## JPA y Hibernate: Persistencia de Datos en Java con Ejemplos Sencillos

La persistencia de datos es un aspecto fundamental en el desarrollo de aplicaciones Java, especialmente cuando se interacciona con bases de datos relacionales. JPA y Hibernate son herramientas clave en este ámbito.

### 1. ¿Qué es JPA?

JPA (Java Persistence API) es la **especificación** o el **conjunto de normas y APIs estándar** que define cómo los objetos Java se pueden mapear a una base de datos relacional y cómo se pueden realizar operaciones de persistencia (guardar, recuperar, actualizar, eliminar datos). Actúa como el "contrato" o el "plano" de lo que se debe hacer en cuanto a persistencia.

### 2. ¿Qué es Hibernate?

Hibernate es una **implementación** o una **herramienta concreta de código abierto para Java** (un framework ORM - Object-Relational Mapping) que sigue y cumple con las normas definidas por la especificación JPA. Es la "construcción" real que se hizo siguiendo esos planos de JPA. Además de implementar JPA, Hibernate a menudo ofrece sus propias características adicionales (extensiones) que van más allá de lo que define la especificación estándar de JPA.

Su propósito principal es simplificar la interacción con bases de datos relacionales, permitiendo a los desarrolladores trabajar con **objetos Java** en lugar de escribir sentencias SQL directamente. En esencia, Hibernate actúa como un puente entre la programación orientada a objetos de Java y el modelo relacional de las bases de datos.

### 3. El Problema que Resuelve Hibernate: Impedancia Objeto-Relacional

En el desarrollo de aplicaciones Java, es común la necesidad de almacenar y recuperar datos de una base de datos relacional. La **impedancia objeto-relacional** es la diferencia fundamental entre el modelo de bases de datos relacionales (tablas, filas, columnas) y el modelo de objetos de Java (objetos con propiedades, métodos y relaciones de herencia/composición).

Tradicionalmente, la interacción con la base de datos se realizaba con JDBC (Java Database Connectivity), lo que implicaba:

* Escribir sentencias SQL para cada operación (SELECT, INSERT, UPDATE, DELETE).
* Mapear manualmente los resultados de las consultas (ResultSet) a objetos Java.
* Gestionar conexiones, transacciones y recursos de forma explícita.
* Repetir mucho código "boilerplate" (repetitivo y estándar).

Este enfoque podía ser tedioso, propenso a errores y difícil de mantener, especialmente en aplicaciones grandes.

### 4. Cómo Hibernate Resuelve el Problema (Mapeo Objeto-Relacional)

Hibernate automatiza gran parte del proceso de mapeo entre objetos Java y tablas de base de datos. Permite definir cómo las clases Java (llamadas **entidades**) se corresponden con las tablas de la base de datos y cómo las propiedades de los objetos se mapean a las columnas.

**Conceptos clave:**

* **Entidades (Java Objects):** Son clases Java POJO (Plain Old Java Object) que representan los datos de tu aplicación (ej., una clase Usuario con propiedades id, nombre, email).
* **Mapeo:** Se define cómo estas clases Java se mapean a las tablas de la base de datos, principalmente de dos maneras:
  + **Anotaciones:** Utilizando anotaciones de Java que forman parte de la especificación JPA (ej., @Entity, @Table, @Id, @Column).
  + **Archivos XML:** Menos común hoy en día, pero usados históricamente para definir los mapeos.
* **Abstracción de SQL:** Una vez configurado el mapeo, no es necesario escribir SQL para la mayoría de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar), ya que Hibernate genera y ejecuta el SQL subyacente.
  + **Guardar:** session.save(miObjeto) genera un INSERT.
  + **Recuperar:** session.get(Clase.class, id) genera un SELECT.
  + **Actualizar:** Modificar el objeto Java y llamar a session.update(miObjeto) (o el objeto se sincroniza automáticamente si está en el contexto de persistencia) genera un UPDATE.
  + **Eliminar:** session.delete(miObjeto) genera un DELETE.
* **HQL (Hibernate Query Language) y Criteria API:** Para consultas más complejas, Hibernate ofrece:
  + **HQL:** Un lenguaje de consulta orientado a objetos, similar a SQL, que opera sobre entidades Java y sus propiedades.
  + **Criteria API:** Permite construir consultas de forma programática usando objetos Java.
* **Gestión de Transacciones:** Hibernate proporciona un marco para gestionar transacciones de base de datos, asegurando la consistencia e integridad de los datos.
* **Caché:** Incluye mecanismos de caché (primer y segundo nivel) para mejorar el rendimiento, reduciendo accesos a la base de datos.
* **Independencia de la Base de Datos:** Al abstraer la capa SQL, permite cambiar la base de datos subyacente con mínimos cambios en el código de la aplicación.

