UD 3. Object/Relational Mapping (ORM). JPA e Hibernate

* [Introducción ORM](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/#introducci%C3%B3n-orm)
* [Persistencia en Sistemas de Bases de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/#persistencia-en-sistemas-de-bases-de-datos)
* [Técnicas de persistencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/#t%C3%A9cnicas-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 otross.

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**: ae 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](https://mariadb.com/).
  + [PostgreSQL](https://www.postgresql.org/).
  + [Oracle](https://www.oracle.com/es/database/).
  + [MySQL](https://www.mysql.com/).
  + [H2 Database Engine](https://www.h2database.com/html/main.html).
  + [HSQLDB](http://hsqldb.org/).
  + [SQLite](https://www.sqlite.org/index.html).
  + [SQL Server](https://www.microsoft.com/es-es/sql-server/sql-server-2019).
  + [DB2](https://www.ibm.com/es-es/analytics/db2)
  + [Access](https://www.microsoft.com/en-us/microsoft-365/access).
  + [Derby](https://db.apache.org/derby/).
  + [DB2](https://www.ibm.com/es-es/analytics/db2).
  + [Informix](https://www.ibm.com/es-es/products/informix).
  + [Firebird](https://firebirdsql.org/).
  + Otros: [Sybase](https://www.sap.com/spain/products/acquired-brands/what-is-sybase.html), [Teradata Database](https://www.teradata.com/resources/datasheets/teradata-database), [Ingres](https://www.actian.com/databases/ingres/). [Adabas D](https://www.softwareag.com/en_corporate/platform/adabas-natural.html), [Progress](https://www.progress.com/es-es/openedge). [InterBase](https://www.embarcadero.com/es/products/interbase).
* **Bases de datos orientadas a documentos**: se basan en el modelo de datos orientado a documentos.
  + [MongoDB](https://www.mongodb.com/es).
  + [Databricks](https://www.databricks.com/). 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](https://aws.amazon.com/es/dynamodb/).
  + [Azure Cosmos DB](https://azure.microsoft.com/es-es/services/cosmos-db/).
  + [Firebase Realtime Database](https://firebase.google.com/).
  + [Cloud Firestore](https://firebase.google.com/products/firestore?hl=es-419).
  + Otros SGBD documentales: [CouchDB](https://couchdb.apache.org/), [RavenDB](https://ravendb.net/), [Couchbase](https://www.couchbase.com/),[MarkLogic](https://www.marklogic.com/), [OrientDB](https://orientdb.com/), [ArangoDB](https://www.arangodb.com/). [RethinkDB](https://rethinkdb.com/), [Cosmos DB](https://azure.microsoft.com/es-es/services/cosmos-db/). [Amazon DocumentDB](https://aws.amazon.com/es/documentdb/), [Elasticsearch](https://www.elastic.co/es/).
* **Bases de datos orientadas a objetos**: se basan en el modelo orientado a objetos.  
  Los datos se almacenan en objetos. Ejemplos son:
  + [ObjectDB](https://www.objectdb.com/)
  + [db4o](https://sourceforge.net/projects/db4o/)
  + [Action NoSQL Databas](https://www.actian.com/databases/nosql/)
  + Otros: [Versant](https://www.versant.com/), [ZODB](https://www.zodb.org/en/latest/)…

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**](https://github.com/hibernate/hibernate-orm), un **framework de persistencia que implementa la especificación JPA**. Otras:
      * [EclipseLink](https://www.eclipse.org/eclipselink/).
      * [OpenJPA](https://openjpa.apache.org/).
      * [DataNucleus](https://datanucleus.org/).
      * [TopLink](https://www.oracle.com/middleware/technologies/toplink.html).
      * [Batoo JPA](https://batoo.jp/).
      * Kundera..

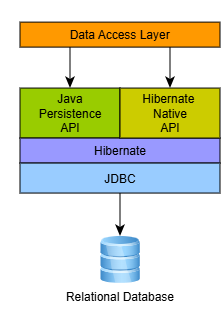
## Mapeo Objeto-Relacional (ORM)

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)**](https://jakarta.ee/specifications/persistence/)**con**[**Hibernate**](https://hibernate.org/), aunque también veremos alguna otra implementación de referencia de JPA, como [EclipseLink](https://www.eclipse.org/eclipselink/) o [DataNucleus](https://datanucleus.org/).  
**SpringBoot Data JPA utiliza Hibernate como implementación de JPA**, pero, en cuanto a rendimiento, no es la mejor opción.



# 01. Jakarta Persistence (JPA).

* [1. Jakarta Persistence](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#1-jakarta-persistence)
  + [1.1. Historia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#11-historia)
  + [1.2. Las versiones de Jakarta Persistence](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#12-las-versiones-de-jakarta-persistence)
    - [Java Persistence API 2.0 (2009)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#java-persistence-api-20-2009)
    - [Java Persistence 2.1 (2013)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#java-persistence-21-2013)
    - [Java Persistence 2.2 (2017)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#java-persistence-22-2017)
    - [Jakarta Persistence 3.0 (2020)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#jakarta-persistence-30-2020)
    - [Jakarta Persistence 3.1 (2021)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#jakarta-persistence-31-2021)
  + [1.3. Referencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#13-referencias)
* [2. Jakarta Persistence (JPA)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#2-jakarta-persistence-jpa)
  + [2.2. Implementaciones JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#22-implementaciones-jpa)
    - [Dependencias Maven](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#dependencias-maven)
  + [2.3 Fichero de configuración persistence.xml](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#23-fichero-de-configuraci%C3%B3n-persistencexml)
  + [2.3. Entidades/Entity](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#23-entidadesentity)
  + [2.4. Relaciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#24-relaciones)
  + [2.5. Tipos de relaciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/01jpa/#25-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](https://projects.eclipse.org/projects/ee4j.jpa/releases/3.2). 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://projects.eclipse.org/projects/ee4j.jpa/releases/3.2): <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.LocalTime, java.time.LocalDateTime, 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](https://jcp.org/aboutJava/communityprocess/maintenance/jsr338/ChangeLog-JakartaPersistence-2.2-MR.txt).

#### 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](https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html).

### 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](https://jakarta.ee/specifications/persistence/3.2/). **En desarrollo las implementaciones**.
* [Jakarta Persistence 3.1](https://jakarta.ee/specifications/persistence/3.1/)
* [Jakarta Persistence 3.0](https://jakarta.ee/specifications/persistence/3.0/)
* [Jakarta Persistence 2.2](https://jakarta.ee/specifications/persistence/2.2/)
* [Jakarta Persistence 1.0 (JSR 220)](https://jcp.org/en/jsr/detail?id=220)

## 2. Jakarta Persistence (JPA)

[Jakarta Persistence](https://jakarta.ee/specifications/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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/module-summary.html), 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](https://github.com/hibernate/hibernate-orm), [EclipseLink](https://www.eclipse.org/eclipselink/), [MyBatis](https://blog.mybatis.org/) (antes [IBatis](https://ibatis.apache.org/)), [DataNucleus](https://datanucleus.org/),.. que emplean la especificación de JPA.
* La [última versión con implementaciones estables es la 3.1](https://jakarta.ee/specifications/persistence/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](https://www.eclipse.org/eclipselink/) 4.0.0-M3 (o superior)
  + [Hibernate](https://github.com/hibernate/hibernate-orm) 6.0.0.Final (o superior)
  + [DataNucleus](https://datanucleus.org/)
* La mayoría de las herramientas ORM como [**Hibernate**](https://github.com/hibernate/hibernate-orm)**,**[**MyBatis**](https://blog.mybatis.org/)**(antes**[**IBatis**](https://ibatis.apache.org/)**) o**[**EclipseLink**](https://www.eclipse.org/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**.

**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](https://hibernate.org/): es una solución de Mapeo Objeto/Relacional (ORM) para programas escritos en Java y otros lenguajes que admiten la JVM.
* [EclipseLink](https://www.eclipse.org/eclipselink/): es una solución de persistencia de objetos para Java.
* [Spring Data JPA](https://spring.io/projects/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](https://openjpa.apache.org/): es una implementación de JPA que puede utilizarse como un almacén de datos independiente o como una extensión de Apache Geronimo.
* [Oracle TopLink](https://www.oracle.com/middleware/technologies/top-link.html): 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](https://datanucleus.org/): 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. 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 habría que indicar el proveedor de persistencia** [org.hibernate.jpa.HibernatePersistenceProvider](https://docs.jboss.org/hibernate/orm/6.4/javadocs/org/hibernate/jpa/package-summary.html), 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;&gt;-->

<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 6](https://docs.jboss.org/hibernate/orm/6.4/javadocs/org/hibernate/dialect/Dialect.html), 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. 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>

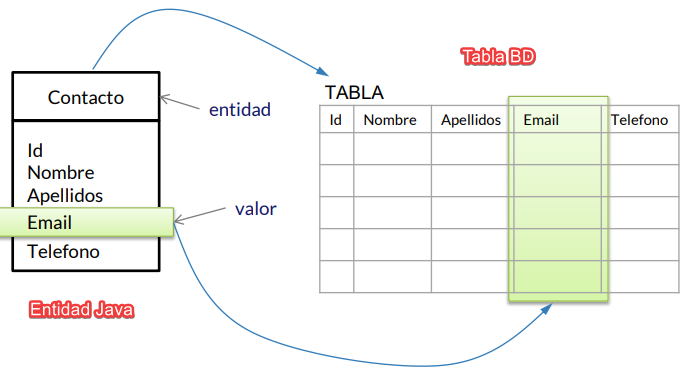
<artifactId>h2</artifactId>

<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”.

### 2.3. Entidades/Entity



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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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[]](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/table#uniqueConstraints())): 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.LocalTime
* java.time.LocalDateTime
* 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. 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. 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.

02. JPA vs Hibernate.

* [JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/02jpavshibernate/#jpa)
* [Hibernate](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/02jpavshibernate/#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 jakarta.persistence (+3.0) javax.persistence (2.3 o inferior). | Está descrito en el paquete org.hibernate. |
| 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 **interfaz EntityManagerFactory** para interactuar con la fábrica del administrador de entidades para la unidad de persistencia. | Utiliza la **interfaz SessionFactory** para crear instancias de sesión. |
| Utiliza la **interfaz EntityManager** para realizar acciones de crear, leer y eliminar para instancias de clases de entidad mapeadas. | Utiliza la **interfaz Session** 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.

# 03. Ejercicio básico de JPA.

* [1. Añadir dependencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#1-a%C3%B1adir-dependencias)
* [2. Creación del archivo de configuración persistence.xml](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#2-creaci%C3%B3n-del-archivo-de-configuraci%C3%B3n-persistencexml)
  + [2.1. Para Hibernate](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#21-para-hibernate)
  + [2.2. Para EclipseLink](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#22-para-eclipselink)
* [3. Creación de la clase de Entidad Usuario](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#3-creaci%C3%B3n-de-la-clase-de-entidad-usuario)
* [4. Creación del EntityManagerFactory y EntityManager](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#4-creaci%C3%B3n-del-entitymanagerfactory-y-entitymanager)
* [5. Creación del ejemplo de persistencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#5-creaci%C3%B3n-del-ejemplo-de-persistencia)
* [6. Creación de la clase TestUsuario](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#6-creaci%C3%B3n-de-la-clase-testusuario)
* [7. Creación de la clase UsuarioDAO](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/03jpaejercicio/#7-creaci%C3%B3n-de-la-clase-usuariodao)

## 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 getionar 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>

</persistence>

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.getTransaction().begin();

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;

}

}

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

Dado el modelo de la aplicación de JokeAPI, en la que tenemos enumeración Category y la clase Joke, vamos a crear una base de datos con JPA. Recuerda que la clase Joke/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;

//...

}

La enumeración Categoria es una enumeración con los siguientes valores:

public enum Categoria {

ANY("Any"),

MISC("Misc"),

PROGRAMMING("Programming"),

DARK("Dark"),

PUN("Pun"),

SPOOKY("Spooky"),

CHRISTMAS("Christmas");

//...

}

Flag es una enumeración con los siguientes valores:

public enum Flag {

EXPLICIT("Explicit"),

NSFW("NSFW"),

RELIGION("Religion"),

POLITICAL("Political"),

RACIST("Racist"),

SEXIST("Sexist");

//...

}

Lenguaje es una enumeración con los siguientes valores:

public enum Lenguaje {

CS("cs"),

DE("de"),

EN("en"),

ES("es"),

FR("fr"),

PT("pt");

//...

}

etc…

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.

 Ejercicio. 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** 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 EntityManagerUtil** para la gestión y obtención de los objetos de tipo EntityManager de una manera eficiente. Emplearemos el **patrón Singleton para el gestor EntityManagerUtil, 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.

**Estructura de la 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 única tabla, Book. 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 |
| ano | int | Año de publicación del libro |
| disponible | boolean | Indica si el libro está disponible |
| portada | Blob | Portada del libro en formato binario |

CREATE TABLE Book (

idBook INT NOT NULL AUTO\_INCREMENT,

isbn VARCHAR(13) NOT NULL,

titulo VARCHAR(255) NOT NULL,

autor VARCHAR(255) NOT NULL,

ano INT NOT NULL,

disponible BOOLEAN NOT NULL DEFAULT true,

portada BLOB,

PRIMARY KEY (id)

);

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"

**Clase EntityManagerUtil:**

Mediante el **patrón Singleton crea una clase EntityManagerUtil**, mediante el patrón Singleton de manera 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**.

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.

**Clase Book implementa Serializable:**

La clase debe tener los siguientes constructores que:

* Book()
* Book(String isbn, String titulo, String autor, int ano, boolean disponible)
* Book(Integer idBook, String isbn, String titulo, String autor, Integer ano, Boolean disponible, byte[] portada)
* Aquellos que consideres necesarios.

La clase Book debe tener los siguientes atributos:

* idBook: Long (autonumérico)
* isbn: String (tamaño 13)
* titulo: String
* autor: String
* ano: Integer
* disponible: Boolean
* portada: byte[]

Métodos de la clase Book:

* 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.

Debes implantar la gestión de sentencias de esta la clase BookDAO, de modo que tenga un objeto de tipo EntityManagercomo atributo (en sistemas empresariales, como la gestión de transacciones no se suele hacer por aplicación, 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**:

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);

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:**

Enumaración de los tipos admitidos de el Factory Method.

public enum TipoDAO {

H2\_JDBC, ORM, HIBERNATE, JSON, POSTGRES\_JDBC;

}

Implementa un método estático getBookDAO que recoge el tipo de DAO que se va a emplear y devuelve el objeto de tipo BookJPADAO.

**AppBiblioteca:**

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

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);

NUEVA VERSION DE LOS APUNTES:

**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** |
| --- | --- | --- |
| 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: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> instancies = 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, avaliable -> 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 EntityManagercomo 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

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.HashMap;

import java.util.Map;

import static com.pepinho.ad.biblioteca.model.BibliotecaLogger.LOG;

public class BibliotecaJpaManager {

public static final String BIBLIOTECA\_H2 = "bibliotecaH2";

public static final String BIBLIOTECA\_POSTGRES = "bibliotecaPostgres";

private static final Map<String, EntityManagerFactory> instancies = new HashMap<>();

private BibliotecaJpaManager() {

}

private static boolean isEntityManagerFactoryClosed(String unidadPersistencia) {

return !instancies.containsKey(unidadPersistencia) || instancies.get(unidadPersistencia) == null ||

!instancies.get(unidadPersistencia).isOpen();

}

public static EntityManagerFactory getEntityManagerFactory(String unidadPersistencia) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

synchronized (BibliotecaJpaManager.class) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

try {

instancies.put(unidadPersistencia, Persistence.createEntityManagerFactory(unidadPersistencia));

} catch (Exception e) {

LOG.error("Erro ó crear a unidade de persistencia " + unidadPersistencia +

": " + e.getMessage());

}

}

}

}

return instancies.get(unidadPersistencia);

}

public static EntityManager getEntityManager(String persistenceUnitName) {

return getEntityManagerFactory(persistenceUnitName).createEntityManager();

}

public static void close(String persistenceUnitName) {

if (instancies.containsKey(persistenceUnitName)) {

instancies.get(persistenceUnitName).close();

instancies.remove(persistenceUnitName);

}

}

}

 Solución: Book

import jakarta.persistence.\*;

import javax.imageio.ImageIO;

import java.awt.\*;

import java.util.ArrayList;

import java.util.List;

import java.io.\*;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.time.LocalDate;

import java.util.Objects;

/\*\*

\* @author pepecalo

\*/

@Entity

public class Book implements Serializable {

// private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long idBook;

@Column(length = 13, nullable = false, unique = true))

private String isbn;

@Column(name = "titulo", nullable = false)

private String title;

@Column(name = "autor")

private String author;

@Column(name = "anho")

private Short ano;

@Column(name = "disponible")

private Boolean available;

private byte[] portada;

private LocalDate dataPublicacion;

// @Transient // Ambas opciones son válidas

transient private List<Contido> contenido = new ArrayList<>();

private static final long serialVersionUID = 1L;

public Book() {

}

public Book(String title, String author, Short year, Boolean available) {

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

}

public Book(String isbn, String title, String author, Short year,

Boolean available) {

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

}

public Book(String isbn, String title, String author, Short year,

Boolean available, byte[] portada) {

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

this.portada = portada;

}

public Book(Long idBook, String isbn, String title, String author,

Short year, Boolean available, byte[] portada) {

this.idBook = idBook;

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

this.portada = portada;

}

public Long getIdBook() {

return idBook;

}

public Book setIdBook(Long idBook) {

this.idBook = idBook;

return this;

}

public String getIsbn() {

return isbn;

}

public Book setIsbn(String isbn) {

this.isbn = isbn;

return this;

}

public String getTitle() {

return title;

}

public Book setTitle(String title) {

this.title = title;

return this;

}

public String getAuthor() {

return author;

}

public Book setAuthor(String author) {

this.author = author;

return this;

}

public Short getYear() {

return ano;

}

public Book setAno(Short ano) {

this.ano = ano;

return this;

}

public Boolean isAvailable() {

return available;

}

public Book setAvailable(Boolean available) {

this.available = available;

return this;

}

public byte[] getCover() {

return portada;

}

public Book setCover(byte[] portada) {

this.portada = portada;

return this;

}

public LocalDate getDataPublicacion() {

return dataPublicacion;

}

public Book setDataPublicacion(LocalDate dataPublicacion) {

this.dataPublicacion = dataPublicacion;

return this;

}

/\*\*

\* Asigna la portada con flujos, leyendo los bytes.

