AMS Bootcamp Fullstack Developer.

**JPA, Hibernate y Spring Data**



Índice

[1. Personal Notebook 4](#_Toc157633257)

[1.1 Datos alumno 4](#_Toc157633258)

[1.2 Personal Notebook 4](#_Toc157633259)

[2. Hibernate JPA y Spring Data 5](#_Toc157633260)

[2.1 Objetivos de aprendizaje 5](#_Toc157633261)

[2.2 ¿Qué es un ORM? 6](#_Toc157633262)

[2.2.1 Objetivos 6](#_Toc157633263)

[2.2.2 Recursos formativos 6](#_Toc157633264)

[2.2.3 Posibles preguntas 6](#_Toc157633265)

[2.2.4 Tareas 6](#_Toc157633266)

[2.2.5 Dudas 6](#_Toc157633267)

[2.3 Patrón DAO vs “Patrón” Repository 7](#_Toc157633268)

[2.3.1 Objetivos 7](#_Toc157633269)

[2.3.2 Recursos formativos 7](#_Toc157633270)

[2.3.3 Posibles preguntas 7](#_Toc157633271)

[2.3.4 Tareas 7](#_Toc157633272)

[2.3.5 Dudas 7](#_Toc157633273)

[2.4 Hibernate JPA 8](#_Toc157633274)

[2.4.1 Objetivos 8](#_Toc157633275)

[2.4.2 Mapeo de entidades y columnas. 8](#_Toc157633276)

[2.4.3 Mapeo de relaciones. 9](#_Toc157633277)

[2.4.4 Recursos formativos 9](#_Toc157633278)

[2.5 JPA Repository: Consultas 10](#_Toc157633279)

[2.5.1 Consultas a partir de propiedades 10](#_Toc157633280)

[2.5.2 Objetivos 11](#_Toc157633281)

[2.5.3 Recursos formativos 11](#_Toc157633282)

[2.5.4 Posibles preguntas 11](#_Toc157633283)

[2.5.5 Tareas 12](#_Toc157633284)

[2.5.6 Dudas 12](#_Toc157633285)

[2.6 JPA Repository: JPQL, Criteria 13](#_Toc157633286)

[2.6.1 ¿Qué es JPQL? 13](#_Toc157633287)

[2.6.2 Uso de @Query 13](#_Toc157633288)

[2.6.3 Objetivos 14](#_Toc157633289)

[2.6.4 Recursos formativos 14](#_Toc157633290)

[2.6.5 Posibles preguntas 14](#_Toc157633291)

[2.6.6 Tareas 14](#_Toc157633292)

[2.6.7 Dudas 14](#_Toc157633293)

[2.7 Ejercicio práctico: Consultas JPA 15](#_Toc157633294)

[2.7.1 Objetivos 15](#_Toc157633295)

[2.7.2 Recursos formativos 15](#_Toc157633296)

[2.7.3 Posibles preguntas 15](#_Toc157633297)

[2.7.4 Tareas 15](#_Toc157633298)

[2.7.5 Dudas 15](#_Toc157633299)

# Personal Notebook

## Datos alumno

* DAS **A926220**
* Nombre Apellidos **Sandra Moriana Herrera**
* email **sandra.moriana@eviden.com**
* URL de GitHub corporativo **https://github.com/Sanmohe**

## Personal Notebook

Este es tu cuaderno de trabajo. En él tendrás que registrar tu aprendizaje, tus avances y tus dudas. ¡Vas a generar tu propio material!

Puede ser una gran herramienta si la utilizas correctamente.

El cuaderno estará compartido con los tutores y el resto de colaboradores para ver tu progreso y ayudarte.

Al final del Bootcamp tendrás que entregarlo como evidencia de tu aprendizaje.



Instrucciones de uso:

El fichero está ubicado en tu carpeta del espacio que tenemos en Teams.

Siempre deberá ser editado en modo online y/o con guardado automático en la nube.

Responde a las preguntas con la información que necesites para dar una respuesta cualificada al formador o a tus compañeros. No es necesario copiar texto solo por “rellenar”. Cita las Fuentes en caso de utilizar otros recursos.

# Hibernate JPA y Spring Data

## Objetivos de aprendizaje

* Objetivo: **Adquirir los conocimientos necesarios para trabajar con sistemas ORM (Hibernate JPA) y con Spring Data.**
* Conceptos principales para adquirir:
  + Hibernate JPA,
  + Patrón DAO vs Repository,
  + Configuración,
  + Consultas y Relaciones.
  + Integración con Spring Data y MVC

Lecciones:

1. ¿Qué es un ORM?
2. Explicación de patrón DAO
3. Hibernate JPA: Entidades, Repositorios, Consultas
4. JPA Repository: Consultas (predefinidas,JPQL,Criteria…)
5. Ejercicio práctico: Consultas JPA
6. Spring Data JPA
7. Ejercicio completo Web + JPA

## ¿Qué es un ORM?

### Objetivos

1. **Introducción a ORM de manera genérica**
2. **Conocer los conceptos que hay detrás de estos enfoques**

### Recursos formativos

* + [(4) ¿Qué es un ORM? Object Relational Mapping - YouTube](https://www.youtube.com/shorts/Egr7EjiamYY))

