EFA
MORATALAZ

2º CFGS Desarrollo de Aplicaciones Web

DESARROLLO WEB EN ENTORNO SERVIDOR

JESÚS SANTIAGO RICO

UT4 – JPA Y PATRÓN DAO/REPOSITORY

JPA

Java Persistence API

EFA
MORATALAZ

2º CFGS Desarrollo de Aplicaciones Web

DESARROLLO WEB EN ENTORNO SERVIDOR

UT4 – JPA Y PATRÓN DAO/REPOSITORY

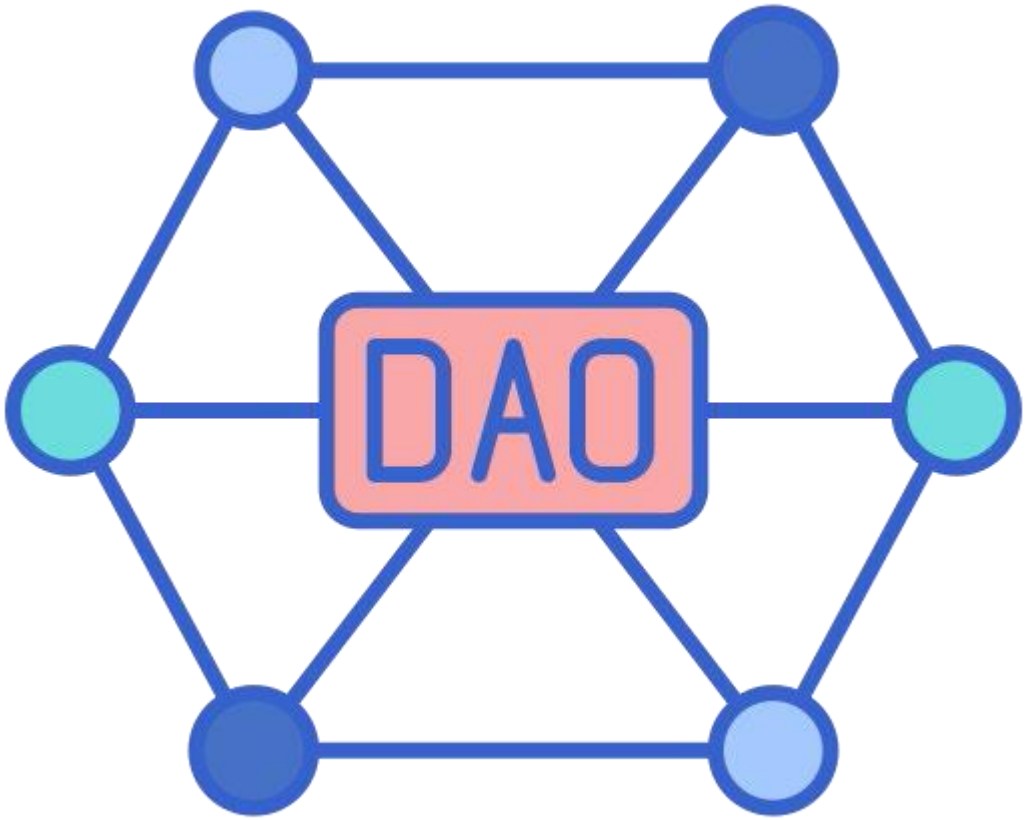
1. INTRODUCCIÓN
2. MAPEANDO UNA ENTIDAD CON @ENTITY
3. REGISTRANDO PROVEEDOR DE BBD
4. VALIDANDO ENTIDADES
5. RELACIONANDO ENTIDADES
6. PATRÓN REPOSITORY

INTRODUCCIÓN

Durante el presente tema abordaremos los temas:

JPA

Java Persistence API



¿Qué es JPA?

JPA define un conjunto de estándares para la persistencia de datos en aplicaciones Java.

¿Qué es un ORM?

Un *ORM* es una herramienta o técnica que permite mapear objetos en un lenguaje de programación (como Java) a registros en bases de datos relacionales.

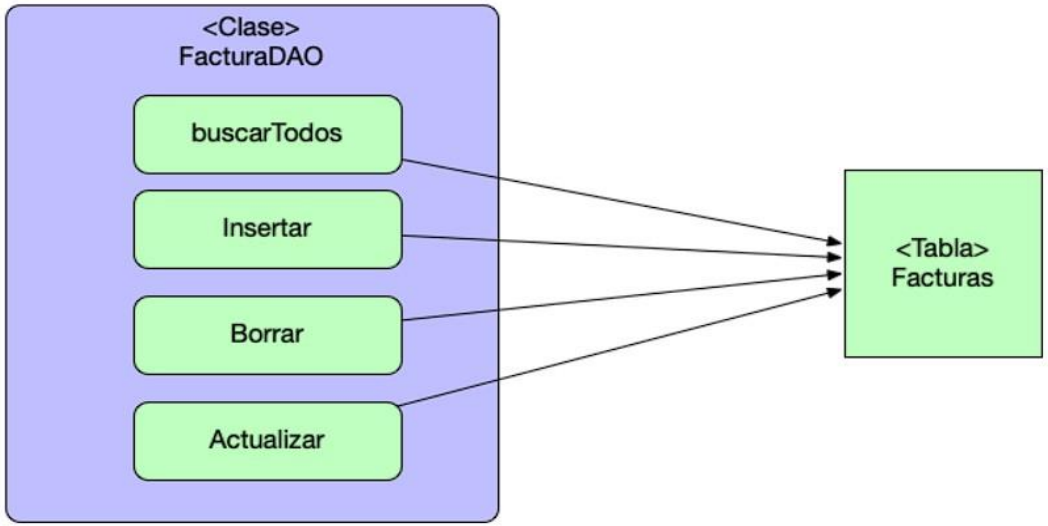
JPA dicta las reglas sobre cómo se deben definir las entidades, relaciones, transacciones y consultas.

