver un `CommandLineRunner` o un `ApplicationRunner`:

```
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

    @Bean
    public CommandLineRunner run() {
        return args -> {
            // Aquí va el código de la aplicación
        };
    }
}
```

## Ejecutores de código al inicio de la Aplicación

Para la realización de una aplicación de consola en Spring Boot, **es necesario crear un proyecto de Spring Boot** y **modificar la clase principal de la aplicación** para que sea una aplicación de consola.

```
package com.miccompanhia.miproyecto;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Existen varias formas de hacerlo:

- **Usando `CommandLineRunner`:** implementa la interfaz `CommandLineRunner` y sobreescribe el método `run`. Ejemplo:

```
@SpringBootApplication
public class MiAplicacion implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(MiAplicacion.class, args);
    }

    @Override
    public void run(String... args) {
        // Aquí va el código de la aplicación
    }
}
```

- **Usando `ApplicationRunner (Interfaz Funcional)`:** implementa la interfaz `ApplicationRunner` y sobreescribe el método `run`.

Ejemplo:

```
@SpringBootApplication
public class MiAplicacion implements ApplicationRunner {

    public static void main(String[] args) {
        SpringApplication.run(MiAplicacion.class, args);
    }

    @Override
    public void run(ApplicationArguments args) {
        // Aquí va el código de la aplicación
    }
}
```

- **Usando un `@Component`:** crea una clase anotada con `@Component` y un método anotado con `@PostConstruct`:

```
@Component
public class MiComponente {

    @PostConstruct
    public void init() {
        // Aquí va el código de la aplicación
    }
}
```

`@Component` es una anotación que marca una clase como un componente de Spring. Spring escaneará las clases anotadas con `@Component` y las registrará en el contexto de la aplicación.

`@PostConstruct` es una anotación que se utiliza en un método que debe ejecutarse después de que se haya completado la construcción de un bean. Spring ejecutará el método anotado con `@PostConstruct` después de que se haya creado el bean.

- **Usando un `@Bean`:** crea un método anotado con `@Bean` en una clase de configuración. El Bean debe devolver un `CommandLineRunner` o un `ApplicationRunner`:

```
@SpringBootApplication
public class MiAplicacion {

    public static void main(String[] args) {
        SpringApplication.run(MiAplicacion.class, args);
    }

    @Bean
    public CommandLineRunner run() {
        return args -> {
            // Aquí va el código de la aplicación
        };
    }
}
```

`@Bean` es una anotación que marca un método como un productor de un bean administrado por Spring. Spring llamará al método anotado con `@Bean` para crear el bean y lo registrará en el contexto de la aplicación.

## Diferencias entre ejecutores de código al inicio de la Aplicación

Estos ejecutores se utilizan para ejecutar la lógica al iniciar la aplicación:

- `ApplicationRunner` (Interfaz Funcional) con el método `run`.

`ApplicationRunner run()` se ejecutará justo después de que se cree el `ApplicationContext` y antes de que inicie la aplicación Spring Boot.

`ApplicationRunner` recoge `ApplicationArguments`, que tiene métodos como `getOptionNames()`, `getOptionValues()` y `getSourceArgs()`.

- `CommandLineRunner` también es una Interfaz Funcional con el método `run`.

`CommandLineRunner run()` se ejecutará justo después de que se cree el `ApplicationContext` y antes de que inicie la aplicación Spring Boot.

Acepta los **argumentos como un array de String** que se pasan en el momento del inicio del servidor.

Ambos proporcionan la misma funcionalidad y **la única diferencia entre `CommandLineRunner` y `ApplicationRunner` es que `CommandLineRunner.run()` acepta un array de `String[]`, mientras que `ApplicationRunner.run()` acepta `ApplicationArguments` como argumento.**

Puedes encontrar más información con ejemplos en la [Guía para Ejecutar Lógica en el Inicio en Spring](#).

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025

# DEPENDENCIAS MAVEN.

---

- [1. Dependencias Maven.](#)
  - [2. Logging](#)
  - [2. Json](#)
    - [2.1 Gson](#)
    - [2.2. Jackson Databind](#)
    - [2.3. Jackson Core](#)
  - [3. JUnit](#)
  - [4. Drivers JDBC](#)
    - [4.1. H2](#)
    - [4.2. SQLite JDBC Driver](#)
    - [4.3. PostgreSQL JDBC Driver](#)
    - [4.4. MySQL Connector/J](#)
    - [4.5. HyperSQL Database \(HSQLDB\)](#)
  - [5. Dependencias para JPA](#)
    - [5.1. Jakarta Persistence API \(JPA\)](#)
    - [5.2. Hibernate](#)
    - [5.3. EclipseLink](#)
  - [6. Dependencias para Spring](#)
    - [6.1. Spring Core](#)
    - [6.2. Spring Boot](#)
    - [6.3. Spring Data JPA](#)
    - [6.4. Spring Boot Starter Data JPA](#)
  - [7. Lenguajes sobre JVM](#)
    - [7.1. Kotlin](#)
    - [7.2. Scala](#)
- [Referencias](#)

## 1. Dependencias Maven.

### 2. Logging

**SLF4J (The Simple Logging Facade for Java)** es una fachada o interfaz para varios sistemas de registro de eventos (logging) en Java. Permite a los desarrolladores cambiar de sistema de registro de eventos en tiempo de ejecución sin tener que modificar el código fuente. Para más información, visita la página oficial de SLF4J: <http://www.slf4j.org/>

<https://mvnrepository.com/artifact/org.slf4j/slf4j-api> <https://central.sonatype.com/artifact/org.slf4j/slf4j-api>

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>2.0.16</version>
</dependency>
```