### Posibles preguntas

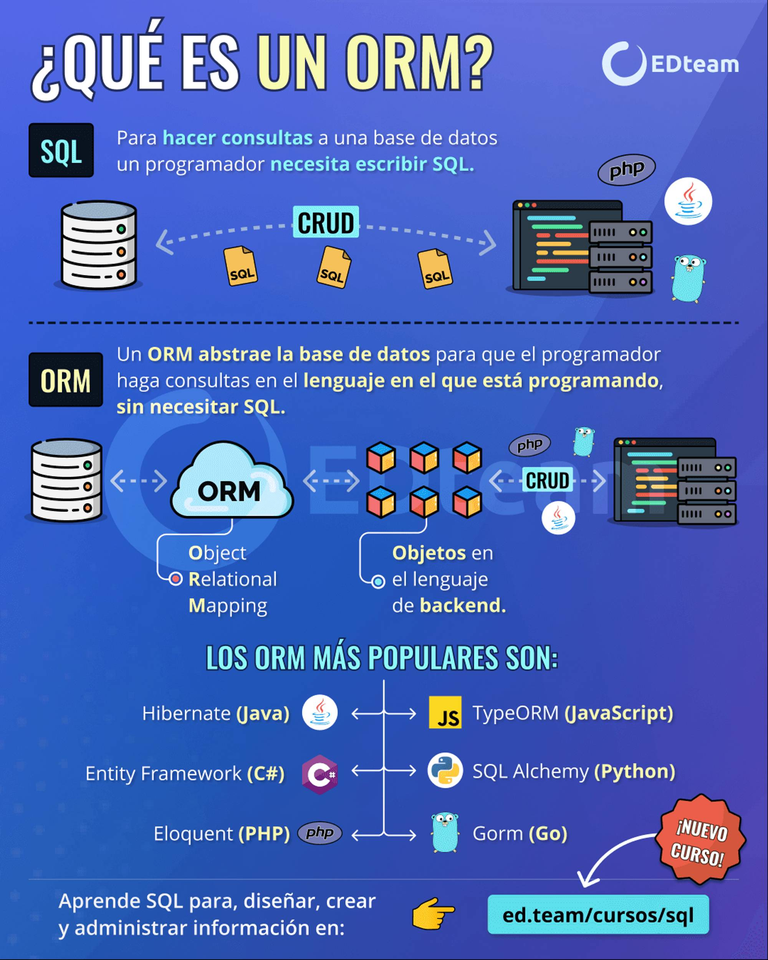
* **¿Qué es un ORM? Indica los principales ORM.**

ORM son las siglas de Object-Relational Mapping. Es una técnica de programación, un conjunto de bibliotecas instalables que permiten utilizar la programación orientada a objetos para gestionar las bases de datos. Permite, desde el propio lenguaje de programación empleado, mediante objetos, manipular los datos de una base de datos sin emplear sintaxis SQL ni tratar directamente con tablas y filas.

El ORM mapea objetos de clases definidas en el lenguaje de programación y se encarga de la traducción entre estos objetos y las entradas de la base de datos.

Los principales ORM en distintos lenguajes de programación son:

* Java: Hibernate, EclipseLink
* Python: Django ORM, SQLAlchemy
* C#: Entity Framework, Dapper
* JavaScript: Sequelize, TypeORM
* PHP: Eloquent



* **Si no uso un ORM ¿cómo puedo acceder a los datos desde Java?**

Hay varias opciones, entre las más destacables:

* **JDBC** (Java Database Connectivity): Es una API de Java que igualmente proporciona clases e interfaces para conectar con una base de datos, enviar consultas SQL y recuperar resultados. Permite establecer conexiones directas con la base de datos y ejecutar consultas SQL mediante código.
* **JPA** (Java Persistance API): Es una API de Java para mapeo objeto-relacional que puede utilizarse sin un ORM. Proporciona un conjunto de interfaces y anotaciones que permiten interactuar con la base de datos. Muchos frameworks ORM están basados en JPA, implementando sus interfaces y anotaciones (Hybernate, EclipseLink...)
* **Spring JDBC**: Proporciona clases para usar de plantilla y soporte para administrar conexiones de base de datos, simplificando la escritura de código JDBC.
* **¿Puedo utilizar cualquier librería ORM para mi proyecto Spring Boot?**

Spring Boot ofrece mucha flexibilidad en cuanto a la elección de tecnologías y herramientas que se pueden utilizar en el desarrollo de una aplicación, por lo que es compatible con gran variedad de librerías ORM enfocadas al lenguaje de programación Java.

No obstante, las principales librerías utilizadas son: Hybernate, Spring Data JPA, EclipseLink, MyBatis

### Tareas

**- Investigar los pros y contras de utilizar ORM**.

Principales **ventajas** de usar ORM:

* Mayor eficacia en el desarrollo de la aplicación, ya que, con el mismo lenguaje de programación, a través de objetos, se puede interactuar con la base de datos, sin necesidad de escribir las consultas SQL directamente, reduciendo así la cantidad de código necesario (mejora la productividad del desarrollador).
* Permite la portabilidad del código, de forma que el mismo código puede emplearse entre diferentes sistemas gestores de base de datos.
* Se reduce el código repetitivo relacionado con la manipulación de datos (consultas SQL) y la conexión a las mismas.
* Suelen ofrecer funciones de seguridad integrada, protegiendo la conexión con la base de datos.
* Facilita el mantenimiento de la aplicación, ya que toda la lógica de acceso a datos se encapsula en clases y métodos reutilizables, quedando separada de la lógica de negocio (lógica de la aplicación).