\*

\* @param f

\*/

public Book setPortada(File f) {

if (f == null || !f.exists())

return this;

Path p = Paths.get(f.getAbsolutePath());

try (BufferedInputStream bi = new BufferedInputStream(Files.newInputStream(p));

ByteArrayOutputStream outputStream = new ByteArrayOutputStream()) {

byte[] buffer = new byte[4096];

int bytesLidos;

while ((bytesLidos = bi.read(buffer)) > 0) {

outputStream.write(buffer, 0, bytesLidos);

}

portada = outputStream.toByteArray();

} catch (FileNotFoundException ex) {

System.err.println("Archivo no encontrado: " + ex.getMessage());

} catch (IOException ex) {

System.err.println("Erro de E/S: " + ex.getMessage());

}

return this;

}

/\*\*

\* Asigna la portada con Java NIO, leyendo los bytes.

\*

\* @param file

\*/

public Book setPortada(String file) {

try {

Path ruta = Paths.get(file);

portada = Files.readAllBytes(ruta);

} catch (IOException ex) {

System.err.println("Error de E/S: " + ex.getMessage());

}

return this;

}

public Image getImage() {

if (portada != null) {

try (ByteArrayInputStream bis = new ByteArrayInputStream(portada)) {

Image imaxe = ImageIO.read(bis);

if(available) {

imaxe.getGraphics().drawLine(0,0, 100, 100);

}

return imaxe;

} catch (IOException e) {

}

}

return null;

}

@Override

public int hashCode() {

int hash = 7;

hash = 41 \* hash + Objects.hashCode(this.isbn);

return hash;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

final Book other = (Book) obj;

return Objects.equals(this.isbn, other.isbn);

}

@Override

public String toString() {

return idBook + "] [isbn: " + isbn + "] " + title + ". "

+ author + " (" + ano + ") [" + ((available) ? '\*' : ' ') + ']';

}

}

 Solución: Clase Contido

De momento, **no hemos declarado Contido como entidad JPA**, pero lo haremos en el futuro, cuando veamos las relaciones.

import java.util.Objects;

/\*\*

\* @autor pepecalo

\* CREATE TABLE PUBLIC.Contido (

\* idContido INTEGER NOT NULL AUTO\_INCREMENT,

\* idBook INTEGER NOT NULL,

\* contido CHARACTER LARGE OBJECT,

\* CONSTRAINT Contido\_PK PRIMARY KEY (idContido),

\* CONSTRAINT FK\_ID\_BOOK FOREIGN KEY (idBook) REFERENCES PUBLIC.Book(idBook) ON DELETE CASCADE ON UPDATE CASCADE

\* );

\* CREATE UNIQUE INDEX PRIMARY\_KEY\_9 ON PUBLIC.Contido (idContido);

\*/

public class Contido {

private Long idContido;

private String contido;

private Book book;

public Contido() {

}

public Contido(Long idBook, String contido) {

this.contido = contido;

}

public Contido(Long idContido, Long idBook) {

this.idContido = idContido;

}

public Contido(Long idContido, Long idBook, String contido) {

this.idContido = idContido;

this.contido = contido;

}

public Long getIdContido() {

return idContido;

}

public void setIdContido(Long idContido) {

this.idContido = idContido;

}

public String getContido() {

return contido;

}

public void setContido(String contido) {

this.contido = contido;

}

public Book getBook() {

return book;

}

public void setBook(Book book) {

this.book = book;

}

@Override

public int hashCode() {

return 97 \* 7 + Objects.hashCode(this.idContido);

}

@Override

public boolean equals(Object obj) {

if (obj == null || !(obj instanceof Contido other)) return false;

return Objects.equals(this.idContido, other.idContido);

}

@Override

public String toString() {

return idContido + ": " + contido;

}

}

 Solución: BookJPADao

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityTransaction;

import jakarta.persistence.TypedQuery;

import java.util.List;

public class BookJPADao implements Dao<Book> {

private final EntityManager em;

public BookJPADao(EntityManager em) {

this.em = em;

}

@Override

public Book get(long id) {

return em.find(Book.class, id);

}

@Override

public List<Book> getAll() {

TypedQuery<Book> query = em.createQuery("SELECT b FROM Book b", Book.class);

return query.getResultList();

}

@Override

public void save(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.persist(book);

tx.commit();

}

@Override

public void update(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.merge(book);

tx.commit();

}

@Override

public void delete(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.remove(book);

tx.commit();

}

@Override

public boolean deleteById(long id) {

Book book = get(id);

if (book != null) {

delete(book);

return true;

}

return false;

}

@Override

public List<Integer> getAllIds() {

TypedQuery<Integer> query = em.createQuery("SELECT b.idBook FROM Book b", Integer.class);

return query.getResultList();

}

@Override

public void updateLOB(Book book, String f) {

book.setPortada(f);

update(book); // La tansacción se hace en el método update

}

@Override

public void updateLOBById(long id, String f) {

Book book = get(id);

if (book != null) {

updateLOB(book, f); // La transacción se hace en el método updateLOB, que a su vez llama a update

}

}

@Override

public void deleteAll() {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.createQuery("DELETE FROM Book").executeUpdate();

tx.commit();

}

}

 Solución: BookDaoFactory

import java.util.Map;

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) {

case JDBC\_H2:

BibliotecaConnectionMaganer bibliotecaConnection = BibliotecaConnectionMaganer.getInstance();

return new BookDao(bibliotecaConnection.getConnection());

case JPA\_H2:

return new BookJPADao((BibliotecaJpaManager.getEntityManager(BibliotecaJpaManager.BIBLIOTECA\_H2)));

// ...

}

return null;

}

}

# 04. Gestión de entidades con EntityManager.

* [1. EntityManager](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#1-entitymanager)
* [2. Creación de un EntityManager](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#2-creaci%C3%B3n-de-un-entitymanager)
* [3. Operaciones CRUD](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#3-operaciones-crud)
  + [3.1. Persistir una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#31-persistir-una-entidad)
  + [3.2. Obtención de una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#32-obtenci%C3%B3n-de-una-entidad)
  + [3.3. Eliminación de una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#33-eliminaci%C3%B3n-de-una-entidad)
  + [3.4. Actualización de una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#34-actualizaci%C3%B3n-de-una-entidad)
* [4. Transacciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#4-transacciones)
* [5. Consultas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#5-consultas)
  + [5.1. Consultas dinámicas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#51-consultas-din%C3%A1micas)
  + [5.2. Consultas con nombre (estáticas)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#52-consultas-con-nombre-est%C3%A1ticas)
  + [5.3. Ejecución de consultas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#53-ejecuci%C3%B3n-de-consultas)
* [5. Consultas (ampliado)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#5-consultas-ampliado)
  + [Tipos de consultas en Jakarta Persistence](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#tipos-de-consultas-en-jakarta-persistence)
* [Ejercicios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/04entitymanager/#ejercicios)

## 1. EntityManager

El [**gestor de entidades (EntityManager)**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager) es el encargado de **gestionar el ciclo de vida de las entidades**. Con él podemos [**persistir (persist)**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#persist(java.lang.Object))**,**[**actualizar**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#merge(T))**,**[**eliminar (remove)**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#remove(java.lang.Object))**y**[**recuperar (find)**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#find(java.lang.Class,java.lang.Object))**entidades**, así como [**realizar consultas (createQuery)**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#createQuery(java.lang.String)).

* **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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/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 gestor de entidades.
* **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 | Persistence | Clase de inicio utilizada para obtener una fábrica de gestores de entidades (EntityManagerFactory) |
| Entity Manager Factory | EntityManagerFactory | Objeto Factory configurado utilizado para obtener gestores de entidades (EntityManager) |
| Persistence Unit | – | **Configuración** con **nombre** que declara las **clases de entidad y la información de la base de datos** |
| Entity Manager | EntityManager | 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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanagerfactory).

En el entorno de Java SE, podemos utilizar una clase de [llamada **Persistence**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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()](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/persistence#createEntityManagerFactory(java.lang.String)), **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()](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#find(java.lang.Class,java.lang.Object)):

<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***](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#remove(java.lang.Object)):

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 getResultList() o getSingleResult():

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

List<Empleado> results = q.getResultList();

El método getResultList() devuelve una **lista de resultados**. El método getSingleResult() devuelve un **único resultado**.

Si la consulta no devuelve ningún resultado, getResultList() devuelve una lista vacía y getSingleResult() lanza una excepción NoResultException.

Si la consulta 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** 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 getResultList:**

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

List<Empleado> emps = query.getResultList();

**Ejemplo usando getResultStream:**

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

Stream<Empleado> employee = query.getResultStream();

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 getResultList() o el método getResultStream() en ella.

El método getResultList() 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 getResultStream() 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 getResultList().stream().

El **método getResultStream() 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();

}

## Ejercicios

 Ejercicio. Creación de una aplicación de persistencia de una biblioteca

Amplia el proyecto de la biblioteca para que sea capaz de almacenar imágenes de portada de los libros. Para ello, crea una clase Book de tipo @Entity.

Recuerda que para almacenar imágenes en una base de datos se utiliza el tipo byte[] y para recuperarlas se utiliza ImageIO.read().

ByteArrayInputStream flujo = new ByteArrayInputStream(portada);

ImageIO.read(flujo);

**Interface DAO:**

Esta interface será implantada por todas aquellas clases DAO que trabajen con objetos con imágenes. Los nombres de los métodos son totalmente descriptivos:

T get(long id);

List<T> getAll();

void save(T t);

void update(T t);

void delete(T t);

public void deleteById(long id);

public void updateImage(T t, String f);

public void updateImageById(long id, String f);

**BookDAO implementa DAO:**

Tiene como atributo un objeto de tipo EntityManagerFactory, emf, que recoge como argumento el constructor.  
La clase BookDAO debe tener los siguientes métodos:

* get(long idBook): devuelve un objeto Book con la información del libro que tiene el identificador pasado como parámetro.
* getAll(): devuelve una lista de todos los libros almacenados en la base de datos
* save(Book book): crea un nuevo registro en la tabla Book con la información del libro pasado como parámetro.
* update(Book book): Actualiza la información del registro correspondiente al libro pasado como parámetro.
* delete(Book idBook): elimina el registro correspondiente al libro con el identificador del libro pasado como parámetro.
* deleteById(int idBook): elimina el registro correspondiente al libro con el identificador pasado como parámetro.
* updateImage(Book b, String f): actualiza el libro en la base de datos con el contenido del archivo recogido como parámetro. Nota: por el momento no implantes este método.
* updateImageByID(long b, String f): actualiza el libro con el id recogido como por parámetro con el contenido del archivo recogido como parámetro. Nota: por el momento no implantes este método.

Debes implantar la gestión de sentencias de esta la clase BookDAO por medio de try-with-resources para manejar los cierres de los EntityManager.

**Aplicación de prueba:**

Haz una aplicación que haga uso del EntityManagerUtil para obtener un EntityManagerFactory se la pase al constructor de BookDAO.

Crea varios libros y añádelos a 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);

Muestra el contenido de la base de datos.

# 03.12. Ejercicios.

* [Bases de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#bases-de-datos)
* [Ejercicios JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#ejercicios-jpa)
  + [0. Creación de EntityManagerFactory con patrón Singleton y Thread-Safe](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#0-creaci%C3%B3n-de-entitymanagerfactory-con-patr%C3%B3n-singleton-y-thread-safe)
    - [0.1. EntityManagerFactory Singleton](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#01-entitymanagerfactory-singleton)
    - [0.2. EntityManagerFactory Singleton con propiedades](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#02-entitymanagerfactory-singleton-con-propiedades)
    - [0.3. EntityManagerFactory Singleton para cada unidad de persistencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#03-entitymanagerfactory-singleton-para-cada-unidad-de-persistencia)
  + [1. Rango Legal y Organismo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#1-rango-legal-y-organismo)
  + [2. Alquiler de películas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#2-alquiler-de-pel%C3%ADculas)
  + [3. Pedidos PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#3-pedidos-postgresql)
  + [4. Pedidos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#4-pedidos)
  + [5. Base de datos de legislación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#5-base-de-datos-de-legislaci%C3%B3n)
    - [Propiedades de la conexión a la base de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#propiedades-de-la-conexi%C3%B3n-a-la-base-de-datos)
    - [Esquema de la base de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#esquema-de-la-base-de-datos)
    - [1. JPAUtil](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#1-jpautil)
    - [2. Clases del modelo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#2-clases-del-modelo)
      * [RangoLegal](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#rangolegal)
      * [Organismo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#organismo)
      * [Publicacion (enumeración) y PublicacionConverter](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#publicacion-enumeraci%C3%B3n-y-publicacionconverter)
      * [Clasificacion](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#clasificacion)
      * [Norma](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#norma)
      * [Documento](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#documento)
      * [DocumentoNorma](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#documentonorma)
    - [3. DTO y Consultas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#3-dto-y-consultas)
    - [4. NormaDAO y NormaRepository](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#4-normadao-y-normarepository)
      * [NormaDAO implements DAO<Norma, Integer>](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#normadao-implements-daonorma-integer)
      * [NormaRepository](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#normarepository)
    - [5. Servicio Rest con Spring Boot Data JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#5-servicio-rest-con-spring-boot-data-jpa)
    - [6. Servicio Rest con Spring Boot Data JPA y paginación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#6-servicio-rest-con-spring-boot-data-jpa-y-paginaci%C3%B3n)
* [Ejecutores de código al inicio de la Aplicación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#ejecutores-de-c%C3%B3digo-al-inicio-de-la-aplicaci%C3%B3n)
  + [Diferencias entre ejecutores de código al inicio de la Aplicación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/#diferencias-entre-ejecutores-de-c%C3%B3digo-al-inicio-de-la-aplicaci%C3%B3n)

## Bases de datos

 Bases de datos

* [bd-lexislacion.tar](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios.files/bd-lexislacion.tar) (24 )
* [bd-pedidosmariadb.sql](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios.files/bd-pedidosmariadb.sql) (1 )
* [bd-pedidospostgre.sql](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios.files/bd-pedidospostgre.sql) (12 )
* [bd-pedidospostgre.tar](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios.files/bd-pedidospostgre.tar) (35 )
* [bd-videoclub.tar](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios.files/bd-videoclub.tar) (2 )

 PostgreSQL

Para algunos ejercicios **usaremos la base de datos PostgreSQL**.

Para su instalación, puedes seguir las instrucciones de la [documentación oficial](https://www.postgresql.org/download/) o, mejor, usar un contenedor Docker con la imagen oficial de PostgreSQL:

docker run --name some-postgres -e POSTGRES\_PASSWORD=mysecretpassword -d postgres

Para conectarte a la base de datos, puedes usar el cliente psql o cualquier cliente de PostgreSQL como [pgAdmin](https://www.pgadmin.org/) o [DBeaver](https://dbeaver.io/), que ya hemos usado en la asignatura.

Los datos de conexión son:

{

"db": "postgres",

"user": "postgres",

"password": "mysecretpassword",

"host": "localhost",

"port": 5432

}

En el **apartado 02.02.03 de la UD 02.02 tienes los pasos de instalación sin privilegios de administrador en Windows, así como su instalación en Docker**.

 restauración de bases de datos PostgreSQL

Para restaurar la base de datos nombreDB en PostgreSQL, sigue los siguientes pasos:

pg\_restore -U postgres -d nombreDB /ruta/al/fichero/nombrearchivo.tar

En la que nombreDB es el nombre de la base de datos y /ruta/al/fichero/nombrearchivo.tar es la ruta al fichero de la base de datos. Además -U postgres es el usuario de la base de datos.

 MariaDB en Docker

También usaremos MariaDB en algunos ejercicios, en cuyo caso recomiendo usar un contenedor Docker con la imagen oficial de MariaDB:

docker run --name some-mariadb -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mariadb

O bien emplear el paquete XAMPP, que incluye MariaDB, Apache y PHP.

 Restauración de MariaDB

Para restaurar la base de datos nombreDB en MariaDB, sigue los siguientes pasos:

mysql -u root -p nombreDB < /ruta/al/fichero/nombreDB.sql

En la que nombreDB es el nombre de la base de datos y /ruta/al/fichero/nombreDB.sql es la ruta al fichero de la base de datos. La opción -u root es el usuario de la base de datos.

## 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 nomeG 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\a\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:

* Pais que mapee la tabla country: country\_id (de tipo serial4), country (varchar(50), last\_update (timestamp).
* Categoria (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.