Además, precisamos alguna implementación de SLF4J. En este caso vamos a usar Logback:

<https://mvnrepository.com/artifact/ch.qos.logback/logback-classic>  
<https://central.sonatype.com/artifact/ch.qos.logback/logback-classic>

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
```

```
<dependency>
<version>1.5.16</version>
</dependency>
```

## 2. Json

### 2.1 Gson

Gson es una biblioteca Java que se utiliza para **convertir objetos Java en su representación JSON**. También puede ser utilizado para convertir una cadena JSON en un objeto Java equivalente. Gson es una biblioteca de código abierto desarrollada por Google. Puedes encontrar más información en la página oficial de Gson: <https://github.com/google/gson>

<https://mvnrepository.com/artifact/com.google.code.gson/gson>  
<https://central.sonatype.com/artifact/com.google.code.gson/gson>

```
<dependency>
<groupId>com.google.code.gson</groupId>
<artifactId>gson</artifactId>
<version>2.11.0</version>
</dependency>
```

### 2.2. Jackson Databind

Jackson es una biblioteca Java de código abierto para **convertir objetos Java en su representación JSON y viceversa**. Jackson es una de las bibliotecas de serialización y deserialización JSON más populares en Java. Puedes encontrar más información en la página oficial de Jackson: <https://github.com/FasterXML/jackson>

<https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind>  
<https://central.sonatype.com/artifact/com.fasterxml.jackson.core/jackson-databind>

```
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.18.2</version>
</dependency>
```

### 2.3. Jackson Core

Jackson Core es una biblioteca Java de código abierto para **procesar JSON** (Stream API). Jackson Core proporciona las clases básicas para trabajar con JSON, como `JsonNode`, `JsonParser` y `JsonGenerator`. Puedes encontrar más información en la página oficial de Jackson: <https://github.com/FasterXML/jackson-core>

<https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core>  
<https://central.sonatype.com/artifact/com.fasterxml.jackson.core/jackson-core>

```
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-core</artifactId>
<version>2.18.2</version>
</dependency>
```

## 3. JUnit

JUnit es un framework open-source que se utiliza para **realizar pruebas unitarias en Java**. JUnit es una herramienta importante en el desarrollo de software, ya que permite a los desarrolladores probar su código de manera eficiente y asegurarse de que funciona correctamente. Puedes encontrar más información en la página oficial de JUnit: <https://junit.org/junit5/>

<https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api>  
<https://central.sonatype.com/artifact/org.junit.jupiter/junit-jupiter-api>

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.11.4</version>
  <scope>test</scope>
</dependency>
```