Principales **inconvenientes** de usar ORM:

* Puede conllevar a la sobrecarga del rendimiento ya que las consultas SQL son menos eficientes al realizarse de una forma “no directa”, sino cargando los objetos relacionados.
* Utilizar ORM requiere aprender una nueva sintaxis, aparte de la propia del lenguaje de programación y del lenguaje SQL. Requiere una formación adicional del programador.
* La aplicación puede volverse más compleja si se trabaja con modelos de datos complejos, que requerirían establecer soluciones más complicadas o específicas a nivel de clases y objetos. En estos casos, las consultas directas suelen ser más ventajosas.
* Aunque ORM puede proporcionar portabilidad en el código, algunos ORM están más orientados hacia ciertas bases de datos, por lo que pueden estar limitados al querer migrar a otro ORM.

### Dudas

¿?

¿?

VIP: registra las dudas que tengas para la sesión con el formador!!

¿?

## Patrón DAO vs “Patrón” Repository

### Objetivos

Entender la diferencia entre un patrón DAO y JPA Repository

### Recursos formativos

El patrón DAO es un patrón de diseño que proporciona una interfaz estándar para acceder a datos en bases de datos relacionales. El patrón DAO puede implementarse utilizando diferentes mecanismos para realizar consultas a la base de datos, como JDBC o un framework de ORM.

El patrón JPA Repository es un término que no se refiere directamente a un patrón de diseño, sino más bien a una característica proporcionada por la especificación JPA (Java Persistence API). El uso de un "Repository" en el contexto de JPA es común y se asocia con el concepto de un repositorio de datos que proporciona métodos para acceder y manipular entidades persistentes. Sin embargo, no es un patrón de diseño en sí mismo.

<https://www.baeldung.com/java-dao-vs-repository>

### Posibles preguntas

* **¿Qué es el patrón DAO? ¿Cuál es su propósito?**

DAO son las siglas de Data Access Object. Es un patrón de diseño cuyo propósito es separar la lógica negocio (lógica de la aplicación) de la lógica de acceso a datos.

Las clases DAO actúan como una capa intermedia y se encargan de interactuar con la fuente de datos y ocultan los detalles de implementación de acceso a datos (consultas como tales) al resto de la aplicación.

Los objetos DAO proporcionan una interfaz común o métodos para realizar las operaciones CRUD en los objetos de dominio. Los cambios en la fuente de datos solo requieren modificaciones en la capa DAO, sin alterar el resto de código correspondiente a la lógica de negocio de la aplicación.

Este patrón de diseño puede implementarse mediante distintos mecanismos, como JDBC o mediante un framework de ORM.

* **Para que sirve la anotación @Repository.**

Es una anotación propia de Spring Framework que indica las clases que son componentes de repositorio, o lo que es lo mismo, las clases responsables de gestionar el acceso a datos de una aplicación (capa de acceso a datos). Estas clases suelen contener la lógica para interactuar con la base de datos.

La anotación se suele utilizar de manera conjunta con objetos DAO, permitiendo su escaneo y detección automática (mediante Spring Data).

### Tareas

NA

### Dudas

¿?

¿?

¿?

¿?

¿?

¿?

## Hibernate JPA

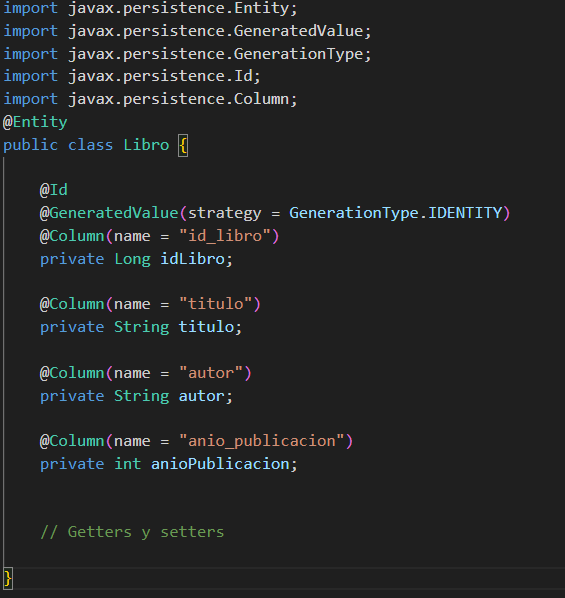
Hibernate es un framework de mapeo objeto-relacional (ORM) para Java. Fue una de las primeras implementaciones populares de la especificación JPA. Hibernate simplifica la interacción con bases de datos relacionales al mapear objetos de Java a tablas de bases de datos y viceversa, eliminando la necesidad de escribir consultas SQL directas.  
  