Un ORM proporciona las herramientas para cumplir con las reglas de JPA y gestionar la comunicación con la base de datos.

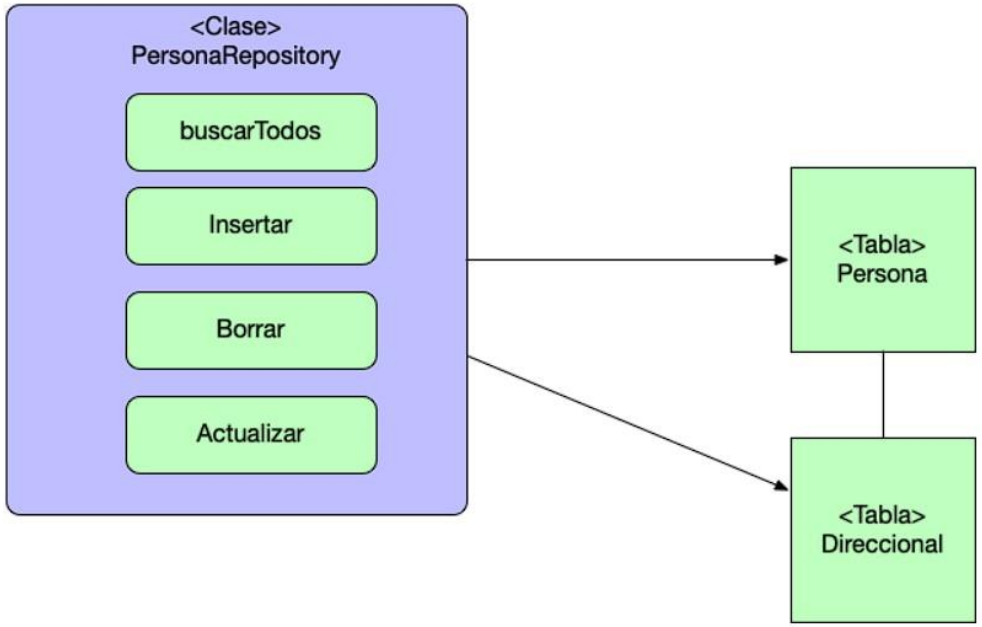
Patrón DAO

El patrón **DAO o Data Access Object** es un patrón de diseño estructural que encapsula la lógica de acceso a datos en un sistema.

En este patrón, una clase específica (el DAO) se encarga de realizar las operaciones CRUD (Create, Read, Update, Delete) contra una base de datos



Patrón Repository – CrudRepository (Spring)