Ejemplo de uso:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyTest {

    @Test
    public void test() {
        assertEquals(2, 1 + 1);
    }
}
```

## 4. Drivers JDBC

Para trabajar con bases de datos, necesitamos los drivers JDBC correspondientes.

### 4.1. H2

H2 es una base de datos relacional escrita en Java. Es muy rápida, de código abierto y **se puede ejecutar en modo embebido o en modo servidor**. Además, admite transacciones, encriptación, funciones de usuario o procedimientos almacenados. Además, puede **almacenarse en memoria o en disco**.

Puedes encontrar más información en la página oficial de H2: <http://www.h2database.com/>

<https://mvnrepository.com/artifact/com.h2database/h2> <https://central.sonatype.com/artifact/com.h2database/h2>

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.3.232</version>
</dependency>
```

Es importante hacer notar que las incompatibilidades entre versiones diferentes de H2, por lo que se recomienda tener control sobre qué versión se está utilizando.

URL: `jdbc:h2:mem:testdb` (base de datos en memoria) Driver: `org.h2.Driver` URL (fichero):  
`jdbc:h2:rutaALaBaseDatos;DATABASE_TO_UPPER=false` (base de datos en fichero)

El Driver JDBC para H2 hace la conversión automática de los nombres de las tablas y columnas a mayúsculas, por lo que si queremos conservar los nombres originales, debemos añadir `DATABASE_TO_UPPER=false` a la URL de conexión.

### 4.2. SQLite JDBC Driver

SQLite es una **base de datos relacional embebida, que no requiere un servidor**. Es muy ligera y rápida, y se puede utilizar en aplicaciones de escritorio, móviles o en la web. Puedes encontrar más información en la página oficial de SQLite: <https://www.sqlite.org/index.html>

Existen varias implementaciones de SQLite en Java, pero vamos a usar Xerial SQLite JDBC Driver:

<https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc> <https://central.sonatype.com/artifact/org.xerial/sqlite-jdbc>

```
<dependency>
  <groupId>org.xerial</groupId>
  <artifactId>sqlite-jdbc</artifactId>
  <version>3.48.0.0</version>
</dependency>
```

URL: `jdbc:sqlite:rutaALaBaseDatos` (base de datos en fichero) Driver: `org.sqlite.JDBC`

Existen otras API para SQLite, como las versiones originales de androidx:

<https://developer.android.com/jetpack/androidx/releases/sqlite>, pero dicha versión no es compatible con Java SE y se usaba antiguamente para android, antes de la aparición de Room.

## 4.3. PostgreSQL JDBC Driver

PostgreSQL es un **sistema de gestión de bases de datos relacional de código abierto y muy potente**.

Puedes encontrar más información en la página oficial de PostgreSQL: <https://www.postgresql.org/>

<https://mvnrepository.com/artifact/org.postgresql/postgresql>

<https://central.sonatype.com/artifact/org.postgresql/postgresql>

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.5</version>
</dependency>
```

URL: `jdbc:postgresql://localhost:5432/nombredelabasededatos` Driver: `org.postgresql.Driver`

El usuario y la contraseña se pasarán como parámetros en la URL de conexión:

```
String url = "jdbc:postgresql://localhost:5432/nombredelabasededatos";
String user = "usuario";
String password = "contraseña";
Connection conn = DriverManager.getConnection(url, user, password);
```

Si queremos añadirlos a la URL:

```
String url = "jdbc:postgresql://localhost:5432/nombredelabasededatos?
user=usuario&password=contraseña";
Connection conn = DriverManager.getConnection(url);
```

## 4.4. MySQL Connector/J

MySQL Connector/J es un **controlador JDBC Tipo 4**, lo que significa que es una implementación Java pura del protocolo MySQL y no depende de las bibliotecas de cliente MySQL. Como los anteriores, este controlador admite el registro automático con DriverManager, lo que significa que no es necesario cargar explícitamente el controlador.

<https://mvnrepository.com/artifact/com.mysql/mysql-connector-j>

<https://central.sonatype.com/artifact/com.mysql/mysql-connector-j>