1. Crea un tipo de entidad, Pelicula, 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 "FKbqsvlyhhs40rh7v7e6qpdto5i" 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

)

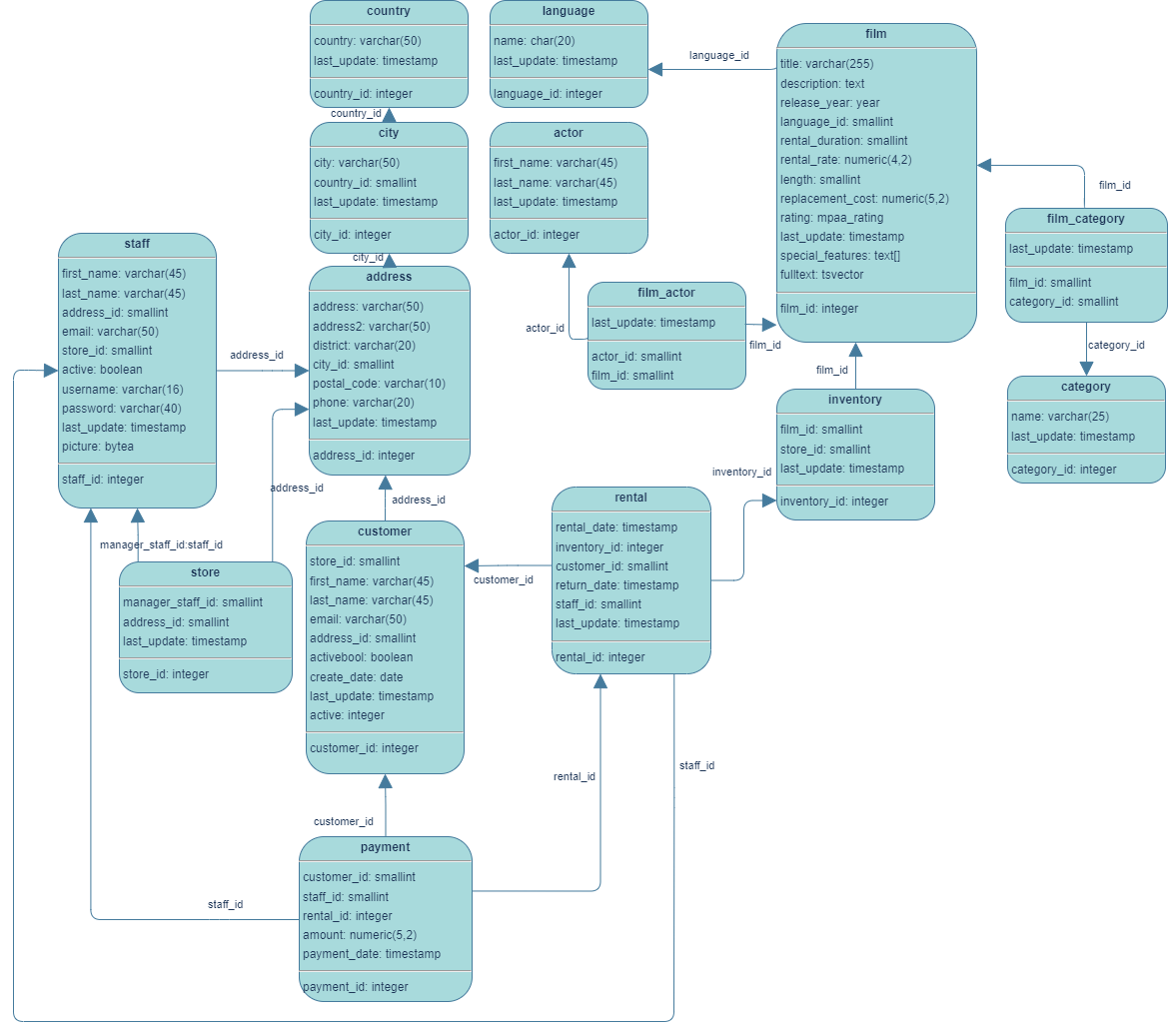
De momento, mapea el campo fulltext como un String:

1. Haz que la entidad Pelicula tenga una con la entidad Idioma (un idioma puede tener muchas películas, pero una película sólo puede tener un idioma).
2. CategoriaPelicula: haz que la entidad Pelicula tenga una relación con la entidad Categoria (una película puede tener muchas categorías y una categoría puede tener muchas películas), para ello, crea una entidad CategoriaPelicula 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.
3. PeliculaActor: haz que la entidad Pelicula 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.
4. Ciudad: mapee 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 Pais (una ciudad pertenece a un único país y un país puede tener muchas ciudades).
5. Direccion: mapee 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 Direccion tenga una relación con la entidad Ciudad (una dirección pertenece a una única ciudad y una ciudad puede tener muchas direcciones).
6. Empleado: mapee 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.
7. 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).
8. 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).
9. 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 tien

1. 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:



### 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 "FKb7xr57df8semvktej7l1lo85e" 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 "FKdne7p3hv47b0l6i5m2efvrpe4" 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'),

('pantalon', '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'),

('22222222B','Lucia'),

('33333333C','Beatriz');

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.
* LineaPedido: 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.
* **LineaPedido debe mapearse como una colección de elementos**. Comprueba el resultado y hazlo como entidad.
* **LineaPedido** 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**.

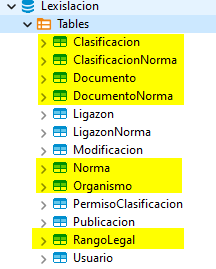
### 5. Base de datos de legislación

##### **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).



#### 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, Organimos no tienen referencia en las normas, Publicacion es una enumeración).

##### **RangoLegal**

**idRangoLegal** (Integer), **nomeG** (String), **nomeC** (String). Los nombres de los atributos nomeG y nomeG 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[]), **tamanho** (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 INDUSTRIAIS [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;

1. Liste los rangos legales y la cantidad de normas que contienen. Debe devolver el nombre de RangoLegal (**nomeG**), 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;

1. 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.
2. 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);

}

1. 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](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.data).

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.micompanhia.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**](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/CommandLineRunner.html): 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)**](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/ApplicationRunner.html): 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**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html): 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**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html): 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)](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/ApplicationRunner.html) 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](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/ApplicationArguments.html), que tiene métodos como getOptionNames(), getOptionValues() y getSourceArgs().

* [CommandLineRunner](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/CommandLineRunner.html) 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](https://www.baeldung.com/spring-boot-run-on-startup).

SE CAMBIO POR ESTA ACTUALIZACION:

### 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 Categoriam 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 Categoria** tiene los siguientes valores:

public enum Categoria {

ANY("Any"),

MISC("Misc"),

PROGRAMMING("Programming"),

DARK("Dark"),

PUN("Pun"),

SPOOKY("Spooky"),

CHRISTMAS("Christmas");

//...

}

 Detalle de implementación de la enumeración Categoría

package com.pepinho.ad.chistes.model;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* <p>

\* Enumeración de categorías de chistes.

\* Pueden ser: Any, Misc, Programming, Dark, Pun, Spooky, Christmas

\* Atributo: nombre, de tipo cadena.

\*/

public enum Categoria {

ANY("Any"),

MISC("Misc"),

PROGRAMMING("Programming"),

DARK("Dark"),

PUN("Pun"),

SPOOKY("Spooky"),

CHRISTMAS("Christmas");

private final String nombre;

Categoria(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

/\*\*

\* Devuelve la categoría a partir de su nombre.

\*

\* @param nombre Nombre de la categoría

\* @return Categoría

\*/

public static Categoria getCategoria(String nombre) {

for (Categoria c : Categoria.values()) {

if (c.getNombre().equals(nombre)) {

return c;

}

}

return null;

}

/\*\*

\* Sobreescribe el método toString() para que devuelva el nombre de la categoría.

\*

\* @return Nombre de la categoría

\* @see java.lang.Enum#toString()

\*/

@Override

public String toString() {

return nombre;

}

}

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

public enum TipoChiste {

SINGLE("single"),

TWOPART("twopart");

//...

}

**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

package com.pepinho.ad.chistes.model;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* Enumeración de banderas de chistes.

\* Pueden ser: NSFW, RELIGION, POLITICAL, RACIST, SEXIST

\* Atributo: String nombre.

\* Constructor: Flag(String nombre)

\* @see Categoria

\* @link <a href="https://v2.jokeapi.dev/flags">https://v2.jokeapi.dev/flags</a>

\*/

public enum Flag {

EXPLICIT("Explicit"),

NSFW("NSFW"),

RELIGION("Religion"),

POLITICAL("Political"),

RACIST("Racist"),

SEXIST("Sexist");

private final String nombre;

Flag(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

/\*\*

\* Devuelve la bandera a partir de su nombre.

\* @param nombre Nombre de la bandera

\* @return Bandera

\*/

public static Flag getFlag(String nombre) {

// Con expresiones lambda:

return java.util.Arrays.stream(Flag.values()).filter(f -> f.getNombre().equals(nombre)).findFirst()

.orElse(null);

/\* // Con un bucle for:

// for (Flag f : Flag.values()) {

// if (f.getNombre().equals(nombre)) {

// return f;

// }

// }

// return null;

\*/

}

/\*\*

\* Sobreescribe el método toString() para que devuelva el nombre de la bandera.

\* @return Nombre de la bandera

\* @see java.lang.Enum#toString()

\*/

@Override

public String toString() {

return nombre;

}

}

**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

package com.pepinho.ad.chistes.model;

import java.util.Arrays;

/\*\*

\* Lenguajes admitidos por la API de chistes.

\* \* "jokeLanguages": [

\* \* "cs",

\* \* "de",

\* \* "en",

\* \* "es",

\* \* "fr",

\* \* "pt"

\* \* ]

\* Atributo con el nombre del lenguaje del chiste.

\*

\* @see <a href="https://sv443.net/jokeapi/v2/#languages">https://sv443.net/jokeapi/v2/#languages</a>

\*/

public enum Lenguaje {

CS("cs"),

DE("de"),

EN("en"),

ES("es"),

FR("fr"),

PT("pt");

private final String lenguaje;

/\*\*

\* Constructor de la clase Lenguajes.

\* @param lenguaje Nombre del lenguaje

\*/

Lenguaje(String lenguaje) {

this.lenguaje = lenguaje;

}

/\*\*

\* Devuelve el nombre del lenguaje.

\* @return Nombre del lenguaje

\*/

public String getLenguaje() {

return lenguaje;

}

public static Lenguaje getLenguaje(String lenguaje) {

// Con expresiones lambda:

return Arrays.stream(Lenguaje.values()).filter(l -> l.getLenguaje().equals(lenguaje)).findFirst()

.orElse(null);

/\* // Con un bucle for:

// for (Lenguaje l : Lenguaje.values()) {

// if (l.getLenguaje().equals(lenguaje)) {

// return l;

// }

// }

// return null;

\*/

}

@Override

public String toString() {

return 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

package com.pepinho.ad.chistes.model;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.Objects;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* <p>

\* Clase que representa un chiste.

\* Atributos: id de tipo int, categoria de tipo Categoria, idiomade tipo Lenguaje, tipo de TipoChiste,

\* List<Flag> banderas, String chiste, String respuesta.

\*/

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;

/\*\*

\* Constructor de la clase Chiste.

\* @param id Identificador del chiste

\* @param categoria Categoría del chiste

\* @param idioma Idioma del chiste

\* @param tipo Tipo del chiste

\* @param chiste Chiste

\* @param respuesta Respuesta del chiste

\*/

public Chiste(int id, Categoria categoria, String idioma, TipoChiste tipo, String chiste, String respuesta) {

this.id = id;

this.categoria = categoria;

this.tipo = tipo;

this.chiste = chiste;

this.respuesta = respuesta;

this.banderas = new ArrayList<>();

this.lenguaje = Lenguaje.getLenguaje(idioma);

}

/\*\*

\* Constructor por defecto de la clase Chiste.

\*

\*/

public Chiste() {

// this.id = 0;

this.categoria = Categoria.ANY;

this.lenguaje = Lenguaje.EN;

this.tipo = TipoChiste.SINGLE;

this.chiste = "";

this.respuesta = "";

this.banderas = new ArrayList<>();

}

/\*\*

\* Devuelve el identificador del chiste.

\* @return Identificador del chiste

\*/

public int getId() {

return id;

}

/\*\*

\* Establece el identificador del chiste.

\* @param id Identificador del chiste

\*/

public void setId(int id) {

this.id = id;

}

/\*\*

\* Devuelve la categoría del chiste.

\* @return Categoría del chiste

\*/

public Categoria getCategoria() {

return categoria;

}

public String getCategoriaString() {

return categoria.getNombre();

}

/\*\*

\* Establece la categoría del chiste.

\* @param categoria Categoría del chiste

\*/

public void setCategoria(Categoria categoria) {

this.categoria = categoria;

}

public void setCategoria(String categoria) {

this.categoria = Categoria.getCategoria(categoria);

}

public Lenguaje getLenguaje() {

return lenguaje;

}

public String getLenguajeString() {

return lenguaje.getLenguaje();

}

public void setLenguaje(String lenguaje) {

this.lenguaje = Lenguaje.getLenguaje(lenguaje);

}

public void setLenguaje(Lenguaje lenguaje) {

this.lenguaje = lenguaje;

}

/\*\*

\* Devuelve el tipo del chiste.

\* @return Tipo del chiste

\*/

public TipoChiste getTipo() {

return tipo;

}

public String getTipoString() {

return tipo.getNombre();

}

/\*\*

\* Establece el tipo del chiste.

\* @param tipo Tipo del chiste

\*/

public void setTipo(TipoChiste tipo) {

this.tipo = tipo;

}

public void setTipo(String tipo) {

this.tipo = TipoChiste.getTipoChiste(tipo);

}

/\*\*

\* Devuelve las banderas del chiste.

\* @return Banderas del chiste

\*/

public List<Flag> getBanderas() {

return banderas;

}

/\*\*

\* Añade una bandera al chiste.

\* @param flag Bandera a añadir

\*/

public void addFlag(Flag flag) {

banderas.add(flag);

}

public boolean removeFlag(Flag bandera) {

return banderas.remove(bandera);

}

/\*\*

\* Si el chiste tiene esa bandera, devuelve true.

\* @param bandera Bandera a comprobar

\* @return true si el chiste tiene esa bandera, false en caso contrario

\*/

public boolean containsFlag(Flag bandera) {

return banderas.contains(bandera);

}

/\*\*

\* Devuelve el chiste como cadena de caracteres.

\* @return Chiste como String

\*/

public String getChiste() {

return chiste;

}

/\*\*

\* Establece el chiste.

\* @param chiste Chiste

\*/

public void setChiste(String chiste) {

this.chiste = chiste;

}

/\*\*

\* Devuelve la respuesta del chiste.

\* @return Respuesta del chiste

\*/

public String getRespuesta() {

return respuesta;

}

/\*\*

\* Establece la respuesta del chiste.

\* @param respuesta Respuesta del chiste

\*/

public void setRespuesta(String respuesta) {

this.respuesta = respuesta;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Chiste chiste = (Chiste) o;

return id == chiste.id;

}

@Override

public int hashCode() {

return Objects.hash(id);

}

/\*\*

\* Sobrescritura del método toString() para que devuelva el chiste.

\* Lo devuelve empleando un StringBuilder y por medio del método forEach() para recorrer la lista de banderas.

\* @return Chiste como String

\*/

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("Chiste: ").append(chiste).append(System.lineSeparator());

sb.append("Respuesta: ").append(respuesta).append(System.lineSeparator());

sb.append("Categoría: ").append(categoria).append(System.lineSeparator());

sb.append("Idioma: ").append(lenguaje).append(System.lineSeparator());

sb.append("Tipo: ").append(tipo).append(System.lineSeparator());

sb.append("Banderas: ");

banderas.forEach(b -> sb.append(b).append(" "));

sb.append(System.lineSeparator());

return sb.toString();

}

}

**B)** El adapter **ChisteDeserializer**:

 Detalle de implementación de la clase ChisteDeserializer

package com.pepinho.ad.chistes.model;

import com.google.gson.\*;

import java.lang.reflect.Type;

/\*

{

"error": false,

"category": "Programming",

"type": "twopart",

"setup": "¿Por qué C consigue todas las chicas y Java no tiene ninguna?",

"delivery": "Porque C no las trata como objetos.",

"flags": {

"nsfw": false,

"religious": false,

"political": false,

"racist": false,

"sexist": false,

"explicit": false

},

"safe": true,

"id": 6,

"lang": "es"

}

\*/

public class ChisteDeserializer implements JsonDeserializer<Chiste> {

@Override

public Chiste deserialize(JsonElement elemento, Type type,

JsonDeserializationContext jsonDeserializationContext) throws JsonParseException {

// Comprobación si es un objeto:

if (!elemento.isJsonObject())

return null;

// Creo un chiste vacío, al que le daré valor a sus atributos:

Chiste chiste = new Chiste();

// Recupero el objeto JSON del chiste

JsonObject jsonChiste = elemento.getAsJsonObject();

// Comprobación de que no hay error en la petición:

if (jsonChiste.get("error") != null && jsonChiste.get("error").getAsBoolean()) {

return null;

}

// Compruebo que cada elemento del objeto existe y lo asigno al objeto Chiste:

// La comprobación se hace con el método get() de la clase JsonObject que devuelve

// un JsonElement. Si es null, no existe el elemento.

if (jsonChiste.get("category") != null) {

chiste.setCategoria(jsonChiste.get("category").getAsString());

}

if (jsonChiste.get("type") != null) {

chiste.setTipo(jsonChiste.get("type").getAsString());

}

// En realidad, dependiendo del tipo de chiste, el setup o el delivery pueden no existir.