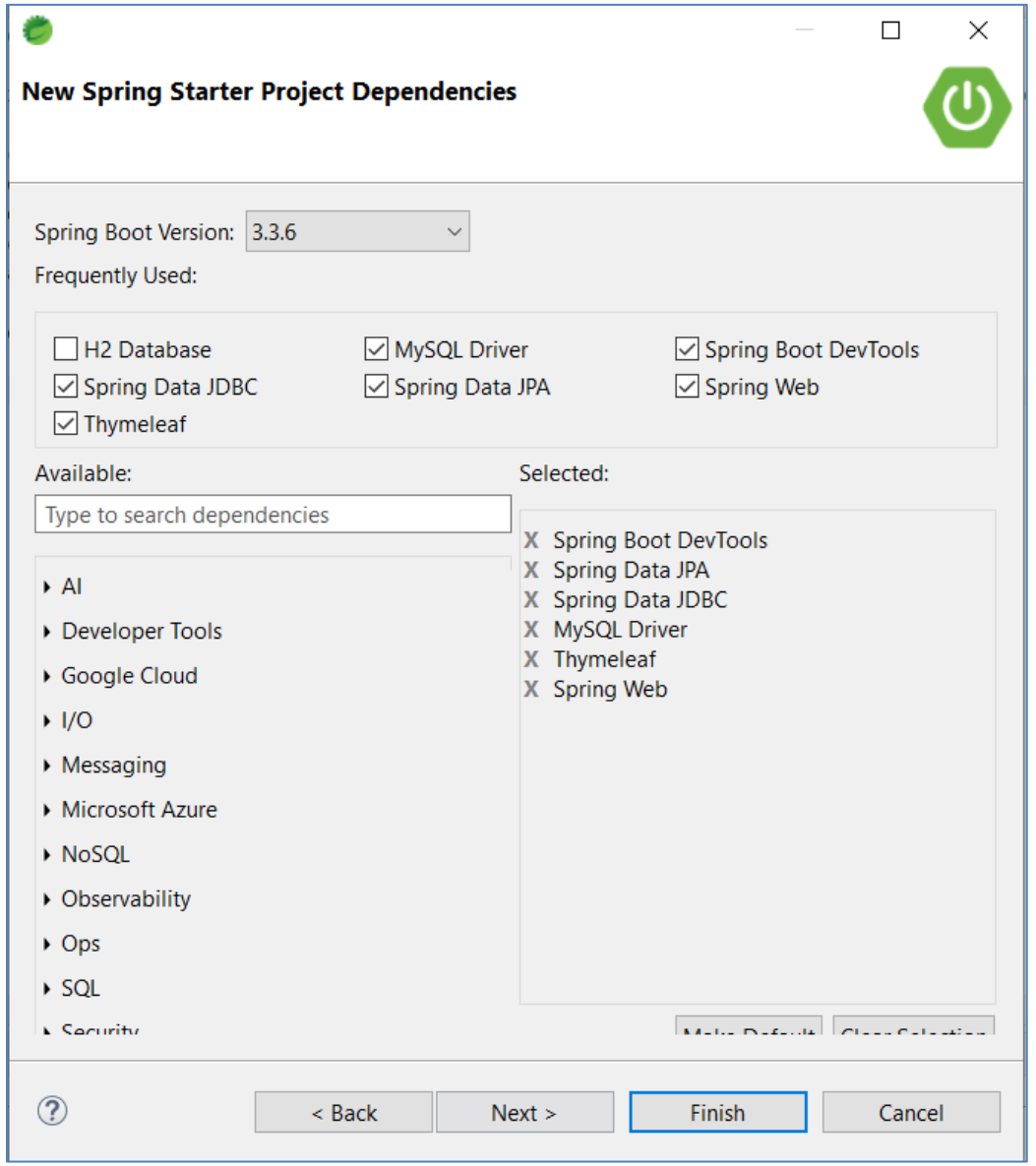


El patrón **Repository** hace referencia a una clase que se encarga de almacenar un objeto y no especifica que tipo de persistencia se ha de usar o si el mapeo debe ser uno a uno con la base de datos. El repositorio aporta mayor flexibilidad y no esta tan fuertemente acoplado a la base de datos

MAPEANDO UNA ENTIDAD CON @ENTITY

En primer lugar, deberemos crear el proyecto con las siguientes opciones:

- ✓ Spring Boot DevTools
- ✓ Spring Data JPA
- ✓ Spring Data JDBC
- ✓ MySQL Driver
- ✓ Thymeleaf
- ✓ Spring Web



Una vez creado el proyecto tal y como se ha indicado anteriormente, debemos crear una clase que representará a una de nuestras tablas, dicha clase deberá implementar la interfaz **Serializable**.

Para indicar que una determinada clase es una **entidad**, que está asociada a una tabla de la base de datos, deberemos usar la siguiente anotación:

@Entity

Adicionalmente a la anterior anotación podemos usar **@Table** para especificar el nombre de la tabla que estamos asociando, esta anotación es opcional y podemos omitirla si el nombre de nuestra clase es el mismo que el de la tabla.

2. Mapeando con @Entity

```java
@Entity
@Table(name="FABRICANTE")
public class Fabricante implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID_FABRICANTE", unique = true, nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idFabricante;
    @Column(name = "NOMBRE")
    private String nombre;

    // Getters y Setters

    public Long getIdFabricante() {
        return idFabricante;
    }

    public void setIdFabricante(Long idFabricante) {
        this.idFabricante = idFabricante;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Otro punto importante para mapear nuestra clase, es indicar cual de los atributos de la clase está asociado con la PK de la tabla, para ellos usamos las siguientes anotaciones:

@Id
@GeneratedValue

La anotación **GeneratedValue** posee, los parámetros que se muestran en la tabla.

Parámetro	Descripción
strategy	La estrategia de generación de clave primaria que el proveedor de persistencia debe utilizar para generar la clave primaria de la entidad anotada. (Para autoIncrementales)
generator	El nombre del generador de claves primarias que se utilizará tal y como se especifica en la anotación SequenceGenerator o TableGenerator.

Las posibles opciones del parámetro **strategy** de **GeneratedValue** son:

Valor	Descripción
GenerationType.AUTO	En función de la compatibilidad de la base de datos con la generación de claves primarias, el marco decide qué tipo de generador debe utilizarse.
GenerationType.IDENTITY	En este caso, la base de datos se encarga de determinar y asignar la siguiente clave primaria. **AUTOINCREMENTAL.**
GenerationType.SEQUENCE	Especifica un objeto (Sequencia Oracle) de base de datos que puede utilizarse como fuente de valores de clave primaria. Utiliza **@SequenceGenerator**.
GenerationType.TABLE	Mantiene una tabla separada con los valores de la clave primaria. Utiliza @TableGenerator.

Cuando se especifica el valor **GenerationType.SEQUENCE**, se debe indicar el parámetro **generator**. En este parámetro es el nombre que le daremos a nuestro generador y que posteriormente asociaremos con la secuencia Oracle con la anotación **@SequenceGenerator**.

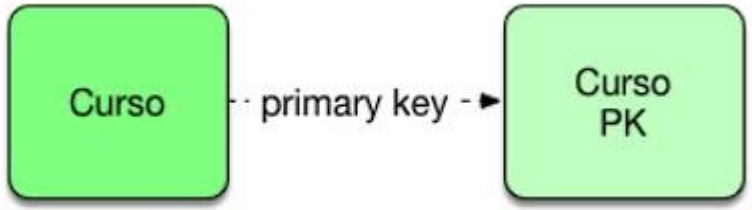
La anotación **@SequenceGenerator** tiene los siguientes parámetros:

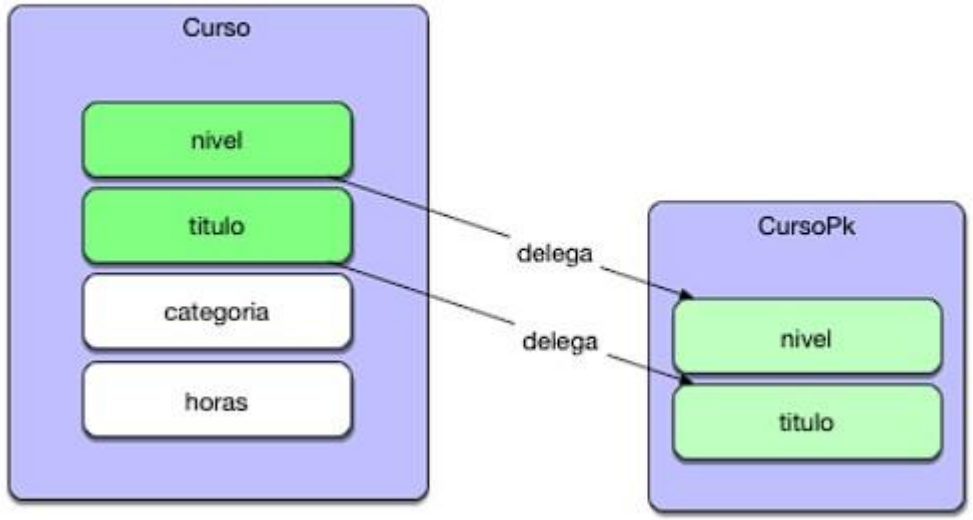
Valor	Descripción
name	Nombre del generador, coincide con el parámetro generator de **@GeneratedValue.**
allocationSize	Indica la cantidad a incrementar
sequenceName	Nombre de la secuencia generada en Oracle.
initialValue	Indica el valor inicial..

Esta anotación presenta más parámetros como catalog y schema, pero los indicados en tabla son los más frecuentemente usamos.

Hasta el momento hemos considerado que una PK, está formada por un único atributo, pero pueden estar compuestas por más de un atributo, si esto ocurre deberemos usar las siguientes anotaciones:

@Embeddable
@EmbeddedId