```
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <version>9.1.0</version>
</dependency>
```

URL: `jdbc:mysql://localhost:3306/nombredelabasededatos` Driver: `com.mysql.cj.jdbc.Driver`

La URL puede recoger parámetros, como:

```
String url = "jdbc:mysql://localhost:3306/nombredelabasededatos?user=usuario&password=contraseña";
Connection conn = DriverManager.getConnection(url);
```

## 4.5. HyperSQL Database (HSQLDB)

HSQLDB es una base de datos relacional escrita en Java. Es muy rápida, de código abierto y **se puede ejecutar en modo embebido o en modo servidor**: <https://hsqldb.org/>

<https://mvnrepository.com/artifact/org.hsqldb/hsqldb> <https://central.sonatype.com/artifact/org.hsqldb/hsqldb>

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.7.4</version>
</dependency>
```

URL: `jdbc:hsqldb:mem:testdb` (base de datos en memoria) Driver: `org.hsqldb.jdbc.JDBCDriver` URL para servidor: `jdbc:hsqldb:hsqldb://localhost/testdb` URL para fichero: `jdbc:hsqldb:file:nombredelabasededatos`

## 5. Dependencias para JPA

### 5.1. Jakarta Persistence API (JPA)

La **Java Persistence API (JPA)** es una **especificación** de Java que describe la gestión de la persistencia de los objetos en las aplicaciones Java. JPA define un conjunto de interfaces y anotaciones que permiten a los desarrolladores mapear objetos Java a tablas de bases de datos y viceversa. Puedes encontrar más información en la página oficial de JPA:

- <https://jakarta.ee/specifications/persistence/>
- <https://github.com/jakartaee/persistence>
- Javadoc: <https://jakartaee.github.io/persistence/latest/api/jakarta.persistence/module-summary.html>
- 

<https://mvnrepository.com/artifact/jakarta.persistence/jakarta.persistence-api>

<https://central.sonatype.com/artifact/jakarta.persistence/jakarta.persistence-api>

JPA 3.1:

```
<dependency>
  <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
  <version>3.1.0</version>
</dependency>
```

JPA 3.2:

```
<dependency>
  <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
  <version>3.2.0</version>
</dependency>
```

## 5.2. Hibernate

**Hibernate** es un **framework de mapeo objeto-relacional (ORM)** para Java. Hibernate simplifica el desarrollo de aplicaciones Java que interactúan con bases de datos relacionales. Puedes encontrar más información en la página oficial de Hibernate: <https://hibernate.org/>

<https://mvnrepository.com/artifact/org.hibernate/hibernate-core>  
<https://central.sonatype.com/artifactory/org.hibernate/hibernate-core>

La versión compatible con JPA 3.1 es la versión 6:

```
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>6.6.5.Final</version>
</dependency>
```

La versión compatible con JPA 3.2 es la versión 7, que todavía está en desarrollo:

```
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>7.0.0.Beta3</version>
</dependency>
```

Pronto se lanzará la versión final de Hibernate 7, que será compatible con JPA 3.2. Esperamos.

## 5.3. EclipseLink

EclipseLink es otro **framework de mapeo objeto-relacional (ORM)** para Java. EclipseLink es una implementación de la especificación JPA y proporciona una serie de características avanzadas, como el mapeo de herencia, el mapeo de tablas, el mapeo de relaciones y la consulta de objetos. Puedes encontrar más información en la página oficial de EclipseLink: <https://www.eclipse.org/eclipselink/>

<https://mvnrepository.com/artifact/org.eclipse.persistence/eclipselink>  
<https://central.sonatype.com/artifactory/org.eclipse.persistence/eclipselink>

La versión compatible con JPA 3.1 es la versión 4:

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>4.0.5</version>
</dependency>
```

La versión compatible con JPA 3.2 es la versión 5, que todavía está en desarrollo:

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>5.0.0-B05</version>
</dependency>
```

Pronto se lanzará la versión final de EclipseLink 5, que será compatible con JPA 3.2. Esperamos.