### 5. Arquitectura Simplificada de Hibernate

Los componentes clave de Hibernate incluyen:

* **Configuration:** Carga los archivos de configuración de Hibernate (ej., hibernate.cfg.xml o anotaciones).
* **SessionFactory:** Una vez configurado, Configuration crea una SessionFactory. Es una clase *thread-safe*, costosa de crear, por lo que se suele crear una sola vez por aplicación. Se usa para obtener objetos Session.
* **Session:** Representa una conversación con la base de datos. Es un objeto *single-threaded* y de corta duración, a través del cual se realizan las operaciones de persistencia.
* **Transaction:** Maneja las unidades de trabajo atómicas con la base de datos, asegurando que un conjunto de operaciones se complete por completo o se revierta si algo falla.

### 6. Ventajas de Usar Hibernate

* **Productividad:** Reduce significativamente el código necesario para interactuar con la base de datos.
* **Mantenibilidad:** El código es más limpio y fácil de mantener al trabajar con objetos Java.
* **Portabilidad:** Facilita el cambio de bases de datos.
* **Rendimiento:** Ofrece caché y carga diferida (lazy loading) para optimizar el rendimiento.
* **Manejo de colecciones y herencia:** Hibernate gestiona el mapeo de colecciones y estructuras de herencia a tablas relacionales de forma natural.
* **Estándar de la industria:** Hibernate es la implementación más popular de JPA.

### 7. Cuándo Usar Hibernate

Hibernate es una excelente opción para:

* Aplicaciones empresariales Java que necesitan persistir datos en bases de datos relacionales.
* Proyectos donde la agilidad en el desarrollo y la mantenibilidad son clave.
* Sistemas que buscan abstraerse de los detalles de SQL específicos de cada base de datos.

En resumen, Hibernate simplifica enormemente el desarrollo de aplicaciones Java que interactúan con bases de datos, permitiendo enfocarse en la lógica de negocio.

## Anotaciones en Hibernate y JPA

Hibernate utiliza un conjunto de anotaciones para mapear clases Java a tablas de base de datos. Muchas de estas anotaciones son parte de la especificación estándar de **JPA**, haciéndolas portables entre diferentes implementaciones (Hibernate, EclipseLink, OpenJPA). Hibernate también tiene anotaciones propias para características específicas.

### I. Anotaciones de Nivel de Clase (Para Entidades y Mapeo General)

Se colocan a nivel de clase para definir la entidad y sus propiedades generales de mapeo.

* **@Entity**: Indica que una clase Java es una entidad persistente y debe mapearse a una tabla de base de datos.
  + Ubicación: Declaración de la clase.

Java  
import jakarta.persistence.Entity; // O javax.persistence.Entity para versiones antiguas de JPA  
  
@Entity // Esta clase es una entidad JPA y se mapeará a una tabla de BD  
public class Producto {  
 // ... atributos y métodos  
}

* **@Table**: Permite configurar detalles de la tabla a la que se mapea la entidad.
  + Atributos comunes: name (nombre de la tabla), schema, catalog, uniqueConstraints (restricciones de unicidad), indexes.
  + Ubicación: Declaración de la clase, junto a @Entity.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Table;  
  
@Entity  
@Table(name = "mis\_productos") // Mapea la entidad Producto a la tabla "mis\_productos"  
public class Producto {  
 // ...  
}

* **@Embeddable**: Indica que una clase es un "componente incrustable" o "objeto de valor". Sus atributos se incrustan en la tabla de la entidad que la contiene, no se mapea a su propia tabla.
  + Ubicación: Declaración de la clase del componente.

Java  
import jakarta.persistence.Embeddable;  
  
@Embeddable // Esta clase puede ser incrustada en otra entidad  
public class Direccion {  
 private String calle;  
 private String ciudad;  
 // ... getters y setters  
}  
  
@Entity  
public class Cliente {  
 // ...  
 @Embedded // Indica que el campo "direccion" es un objeto incrustable  
 private Direccion direccion;  
 // ...  
}

* **@MappedSuperclass**: Indica que una clase es una superclase de mapeo. Sus atributos persistentes se heredan a sus subclases de entidad, pero la superclase en sí no se mapea a una tabla. Útil para compartir atributos comunes.
  + Ubicación: Declaración de la clase superclase.

Java  
import jakarta.persistence.MappedSuperclass;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
  
@MappedSuperclass // Esta clase no se mapea a una tabla, pero sus atributos sí se heredan  
public abstract class EntidadBase {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 // ... getters y setters (ej. para id)  
}  
  
@Entity  
@Table(name = "usuarios")  
public class Usuario extends EntidadBase { // Hereda el 'id' de EntidadBase  
 private String nombre;  
 // ...  
}

* **@Inheritance (JPA)**: Define la estrategia de mapeo de herencia para una jerarquía de clases de entidades.
  + Atributo común: strategy (SINGLE\_TABLE, JOINED, TABLE\_PER\_CLASS).
  + Ubicación: Clase raíz de la jerarquía de herencia.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Inheritance;  
import jakarta.persistence.InheritanceType;  
  
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE\_TABLE) // Todas las subclases en una única tabla  
public abstract class Empleado {  
 private String nombre;  
 // ...  
}  
  
@Entity  
public class Desarrollador extends Empleado {  
 private String lenguajePrincipal;  
 // ...  
}

### II. Anotaciones de Nivel de Campo/Propiedad (Para Columnas y Atributos)

Se colocan sobre los campos o los métodos getter de una entidad para definir su mapeo a columnas de la base de datos.