```java
@Embeddable
public class CursoPKEntity implements Serializable {
	/**
	*
	*/
	private static final long serialVersionUID = 1L;

	@Column(name = "TITULO")
	private String titulo;
	@Column(name = "NIVEL")
	private Long nivel;


	public CursoPKEntity() {
	super();
}
```

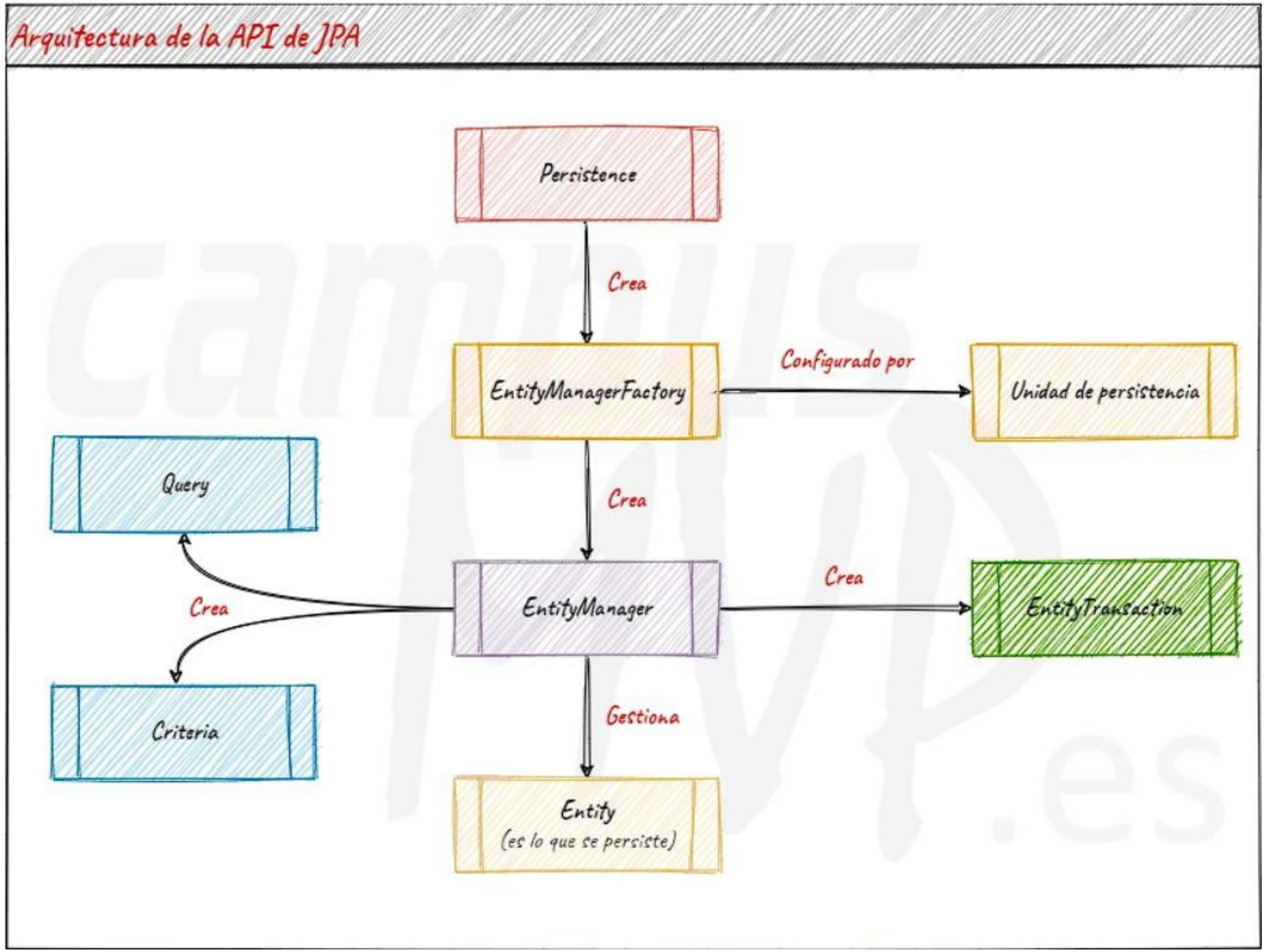
```java
@Entity
@Table(name = "CURSO")
public class CursoEntity implements Serializable {