// Por lo que podría hacer comprobando el valor de type, pero lo dejo así para que veáis

// como se puede hacer con el método get() de la clase JsonObject.

if (jsonChiste.get("setup") != null) {

chiste.setChiste(jsonChiste.get("setup").getAsString());

if (jsonChiste.get("delivery") != null) {

chiste.setRespuesta(jsonChiste.get("delivery").getAsString());

}

} else if (jsonChiste.get("joke") != null) {

chiste.setChiste(jsonChiste.get("joke").getAsString());

}

if (jsonChiste.get("lang") != null) {

chiste.setLenguaje(jsonChiste.get("lang").getAsString());

}

if (jsonChiste.get("id") != null) {

chiste.setId(jsonChiste.get("id").getAsInt());

}

if (jsonChiste.get("flags") != null) {

JsonObject flags = jsonChiste.get("flags").getAsJsonObject();

if (flags.get("nsfw").getAsBoolean()) {

chiste.addFlag(Flag.NSFW);

}

if (flags.get("religious").getAsBoolean()) {

chiste.addFlag(Flag.RELIGION);

}

if (flags.get("political").getAsBoolean()) {

chiste.addFlag(Flag.POLITICAL);

}

if (flags.get("racist").getAsBoolean()) {

chiste.addFlag(Flag.RACIST);

}

if (flags.get("sexist").getAsBoolean()) {

chiste.addFlag(Flag.SEXIST);

}

if (flags.get("explicit").getAsBoolean()) {

chiste.addFlag(Flag.EXPLICIT);

}

}

return chiste;

}

}

**C)** La clase **ChisteTypeAdapter**:

 Detalle de implementación de la clase ChisteTypeAdapter

package com.pepinho.ad.chistes.model;

import com.google.gson.TypeAdapter;

import com.google.gson.stream.JsonReader;

import com.google.gson.stream.JsonToken;

import com.google.gson.stream.JsonWriter;

import java.io.IOException;

/\*

Formato de JSON:

{

"id": 1,

"category": "Programming",

"type": "single",

"joke": "Chuck Norris can write multithreaded applications with a single thread.",

"flags": {

"nsfw": false,

"religious": false,

"political": false,

"racist": false,

"sexist": false

},

"lang": "en"

\*/

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* Clase que adaptará el tipo Chiste para que pueda ser serializado y deserializado por Gson.

\*

\* @see com.google.gson.Gson

\* @see com.google.gson.TypeAdapter

\* @see com.google.gson.GsonBuilder

\* @see com.google.gson.JsonDeserializer

\*/

public class ChisteTypeAdapter extends TypeAdapter<Chiste> {

@Override

public void write(JsonWriter jsonWriter, Chiste chiste) throws IOException {

jsonWriter.beginObject();

jsonWriter.name("id").value(chiste.getId());

jsonWriter.name("category").value(chiste.getCategoriaString());

jsonWriter.name("type").value(chiste.getTipoString());

if (chiste.getTipo() == TipoChiste.SINGLE) {

jsonWriter.name("joke").value(chiste.getChiste());

} else {

jsonWriter.name("setup").value(chiste.getChiste());

jsonWriter.name("delivery").value(chiste.getRespuesta());

}

jsonWriter.name("flags");

jsonWriter.beginObject();

// Recorremos todas las banderas y asignamos el valor verdadero o falso si el chiste la contiene o no, respectivamente.

// Puede hacerse por medio del método containsFlag() de la clase Chiste o recoger las banderas

// del chiste e invocar el método contains() de la clase List.

for (Flag flag : Flag.values()) {

jsonWriter.name(flag.getNombre().toLowerCase()).value(chiste.containsFlag(flag));

}

jsonWriter.endObject();

jsonWriter.name("lang").value(chiste.getLenguajeString());

jsonWriter.endObject();

}

/\*\*

\* Método que deserializa un objeto Chiste a partir de un JsonReader.

\*

\* @param reader JsonReader que contiene el objeto Chiste

\* @return Objeto Chiste

\* @throws IOException Si hay un error de E/S

\* @see com.google.gson.stream.JsonReader

\* @see com.google.gson.stream.JsonToken

\*/

@Override

public Chiste read(JsonReader reader) throws IOException {

if(reader.peek()== JsonToken.NULL || reader.peek()!= JsonToken.BEGIN\_OBJECT){

// reader.nextNull();

return null;

}

reader.beginObject();

Chiste chiste = new Chiste();

while (reader.peek() != JsonToken.END\_OBJECT) {

String name = reader.nextName();

switch (name) {

case "id" -> chiste.setId(reader.nextInt());

case "category" -> chiste.setCategoria(Categoria.getCategoria(reader.nextString()));

case "type" -> chiste.setTipo(TipoChiste.getTipoChiste(reader.nextString()));

case "joke", "setup" -> chiste.setChiste(reader.nextString());

case "delivery" -> chiste.setRespuesta(reader.nextString());

case "flags" -> // Para hacerlo más modular he puesto el código en un método aparte.

readFlags(reader, chiste);

case "lang" -> chiste.setLenguaje(reader.nextString());

default -> reader.skipValue();

}

}

reader.endObject();

return chiste;

}

private void readFlags(JsonReader reader, Chiste chiste) throws IOException {

reader.beginObject();

while (reader.peek() != JsonToken.END\_OBJECT) {

String flagName = reader.nextName();

switch (flagName) {

case "nsfw" -> {

if (reader.nextBoolean()) chiste.addFlag(Flag.NSFW);

}

case "religious" -> {

if (reader.nextBoolean()) chiste.addFlag(Flag.RELIGION);

}

case "political" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.POLITICAL);

}

case "racist" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.RACIST);

}

case "sexist" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.SEXIST);

}

case "explicit" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.EXPLICIT);

}

default -> reader.skipValue();

}

}

reader.endObject();

}

}

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

 Detalle de implementación de la interfaz IChisteDAO

package com.pepinho.ad.chistes.model;

import java.io.Writer;

public interface IChisteDAO {

String getRandomJokeAsString();

String getJokeAsString(String categoria, String[] tipo, String[] banderas);

String getJokeAsString(String categoria, String[] tipo, String[] banderas, String idioma);

Chiste getRandomJoke();

Chiste getJoke(String categoria, String[] tipo, String[] banderas);

Chiste getJoke(String categoria, String[] tipo, String[] banderas, String idioma);

Chiste getJokeById(int id);

void saveJokeAsJson(Chiste chiste, Writer writer);

}

*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

package com.pepinho.ad.chistes.model;

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.Writer;

import java.net.MalformedURLException;

import java.net.URI;

import java.net.URISyntaxException;

import java.util.Arrays;

import java.util.Objects;

/\*\*

\* Created by Pepe Calo on 07/11/2023

\* Implementación de la interfaz IChisteDAO que consulta un chiste en un archivo Json

\* mediante la librería Gson.

\* La API de chistes utilizada es:

\* <a href="https://v2.jokeapi.dev/joke/">...</a>

\*

\* @see IChisteDAO

\* @see Chiste

\* @see Gson

\* @see GsonBuilder

\* @see com.google.gson.JsonObject

\* @see com.google.gson.JsonParser

\*/

public class ChisteDAO implements IChisteDAO {

private final Gson gson;

// https://v2.jokeapi.dev/joke/Programming,Miscellaneous?blacklistFlags=nsfw,religious

private static final String BASE\_URL = "https://v2.jokeapi.dev/joke/";

private static final String ENDPOINT = "?format=json";

private static final int NO\_ID = 0;

private static final String SINGLE = "single";

/\*\*

\* Constructor de la clase ChisteDAO.

\* Si deseas emplear las clases ChisteSerializer y ChisteDeserializer, debes comentar la línea con ChisteTypeAdapter

\* y no comentar las de los otros dos adaptadores.

\*/

public ChisteDAO() {

gson = new GsonBuilder().setPrettyPrinting()

// .registerTypeAdapter(Chiste.class, new ChisteDeserializer())

// .registerTypeAdapter(Chiste.class, new ChisteSerializer())

.registerTypeAdapter(Chiste.class, new ChisteTypeAdapter())

.create();

}

private String getURL(String categoria, String[] tipo, String[] banderas, String idioma, int id) {

String url = BASE\_URL + categoria + ENDPOINT;

if (tipo != null && tipo.length > 0) {

// Concateno los elementos no nulos media stream de un array de String. En el caso de que no haya ninguno, devuelvo un Optional vacío.

String tipos = Arrays.stream(tipo).filter(Objects::nonNull).reduce((s, s2) -> s + "," + s2).orElse(null);

if(tipos!=null && !tipos.isEmpty()){

url += "&type=" + tipos;

}

}

if (banderas != null && banderas.length > 0) {

String flags = Arrays.stream(banderas).filter(Objects::nonNull).reduce((s, s2) -> s + "," + s2).orElse(null);

if(flags!=null && !flags.isEmpty()){

url += "&blacklistFlags=" + flags;

}

}

if (idioma != null && !idioma.isEmpty()) {

url += "&lang=" + idioma;

}

if (id > 0) {

url += "&idRange=" + id;

}

System.out.println("url = " + url);

return url;

}

private Chiste getJoke(String url) {

try (BufferedReader is = new BufferedReader(new InputStreamReader(new URI(url).toURL().openStream()))) {

return gson.fromJson(is, Chiste.class);

} catch (MalformedURLException e) {

System.err.println("Error en la URL: " + e.getMessage());

} catch (IOException e) {

System.err.println("Erro E/S: " + e.getMessage());

} catch (URISyntaxException e) {

throw new RuntimeException(e);

}

return null;

}

private String getJokeAsString(String url) {

Chiste chiste = getJoke(url);

return (chiste!=null) ? chiste.getChiste() + System.lineSeparator() + chiste.getRespuesta() : "";

}

@Override

public String getJokeAsString(String categoria, String[] tipo, String[] banderas) {

return getJokeAsString(getURL(categoria, tipo, banderas, null, NO\_ID));

}

@Override

public Chiste getJoke(String categoria, String[] tipo, String[] banderas) {

return getJoke(getURL(categoria, tipo, banderas, null, NO\_ID));

}

@Override

public String getJokeAsString(String categoria, String[] tipo, String[] banderas, String idioma) {

return getJokeAsString(getURL(categoria, tipo, banderas, idioma, NO\_ID));

}

@Override

public Chiste getJoke(String categoria, String[] tipo, String[] banderas, String idioma) {

return getJoke(getURL(categoria, tipo, banderas, idioma, NO\_ID));

}

@Override

public Chiste getJokeById(int id) {

return getJoke(getURL("Any", null, null, null, id));

}

@Override

public void saveJokeAsJson(Chiste chiste, Writer writer) {

gson.toJson(chiste, writer);

}

@Override

public String getRandomJokeAsString() {

System.out.println(BASE\_URL + "Any");

return getJokeAsString(BASE\_URL + "Any");

}

@Override

public Chiste getRandomJoke() {

return getJoke(BASE\_URL + "Any");

}

}

#### 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

package com.pepinho.ad.chistes.model;

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.HashMap;

import java.util.Map;

import static com.pepinho.ad.chistes.model.ChisteLogger.LOG;

public class ChisteJpaManager {

public static final String BIBLIOTECA\_H2 = "chistesH2";

public static final String BIBLIOTECA\_POSTGRES = "chistesPostgres";

private static final Map<String, EntityManagerFactory> instancies = new HashMap<>();

private ChisteJpaManager() {

}

private static boolean isEntityManagerFactoryClosed(String unidadPersistencia) {

return !instancies.containsKey(unidadPersistencia) || instancies.get(unidadPersistencia) == null ||

!instancies.get(unidadPersistencia).isOpen();

}

public static EntityManagerFactory getEntityManagerFactory(String unidadPersistencia) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

synchronized (ChisteJpaManager.class) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

try {

instancies.put(unidadPersistencia, Persistence.createEntityManagerFactory(unidadPersistencia));

} catch (Exception e) {

LOG.error("Erro ó crear a unidade de persistencia " + unidadPersistencia +

": " + e.getMessage());

}

}

}

}

return instancies.get(unidadPersistencia);

}

public static EntityManager getEntityManager(String persistenceUnitName) {

return getEntityManagerFactory(persistenceUnitName).createEntityManager();

}

public static void close(String persistenceUnitName) {

if (instancies.containsKey(persistenceUnitName)) {

instancies.get(persistenceUnitName).close();

instancies.remove(persistenceUnitName);

}

}

}

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

 Solución de Chiste

package com.pepinho.ad.chistes.model;

import jakarta.persistence.\*;

import java.util.ArrayList;

@Entity

public class Chiste implements java.io.Serializable {

@Id

@Column(name = "idChiste")

private int id;

private Categoria categoria;

private TipoChiste tipo;

// Como se trata de una relación muchos a muchos, se emplea la anotación @ElementCollection

// H2 admite el tipo de dato Array de enteros (TINYINT ARRAY), prueba a no poner la anotación @ElementCollection ni @CollectionTable

@ElementCollection // Para que se cree una tabla intermedia

@Enumerated(EnumType.STRING)

@CollectionTable(name = "FlagsChiste", joinColumns = @JoinColumn(name = "idChiste"))

private final List<Flag> banderas;

private String chiste;

private String respuesta;

private Lenguaje lenguaje;

//...

}

**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
* package com.pepinho.ad.chistes;
* import com.pepinho.ad.chistes.model.Chiste;
* import com.pepinho.ad.chistes.model.ChisteDAO;
* import com.pepinho.ad.chistes.model.ChisteJpaManager;
* import static java.lang.Thread.sleep;
* public class ChisteDownloader implements Runnable {
* private static final long tiempoEspera = 550;
* private static ChisteDownloader instance;
* private static final int MAX\_CHISTES = 200;
* private ChisteDAO chisteDAO;
* private int numeroChistes = MAX\_CHISTES;
* private ChisteDownloader() {
* chisteDAO = new ChisteDAO();
* }
* public static ChisteDownloader getInstance() {
* if (instance == null) {
* synchronized (ChisteDownloader.class) {
* if (instance == null) {
* instance = new ChisteDownloader();
* }
* }
* }
* return instance;
* }
* public void setNumeroChistes(int numeroChistes) {
* this.numeroChistes = numeroChistes;
* }
* @Override
* public void run() {
* chisteDAO = new ChisteDAO();
* var emf = ChisteJpaManager.getEntityManagerFactory(ChisteJpaManager.BIBLIOTECA\_H2);
* var em = emf.createEntityManager();
* for (int i = 0; i < numeroChistes; i++) {
* Chiste chiste = chisteDAO.getJokeById(i);
* if (chiste != null) {
* try {
* em.getTransaction().begin();
* em.persist(chiste);
* em.getTransaction().commit();
* } catch (Exception e) {
* em.getTransaction().rollback();
* }
* System.out.print("\*");
* try {
* sleep(tiempoEspera);
* } catch (InterruptedException e) {
* System.out.println("Error en el hilo");
* }
* }
* }
* }
* }
* **D)** **Main** que descarga los chistes y los guarda en la base de datos.
* Solución de Main
* public static void main(String[] args) {
* ChisteDownloader chisteDownloader = ChisteDownloader.getInstance();
* chisteDownloader.setNumeroChistes(300);
* Thread thread = new Thread(chisteDownloader);
* thread.start();

# 05. Mapeo de entidades.

* [1. Mapeo Objeto-Relacional](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1-mapeo-objeto-relacional)
* [2. Anotaciones de Persistencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#2-anotaciones-de-persistencia)
* [1. Modo de acceso a una Entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1-modo-de-acceso-a-una-entidad)
  + [1.2. Acceso por atributo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#12-acceso-por-atributo)
  + [1.2. Acceso por propiedad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#12-acceso-por-propiedad)
  + [1.3. Acceso Mixto: @Access(AccessType.FIELD|AccessType.PROPERTY))](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#13-acceso-mixto-accessaccesstypefieldaccesstypeproperty)
* [2. Mapeo a una Tabla concreta: @Table](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#2-mapeo-a-una-tabla-concreta-table)
* [2.1. Nombres sensibles a mayúsculas de tablas y columnas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#21-nombres-sensibles-a-may%C3%BAsculas-de-tablas-y-columnas)
* [3. Mapeo de Tipos Simples](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#3-mapeo-de-tipos-simples)
* [4. Mapeo de columnas: @Column](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#4-mapeo-de-columnas-column)
* [5. Carga perezosa (Lazy Fetching): @Basic(fetch=FetchType.LAZY)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#5-carga-perezosa-lazy-fetching-basicfetchfetchtypelazy)
* [6. Objetos Grandes (LOBs): @Lob](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#6-objetos-grandes-lobs-lob)
* [7. Tipos Enumerados (enum): @Enumerated](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#7-tipos-enumerados-enum-enumerated)
  + [7.1. Mapeo enumeraciones como cadenas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#71-mapeo-enumeraciones-como-cadenas)
  + [7.2. Mapeo de enumeraciones con @PostLoad y @PrePersist](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#72-mapeo-de-enumeraciones-con-postload-y-prepersist)
  + [7.3 Mapeo de enumeraciones con @Converter](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#73-mapeo-de-enumeraciones-con-converter)
  + [7.4. Uso de Enums en JPQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#74-uso-de-enums-en-jpql)
* [8. Tipos temporales: @Temporal](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#8-tipos-temporales-temporal)
* [9. Atributos transitorios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#9-atributos-transitorios)
* [10. Mapeo de clave primaria: @Id](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#10-mapeo-de-clave-primaria-id)
  + [10.1. Sobrescritura de la clave primaria](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#101-sobrescritura-de-la-clave-primaria)
  + [10.2. Tipos de claves primarias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#102-tipos-de-claves-primarias)
  + [10.3. Generación de claves primarias: @GeneratedValue](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#103-generaci%C3%B3n-de-claves-primarias-generatedvalue)
    - [10.3.1. Generación Automática de ID: GenerationType.AUTO](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1031-generaci%C3%B3n-autom%C3%A1tica-de-id-generationtypeauto)
    - [10.3.2. Generación de ID utilizando una tabla: GenerationType.TABLE](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1032-generaci%C3%B3n-de-id-utilizando-una-tabla-generationtypetable)
    - [10.3.3. Generación de ID Utilizando una Secuencia de Base de Datos: GenerationType.SEQUENCE](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1033-generaci%C3%B3n-de-id-utilizando-una-secuencia-de-base-de-datos-generationtypesequence)
    - [10.3.4. Generación de ID utilizando una Identidad de Base de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1034-generaci%C3%B3n-de-id-utilizando-una-identidad-de-base-de-datos)
    - [10.3.5. Generación de ID Utilizando un UUID: GenerationType.UUID](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#1035-generaci%C3%B3n-de-id-utilizando-un-uuid-generationtypeuuid)
* [11. Ejercicio. Persistencia de una biblioteca](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/05mapeoentidades/index.html#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**](https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html)**(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 prohíben 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 {

// ...