La librería que está incluida proyecto Spring Data JPA es Hibernate JPA, que es como por así decirlo, una versión más integrada con Spring Data del ORM, ya que Hibernate como tal existía desde antes.

### Objetivos

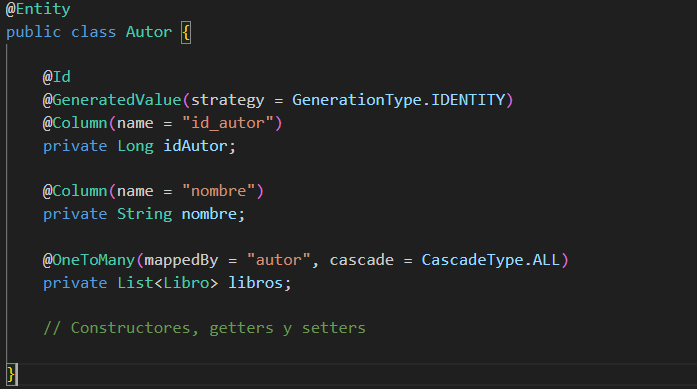
1. Aprender a mapear entidades y columnas.
2. Aprender a mapear los diferentes tipos de relaciones entre entidades.
3. Crear el esquema de la base de datos a partir del mapeo de Hibernate y no con scripts de bdd.
4. Utilizar secuencias autoincrementables en claves primarias.

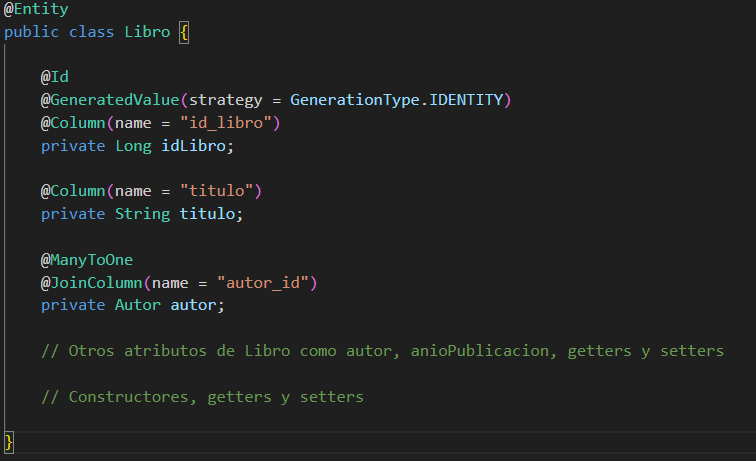
### Mapeo de entidades y columnas.

En este ejemplo se muestra el mapeo de una tabla Libro mediante el uso de anotaciones. Las anotaciones que usamos para mapear la tabla es @Entity (*Si el nombre de la clase coincide con el nombre de la tabla no es necesario añadir el atributo “name”*). También se usa la anotación @Column para definir las columnas en cada propiedad de nuestra clase. IMPORTANTE: Si nos equivocamos en algún nombre, o tipo no nos compilará el proyecto. Ya que Hibernate JPA se encarga de validar nuestro esquema.



### Mapeo de relaciones.

Esto sería otro ejemplo de como mapeariamos una relación entre dos entidades 1 a N.  
  




El atributo **mappedBy** se utiliza para indicar que la relación es propiedad de la entidad Libro, y **cascade = CascadeType.ALL** indica que las operaciones de persistencia (guardar, actualizar, eliminar) en un autor también deben afectar a sus libros.

### Recursos formativos

[Hibernate / JPA One-to-Many Mappings (howtodoinjava.com)](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping/)

<https://www.youtube.com/watch?v=qocxxWNvHg4>

## JPA Repository: Consultas

Creación de interfaz con anotación @Repository, que extienda de JpaRepository o CrudRepository.

|  |
| --- |
| @Repository  public interface BookRepository extends JpaRepository<Book, Long> {  //Métodos adicionales según sea necesario  } |

Esta interfaz, al heredar de otra perteneciente a JPA, nos va a permitir hacer uso de un CRUD básico y operaciones usando el Id de la entidad. Los métodos más usados son **findById(), findAll(), save(), delete()...**

Es importante tener en cuenta que JpaRepository es una interfaz de JPA, pero existen algunas otras, como CrudRepository, que es la interfaz padre de JpaRepository, es decir, JpaRepository hereda los métodos de CrudRepository y además añade algunos más referentes a paginación etc.