	/**
	*
	*/
	private static final long serialVersionUID = 1L;

	@EmbeddedId
	private CursoPKEntity pk;

	@Column(name = "CATEGORIA")
	private String categoria;
	@Column(name = "HORAS")
	private Long horas;
}
```

Una vez que la clase este mapeada con **@Entity - @Table** y se ha definido la clave primaria y la forma en la que se generan los valores, no haría falta mapear el resto de los campos, siempre y cuando estos tuvieran los mismos nombres que los campos de las tablas,

En caso contrario deberíamos usar la anotación **@Column,** pero a pesar de que este campo sea optativo se recomienda volver a usarlo.

Los campos fecha de nuestra clase se deben declarar de tipo Date (java.util) y se anotan con **@Temporal**, y como parámetro obligatorio alguno de los que se indican a continuación.

Parámetro	Descripcción
TemporalType.DATE	Especifica sólo la fecha.
TemporalType.TIME	Especifica solo la hora.
TemporalType.TIMESTAMP	Especifica tanto fecha como hora.

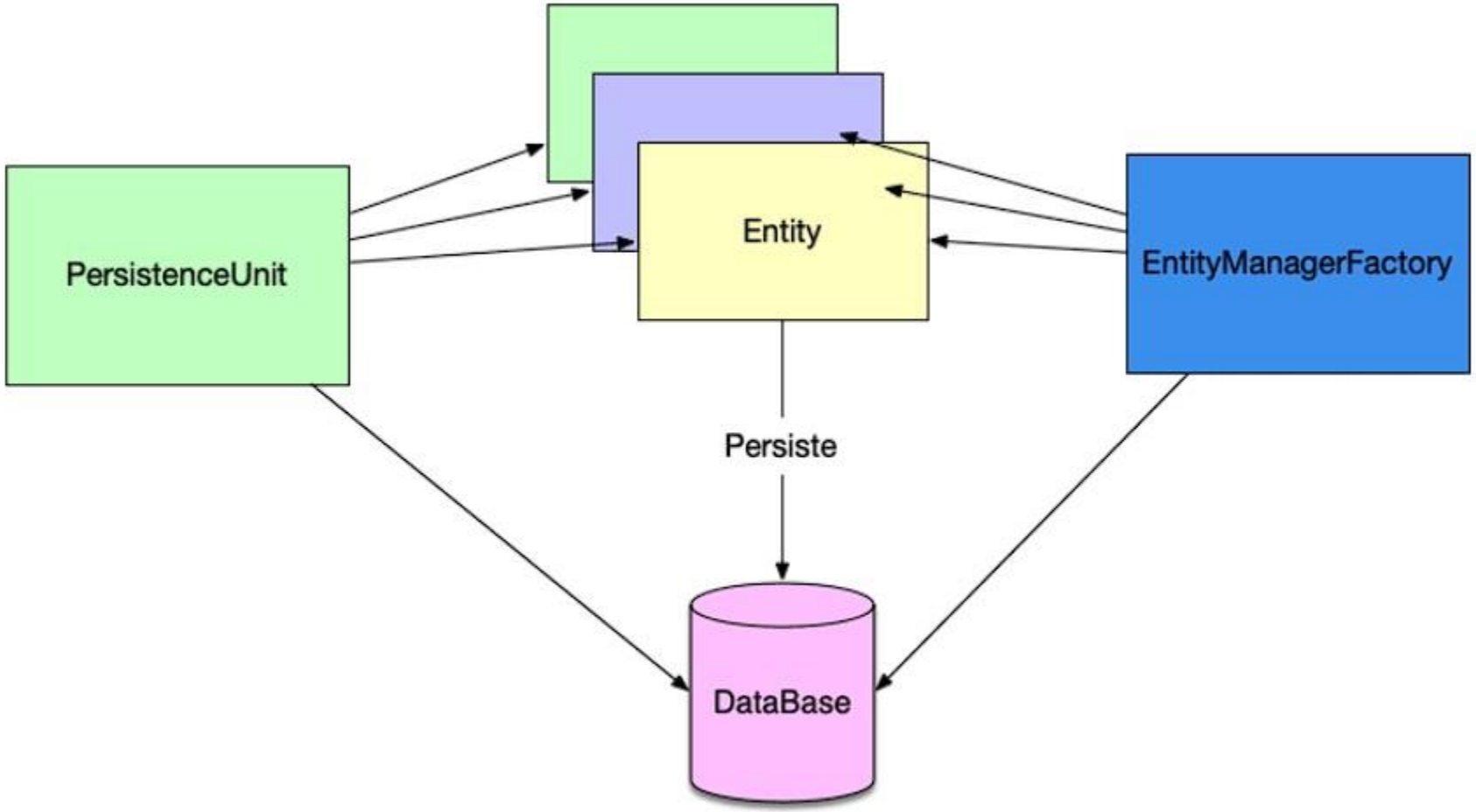

Arquitectura de la API de JPA
Persistence
Crea
EntityManagerFactory
Configurado por
Unidad de persistencia
Query
Crea
Crea
EntityManager
Crea
EntityTransaction
Criteria
Gestiona
Entity
(es lo que se persiste)

EntityManagerFactory

EntityManagerFactory es un componente que se encarga de crear instancias de **EntityManager** puede ver como una fábrica de objetos EntityManager que se necesita para interactuar con la base de datos.

Es el punto de inicio para cualquier operación de persistencia. Antes de poder realizar cualquier acción (como insertar, actualizar o eliminar datos), primero necesitamos tener un **EntityManagerFactory y un EntityManager.**

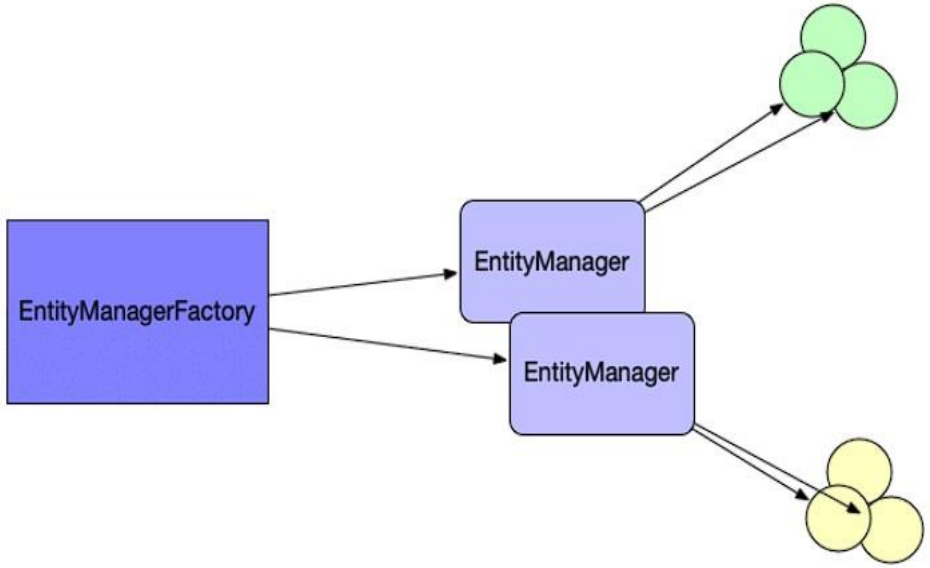
EntityManagerFactory



EntityManager

Es el **objeto** principal que se utiliza para interactuar con la base de datos en una aplicación Java que utiliza **JPA**. Permite realizar **operaciones de persistencia** (guardar, actualizar, eliminar y consultar entidades).