}

1. 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() {

// ...

}

1. 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:

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.LocalTime
  + java.time.LocalDateTime
  + 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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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, **alguna parte 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 ineficientes 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, querríamos 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 quisiéramos 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 (preescritrua) 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 creemos 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 Categoria**. De lo contrario, **tendríamos que poner**[**la anotación @Convert**](https://jakarta.ee/specifications/persistence/3.2/apidocs/jakarta.persistence/jakarta/persistence/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(Categoria.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 Categoria.DEPORTE:

String jpql = "select a from Articulo a where a.categoria = com.pepinho.ad.jpa.Categoria.DEPORTE";

List<Articulo> articulos = 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.categoria = :categoria";

TypedQuery<Articulo> query = em.createQuery(jpql, Articulo.class);

query.setParameter("categoria", Categoria.TECNOLOGIA);

List<Article> articulos = 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 **fotografia** 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**: java.sql.Date, java.sql.Time y java.sql.Timestamp, así como los dos tipos java.util: 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**](https://jakarta.ee/specifications/persistence/3.2/apidocs/jakarta.persistence/jakarta/persistence/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)**](https://jakarta.ee/specifications/persistence/3.2/apidocs/jakarta.persistence/jakarta/persistence/temporal). **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.LocalTime
  + 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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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](https://jakarta.ee/specifications/persistence/3.1/apidocs/).

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 desee 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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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 nombra 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 pkColumnValue**. 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 ejecicio anterior con Persona, haz aque 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 ob**jeto 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 compartirla. 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 IDENTIDAD debe ocurrir, l**a 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 x 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.

# 06. Relaciones JPA.

* [1. Relaciones entre entidades](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#1-relaciones-entre-entidades)
  + [1.1. Roles de las entidades en las relaciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#11-roles-de-las-entidades-en-las-relaciones)
  + [1.2. Direccionalidad de las relaciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#12-direccionalidad-de-las-relaciones)
  + [1.3. Cardinalidad de las relaciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#13-cardinalidad-de-las-relaciones)
  + [1.4. Ordinalidad de las relaciones (opcionalidad)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#14-ordinalidad-de-las-relaciones-opcionalidad)
* [2. Relaciones entre Entidades](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#2-relaciones-entre-entidades)
* [3. Relaciones mono-valuadas: OneToOne y ManyToOne](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#3-relaciones-mono-valuadas-onetoone-y-manytoone)
  + [3.1. @OneToOne unidireccionales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#31-onetoone-unidireccionales)
  + [3.2. @OneToOne bidireccionales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#32-onetoone-bidireccionales)
  + [Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#ejercicio-0601-relaci%C3%B3n-uno-a-uno-bidireccional-equipo-entrenador)
  + [3.3. @ManyToOne unidireccional](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#33-manytoone-unidireccional)
    - [Empleando @JoinColum](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#empleando-joincolum)
  + [Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#ejercicio-0602-relaci%C3%B3n-muchos-a-uno-unidireccional-jugador-equipo)
* [4. Relaciones multi-valuadas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#4-relaciones-multi-valuadas)
  + [4.1. @OneToMany](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#41-onetomany)
    - [4.1.1. @OneToMany bidireccional](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#411-onetomany-bidireccional)
    - [4.1.2 @OneToMany unidireccional](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#412-onetomany-unidireccional)
  + [4.2. @ManyToMany](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#42-manytomany)
* [5. Nombre de la columna de Clave foránea](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#5-nombre-de-la-columna-de-clave-for%C3%A1nea)
* [Ejercicios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#ejercicios)
* [6. Claves compartidas en relaciones uno a uno](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#6-claves-compartidas-en-relaciones-uno-a-uno)
  + [6.1. Claves compartidas con @MapsId](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#61-claves-compartidas-con-mapsid)
  + [6.2. PrimaryKeyJoinColumn y PrimaryKeyJoinColumns](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#62-primarykeyjoincolumn-y-primarykeyjoincolumns)
  + [6.3. Ejercicio](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones/index.html#63-ejercicio)

## 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:

NOTFOUND

@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:

NOTFOUND

@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:

NOTFOUND  
NOTFOUND

@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**:

NOTFOUND

**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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytoone)
* **One-to-One** [(Uno a Uno): @OneToOne](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetoone)
* **One-to-Many** [(Uno a Muchos): @OneToMany](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetomany)
* **Many-to-Many** [(Muchos a Muchos): @ManyToMany](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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/onetoone>
* **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/onetoone>

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**:

NOT FOUND

**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 Apartamiento).

@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:

NOT FOUND

### 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**:

NOTFOUND

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;

// ...

}

NOTFOUND

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;

// ...

}

NOTFOUND

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

{

"Equipo": [

{

"abreviatura" : "ATL",

"idEquipo" : 1,

"ciudad" : "Atlanta",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Hawks",

"nombreCompleto" : "Atlanta Hawks"

},

{

"abreviatura" : "BOS",

"idEquipo" : 2,

"ciudad" : "Boston",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Celtics",

"nombreCompleto" : "Boston Celtics"

},

{

"abreviatura" : "BKN",

"idEquipo" : 3,

"ciudad" : "Brooklyn",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Nets",

"nombreCompleto" : "Brooklyn Nets"

},

{

"abreviatura" : "CHA",

"idEquipo" : 4,

"ciudad" : "Charlotte",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Hornets",

"nombreCompleto" : "Charlotte Hornets"

},

{

"abreviatura" : "CHI",

"idEquipo" : 5,

"ciudad" : "Chicago",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Bulls",

"nombreCompleto" : "Chicago Bulls"

},

{

"abreviatura" : "CLE",

"idEquipo" : 6,

"ciudad" : "Cleveland",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Cavaliers",

"nombreCompleto" : "Cleveland Cavaliers"

},

{

"abreviatura" : "DAL",

"idEquipo" : 7,

"ciudad" : "Dallas",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Mavericks",

"nombreCompleto" : "Dallas Mavericks"

},

{

"abreviatura" : "DEN",

"idEquipo" : 8,

"ciudad" : "Denver",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Nuggets",

"nombreCompleto" : "Denver Nuggets"

},

{

"abreviatura" : "DET",

"idEquipo" : 9,

"ciudad" : "Detroit",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Pistons",

"nombreCompleto" : "Detroit Pistons"

},

{

"abreviatura" : "GSW",

"idEquipo" : 10,

"ciudad" : "Golden State",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Warriors",

"nombreCompleto" : "Golden State Warriors"

},

{

"abreviatura" : "HOU",

"idEquipo" : 11,

"ciudad" : "Houston",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Rockets",

"nombreCompleto" : "Houston Rockets"

},

{

"abreviatura" : "IND",

"idEquipo" : 12,

"ciudad" : "Indiana",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Pacers",

"nombreCompleto" : "Indiana Pacers"

},

{

"abreviatura" : "LAC",

"idEquipo" : 13,

"ciudad" : "LA",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Clippers",

"nombreCompleto" : "LA Clippers"

},

{

"abreviatura" : "LAL",

"idEquipo" : 14,

"ciudad" : "Los Angeles",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Lakers",

"nombreCompleto" : "Los Angeles Lakers"

},

{

"abreviatura" : "MEM",

"idEquipo" : 15,

"ciudad" : "Memphis",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Grizzlies",

"nombreCompleto" : "Memphis Grizzlies"

},

{

"abreviatura" : "MIA",

"idEquipo" : 16,

"ciudad" : "Miami",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Heat",

"nombreCompleto" : "Miami Heat"

},

{

"abreviatura" : "MIL",

"idEquipo" : 17,

"ciudad" : "Milwaukee",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Bucks",

"nombreCompleto" : "Milwaukee Bucks"

},

{

"abreviatura" : "MIN",

"idEquipo" : 18,

"ciudad" : "Minnesota",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Timberwolves",

"nombreCompleto" : "Minnesota Timberwolves"

},

{

"abreviatura" : "NOP",

"idEquipo" : 19,

"ciudad" : "New Orleans",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Pelicans",

"nombreCompleto" : "New Orleans Pelicans"

},

{

"abreviatura" : "NYK",

"idEquipo" : 20,

"ciudad" : "New York",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Knicks",

"nombreCompleto" : "New York Knicks"

},

{

"abreviatura" : "OKC",

"idEquipo" : 21,

"ciudad" : "Oklahoma City",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Thunder",

"nombreCompleto" : "Oklahoma City Thunder"

},

{

"abreviatura" : "ORL",

"idEquipo" : 22,

"ciudad" : "Orlando",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Magic",

"nombreCompleto" : "Orlando Magic"

},

{

"abreviatura" : "PHI",

"idEquipo" : 23,

"ciudad" : "Philadelphia",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "76ers",

"nombreCompleto" : "Philadelphia 76ers"

},

{

"abreviatura" : "PHX",

"idEquipo" : 24,

"ciudad" : "Phoenix",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Suns",

"nombreCompleto" : "Phoenix Suns"

},

{

"abreviatura" : "POR",

"idEquipo" : 25,

"ciudad" : "Portland",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Trail Blazers",

"nombreCompleto" : "Portland Trail Blazers"

},

{

"abreviatura" : "SAC",

"idEquipo" : 26,

"ciudad" : "Sacramento",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Kings",

"nombreCompleto" : "Sacramento Kings"

},

{

"abreviatura" : "SAS",

"idEquipo" : 27,

"ciudad" : "San Antonio",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Spurs",

"nombreCompleto" : "San Antonio Spurs"

},

{

"abreviatura" : "TOR",

"idEquipo" : 28,

"ciudad" : "Toronto",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Raptors",

"nombreCompleto" : "Toronto Raptors"

},

{

"abreviatura" : "UTA",

"idEquipo" : 29,

"ciudad" : "Utah",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Jazz",

"nombreCompleto" : "Utah Jazz"

},

{

"abreviatura" : "WAS",

"idEquipo" : 30,

"ciudad" : "Washington",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Wizards",

"nombreCompleto" : "Washington Wizards"

}

]}

* 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:

NOTFOUND

Para ello se usa [la anotación @ManyToOne](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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 @JoinColum

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 @JoinColum**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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 embebible “\_”; 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 quisiéramos 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.

NOTFOUND

**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, pívot, como enumeración, que debe guardarse como ‘G’, ‘C’, ‘F’, ‘F-C’, ‘C-F’).
* pais: 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

{

"Jugador": [

{

"altura" : 198.12,

"anoDraft" : 2013,

"idEquipo" : 21,

"idJugador" : 1,

"numero" : 8,

"numeroDraft" : 32,

"peso" : 86.1825503,

"posicion" : "G",

"rondaDraft" : 2,

"pais" : "Spain",

"colegio" : "FC Barcelona",

"nombre" : "Alex",

"apellido" : "Abrines",

"foto" : null

},

{

"altura" : 182.88,

"anoDraft" : null,

"idEquipo" : 1,

"idJugador" : 2,

"numero" : 10,

"numeroDraft" : null,

"peso" : 102.05828325,

"posicion" : "G",

"rondaDraft" : null,

"pais" : "USA",

"colegio" : "St. Bonaventure",

"nombre" : "Jaylen",

"apellido" : "Adams",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2013,

"idEquipo" : 11,

"idJugador" : 3,

"numero" : 12,

"numeroDraft" : 12,

"peso" : 120.20197805000001,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "New Zealand",

"colegio" : "Pittsburgh",

"nombre" : "Steven",

"apellido" : "Adams",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2017,

"idEquipo" : 16,

"idJugador" : 4,

"numero" : 13,

"numeroDraft" : 14,

"peso" : 115.66605435000001,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Kentucky",

"nombre" : "Bam",

"apellido" : "Adebayo",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2006,

"idEquipo" : 3,

"idJugador" : 6,

"numero" : 21,

"numeroDraft" : 2,

"peso" : 113.3980925,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Texas",

"nombre" : "LaMarcus",

"apellido" : "Aldridge",

"foto" : null

},

{

"altura" : 193.04,

"anoDraft" : 2018,

"idEquipo" : 24,

"idJugador" : 8,

"numero" : 8,

"numeroDraft" : 21,

"peso" : 89.81128926000001,

"posicion" : "G",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Duke",

"nombre" : "Grayson",

"apellido" : "Allen",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2017,

"idEquipo" : 6,

"idJugador" : 9,

"numero" : 31,

"numeroDraft" : 22,

"peso" : 110.22294591,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Texas",

"nombre" : "Jarrett",

"apellido" : "Allen",

"foto" : null

},

{

"altura" : 203.2,

"anoDraft" : 2010,

"idEquipo" : 25,

"idJugador" : 10,

"numero" : 5,

"numeroDraft" : 8,

"peso" : 99.79032140000001,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Wake Forest",

"nombre" : "Al-Farouq",

"apellido" : "Aminu",

"foto" : null

},

{

"altura" : 195.57999999999998,

"anoDraft" : 2015,

"idEquipo" : 12,

"idJugador" : 11,

"numero" : 10,

"numeroDraft" : 21,

"peso" : 104.77983747,

"posicion" : "G",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Virginia",

"nombre" : "Justin",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2014,

"idEquipo" : 18,

"idJugador" : 12,

"numero" : 1,

"numeroDraft" : 30,

"peso" : 104.32624510000001,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "UCLA",

"nombre" : "Kyle",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 208.28,

"anoDraft" : 2008,

"idEquipo" : 19,

"idJugador" : 13,

"numero" : 31,

"numeroDraft" : 21,

"peso" : 108.8621688,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "California",

"nombre" : "Ryan",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2013,

"idEquipo" : 17,

"idJugador" : 15,

"numero" : 34,

"numeroDraft" : 15,

"peso" : 110.22294591,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "Greece",

"colegio" : "Filathlitikos",

"nombre" : "Giannis",

"apellido" : "Antetokounmpo",

"foto" : null

},

{

"altura" : 208.28,

"anoDraft" : 2018,

"idEquipo" : 5,

"idJugador" : 16,

"numero" : 37,

"numeroDraft" : 60,

"peso" : 90.718474,

"posicion" : "F",

"rondaDraft" : 2,

"pais" : "Greece",

"colegio" : "Dayton",

"nombre" : "Kostas",

"apellido" : "Antetokounmpo",

"foto" : null

},

{

"altura" : 200.66,

"anoDraft" : 2003,

"idEquipo" : 14,

"idJugador" : 17,

"numero" : 7,

"numeroDraft" : 3,

"peso" : 107.95498406,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Syracuse",

"nombre" : "Carmelo",

"apellido" : "Anthony",

"foto" : null

},

{

"altura" : 200.66,

"anoDraft" : 2017,

"idEquipo" : 20,

"idJugador" : 18,

"numero" : 8,

"numeroDraft" : 23,

"peso" : 108.8621688,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "United Kingdom",

"colegio" : "Indiana",

"nombre" : "OG",

"apellido" : "Anunoby",

"foto" : null

},

{

"altura" : 193.04,

"anoDraft" : null,

"idEquipo" : 24,

"idJugador" : 30053472,

"numero" : 19,

"numeroDraft" : null,

"peso" : 92.07925111,

"posicion" : null,

"rondaDraft" : null,

"pais" : "Denmark",

"colegio" : "CSKA Moscow",

"nombre" : "Gabriel",

"apellido" : "Lundberg",

"foto" : null

}

]}

**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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetomany).
* **@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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collection.html)) 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:

NOTFOUND

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**:

NOTFOUND

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 bidireccionaluno a muchos** . Si no se especifica mappedBy, el proveedor puede crear una tabla intermedia para la relación, que no es lo que se desea:

NOT FOUND

**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.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:

NOTFOUND

**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**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetomany#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](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytomany).

Por ejemplo, entre Empleado y Proyecto:

NOTFOUND

Ambos lados se mapean con la anotación @ManyToMany, **especificando** los parámetros de la **tabla de unión con**[**la anotación @JoinTable**](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/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"),

inverseJoinColumns=@JoinColumn(name="idProyecto"))

private Collection<Proyecto> proyectos;

// ...

}

joinColumns es un array y podría tener más de un elemento, si la tabla de unión tiene más de una columna de clave foránea.

@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:

NOTFOUND

Si no se indica la tabla de unión, el proveedor de JPA creará una tabla de unión con los valores por defecto:

NOTFOUND

**@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**.

### 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.

## 5. Nombre de la columna de Clave foránea

El nombre de la **columna de clave foránea se especifica con name**. 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 externa está en una tabla de colección.

## Ejercicios

 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. 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, descripcion, 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 requeriminetos 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 royle, and \u201cPlunder\u201d. The latter mode is described as a \u201crace to deposit the most Cash\u201d. 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

* [bdFreeToGame.zip](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones.files/bdFreeToGame.zip) (4 MB)
* [bdjuegosh2.mv.db](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones.files/bdjuegosh2.mv.db) (36 KB)
* [bdjuegosh2.trace.db](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/06relaciones.files/bdjuegosh2.trace.db) (161 KB)

## 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

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 (nombreEnidad\_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**.

 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;

// ...

}

@Entity

public class HistorialEmpleado {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String nombre;

private long salario;

// ...

}

En este caso, la clave primaria de la tabla HistorialEmpleado sería la clave foránea de la tabla Empleado. La entidad HistorialEmpleado tiene un campo @Id que actúa como clave primaria de la tabla.

### 6.3. Ejercicio

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.