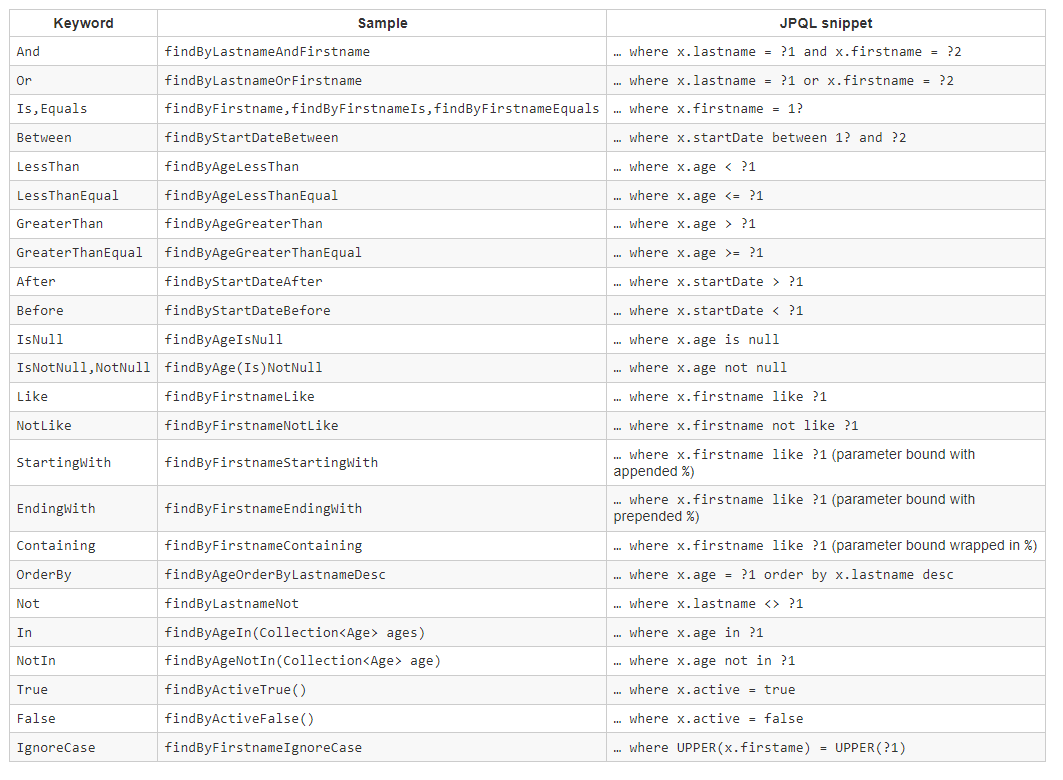
### Consultas a partir de propiedades

Si nosotros tenemos una entidad Libro con una propiedad título, que no es clave primaria, y queremos crear un método en el repositorio para buscar (Como si de una cláusula WHERE se tratase), solo tenemos que declarar un método de la siguiente forma.

|  |
| --- |
| @Repository  public interface BookRepository extends JpaRepository<Book, Long> {  List<Book> findByTittle(String tittle);  } |

JPA detecta automáticamente la propiedad nombre de la entidad libro en el propio nombre del método. Esto es un avance muy significativo ya que nos ahorra mucho tiempo con respecto a otro tipo de herramientas como JdbcTemplate o EntityManager.

Introduciremos las cláusulas necesarias en el propio nombre del método, y nos ahorraremos crear una implementación para definirlos, así de sencillo, a continuación, se muestra una tabla con todos los JPA Criterias (Cláusulas) disponibles.



**Aplica lo mismo para los deleteBy.**

Por último, cabe destacar que, JpaRepository respeta las relaciones y las operaciones en cascada definidas en cada entidad. De manera que, si realizamos una consulta sobre una entidad padre, nos traerá la entidad o entidades hijas. Del mismo modo si realizamos operaciones de escritura o borrado, ésta las realizará en cascada, gracias a las anotaciones CascadeType de Hibernate.

### Objetivos

* Comprender el funcionamiento de JPA, conocer los principales JPA Criterias.
* Cómo hacer consultas y las operaciones básicas de CRUD.
* Realizar consultas más complejas utilizando cláusula BY.
* Leer y escribir en cascada.
* Tomar conciencia de los efectos de determinados atributos de Hibernate.

### Recursos formativos

* <https://www.youtube.com/watch?v=VlpIXkHU3EQ>

### Posibles preguntas

* **¿Qué significa JPA? ¿Para qué se usa?**

JPA son las siglas de Java Persistance API. Es una API de Java con una interfaz de programación que permite gestionar los datos con la base de datos. Mapea las tablas de la base de datos relacional e interactúa con ellas utilizando objetos Java (código de programación) en lugar de sintaxis SQL.

Se usa para no tener que manipular directamente las tablas de las bases de datos, ya que todo se realiza con las “entidades” JPA (objetos o clases Java) y sus métodos que permiten realizar las operaciones CRUD. De esta forma se independiza el desarrollo de la aplicación del sistema gestor de datos que se utilice (MySQL, Oracle...), utilizando un conjunto común de interfaces y anotaciones.

* **¿Es lo mismo JPA que Hibernate?**

No, mientras que JPA es una API que proporciona una interfaz básica o común para interactuar con las bases de datos, Hibernate es un framework ORM que implementa JPA y funcionalidades adicionales más avanzadas. Es decir, JPA contiene las reglas o herramientas base y estándares mientras que Hibernate es una implementación avanzada de JPA.

* **¿Qué es una Entidad? ¿Cómo consigo que una clase sea una Entidad?**

Es una clase Java, que representa un objeto del mundo real, y que está mapeada a una tabla en una base datos, lo que significa que existe una correspondencia entre la estructura de dicha clase (atributos, métodos, etc.) y la estructura de una tabla en la base de datos relacional. Los objetos entidad se tratan como registros que pueden ser almacenados y gestionados en la base de datos a través de código Java de programación, sin necesidad de escribir consultas SQL manuamente (ORM).