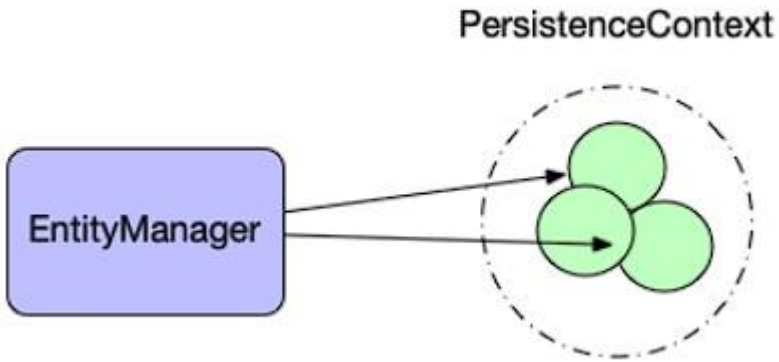
Es el intermediario entre el código Java y la base de datos.



PersistenceContext

El **PersistenceContext** es un **conjunto de entidades** que están siendo gestionadas por el EntityManager en un momento dado. Este contexto es como una **memoria temporal** en la que se almacenan las entidades que estamos manipulando mientras dura una transacción o un ciclo de vida.

El PersistenceContext garantiza que los cambios en las entidades gestionadas se sincronicen con la base de datos al final de la transacción. Evita que los datos se pierdan o queden inconsistentes.



Conclusiones

- ✓ Spring Boot
- ✓ Spring Data JPA

Nos ahorra gran parte de toda esta configuración que se ha visto anteriormente y nos permite centrarnos en la implementación del DAO/Repository que se verá a continuación.

Lo único que deberemos configurar para la gestión de base de datos en spring será:

- ✓ Url datasource
- ✓ Username y passwd del datasource
- ✓ Driver que usa el data source
- ✓ El dialecto que usará JPA

REGISTRANDO PROVEEDOR BBDD

Propiedades a configurar

Propiedad	Descripción
spring.datasource.url	URL de conexión a la BBDD
spring.datasource.username	Nombre de usuario para conectarse a la BBDD.
spring.datasource.password	Contraseña para conectarse a la BBDD.
spring.datasource.driver-class-name	Driver de conexión.
spring.jpa.database-platform	Dialecto usado en Hibernate según el proveedor, este cambia según el proveedor y la versión que se esté usando.
spring.jpa.hibernate.ddl-auto	Estrategia de generación de tablas cuando se levanta la aplicación se crean las tablas y cuando se baja el servidor se eliminan, para un entorno real dar el valor **none**.
logging.level.org.hibernate.SQL	Propiedad con la que podemos ver las querys que JPA lanza, dar valor **debug** para ello.

Configuración en Oracle