# Ejercicios de la unidad de JPA

* [Ejercicios de apuntes](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicios-de-apuntes)
  + [Ejercicio 01.01. Creación de un proyecto con JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0101-creaci%C3%B3n-de-un-proyecto-con-jpa)
  + [Ejercicio 01.02. Creación de un archivo de configuración de persistencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0102-creaci%C3%B3n-de-un-archivo-de-configuraci%C3%B3n-de-persistencia)
  + [Ejercicio 01.03. Creación de una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0103-creaci%C3%B3n-de-una-entidad)
  + [Ejercicio 01.04. Creación de una entidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0104-creaci%C3%B3n-de-una-entidad)
  + [Ejercicio 03.01. Creación de una aplicación de persistencia de una biblioteca](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0301-creaci%C3%B3n-de-una-aplicaci%C3%B3n-de-persistencia-de-una-biblioteca)
    - [03.01. Solución](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#0301-soluci%C3%B3n)
  + [Ejercicio 04.01. Descarga y creación de la base de datos de JokeAPI](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0401-descarga-y-creaci%C3%B3n-de-la-base-de-datos-de-jokeapi)
    - [Enumeraciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#enumeraciones)
    - [Clases](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#clases)
    - [Ejercicio](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio)
  + [Ejercicio 05.01. Acceso combinado a la entidad Chiste.](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0501-acceso-combinado-a-la-entidad-chiste)
  + [Ejercicio 05.02. CLOB y BLOB de una entidad Documento](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0502-clob-y-blob-de-una-entidad-documento)
  + [Ejercicio 05.03. Conversores personalizados y enumeraciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0503-conversores-personalizados-y-enumeraciones)
  + [Ejercicio 05.04. Generación de ids con tabla](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0504-generaci%C3%B3n-de-ids-con-tabla)
  + [Ejercicio 05.05. Generación de ids con una secuencia](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0505-generaci%C3%B3n-de-ids-con-una-secuencia)
  + [Ejercicio 05.06. Ampliación de la aplicación de persistencia de una biblioteca](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0506-ampliaci%C3%B3n-de-la-aplicaci%C3%B3n-de-persistencia-de-una-biblioteca)
  + [Ejercicio 06.01. Relación uno a uno bidireccional Equipo-Entrenador](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0601-relaci%C3%B3n-uno-a-uno-bidireccional-equipo-entrenador)
  + [Ejercicio 06.02. Relación muchos a uno unidireccional Jugador-Equipo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0602-relaci%C3%B3n-muchos-a-uno-unidireccional-jugador-equipo)
  + [Ejercicio 06.03. Relación muchos a muchos unidireccional Jugador-Posición](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/14ejercicios/index.html#ejercicio-0603-relaci%C3%B3n-muchos-a-muchos-unidireccional-jugador-posici%C3%B3n)

## 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: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> instancies = 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, avaliable -> 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 EntityManagercomo 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

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.HashMap;

import java.util.Map;

import static com.pepinho.ad.biblioteca.model.BibliotecaLogger.LOG;

public class BibliotecaJpaManager {

public static final String BIBLIOTECA\_H2 = "bibliotecaH2";

public static final String BIBLIOTECA\_POSTGRES = "bibliotecaPostgres";

private static final Map<String, EntityManagerFactory> instancies = new HashMap<>();

private BibliotecaJpaManager() {

}

private static boolean isEntityManagerFactoryClosed(String unidadPersistencia) {

return !instancies.containsKey(unidadPersistencia) || instancies.get(unidadPersistencia) == null ||

!instancies.get(unidadPersistencia).isOpen();

}

public static EntityManagerFactory getEntityManagerFactory(String unidadPersistencia) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

synchronized (BibliotecaJpaManager.class) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

try {

instancies.put(unidadPersistencia, Persistence.createEntityManagerFactory(unidadPersistencia));

} catch (Exception e) {

LOG.error("Erro ó crear a unidade de persistencia " + unidadPersistencia +

": " + e.getMessage());

}

}

}

}

return instancies.get(unidadPersistencia);

}

public static EntityManager getEntityManager(String persistenceUnitName) {

return getEntityManagerFactory(persistenceUnitName).createEntityManager();

}

public static void close(String persistenceUnitName) {

if (instancies.containsKey(persistenceUnitName)) {

instancies.get(persistenceUnitName).close();

instancies.remove(persistenceUnitName);

}

}

}

 Solución: Book

import jakarta.persistence.\*;

import javax.imageio.ImageIO;

import java.awt.\*;

import java.util.ArrayList;

import java.util.List;

import java.io.\*;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.time.LocalDate;

import java.util.Objects;

/\*\*

\* @author pepecalo

\*/

@Entity

public class Book implements Serializable {

// private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long idBook;

@Column(length = 13, nullable = false, unique = true))

private String isbn;

@Column(name = "titulo", nullable = false)

private String title;

@Column(name = "autor")

private String author;

@Column(name = "anho")

private Short ano;

@Column(name = "disponible")

private Boolean available;

private byte[] portada;

private LocalDate dataPublicacion;

// @Transient // Ambas opciones son válidas

transient private List<Contido> contenido = new ArrayList<>();

private static final long serialVersionUID = 1L;

public Book() {

}

public Book(String title, String author, Short year, Boolean available) {

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

}

public Book(String isbn, String title, String author, Short year,

Boolean available) {

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

}

public Book(String isbn, String title, String author, Short year,

Boolean available, byte[] portada) {

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

this.portada = portada;

}

public Book(Long idBook, String isbn, String title, String author,

Short year, Boolean available, byte[] portada) {

this.idBook = idBook;

this.isbn = isbn;

this.title = title;

this.author = author;

this.ano = year;

this.available = available;

this.portada = portada;

}

public Long getIdBook() {

return idBook;

}

public Book setIdBook(Long idBook) {

this.idBook = idBook;

return this;

}

public String getIsbn() {

return isbn;

}

public Book setIsbn(String isbn) {

this.isbn = isbn;

return this;

}

public String getTitle() {

return title;

}

public Book setTitle(String title) {

this.title = title;

return this;

}

public String getAuthor() {

return author;

}

public Book setAuthor(String author) {

this.author = author;

return this;

}

public Short getYear() {

return ano;

}

public Book setAno(Short ano) {

this.ano = ano;

return this;

}

public Boolean isAvailable() {

return available;

}

public Book setAvailable(Boolean available) {

this.available = available;

return this;

}

public byte[] getCover() {

return portada;

}

public Book setCover(byte[] portada) {

this.portada = portada;

return this;

}

public LocalDate getDataPublicacion() {

return dataPublicacion;

}

public Book setDataPublicacion(LocalDate dataPublicacion) {

this.dataPublicacion = dataPublicacion;

return this;

}

/\*\*

\* Asigna la portada con flujos, leyendo los bytes.

\*

\* @param f

\*/

public Book setPortada(File f) {

if (f == null || !f.exists())

return this;

Path p = Paths.get(f.getAbsolutePath());

try (BufferedInputStream bi = new BufferedInputStream(Files.newInputStream(p));

ByteArrayOutputStream outputStream = new ByteArrayOutputStream()) {

byte[] buffer = new byte[4096];

int bytesLidos;

while ((bytesLidos = bi.read(buffer)) > 0) {

outputStream.write(buffer, 0, bytesLidos);

}

portada = outputStream.toByteArray();

} catch (FileNotFoundException ex) {

System.err.println("Archivo no encontrado: " + ex.getMessage());

} catch (IOException ex) {

System.err.println("Erro de E/S: " + ex.getMessage());

}

return this;

}

/\*\*

\* Asigna la portada con Java NIO, leyendo los bytes.

\*

\* @param file

\*/

public Book setPortada(String file) {

try {

Path ruta = Paths.get(file);

portada = Files.readAllBytes(ruta);

} catch (IOException ex) {

System.err.println("Error de E/S: " + ex.getMessage());

}

return this;

}

public Image getImage() {

if (portada != null) {

try (ByteArrayInputStream bis = new ByteArrayInputStream(portada)) {

Image imaxe = ImageIO.read(bis);

if(available) {

imaxe.getGraphics().drawLine(0,0, 100, 100);

}

return imaxe;

} catch (IOException e) {

}

}

return null;

}

@Override

public int hashCode() {

int hash = 7;

hash = 41 \* hash + Objects.hashCode(this.isbn);

return hash;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

final Book other = (Book) obj;

return Objects.equals(this.isbn, other.isbn);

}

@Override

public String toString() {

return idBook + "] [isbn: " + isbn + "] " + title + ". "

+ author + " (" + ano + ") [" + ((available) ? '\*' : ' ') + ']';

}

}

 Solución: Clase Contido

De momento, **no hemos declarado Contido como entidad JPA**, pero lo haremos en el futuro, cuando veamos las relaciones.

import java.util.Objects;

/\*\*

\* @autor pepecalo

\* CREATE TABLE PUBLIC.Contido (

\* idContido INTEGER NOT NULL AUTO\_INCREMENT,

\* idBook INTEGER NOT NULL,

\* contido CHARACTER LARGE OBJECT,

\* CONSTRAINT Contido\_PK PRIMARY KEY (idContido),

\* CONSTRAINT FK\_ID\_BOOK FOREIGN KEY (idBook) REFERENCES PUBLIC.Book(idBook) ON DELETE CASCADE ON UPDATE CASCADE

\* );

\* CREATE UNIQUE INDEX PRIMARY\_KEY\_9 ON PUBLIC.Contido (idContido);

\*/

public class Contido {

private Long idContido;

private String contido;

private Book book;

public Contido() {

}

public Contido(Long idBook, String contido) {

this.contido = contido;

}

public Contido(Long idContido, Long idBook) {

this.idContido = idContido;

}

public Contido(Long idContido, Long idBook, String contido) {

this.idContido = idContido;

this.contido = contido;

}

public Long getIdContido() {

return idContido;

}

public void setIdContido(Long idContido) {

this.idContido = idContido;

}

public String getContido() {

return contido;

}

public void setContido(String contido) {

this.contido = contido;

}

public Book getBook() {

return book;

}

public void setBook(Book book) {

this.book = book;

}

@Override

public int hashCode() {

return 97 \* 7 + Objects.hashCode(this.idContido);

}

@Override

public boolean equals(Object obj) {

if (obj == null || !(obj instanceof Contido other)) return false;

return Objects.equals(this.idContido, other.idContido);

}

@Override

public String toString() {

return idContido + ": " + contido;

}

 Solución: BookJPADao

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityTransaction;

import jakarta.persistence.TypedQuery;

import java.util.List;

public class BookJPADao implements Dao<Book> {

private final EntityManager em;

public BookJPADao(EntityManager em) {

this.em = em;

}

@Override

public Book get(long id) {

return em.find(Book.class, id);

}

@Override

public List<Book> getAll() {

TypedQuery<Book> query = em.createQuery("SELECT b FROM Book b", Book.class);

return query.getResultList();

}

@Override

public void save(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.persist(book);

tx.commit();

}

@Override

public void update(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.merge(book);

tx.commit();

}

@Override

public void delete(Book book) {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.remove(book);

tx.commit();

}

@Override

public boolean deleteById(long id) {

Book book = get(id);

if (book != null) {

delete(book);

return true;

}

return false;

}

@Override

public List<Integer> getAllIds() {

TypedQuery<Integer> query = em.createQuery("SELECT b.idBook FROM Book b", Integer.class);

return query.getResultList();

}

@Override

public void updateLOB(Book book, String f) {

book.setPortada(f);

update(book); // La tansacción se hace en el método update

}

@Override

public void updateLOBById(long id, String f) {

Book book = get(id);

if (book != null) {

updateLOB(book, f); // La transacción se hace en el método updateLOB, que a su vez llama a update

}

}

@Override

public void deleteAll() {

EntityTransaction tx = em.getTransaction();

tx.begin();

em.createQuery("DELETE FROM Book").executeUpdate();

tx.commit();

 Solución: BookDaoFactory

import java.util.Map;

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) {

case JDBC\_H2:

BibliotecaConnectionMaganer bibliotecaConnection = BibliotecaConnectionMaganer.getInstance();

return new BookDao(bibliotecaConnection.getConnection());

case JPA\_H2:

return new BookJPADao((BibliotecaJpaManager.getEntityManager(BibliotecaJpaManager.BIBLIOTECA\_H2)));

// ...

}

return null;

}

}

### 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 Categoriam 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 Categoria** tiene los siguientes valores:

public enum Categoria {

ANY("Any"),

MISC("Misc"),

PROGRAMMING("Programming"),

DARK("Dark"),

PUN("Pun"),

SPOOKY("Spooky"),

CHRISTMAS("Christmas");

//...

}

 Detalle de implementación de la enumeración Categoría

package com.pepinho.ad.chistes.model;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* <p>

\* Enumeración de categorías de chistes.

\* Pueden ser: Any, Misc, Programming, Dark, Pun, Spooky, Christmas

\* Atributo: nombre, de tipo cadena.

\*/

public enum Categoria {

ANY("Any"),

MISC("Misc"),

PROGRAMMING("Programming"),

DARK("Dark"),

PUN("Pun"),

SPOOKY("Spooky"),

CHRISTMAS("Christmas");

private final String nombre;

Categoria(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

/\*\*

\* Devuelve la categoría a partir de su nombre.

\*

\* @param nombre Nombre de la categoría

\* @return Categoría

\*/

public static Categoria getCategoria(String nombre) {

for (Categoria c : Categoria.values()) {

if (c.getNombre().equals(nombre)) {

return c;

}

}

return null;

}

/\*\*

\* Sobreescribe el método toString() para que devuelva el nombre de la categoría.

\*

\* @return Nombre de la categoría

\* @see java.lang.Enum#toString()

\*/

@Override

public String toString() {

return nombre;

}

}

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

public enum TipoChiste {

SINGLE("single"),

TWOPART("twopart");

//...

}

 Detalle de implementación de la enumeración TipoChiste

package com.pepinho.ad.chistes.model;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* Enumeración de tipos de chistes.

\* Pueden ser: single, twopart

\* Atributo: String nombre.

\* Constructor: TipoChiste(String nombre)

\* @see Categoria

\* @see Flag

\* @see Chiste

\*

\*/

public enum TipoChiste {

SINGLE("single"),

TWOPART("twopart");

private final String nombre;

TipoChiste(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

/\*\*

\* Devuelve el tipo de chiste a partir de su nombre.

\* @param nombre Nombre del tipo de chiste

\* @return Tipo de chiste

\*/

public static TipoChiste getTipoChiste(String nombre) {

for (TipoChiste tc : TipoChiste.values()) {

if (tc.getNombre().equals(nombre)) {

return tc;

}

}

return null;

}

/\*\*

\* Sobreescribe el método toString() para que devuelva el nombre del tipo de chiste.

\* @return Nombre del tipo de chiste

\* @see java.lang.Enum#toString()

\*/

@Override

public String toString() {

return nombre;

}

}

**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

package com.pepinho.ad.chistes.model;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* Enumeración de banderas de chistes.

\* Pueden ser: NSFW, RELIGION, POLITICAL, RACIST, SEXIST

\* Atributo: String nombre.

\* Constructor: Flag(String nombre)

\* @see Categoria

\* @link <a href="https://v2.jokeapi.dev/flags">https://v2.jokeapi.dev/flags</a>

\*/

public enum Flag {

EXPLICIT("Explicit"),

NSFW("NSFW"),

RELIGION("Religion"),

POLITICAL("Political"),

RACIST("Racist"),

SEXIST("Sexist");

private final String nombre;

Flag(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

/\*\*

\* Devuelve la bandera a partir de su nombre.

\* @param nombre Nombre de la bandera

\* @return Bandera

\*/

public static Flag getFlag(String nombre) {

// Con expresiones lambda:

return java.util.Arrays.stream(Flag.values()).filter(f -> f.getNombre().equals(nombre)).findFirst()

.orElse(null);

/\* // Con un bucle for:

// for (Flag f : Flag.values()) {

// if (f.getNombre().equals(nombre)) {

// return f;

// }

// }

// return null;

\*/

}

/\*\*

\* Sobreescribe el método toString() para que devuelva el nombre de la bandera.

\* @return Nombre de la bandera

\* @see java.lang.Enum#toString()

\*/

@Override

public String toString() {

return nombre;

}

}

**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

package com.pepinho.ad.chistes.model;

import java.util.Arrays;

/\*\*

\* Lenguajes admitidos por la API de chistes.

\* \* "jokeLanguages": [

\* \* "cs",

\* \* "de",

\* \* "en",

\* \* "es",

\* \* "fr",

\* \* "pt"

\* \* ]

\* Atributo con el nombre del lenguaje del chiste.

\*

\* @see <a href="https://sv443.net/jokeapi/v2/#languages">https://sv443.net/jokeapi/v2/#languages</a>

\*/

public enum Lenguaje {

CS("cs"),

DE("de"),

EN("en"),

ES("es"),

FR("fr"),

PT("pt");

private final String lenguaje;

/\*\*

\* Constructor de la clase Lenguajes.

\* @param lenguaje Nombre del lenguaje

\*/

Lenguaje(String lenguaje) {

this.lenguaje = lenguaje;

}

/\*\*

\* Devuelve el nombre del lenguaje.

\* @return Nombre del lenguaje

\*/

public String getLenguaje() {

return lenguaje;

}

public static Lenguaje getLenguaje(String lenguaje) {

// Con expresiones lambda:

return Arrays.stream(Lenguaje.values()).filter(l -> l.getLenguaje().equals(lenguaje)).findFirst()

.orElse(null);

/\* // Con un bucle for:

// for (Lenguaje l : Lenguaje.values()) {

// if (l.getLenguaje().equals(lenguaje)) {

// return l;

// }

// }

// return null;

\*/

}

@Override

public String toString() {

return 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

package com.pepinho.ad.chistes.model;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.Objects;

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* <p>

\* Clase que representa un chiste.

\* Atributos: id de tipo int, categoria de tipo Categoria, idiomade tipo Lenguaje, tipo de TipoChiste,

\* List<Flag> banderas, String chiste, String respuesta.