En JPA, una entidad se marca con la anotación @Entity.

Dentro de dicha entidad, el atributo que actuará como clave primaria se marca con @Id. Opcionalmente, se puede especificar estrategias para autogenerar valores de la clave primaria con la anotación @GeneratedValue (autoincrementos, sencuencias...). Por su parte, @Column, marca la correspondencia entre el atributo y la columna de tabla en la base de datos.

* **Explica los tipos de relaciones de JPA y cuando usarlos.**

Los tipos de relaciones entre entidades equivalen a los tipos de relaciones o cardinalidades existentes entre tablas de una base de datos. Pueden ser:

* + **Relación Uno a Uno (@OneToOne):** Se emplea cuando una entidad tiene relación directa y exclusiva con una única instancia de otra entidad. Por ejemplo, a una entidad Capital le corresponde una única entidad País.
  + **Relación Uno a Muchos (@OneToMany):** Se emplea cuando una entidad tiene relación con varias instancias de otra entidad. Por ejemplo, una entidad Departamento puede relacionarse con varios objetos de la entidad Profesor.
  + **Relación Muchos a Uno (@ManyToOne)**: Se emplea cuando varias instancias de una entidad se relacionan con una única instancia de otra entidad. Por ejemplo, el caso contrario al anterior, varios objetos de la entidad Profesor pueden pertenecer al mismo objeto Departamento.
  + **Relación Muchos a Muchos (@ManyToMany):** Se emplea cuando cada instancia de una entidad puede relacionarse con múltiples instancias de otra entidad y viceversa. Por ejemplo, un objeto Alumno puede matricularse en varios objetos Curso, mientras que cada objeto Curso puede relacionarse a su vez con varios objetos Alumnos.
* **¿Cuáles son los mecanismos de fetching? ¿diferencias? ¿implicaciones?**

Un mecanismo fetching se emplea para establecer la forma en que se recuperan los datos de una consulta que involucra a entidades relacionadas en una base de datos. Por ejemplo, si se consulta sobre un objeto Cliente, puede que también se desee acceder a los objetos Pedido con los que está relacionado.

Los mecanismos de fetching son:

* **Eager Fetching (Carga ansiosa)**: Recupera todos los datos relacionados junto con los propios de la entidad principal. Todos los datos se cargan en memoria tan pronto se acceda a la entidad principal. Ello implica un mayor consumo de recursos (memoria) y tiempos de espera más largos.
* **Lazy Fetching (Carga perezosa)**: Los datos relacionados se recuperan de la base de datos solo cuando se accede explícitamente a ellos. Los datos relacionados no se cargan en memoria al acceder a la entidad principal. Ello implica un menor consumo de memoria y una reducción de tiempos de carga, pero requiere mayor número de consultas para acceder a los datos relacionados.

Por tanto, la principal diferencia entre Eager y Lazy (fetching) es que la primera carga conjuntamente los datos relacionados y los datos propios de la entidad principal, mientras que la segunda solo carga los de la entidad principal. Se elije entre una u otra en función del volumen de datos a manejar, la naturaleza de las entidades y los requisitos de rendimiento.

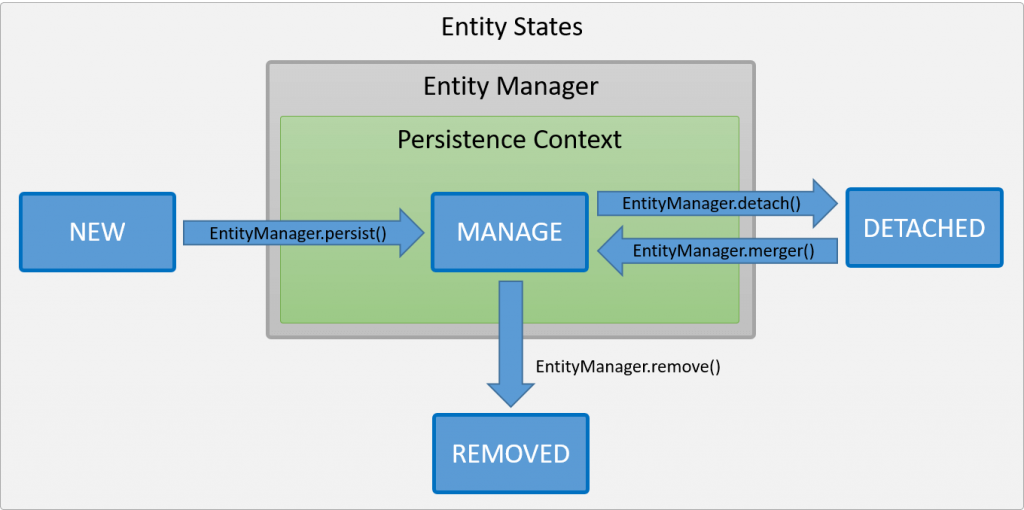
* **¿Qué métodos usamos para un CRUD? ¿Puedes explicar el ciclo de vida de una Entity?**

Al trabajar con Hibernate JPA en un entorno spring, se emplea la anotación @Repository para indicar la clase responsable de getionar el acceso a datos de la aplicación. Los **métodos CRUD** que se pueden emplear en este conteto son:

* **save()**: Operación **CREATE** o **UPDATE**. Para guardar o actualizar una entidad en la base de datos. Si la entidad no existe, se crea; si exite, se actualiza.
* **find()**: Operación **READ**.
  + **findAll():** Para leer todas las entidades de una tabla.
  + **findByX()**: Operación READ. Para leer una entidad por la cláusula que se pase en X (Id, FirstName, LastName, Age...) con posibilidad de usar una keyword que aporte mayor funcionalidad o filtrado a la consulta.
* **delete()**: Operación **DELETE**. Borra una entidad que se le pasa como parámetro. También se puede utilizar con una sintaxis análoga a la de find() (mismas cláusulas).

