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..

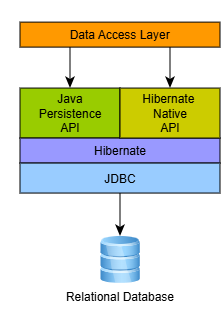
## Mapeo Objeto-Relacional (ORM)

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>

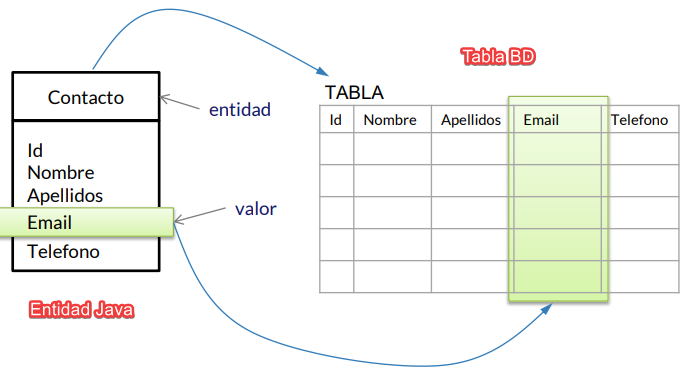
<artifactId>h2</artifactId>

<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);

# 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).

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

)

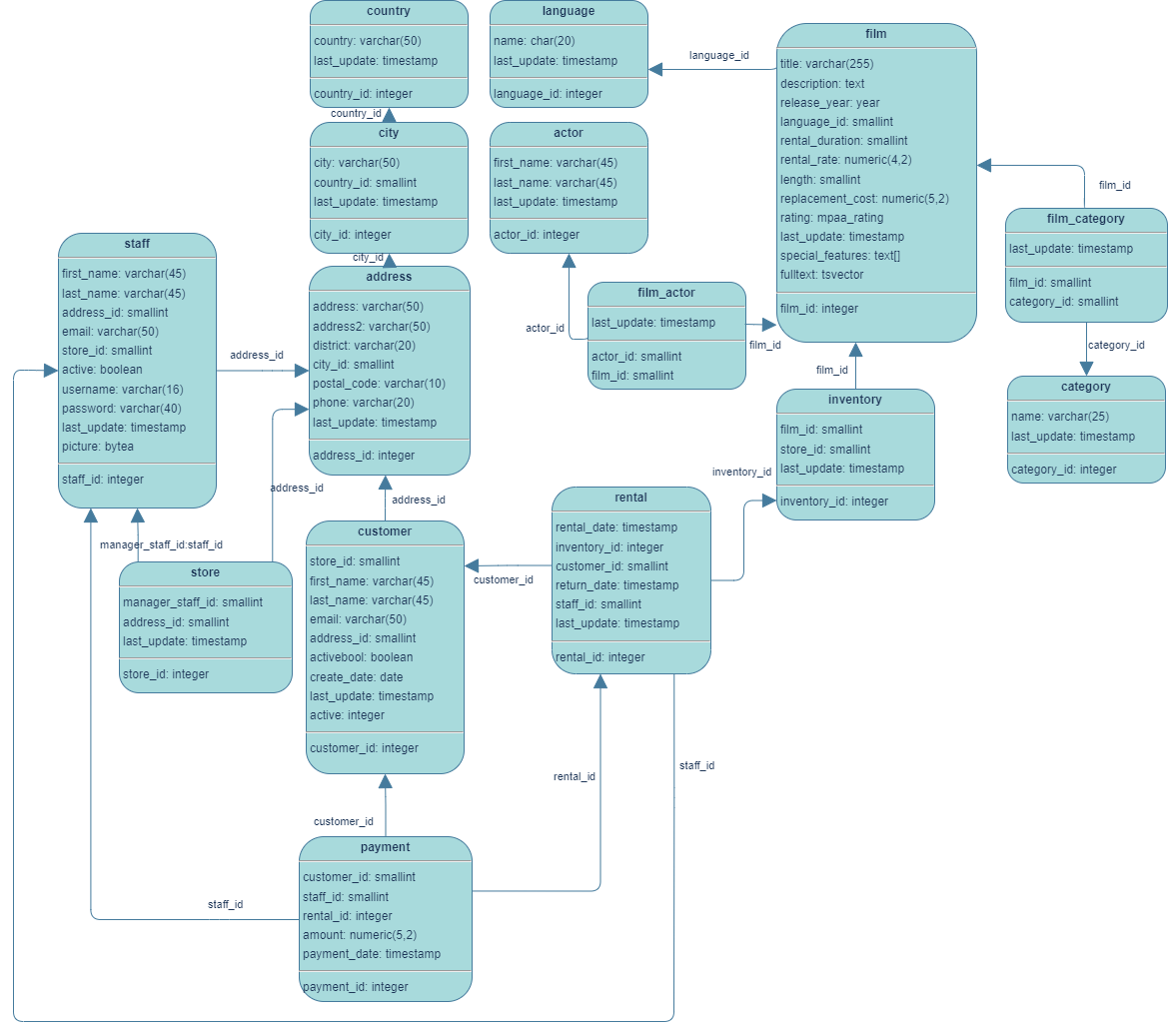
De momento, mapea el campo fulltext como un String:

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**.

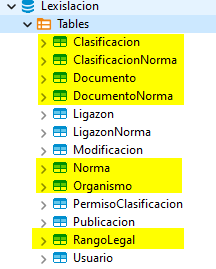
### 5. Base de datos de legislación

##### **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).