\*/

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;

/\*\*

\* Constructor de la clase Chiste.

\* @param id Identificador del chiste

\* @param categoria Categoría del chiste

\* @param idioma Idioma del chiste

\* @param tipo Tipo del chiste

\* @param chiste Chiste

\* @param respuesta Respuesta del chiste

\*/

public Chiste(int id, Categoria categoria, String idioma, TipoChiste tipo, String chiste, String respuesta) {

this.id = id;

this.categoria = categoria;

this.tipo = tipo;

this.chiste = chiste;

this.respuesta = respuesta;

this.banderas = new ArrayList<>();

this.lenguaje = Lenguaje.getLenguaje(idioma);

}

/\*\*

\* Constructor por defecto de la clase Chiste.

\*

\*/

public Chiste() {

// this.id = 0;

this.categoria = Categoria.ANY;

this.lenguaje = Lenguaje.EN;

this.tipo = TipoChiste.SINGLE;

this.chiste = "";

this.respuesta = "";

this.banderas = new ArrayList<>();

}

/\*\*

\* Devuelve el identificador del chiste.

\* @return Identificador del chiste

\*/

public int getId() {

return id;

}

/\*\*

\* Establece el identificador del chiste.

\* @param id Identificador del chiste

\*/

public void setId(int id) {

this.id = id;

}

/\*\*

\* Devuelve la categoría del chiste.

\* @return Categoría del chiste

\*/

public Categoria getCategoria() {

return categoria;

}

public String getCategoriaString() {

return categoria.getNombre();

}

/\*\*

\* Establece la categoría del chiste.

\* @param categoria Categoría del chiste

\*/

public void setCategoria(Categoria categoria) {

this.categoria = categoria;

}

public void setCategoria(String categoria) {

this.categoria = Categoria.getCategoria(categoria);

}

public Lenguaje getLenguaje() {

return lenguaje;

}

public String getLenguajeString() {

return lenguaje.getLenguaje();

}

public void setLenguaje(String lenguaje) {

this.lenguaje = Lenguaje.getLenguaje(lenguaje);

}

public void setLenguaje(Lenguaje lenguaje) {

this.lenguaje = lenguaje;

}

/\*\*

\* Devuelve el tipo del chiste.

\* @return Tipo del chiste

\*/

public TipoChiste getTipo() {

return tipo;

}

public String getTipoString() {

return tipo.getNombre();

}

/\*\*

\* Establece el tipo del chiste.

\* @param tipo Tipo del chiste

\*/

public void setTipo(TipoChiste tipo) {

this.tipo = tipo;

}

public void setTipo(String tipo) {

this.tipo = TipoChiste.getTipoChiste(tipo);

}

/\*\*

\* Devuelve las banderas del chiste.

\* @return Banderas del chiste

\*/

public List<Flag> getBanderas() {

return banderas;

}

/\*\*

\* Añade una bandera al chiste.

\* @param flag Bandera a añadir

\*/

public void addFlag(Flag flag) {

banderas.add(flag);

}

public boolean removeFlag(Flag bandera) {

return banderas.remove(bandera);

}

/\*\*

\* Si el chiste tiene esa bandera, devuelve true.

\* @param bandera Bandera a comprobar

\* @return true si el chiste tiene esa bandera, false en caso contrario

\*/

public boolean containsFlag(Flag bandera) {

return banderas.contains(bandera);

}

/\*\*

\* Devuelve el chiste como cadena de caracteres.

\* @return Chiste como String

\*/

public String getChiste() {

return chiste;

}

/\*\*

\* Establece el chiste.

\* @param chiste Chiste

\*/

public void setChiste(String chiste) {

this.chiste = chiste;

}

/\*\*

\* Devuelve la respuesta del chiste.

\* @return Respuesta del chiste

\*/

public String getRespuesta() {

return respuesta;

}

/\*\*

\* Establece la respuesta del chiste.

\* @param respuesta Respuesta del chiste

\*/

public void setRespuesta(String respuesta) {

this.respuesta = respuesta;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Chiste chiste = (Chiste) o;

return id == chiste.id;

}

@Override

public int hashCode() {

return Objects.hash(id);

}

/\*\*

\* Sobrescritura del método toString() para que devuelva el chiste.

\* Lo devuelve empleando un StringBuilder y por medio del método forEach() para recorrer la lista de banderas.

\* @return Chiste como String

\*/

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("Chiste: ").append(chiste).append(System.lineSeparator());

sb.append("Respuesta: ").append(respuesta).append(System.lineSeparator());

sb.append("Categoría: ").append(categoria).append(System.lineSeparator());

sb.append("Idioma: ").append(lenguaje).append(System.lineSeparator());

sb.append("Tipo: ").append(tipo).append(System.lineSeparator());

sb.append("Banderas: ");

banderas.forEach(b -> sb.append(b).append(" "));

sb.append(System.lineSeparator());

return sb.toString();

}

}

**B)** El adapter **ChisteDeserializer**:

 Detalle de implementación de la clase ChisteDeserializer

package com.pepinho.ad.chistes.model;

import com.google.gson.\*;

import java.lang.reflect.Type;

/\*

{

"error": false,

"category": "Programming",

"type": "twopart",

"setup": "¿Por qué C consigue todas las chicas y Java no tiene ninguna?",

"delivery": "Porque C no las trata como objetos.",

"flags": {

"nsfw": false,

"religious": false,

"political": false,

"racist": false,

"sexist": false,

"explicit": false

},

"safe": true,

"id": 6,

"lang": "es"

}

\*/

public class ChisteDeserializer implements JsonDeserializer<Chiste> {

@Override

public Chiste deserialize(JsonElement elemento, Type type,

JsonDeserializationContext jsonDeserializationContext) throws JsonParseException {

// Comprobación si es un objeto:

if (!elemento.isJsonObject())

return null;

// Creo un chiste vacío, al que le daré valor a sus atributos:

Chiste chiste = new Chiste();

// Recupero el objeto JSON del chiste

JsonObject jsonChiste = elemento.getAsJsonObject();

// Comprobación de que no hay error en la petición:

if (jsonChiste.get("error") != null && jsonChiste.get("error").getAsBoolean()) {

return null;

}

// Compruebo que cada elemento del objeto existe y lo asigno al objeto Chiste:

// La comprobación se hace con el método get() de la clase JsonObject que devuelve

// un JsonElement. Si es null, no existe el elemento.

if (jsonChiste.get("category") != null) {

chiste.setCategoria(jsonChiste.get("category").getAsString());

}

if (jsonChiste.get("type") != null) {

chiste.setTipo(jsonChiste.get("type").getAsString());

}

// En realidad, dependiendo del tipo de chiste, el setup o el delivery pueden no existir.

// Por lo que podría hacer comprobando el valor de type, pero lo dejo así para que veáis

// como se puede hacer con el método get() de la clase JsonObject.

if (jsonChiste.get("setup") != null) {

chiste.setChiste(jsonChiste.get("setup").getAsString());

if (jsonChiste.get("delivery") != null) {

chiste.setRespuesta(jsonChiste.get("delivery").getAsString());

}

} else if (jsonChiste.get("joke") != null) {

chiste.setChiste(jsonChiste.get("joke").getAsString());

}

if (jsonChiste.get("lang") != null) {

chiste.setLenguaje(jsonChiste.get("lang").getAsString());

}

if (jsonChiste.get("id") != null) {

chiste.setId(jsonChiste.get("id").getAsInt());

}

if (jsonChiste.get("flags") != null) {

JsonObject flags = jsonChiste.get("flags").getAsJsonObject();

if (flags.get("nsfw").getAsBoolean()) {

chiste.addFlag(Flag.NSFW);

}

if (flags.get("religious").getAsBoolean()) {

chiste.addFlag(Flag.RELIGION);

}

if (flags.get("political").getAsBoolean()) {

chiste.addFlag(Flag.POLITICAL);

}

if (flags.get("racist").getAsBoolean()) {

chiste.addFlag(Flag.RACIST);

}

if (flags.get("sexist").getAsBoolean()) {

chiste.addFlag(Flag.SEXIST);

}

if (flags.get("explicit").getAsBoolean()) {

chiste.addFlag(Flag.EXPLICIT);

}

}

return chiste;

}

}

**C)** La clase **ChisteTypeAdapter**:

 Detalle de implementación de la clase ChisteTypeAdapter

package com.pepinho.ad.chistes.model;

import com.google.gson.TypeAdapter;

import com.google.gson.stream.JsonReader;

import com.google.gson.stream.JsonToken;

import com.google.gson.stream.JsonWriter;

import java.io.IOException;

/\*

Formato de JSON:

{

"id": 1,

"category": "Programming",

"type": "single",

"joke": "Chuck Norris can write multithreaded applications with a single thread.",

"flags": {

"nsfw": false,

"religious": false,

"political": false,

"racist": false,

"sexist": false

},

"lang": "en"

\*/

/\*\*

\* Updated by Pepinho on 16/01/2025.

\* Clase que adaptará el tipo Chiste para que pueda ser serializado y deserializado por Gson.

\*

\* @see com.google.gson.Gson

\* @see com.google.gson.TypeAdapter

\* @see com.google.gson.GsonBuilder

\* @see com.google.gson.JsonDeserializer

\*/

public class ChisteTypeAdapter extends TypeAdapter<Chiste> {

@Override

public void write(JsonWriter jsonWriter, Chiste chiste) throws IOException {

jsonWriter.beginObject();

jsonWriter.name("id").value(chiste.getId());

jsonWriter.name("category").value(chiste.getCategoriaString());

jsonWriter.name("type").value(chiste.getTipoString());

if (chiste.getTipo() == TipoChiste.SINGLE) {

jsonWriter.name("joke").value(chiste.getChiste());

} else {

jsonWriter.name("setup").value(chiste.getChiste());

jsonWriter.name("delivery").value(chiste.getRespuesta());

}

jsonWriter.name("flags");

jsonWriter.beginObject();

// Recorremos todas las banderas y asignamos el valor verdadero o falso si el chiste la contiene o no, respectivamente.

// Puede hacerse por medio del método containsFlag() de la clase Chiste o recoger las banderas

// del chiste e invocar el método contains() de la clase List.

for (Flag flag : Flag.values()) {

jsonWriter.name(flag.getNombre().toLowerCase()).value(chiste.containsFlag(flag));

}

jsonWriter.endObject();

jsonWriter.name("lang").value(chiste.getLenguajeString());

jsonWriter.endObject();

}

/\*\*

\* Método que deserializa un objeto Chiste a partir de un JsonReader.

\*

\* @param reader JsonReader que contiene el objeto Chiste

\* @return Objeto Chiste

\* @throws IOException Si hay un error de E/S

\* @see com.google.gson.stream.JsonReader

\* @see com.google.gson.stream.JsonToken

\*/

@Override

public Chiste read(JsonReader reader) throws IOException {

if(reader.peek()== JsonToken.NULL || reader.peek()!= JsonToken.BEGIN\_OBJECT){

// reader.nextNull();

return null;

}

reader.beginObject();

Chiste chiste = new Chiste();

while (reader.peek() != JsonToken.END\_OBJECT) {

String name = reader.nextName();

switch (name) {

case "id" -> chiste.setId(reader.nextInt());

case "category" -> chiste.setCategoria(Categoria.getCategoria(reader.nextString()));

case "type" -> chiste.setTipo(TipoChiste.getTipoChiste(reader.nextString()));

case "joke", "setup" -> chiste.setChiste(reader.nextString());

case "delivery" -> chiste.setRespuesta(reader.nextString());

case "flags" -> // Para hacerlo más modular he puesto el código en un método aparte.

readFlags(reader, chiste);

case "lang" -> chiste.setLenguaje(reader.nextString());

default -> reader.skipValue();

}

}

reader.endObject();

return chiste;

}

private void readFlags(JsonReader reader, Chiste chiste) throws IOException {

reader.beginObject();

while (reader.peek() != JsonToken.END\_OBJECT) {

String flagName = reader.nextName();

switch (flagName) {

case "nsfw" -> {

if (reader.nextBoolean()) chiste.addFlag(Flag.NSFW);

}

case "religious" -> {

if (reader.nextBoolean()) chiste.addFlag(Flag.RELIGION);

}

case "political" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.POLITICAL);

}

case "racist" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.RACIST);

}

case "sexist" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.SEXIST);

}

case "explicit" -> {

if (reader.nextBoolean())

chiste.addFlag(Flag.EXPLICIT);

}

default -> reader.skipValue();

}

}

reader.endObject();

}

}

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

 Detalle de implementación de la interfaz IChisteDAO

package com.pepinho.ad.chistes.model;

import java.io.Writer;

public interface IChisteDAO {

String getRandomJokeAsString();

String getJokeAsString(String categoria, String[] tipo, String[] banderas);

String getJokeAsString(String categoria, String[] tipo, String[] banderas, String idioma);

Chiste getRandomJoke();

Chiste getJoke(String categoria, String[] tipo, String[] banderas);

Chiste getJoke(String categoria, String[] tipo, String[] banderas, String idioma);

Chiste getJokeById(int id);

void saveJokeAsJson(Chiste chiste, Writer writer);

}

*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

package com.pepinho.ad.chistes.model;

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.Writer;

import java.net.MalformedURLException;

import java.net.URI;

import java.net.URISyntaxException;

import java.util.Arrays;

import java.util.Objects;

/\*\*

\* Created by Pepe Calo on 07/11/2023

\* Implementación de la interfaz IChisteDAO que consulta un chiste en un archivo Json

\* mediante la librería Gson.

\* La API de chistes utilizada es:

\* <a href="https://v2.jokeapi.dev/joke/">...</a>

\*

\* @see IChisteDAO

\* @see Chiste

\* @see Gson

\* @see GsonBuilder

\* @see com.google.gson.JsonObject

\* @see com.google.gson.JsonParser

\*/

public class ChisteDAO implements IChisteDAO {

private final Gson gson;

// https://v2.jokeapi.dev/joke/Programming,Miscellaneous?blacklistFlags=nsfw,religious

private static final String BASE\_URL = "https://v2.jokeapi.dev/joke/";

private static final String ENDPOINT = "?format=json";

private static final int NO\_ID = 0;

private static final String SINGLE = "single";

/\*\*

\* Constructor de la clase ChisteDAO.

\* Si deseas emplear las clases ChisteSerializer y ChisteDeserializer, debes comentar la línea con ChisteTypeAdapter

\* y no comentar las de los otros dos adaptadores.

\*/

public ChisteDAO() {

gson = new GsonBuilder().setPrettyPrinting()

// .registerTypeAdapter(Chiste.class, new ChisteDeserializer())

// .registerTypeAdapter(Chiste.class, new ChisteSerializer())

.registerTypeAdapter(Chiste.class, new ChisteTypeAdapter())

.create();

}

private String getURL(String categoria, String[] tipo, String[] banderas, String idioma, int id) {

String url = BASE\_URL + categoria + ENDPOINT;

if (tipo != null && tipo.length > 0) {

// Concateno los elementos no nulos media stream de un array de String. En el caso de que no haya ninguno, devuelvo un Optional vacío.

String tipos = Arrays.stream(tipo).filter(Objects::nonNull).reduce((s, s2) -> s + "," + s2).orElse(null);

if(tipos!=null && !tipos.isEmpty()){

url += "&type=" + tipos;

}

}

if (banderas != null && banderas.length > 0) {

String flags = Arrays.stream(banderas).filter(Objects::nonNull).reduce((s, s2) -> s + "," + s2).orElse(null);

if(flags!=null && !flags.isEmpty()){

url += "&blacklistFlags=" + flags;

}

}

if (idioma != null && !idioma.isEmpty()) {

url += "&lang=" + idioma;

}

if (id > 0) {

url += "&idRange=" + id;

}

System.out.println("url = " + url);

return url;

}

private Chiste getJoke(String url) {

try (BufferedReader is = new BufferedReader(new InputStreamReader(new URI(url).toURL().openStream()))) {

return gson.fromJson(is, Chiste.class);

} catch (MalformedURLException e) {

System.err.println("Error en la URL: " + e.getMessage());

} catch (IOException e) {

System.err.println("Erro E/S: " + e.getMessage());

} catch (URISyntaxException e) {

throw new RuntimeException(e);

}

return null;

}

private String getJokeAsString(String url) {

Chiste chiste = getJoke(url);

return (chiste!=null) ? chiste.getChiste() + System.lineSeparator() + chiste.getRespuesta() : "";

}

@Override

public String getJokeAsString(String categoria, String[] tipo, String[] banderas) {

return getJokeAsString(getURL(categoria, tipo, banderas, null, NO\_ID));

}

@Override

public Chiste getJoke(String categoria, String[] tipo, String[] banderas) {

return getJoke(getURL(categoria, tipo, banderas, null, NO\_ID));

}

@Override

public String getJokeAsString(String categoria, String[] tipo, String[] banderas, String idioma) {

return getJokeAsString(getURL(categoria, tipo, banderas, idioma, NO\_ID));

}

@Override

public Chiste getJoke(String categoria, String[] tipo, String[] banderas, String idioma) {

return getJoke(getURL(categoria, tipo, banderas, idioma, NO\_ID));

}

@Override

public Chiste getJokeById(int id) {

return getJoke(getURL("Any", null, null, null, id));

}

@Override

public void saveJokeAsJson(Chiste chiste, Writer writer) {

gson.toJson(chiste, writer);

}

@Override

public String getRandomJokeAsString() {

System.out.println(BASE\_URL + "Any");

return getJokeAsString(BASE\_URL + "Any");

}

@Override

public Chiste getRandomJoke() {

return getJoke(BASE\_URL + "Any");

}

}

#### 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

package com.pepinho.ad.chistes.model;

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.HashMap;

import java.util.Map;

import static com.pepinho.ad.chistes.model.ChisteLogger.LOG;

public class ChisteJpaManager {

public static final String BIBLIOTECA\_H2 = "chistesH2";

public static final String BIBLIOTECA\_POSTGRES = "chistesPostgres";

private static final Map<String, EntityManagerFactory> instancies = new HashMap<>();

private ChisteJpaManager() {

}

private static boolean isEntityManagerFactoryClosed(String unidadPersistencia) {

return !instancies.containsKey(unidadPersistencia) || instancies.get(unidadPersistencia) == null ||

!instancies.get(unidadPersistencia).isOpen();

}

public static EntityManagerFactory getEntityManagerFactory(String unidadPersistencia) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

synchronized (ChisteJpaManager.class) {

if (isEntityManagerFactoryClosed(unidadPersistencia)) {

try {

instancies.put(unidadPersistencia, Persistence.createEntityManagerFactory(unidadPersistencia));

} catch (Exception e) {

LOG.error("Erro ó crear a unidade de persistencia " + unidadPersistencia +

": " + e.getMessage());

}

}

}

}

return instancies.get(unidadPersistencia);

}

public static EntityManager getEntityManager(String persistenceUnitName) {

return getEntityManagerFactory(persistenceUnitName).createEntityManager();

}

public static void close(String persistenceUnitName) {

if (instancies.containsKey(persistenceUnitName)) {

instancies.get(persistenceUnitName).close();

instancies.remove(persistenceUnitName);

}

}

}

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

 Solución de Chiste

package com.pepinho.ad.chistes.model;

import jakarta.persistence.\*;

import java.util.ArrayList;

@Entity

public class Chiste implements java.io.Serializable {

@Id

@Column(name = "idChiste")

private int id;

private Categoria categoria;

private TipoChiste tipo;

// Como se trata de una relación muchos a muchos, se emplea la anotación @ElementCollection

// H2 admite el tipo de dato Array de enteros (TINYINT ARRAY), prueba a no poner la anotación @ElementCollection ni @CollectionTable

@ElementCollection // Para que se cree una tabla intermedia

@Enumerated(EnumType.STRING)

@CollectionTable(name = "FlagsChiste", joinColumns = @JoinColumn(name = "idChiste"))

private final List<Flag> banderas;

private String chiste;

private String respuesta;

private Lenguaje lenguaje;

//...