Por otro lado, el **ciclo de vida** de una entidad es el conjunto de estados por los que puede pasar una entidad JPA. Dichos estados principales son:

* **New o Transient** (transitoria): La entidad ha sido creada pero aún no existe en la base de datos ni está asociada a un contexto de persistencia (un entorno que gestiona las entidades y su interacción con la base de datos).
* **Managed** (gestionada): La entidad está siendo gestionada por un contexto de persistencia, cualquier actividad de la entidad será rastreada por JPA y se sincronizará con la base de datos cuando se realice una operación CRUD sobre la misma.
* **Detached** (desconectada): La entidad deja de ser administrada por el contexto de persistencia, aunque siga perteneciendo a la base de datos. Por tanto, lo cambios en la entidad no serán rastreados por JPA.
* **Removed** (eliminada): La entidad está marcada para ser eliminada. Al finalizar la transacción, es eliminada de la base de datos y del contexto de persistencia.



* **¿Cuáles son los principales ficheros de configuración del Hibernate?**
* **hibernate.cfg.xml**: Es el fichero principal de configuración. Contiene la configuración de sesión, la configuración de la conexión a la base de datos, las propiedades de Hibernate y los mapeos de las entidades.
* **hbm.xml**: Hibernate Mapping Files. Este fichero contiene los mapeos entre las clases de entidad y las tablas de bases de datos. Cada fichero de este tipo, por lo general mapea una clase a una tabla y define la relación entre propiedades de clase y columnas.
* **hibernate.properties**: También se puede configurar Hibernate mediante un archivo de propiedades que recoge todas las propiedades de configuración en formato clave-valor.

### Tareas

NA

### Dudas

## JPA Repository: JPQL, Criteria

JpaRepository, tiene una limitación a la hora de realizar subconsultas, ya que no es capaz de realizar el mapeo a través de la declaración del método. Es por eso que se hace uso de JPQL y la anotación @Query.

### ¿Qué es JPQL?

JPQL no es más que un sistema de mapeo de SQL a Java, es, básicamente interpretar una consulta SQL pero utilizando las clases Java (Entidades), en lugar de las tablas de la bbdd.  
  
De forma que si uno tiene una tabla TB\_LIBRO y un @Entity Libro puedo realizar la siguiente sentendia JPQL :  
  
**SELECT A FROM Libro A;**  
  
JPQL mapea automáticamente la palabra “Libro” con la entidad, y a diferencia de SQL en este caso no se utiliza nunca el “\*” asterisco.   
  
Siguiendo esta premisa, se puede hacer uso de las propiedades y no de las columnas.   
  
**SELECT A FROM Libro A WHERE A.nombre = ‘Título de libro’;**

Además de esto, JPQL nos permite parametrizar las consultas, haciendo uso de ( : ) delante del parámetro.  
  
**SELECT A FROM Libro A WHERE A.nombre = :nombreTitulo;**

### Uso de @Query

La anotación @Query se sitúa sobre la declaración del método en la interfaz @Repository, y esta anotación nos permite recibir una consulta JPQL. De esta forma podemos hacer uso de cláusulas JOIN para poder realizar subconsultas

|  |
| --- |
| public interface LibroRepository extends JpaRepository<Libro, Long> {  @Query("SELECT l.titulo FROM Libro l JOIN l.autores a WHERE a.nombre = :nombreAutor")  List<String> buscarTitulosPorNombreAutor(@Param("nombreAutor") String nombreAutor);  } |

### Objetivos

* Comprender el funcionamiento de los mecanismos para realizar consultas en JPA e Hibernate

### Recursos formativos

* <https://www.youtube.com/watch?v=FMkJhxopJcY>
* <https://www.baeldung.com/spring-data-criteria-queries>

### Posibles preguntas

* **¿Cómo puedo realizar consultas (select) con JPA/Hibernate?**

Se puede realizar mediante **JPQL** (lenguaje de consultas orientado a objetos). Emplea entidades JPA en lugar de tablas y columnas de la base de datos. Por tanto, permite realizar las consultas a través de los nombres de entidades y sus atributos (objetos Java). La sintaxis es muy similar a SQL, pero empleando objetos y sus atributos. Ejemplo:

SELECT e FROM Empleado e WHERE e.departamento = :departamento

En este caso, “:departamento” es el parámetro que aportará el valor.