```
spring.datasource.url= jdbc:oracle:thin:@localhost:1521:XE
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
#hibernate config
spring.jpa.database-platform=org.hibernate.dialect.OracleDialect
spring.jpa.hibernate.ddl-auto=none logging.level.org.hibernate.SQL=debug
```

Configuración en MySql

```
spring.datasource.url=jdbc:mysql://localhost:3306/tienda?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#hibernate config
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

RELACIONANDO ENTIDADES

En esta sección veremos como establecer las relaciones (FK) con JPA. Las casuísticas que veremos sobre las relaciones serán las siguientes:

- ✓OneToOne
- ✓OneToMany
- ✓ManyToOne
- ✓ManyToMany

OneToOne: Una relación donde **una entidad está asociada con exactamente otra entidad**.

```
@Entity
public class Fabricante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_FABRICANTE")
    private Long idFabricante;

    @Column(name="NOMBRE")
    private String nombre;

    @OneToOne
    @JoinColumn(name = "ID_ARTICULO") // Nombre de la columna en la tabla Articulo
    private Articulo articulo;

}
```

```
@Entity
public class Articulo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_ARTICULO")
    private Long idArticulo;

    @Column(name="NOMBRE")
    private String nombre;

}
```

OneToMany:

Una entidad está asociada con muchas otras entidades

ManyToOne:

Muchas entidades están asociadas a una sola entidad.

```
@Entity
public class Fabricante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_FABRICANTE")
    private Long idFabricante;

    @Column(name="NOMBRE")
    private String nombre;

    @OneToMany(mappedBy = "fabricante")
    private Set<Articulo> productos = new HashSet<>();

}
```

```
@Entity
public class Articulo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_ARTICULO")
    private Long idArticulo;

    @Column(name="NOMBRE")
    private String nombre;

    @ManyToOne
    @JoinColumn(name = "ID_FABRICANTE") // Nombre de la columna en la tabla Articulo
    private Fabricante fabricante;

}
```

ManyToMany: Muchas entidades están asociadas con muchas otras entidades.

```java
@Entity
public class Fabricante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_FABRICANTE")
    private Long idFabricante;

    @Column(name="NOMBRE")
    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "FABRICANTE_ARTICULO", // Nombre de la tabla intermedia
        joinColumns = @JoinColumn(name = "ID_FABRICANTE"),
        inverseJoinColumns = @JoinColumn(name = "ID_ARTICULO")
    )
    private Set<Articulo> articulos = new HashSet<>();
}
```

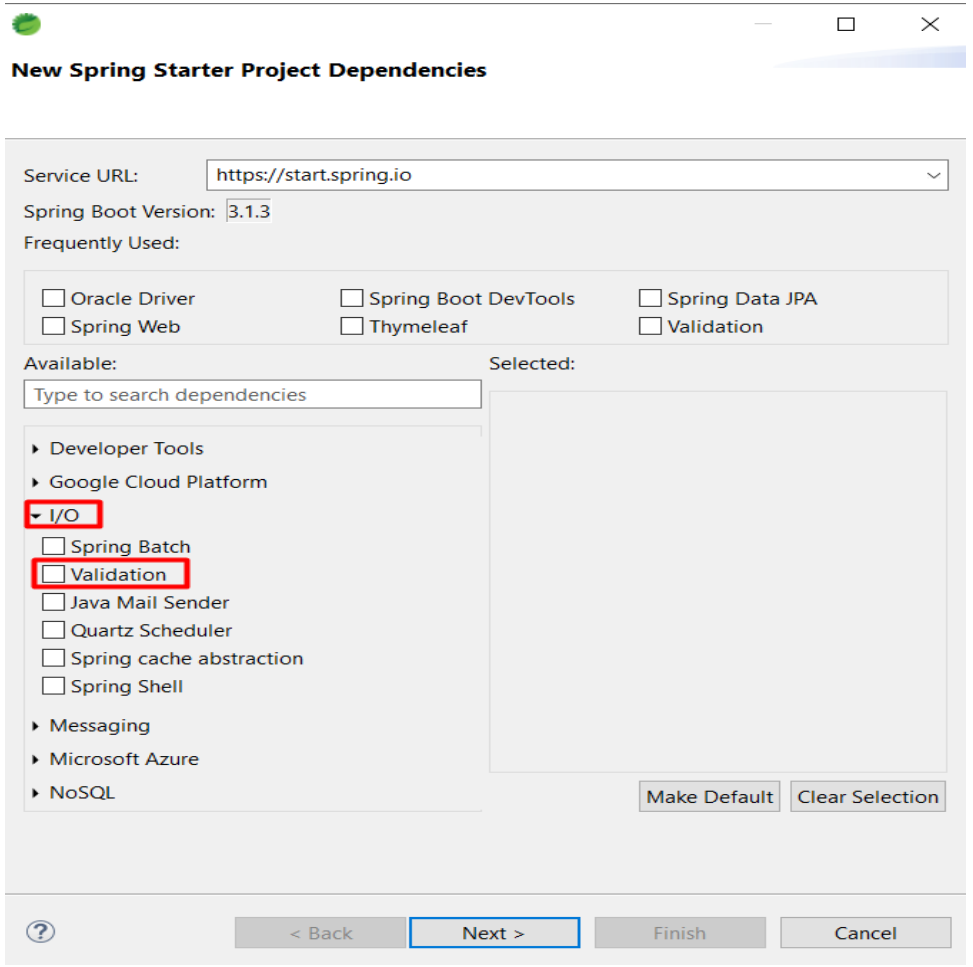
```java
@Entity
public class Articulo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="ID_ARTICULO")
    private Long idArticulo;

    @Column(name="NOMBRE")
    private String nombre;

    @ManyToMany(mappedBy = "articulos")
    private Set<Fabricante> fabricantes = new HashSet<>();
}
```