}

**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
* package com.pepinho.ad.chistes;
* import com.pepinho.ad.chistes.model.Chiste;
* import com.pepinho.ad.chistes.model.ChisteDAO;
* import com.pepinho.ad.chistes.model.ChisteJpaManager;
* import static java.lang.Thread.sleep;
* public class ChisteDownloader implements Runnable {
* private static final long tiempoEspera = 550;
* private static ChisteDownloader instance;
* private static final int MAX\_CHISTES = 200;
* private ChisteDAO chisteDAO;
* private int numeroChistes = MAX\_CHISTES;
* private ChisteDownloader() {
* chisteDAO = new ChisteDAO();
* }
* public static ChisteDownloader getInstance() {
* if (instance == null) {
* synchronized (ChisteDownloader.class) {
* if (instance == null) {
* instance = new ChisteDownloader();
* }
* }
* }
* return instance;
* }
* public void setNumeroChistes(int numeroChistes) {
* this.numeroChistes = numeroChistes;
* }
* @Override
* public void run() {
* chisteDAO = new ChisteDAO();
* var emf = ChisteJpaManager.getEntityManagerFactory(ChisteJpaManager.BIBLIOTECA\_H2);
* var em = emf.createEntityManager();
* for (int i = 0; i < numeroChistes; i++) {
* Chiste chiste = chisteDAO.getJokeById(i);
* if (chiste != null) {
* try {
* em.getTransaction().begin();
* em.persist(chiste);
* em.getTransaction().commit();
* } catch (Exception e) {
* em.getTransaction().rollback();
* }
* System.out.print("\*");
* try {
* sleep(tiempoEspera);
* } catch (InterruptedException e) {
* System.out.println("Error en el hilo");
* }
* }
* }
* }
* }
* **D)** **Main** que descarga los chistes y los guarda en la base de datos.

 Solución de Main

public static void main(String[] args) {

ChisteDownloader chisteDownloader = ChisteDownloader.getInstance();

chisteDownloader.setNumeroChistes(300);

Thread thread = new Thread(chisteDownloader);

thread.start();

}

### 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;

// ...

}

### 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 **fotografia** 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 ejecicio anterior con Persona, haz aque 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 compartirla. 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 x 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.

### 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

{

"Equipo": [

{

"abreviatura" : "ATL",

"idEquipo" : 1,

"ciudad" : "Atlanta",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Hawks",

"nombreCompleto" : "Atlanta Hawks"

},

{

"abreviatura" : "BOS",

"idEquipo" : 2,

"ciudad" : "Boston",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Celtics",

"nombreCompleto" : "Boston Celtics"

},

{

"abreviatura" : "BKN",

"idEquipo" : 3,

"ciudad" : "Brooklyn",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Nets",

"nombreCompleto" : "Brooklyn Nets"

},

{

"abreviatura" : "CHA",

"idEquipo" : 4,

"ciudad" : "Charlotte",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Hornets",

"nombreCompleto" : "Charlotte Hornets"

},

{

"abreviatura" : "CHI",

"idEquipo" : 5,

"ciudad" : "Chicago",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Bulls",

"nombreCompleto" : "Chicago Bulls"

},

{

"abreviatura" : "CLE",

"idEquipo" : 6,

"ciudad" : "Cleveland",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Cavaliers",

"nombreCompleto" : "Cleveland Cavaliers"

},

{

"abreviatura" : "DAL",

"idEquipo" : 7,

"ciudad" : "Dallas",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Mavericks",

"nombreCompleto" : "Dallas Mavericks"

},

{

"abreviatura" : "DEN",

"idEquipo" : 8,

"ciudad" : "Denver",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Nuggets",

"nombreCompleto" : "Denver Nuggets"

},

{

"abreviatura" : "DET",

"idEquipo" : 9,

"ciudad" : "Detroit",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Pistons",

"nombreCompleto" : "Detroit Pistons"

},

{

"abreviatura" : "GSW",

"idEquipo" : 10,

"ciudad" : "Golden State",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Warriors",

"nombreCompleto" : "Golden State Warriors"

},

{

"abreviatura" : "HOU",

"idEquipo" : 11,

"ciudad" : "Houston",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Rockets",

"nombreCompleto" : "Houston Rockets"

},

{

"abreviatura" : "IND",

"idEquipo" : 12,

"ciudad" : "Indiana",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Pacers",

"nombreCompleto" : "Indiana Pacers"

},

{

"abreviatura" : "LAC",

"idEquipo" : 13,

"ciudad" : "LA",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Clippers",

"nombreCompleto" : "LA Clippers"

},

{

"abreviatura" : "LAL",

"idEquipo" : 14,

"ciudad" : "Los Angeles",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Lakers",

"nombreCompleto" : "Los Angeles Lakers"

},

{

"abreviatura" : "MEM",

"idEquipo" : 15,

"ciudad" : "Memphis",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Grizzlies",

"nombreCompleto" : "Memphis Grizzlies"

},

{

"abreviatura" : "MIA",

"idEquipo" : 16,

"ciudad" : "Miami",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Heat",

"nombreCompleto" : "Miami Heat"

},

{

"abreviatura" : "MIL",

"idEquipo" : 17,

"ciudad" : "Milwaukee",

"conferencia" : "EAST",

"division" : "CENTRAL",

"nombre" : "Bucks",

"nombreCompleto" : "Milwaukee Bucks"

},

{

"abreviatura" : "MIN",

"idEquipo" : 18,

"ciudad" : "Minnesota",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Timberwolves",

"nombreCompleto" : "Minnesota Timberwolves"

},

{

"abreviatura" : "NOP",

"idEquipo" : 19,

"ciudad" : "New Orleans",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Pelicans",

"nombreCompleto" : "New Orleans Pelicans"

},

{

"abreviatura" : "NYK",

"idEquipo" : 20,

"ciudad" : "New York",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Knicks",

"nombreCompleto" : "New York Knicks"

},

{

"abreviatura" : "OKC",

"idEquipo" : 21,

"ciudad" : "Oklahoma City",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Thunder",

"nombreCompleto" : "Oklahoma City Thunder"

},

{

"abreviatura" : "ORL",

"idEquipo" : 22,

"ciudad" : "Orlando",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Magic",

"nombreCompleto" : "Orlando Magic"

},

{

"abreviatura" : "PHI",

"idEquipo" : 23,

"ciudad" : "Philadelphia",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "76ers",

"nombreCompleto" : "Philadelphia 76ers"

},

{

"abreviatura" : "PHX",

"idEquipo" : 24,

"ciudad" : "Phoenix",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Suns",

"nombreCompleto" : "Phoenix Suns"

},

{

"abreviatura" : "POR",

"idEquipo" : 25,

"ciudad" : "Portland",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Trail Blazers",

"nombreCompleto" : "Portland Trail Blazers"

},

{

"abreviatura" : "SAC",

"idEquipo" : 26,

"ciudad" : "Sacramento",

"conferencia" : "WEST",

"division" : "PACIFIC",

"nombre" : "Kings",

"nombreCompleto" : "Sacramento Kings"

},

{

"abreviatura" : "SAS",

"idEquipo" : 27,

"ciudad" : "San Antonio",

"conferencia" : "WEST",

"division" : "SOUTHWEST",

"nombre" : "Spurs",

"nombreCompleto" : "San Antonio Spurs"

},

{

"abreviatura" : "TOR",

"idEquipo" : 28,

"ciudad" : "Toronto",

"conferencia" : "EAST",

"division" : "ATLANTIC",

"nombre" : "Raptors",

"nombreCompleto" : "Toronto Raptors"

},

{

"abreviatura" : "UTA",

"idEquipo" : 29,

"ciudad" : "Utah",

"conferencia" : "WEST",

"division" : "NORTHWEST",

"nombre" : "Jazz",

"nombreCompleto" : "Utah Jazz"

},

{

"abreviatura" : "WAS",

"idEquipo" : 30,

"ciudad" : "Washington",

"conferencia" : "EAST",

"division" : "SOUTHEAST",

"nombre" : "Wizards",

"nombreCompleto" : "Washington Wizards"

}

]}

* 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, pívot, como enumeración, que debe guardarse como ‘G’, ‘C’, ‘F’, ‘F-C’, ‘C-F’).
* pais: 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

{

"Jugador": [

{

"altura" : 198.12,

"anoDraft" : 2013,

"idEquipo" : 21,

"idJugador" : 1,

"numero" : 8,

"numeroDraft" : 32,

"peso" : 86.1825503,

"posicion" : "G",

"rondaDraft" : 2,

"pais" : "Spain",

"colegio" : "FC Barcelona",

"nombre" : "Alex",

"apellido" : "Abrines",

"foto" : null

},

{

"altura" : 182.88,

"anoDraft" : null,

"idEquipo" : 1,

"idJugador" : 2,

"numero" : 10,

"numeroDraft" : null,

"peso" : 102.05828325,

"posicion" : "G",

"rondaDraft" : null,

"pais" : "USA",

"colegio" : "St. Bonaventure",

"nombre" : "Jaylen",

"apellido" : "Adams",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2013,

"idEquipo" : 11,

"idJugador" : 3,

"numero" : 12,

"numeroDraft" : 12,

"peso" : 120.20197805000001,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "New Zealand",

"colegio" : "Pittsburgh",

"nombre" : "Steven",

"apellido" : "Adams",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2017,

"idEquipo" : 16,

"idJugador" : 4,

"numero" : 13,

"numeroDraft" : 14,

"peso" : 115.66605435000001,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Kentucky",

"nombre" : "Bam",

"apellido" : "Adebayo",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2006,

"idEquipo" : 3,

"idJugador" : 6,

"numero" : 21,

"numeroDraft" : 2,

"peso" : 113.3980925,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Texas",

"nombre" : "LaMarcus",

"apellido" : "Aldridge",

"foto" : null

},

{

"altura" : 193.04,

"anoDraft" : 2018,

"idEquipo" : 24,

"idJugador" : 8,

"numero" : 8,

"numeroDraft" : 21,

"peso" : 89.81128926000001,

"posicion" : "G",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Duke",

"nombre" : "Grayson",

"apellido" : "Allen",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2017,

"idEquipo" : 6,

"idJugador" : 9,

"numero" : 31,

"numeroDraft" : 22,

"peso" : 110.22294591,

"posicion" : "C",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Texas",

"nombre" : "Jarrett",

"apellido" : "Allen",

"foto" : null

},

{

"altura" : 203.2,

"anoDraft" : 2010,

"idEquipo" : 25,

"idJugador" : 10,

"numero" : 5,

"numeroDraft" : 8,

"peso" : 99.79032140000001,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Wake Forest",

"nombre" : "Al-Farouq",

"apellido" : "Aminu",

"foto" : null

},

{

"altura" : 195.57999999999998,

"anoDraft" : 2015,

"idEquipo" : 12,

"idJugador" : 11,

"numero" : 10,

"numeroDraft" : 21,

"peso" : 104.77983747,

"posicion" : "G",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Virginia",

"nombre" : "Justin",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 205.74,

"anoDraft" : 2014,

"idEquipo" : 18,

"idJugador" : 12,

"numero" : 1,

"numeroDraft" : 30,

"peso" : 104.32624510000001,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "UCLA",

"nombre" : "Kyle",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 208.28,

"anoDraft" : 2008,

"idEquipo" : 19,

"idJugador" : 13,

"numero" : 31,

"numeroDraft" : 21,

"peso" : 108.8621688,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "California",

"nombre" : "Ryan",

"apellido" : "Anderson",

"foto" : null

},

{

"altura" : 210.82,

"anoDraft" : 2013,

"idEquipo" : 17,

"idJugador" : 15,

"numero" : 34,

"numeroDraft" : 15,

"peso" : 110.22294591,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "Greece",

"colegio" : "Filathlitikos",

"nombre" : "Giannis",

"apellido" : "Antetokounmpo",

"foto" : null

},

{

"altura" : 208.28,

"anoDraft" : 2018,

"idEquipo" : 5,

"idJugador" : 16,

"numero" : 37,

"numeroDraft" : 60,

"peso" : 90.718474,

"posicion" : "F",

"rondaDraft" : 2,

"pais" : "Greece",

"colegio" : "Dayton",

"nombre" : "Kostas",

"apellido" : "Antetokounmpo",

"foto" : null

},

{

"altura" : 200.66,

"anoDraft" : 2003,

"idEquipo" : 14,

"idJugador" : 17,

"numero" : 7,

"numeroDraft" : 3,

"peso" : 107.95498406,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "USA",

"colegio" : "Syracuse",

"nombre" : "Carmelo",

"apellido" : "Anthony",

"foto" : null

},

{

"altura" : 200.66,

"anoDraft" : 2017,

"idEquipo" : 20,

"idJugador" : 18,

"numero" : 8,

"numeroDraft" : 23,

"peso" : 108.8621688,

"posicion" : "F",

"rondaDraft" : 1,

"pais" : "United Kingdom",

"colegio" : "Indiana",

"nombre" : "OG",

"apellido" : "Anunoby",

"foto" : null

},

{

"altura" : 193.04,

"anoDraft" : null,

"idEquipo" : 24,

"idJugador" : 30053472,

"numero" : 19,

"numeroDraft" : null,

"peso" : 92.07925111,

"posicion" : null,

"rondaDraft" : null,

"pais" : "Denmark",

"colegio" : "CSKA Moscow",

"nombre" : "Gabriel",

"apellido" : "Lundberg",

"foto" : null

}

]}

**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.

# Dependencias Maven.

* [1. Dependencias Maven.](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#1-dependencias-maven)
  + [2. Logging](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#2-logging)
  + [2. Json](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#2-json)
    - [2.1 Gson](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#21-gson)
    - [2.2. Jackson Databind](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#22-jackson-databind)
    - [2.3. Jackson Core](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#23-jackson-core)
  + [3. JUnit](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#3-junit)
  + [4. Drivers JDBC](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#4-drivers-jdbc)
    - [4.1. H2](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#41-h2)
    - [4.2. SQLite JDBC Driver](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#42-sqlite-jdbc-driver)
    - [4.3. PostgreSQL JDBC Driver](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#43-postgresql-jdbc-driver)
    - [4.4. MySQL Connector/J](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#44-mysql-connectorj)
    - [4.5. HyperSQL Database (HSQLDB)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#45-hypersql-database-hsqldb)
  + [5. Dependencias para JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#5-dependencias-para-jpa)
    - [5.1. Jakarta Persistence API (JPA)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#51-jakarta-persistence-api-jpa)
    - [5.2. Hibernate](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#52-hibernate)
    - [5.3. EclipseLink](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#53-eclipselink)
  + [6. Dependencias para Spring](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#6-dependencias-para-spring)
    - [6.1. Spring Core](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#61-spring-core)
    - [6.2. Spring Boot](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#62-spring-boot)
    - [6.3. Spring Data JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#63-spring-data-jpa)
    - [6.4. Spring Boot Starter Data JPA](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#64-spring-boot-starter-data-jpa)
  + [7. Lenguajes sobre JVM](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#7-lenguajes-sobre-jvm)
    - [7.1. Kotlin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#71-kotlin)
    - [7.2. Scala](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#72-scala)
* [Referencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/03orm/jpa/16dependencias/index.html#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>

<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 aparicion 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 DriverMaganer, 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:hsql://localhost/testdb URL para fichero: jdbc:hsqldb:file:nombrebasededatos

### 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/artifact/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/artifact/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://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](https://mvnrepository.com/)
* [Central Sonatype Repository](https://central.sonatype.com/)
* [SLF4J](http://www.slf4j.org/)
* [Gson](https://github.com/google/gson)
* [Spring](https://spring.io/)