Por otro lado, se pueden realizar consultas mediante la anotación **@Query (Spring Data JPA)**. Para ello, en la interfaz repositorio (marcada con @Repository) se marca el método que realiza la consulta con la anotación @Query. Aplicándolo al mismo ejemplo:

@Repository  
public interface EmpleadoRepository extends JpaRepository<Empleado, String> {  
  
 @Query("SELECT e FROM Empleado e WHERE e.departamento = ?1")  
 List<Empleado> findByDepartamento(String departamento);  
}

Este método devuelve una lista de empleados cuyo Departamento coincida con el departamento pasado como parámetro. El término “?1” del select se refiere al primer parámetro (1) que recibe el método (parámetro posicional).

Otra forma equivalente sería:

@Repository  
public interface EmpleadoRepository extends JpaRepository<Empleado, String> {  
  
 @Query("SELECT e FROM Empleado e WHERE e.departamento = :departamento")  
 List<Empleado> findByDepartamento(@Param(departamento) String

departamento);  
}

En este caso, se emplea lo que ese denomina parámetro nombrado (:departamento). Dicho parámetro se define en el método mediante la anotación @Param.

* **¿Cuándo debo de usar uno u otro? ¿ventajas e inconvenientes? Explicación de su funcionamiento.**

**Ventajas de JPQL:**

* Permite generar consultas con mayor flexibilidad y de mayor complejidad.
* Es independiente del proveedor de persistencia JPA (Ya sea, Hibernate, EclipseLink...), de forma que se puede portar el codigo sin modificar la conulta.

**Inconvenientes de JPQL:**

* JPQL se escribe como cadena de texto (string), lo que puede dificultar la detección de errores de sintaxis y el mantenimiento de consultas complejas.

**Ventajas de @Query:**

* Permite definir métodos de repositorio que tienen embebidas las consultas.
* Las consultas se definen directamente en las interfaces de repositorio, lo que facilita su comprensión y la detección de errores de sintaxis.

**Inconvenientes de @Query:**

* Frente a JPQL, @Query ofrece menor flexibilidad y puede ser más complejo de utilizar a la hora de realizar consultas más complejas o dinámicas.
* Está vinculado a Spring Data JPA, por lo que puede afectar a la portabilidad si se decide cambiar el proveedor de persistencia.

Por tanto, JPQL está más indicado cuando se necesite flexibilidad para construir consultas complejas y cuando se requiera independencia del proveedor de persistencia, mientras que @Query se emplea cuando se quiera tener un mayor control y comprensión sobre las consultas realizadas.

**JPQL funciona de la siguiente forma**:  
String jpql = "SELECT e FROM Empleado e";  
Query query = entityManager.createQuery(jpql);  
List<EntityName> resultList = query.getResultList();

La consulta se establece como un string. A continuación se utiliza el EntityManager para crear una instancia de tipo Query a la que se le pasa el string de la consulta como parámetro. Finalmente, se ejecuta la consulta introduciendo sus resultados en una lista.

**La misma consulta, pero utilizando @Query, funciona de la siguiente forma:**

@Repository  
public interface EmpleadoRepository extends JpaRepository<Empleado, String> {  
 @Query("SELECT e FROM Empleado e")  
 List<EntityName> findAllEmpleados();  
}

En la interfaz repositorio se declara un método marcado con la anotación @Query que incluye la consulta a realizar. El método devolverá la consulta indicada en @Query.

* **¿Qué parámetro tengo que usar para poder ver las consultas realizadas por consola?**

Se debe habilitar la propiedad hibernate.show\_sql en el archivo de configuración de Hibernate o en el archivo de propiedades de la aplicación.

Por ejemplo, en el archivo de propiedades application.properties habría que introducir:

spring.jpa.show-sql=true

### Tareas

NA

### Dudas

* ¿?
* ¿?

## Ejercicio práctico: Consultas JPA

### Objetivos

* + **Poner en práctica los conocimientos adquiridos mediante un caso de uso**

### Recursos formativos

NA

### Posibles preguntas

NA

### Tareas

* + El programa tiene que ser ejecutado desde el Main de la aplicación (mínimo) desde el propio IDE (la forma óptima será usando Junit).
  + La bbdd puede ser MySQL, Oracle o H2Database
  + Caso de uso:
  + Se quiere diseñar una base de datos relacional para gestionar los datos de los socios de un club náutico.
  + De cada socio se guardan los datos personales y los datos del barco o barcos que posee: número de matrícula, nombre, número del amarre y cuota que paga por el mismo.
  + Además, se quiere mantener información sobre las salidas realizadas por cada barco, como la fecha y hora de salida, el destino y los datos personales del patrón, que no tiene por qué ser el propietario del barco, ni es necesario que sea socio del club.
  + El programa tiene que realizar todas las operaciones CRUD necesarias para cada una de las entidades definidas.
  + Se deben anotar las entidades para utilizar los métodos de validación definidos por la especificación [JSR 380](https://jcp.org/en/jsr/detail?id=380).
  + Las operaciones de consulta tienen que realizarse utilizando los distintos métodos aprendidos durante la formación (consultas predefinidas, JPQL, Criteria) (se deja a cada alumno la decisión de definir las consultas que obtengan datos necesarios para la gestión del club náutico y los métodos JPA empleados)

**Enlace repositorio**

**https://github.com/Sanmohe/APIclubNautico.git**

### Dudas

¿?