## 6. Dependencias para Spring

### 6.1. Spring Core

Spring Core es el **núcleo del framework Spring**. Proporciona las funcionalidades básicas de Spring, como la **inyección de dependencias** y la **gestión de transacciones**. Puedes encontrar más información en la página

oficial de Spring: <https://spring.io/projects/spring-framework>

<https://mvnrepository.com/artifact/org.springframework/spring-core>

<https://central.sonatype.com/artifact/org.springframework/spring-core>

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>6.2.2</version>
</dependency>
```

## 6.2. Spring Boot

Spring Boot es un **proyecto de Spring que simplifica el desarrollo de aplicaciones Java**. Proporciona una serie de características, como la **configuración automática**, el **embebido de servidores**, la **gestión de dependencias** y la **creación de aplicaciones ejecutables**. Puedes encontrar más información en la página oficial de Spring Boot: <https://spring.io/projects/spring-boot>

<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot>

<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot>

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot</artifactId>
  <version>3.4.1</version>
</dependency>
```

Spring Boot Starter es una colección de dependencias que se utilizan comúnmente en las aplicaciones Spring Boot. Puedes encontrar más información en la página oficial de Spring Boot: <https://spring.io/projects/spring-boot>

<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter>

<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot-starter>

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <version>3.4.1</version>
</dependency>
```

## 6.3. Spring Data JPA

Spring Data JPA es un **proyecto de Spring que simplifica el acceso a datos en aplicaciones Java**.

Proporciona una serie de características, como la **creación de repositorios**, la **generación de consultas** y la **gestión de transacciones**. Puedes encontrar más información en la página oficial de Spring Data JPA:

<https://spring.io/projects/spring-data-jpa>

<https://mvnrepository.com/artifact/org.springframework.data/spring-data-jpa>

<https://central.sonatype.com/artifact/org.springframework.data/spring-data-jpa>

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-jpa</artifactId>
  <version>3.4.2</version>
</dependency>
```

## 6.4. Spring Boot Starter Data JPA

Spring Boot Starter Data JPA es una colección de dependencias que se utilizan comúnmente en las aplicaciones Spring Boot que utilizan Spring Data JPA. Puedes encontrar más información en la página oficial de Spring Boot: <https://spring.io/projects/spring-boot>

<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa>  
<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa>

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>3.4.1</version>
</dependency>
```

## 7. Lenguajes sobre JVM

### 7.1. Kotlin

Kotlin es un lenguaje de programación moderno y conciso que se ejecuta sobre la JVM. Kotlin es interoperable con Java, lo que significa que puedes utilizar las bibliotecas de Java en Kotlin y viceversa. Puedes encontrar más información en la página oficial de Kotlin: <https://kotlinlang.org/>

IDEs como IntelliJ IDEA o Android Studio soportan Kotlin de forma nativa, pero también puedes usar Kotlin en otros IDE añadiendo las dependencias necesarias.

<https://mvnrepository.com/artifact/org.jetbrains.kotlin/kotlin-stdlib>  
<https://central.sonatype.com/artifact/org.jetbrains.kotlin/kotlin-stdlib>

```
<dependency>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-stdlib</artifactId>
  <version>2.1.0</version>
</dependency>
```

### 7.2. Scala

Scala es un lenguaje de programación funcional y orientado a objetos que se ejecuta sobre la JVM. **Scala es interoperable con Java, lo que significa que puedes utilizar las bibliotecas de Java en Scala y viceversa.** Puedes encontrar más información en la página oficial de Scala: <https://www.scala-lang.org/>

IDEs como IntelliJ IDEA o Eclipse soportan Scala de forma nativa, pero también puedes usar Scala en otros IDE añadiendo las dependencias necesarias.

[https://mvnrepository.com/artifact/org.scala-lang/scala3-library\\_3](https://mvnrepository.com/artifact/org.scala-lang/scala3-library_3) [https://central.sonatype.com/artifact/org.scala-lang/scala3-library\\_3](https://central.sonatype.com/artifact/org.scala-lang/scala3-library_3)

```
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala3-library_3</artifactId>
  <version>3.6.3</version>
</dependency>
```

## Referencias

- [Maven Repository](#)
- [Central Sonatype Repository](#)

- [SLF4J](#)
- [Gson](#)
- [Spring](#)

---

👤 Autor/a: Pepinho 📅 Última actualización: 13.02.2025