JoinTable:

Parámetro	Descripción
name	Nombre de la tabla que será creada físicamente en la base de datos
joinColumns	Corresponde al nombre para el ID de la Entidad donde este la anotación @JoinTable
inverseJoinColumns	Corresponde al nombre para el ID de la Entidad que no tiene la anotación @JoinTable

VALIDANDO ENTIDADES

Adicionalmente a mapear la clase con una tabla de BBDD, también podemos para añadir otras validaciones para que esta sea más robusta, pero para ello debemos agregar al **pom.xml.**



Alguna de las validaciones que podemos realizar a las entities, para hacer nuestra aplicación más robusta, son las siguientes:

Anotación	Descripción
@NotEmpty	Usado generalmente en el tipo String e indica obligatoriedad.
@NotNull	Usando en tipos que no String e indica obligatoriedad.
@Email	Valida que un determinado campo respete el formato que poseen los E_mails.
@Size	Indica el rango de caracteres que puede tener un determinado campo, para ellos se usan los parámetros min y max.
@Min	Indica el valor mínimo que debe tener el campo.
@Max	Indica el valor máximo que debe tener el campo.

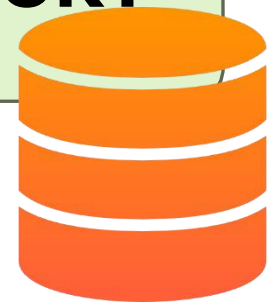

6

Controller

Service

REPOSITORY

PersistenceContext



El patrón Repository es un patrón de diseño que abstrae el acceso a la base de datos.

Los métodos por defecto incluidos para las operaciones CRUD son (entre otras):

- ✓ save(E entity): Guarda o actualiza una entidad.
- ✓ findById(Long id): Busca una entidad por su ID
- ✓ findAll(): Devuelve una lista de todas las entidades
- ✓ delete(E entity): Elimina una entidad específica.
- ✓ deleteById(Long id): Elimina una entidad específica.

Para poder utilizar estas operaciones, tendremos que crear una interfaz que extiende de JpaRepository

```
public interface FabricanteRepository extends JpaRepository<Fabricante, Long> {
    // JpaRepository ya incluye métodos para CRUD:
    // save() - Crear/Actualizar
    // findById() - Leer por ID
    // findAll() - Leer todos
    // deleteById() - Eliminar por ID


}
```

Adicionalmente, tenemos que tener en cuenta que existen dos tipos de funciones:

- ✓ Escritura
- ✓ Lectura

Como trabajamos con base de datos que gestionan transacciones, debemos usar en estos métodos la anotación de Spring:

@Transactional
@Transactional(readOnly=true)

¿QUÉ ES UN TRANSACCIÓN?

Una transacción de base de datos es una serie de una o más operaciones ejecutadas como una única unidad atómica de trabajo.

Esto significa que, o bien todas las operaciones de la transacción se completan con éxito, o bien ninguna de ellas se aplica a la base de datos.

Las transacciones se utilizan para garantizar la coherencia e integridad de los datos, asegurando que la base de datos siga siendo coherente incluso en caso de fallos o errores del sistema.

Operaciones del EntityManager

La siguiente tabla nos muestras las operaciones que nos permite realizar JPA

Operación	Descripción
persist	Nos permite realizar un INSERT o UPDATE en BBDD, en función de que si el dato ya existe o no.
remove	Nos permite eliminar un registro de la base de datos.
find	Nos permite buscar en la base de datos usando el atributo definido como PK.
merge	Se utiliza para actualizar un registro existente en base de datos.
refresh	Permite actualizar la Entidad con la base de datos, tan solo es necesario mandar como parámetro la Entidad a actualizar, esto actualizará todos los atributos con los de la base de datos
contains	Permite saber si una entidad se encuentra en el Contexto de persistencia. Se envía la Entidad a validar y retorna una booleano, indicando si está o no en el Contexto de Persistencia

¿Qué es JPQL?

JPQL (Java Persistence Query Language) es un lenguaje de consulta similar a SQL, pero diseñado específicamente para trabajar con objetos de persistencia en Java. JPQL se utiliza para realizar consultas de bases de datos en aplicaciones Java que utilizan JPA (Java Persistence API).

```java
@Repository
public interface FabricanteRepository extends JpaRepository<Fabricante, Long> {
    // JpaRepository ya incluye métodos para CRUD:
    // save() - Crear/Actualizar
    // findById() - Leer por ID
    // findAll() - Leer todos
    // deleteById() - Eliminar por ID

    // Consulta JPQL con parámetro nombrado
    @Query("SELECT f FROM Fabricante f WHERE f.nombre = :nombre")
    List<Fabricante> buscarPorNombre(@Param("nombre") String nombre);

    // Consulta JPQL con parámetro posicional
    @Query("SELECT f FROM Fabricante f WHERE f.pais = ?1")
    List<Fabricante> buscarPorPais(String pais);

}
```