* **@Id**: Marca un campo como la clave primaria de la entidad. Cada entidad debe tener una.
  + Ubicación: Campo que representa la clave primaria.

Java  
import jakarta.persistence.Id;  
  
@Entity  
public class Coche {  
 @Id // Este campo es la clave primaria de la tabla 'Coche'  
 private String matricula;  
 private String modelo;  
 // ...  
}

* **@GeneratedValue**: Especifica cómo se generarán automáticamente los valores de la clave primaria.
  + Atributos comunes: strategy (AUTO, IDENTITY, SEQUENCE, TABLE), generator.
  + Ubicación: Junto a @Id.

Java  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
  
@Entity  
public class Cliente {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY) // El ID se auto-incrementa por la BD  
 private Long id;  
 private String nombre;  
 // ...  
}

* **@Column**: Configura detalles de la columna a la que se mapea un atributo.
  + Atributos comunes: name (nombre de la columna), length, nullable, unique, precision, scale, insertable, updatable.
  + Ubicación: Campo o getter.

Java  
import jakarta.persistence.Column;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class Usuario {  
 @Id  
 private Long id;  
  
 @Column(name = "nombre\_completo", length = 100, nullable = false, unique = true)  
 private String nombre; // Este campo se mapea a la columna "nombre\_completo"  
 // que no puede ser nula y debe ser única, con longitud máxima de 100  
  
 @Column(name = "email\_usuario", updatable = false) // Esta columna no se actualizará una vez insertada  
 private String email;  
 // ...  
}

* **@Transient**: Indica que un campo **no debe ser persistido** en la base de datos. Hibernate lo ignorará.
  + Ubicación: Campo.

Java  
import jakarta.persistence.Transient;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class Tarea {  
 @Id  
 private Long id;  
 private String descripcion;  
  
 @Transient // Este campo no se guardará en la base de datos  
 private boolean estaCompletaEnMemoria; // Solo para lógica en memoria, no persistente  
 // ...  
}

* **@Temporal**: Para campos java.util.Date o java.util.Calendar, especifica el tipo de dato temporal a persistir.
  + Atributos comunes: TemporalType.DATE, TemporalType.TIME, TemporalType.TIMESTAMP.
  + Ubicación: Campo de fecha/hora.

Java  
import jakarta.persistence.Temporal;  
import jakarta.persistence.TemporalType;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import java.util.Date;  
  
@Entity  
public class Evento {  
 @Id  
 private Long id;  
  
 @Temporal(TemporalType.DATE) // Solo se guarda la fecha (año, mes, día)  
 private Date fechaEvento;  
  
 @Temporal(TemporalType.TIMESTAMP) // Se guarda la fecha y hora completa  
 private Date momentoRegistro;  
 // ...  
}

* **@Lob**: Indica que un campo es un "Large Object" (BLOB o CLOB) para datos binarios grandes o texto largo.
  + Ubicación: Campo.

Java  
import jakarta.persistence.Lob;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class Documento {  
 @Id  
 private Long id;  
 private String titulo;  
  
 @Lob // Este campo almacenará un texto muy largo (CLOB)  
 private String contenidoCompleto;  
  
 @Lob // Este campo almacenará datos binarios grandes (BLOB), como una imagen  
 private byte[] archivoBinario;  
 // ...  
}

* **@Enumerated**: Define cómo se persistirá un tipo enumerado (enum) en la base de datos.
  + Atributos comunes: EnumType.ORDINAL (posición, no recomendado), EnumType.STRING (nombre, más robusto).
  + Ubicación: Campo enum.

Java  
import jakarta.persistence.Enumerated;  
import jakarta.persistence.EnumType;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
public enum EstadoPedido {  
 PENDIENTE, PROCESANDO, ENVIADO, ENTREGADO, CANCELADO  
}  
  
@Entity  
public class Pedido {  
 @Id  
 private Long id;  
  
 @Enumerated(EnumType.STRING) // Guarda el nombre del enum (ej. "PENDIENTE")  
 private EstadoPedido estado;  
 // ...  
}

* **@Embedded**: Marca un campo que es una instancia de una clase @Embeddable. Sus atributos se incrustan en la tabla de la entidad propietaria.
  + Ubicación: Campo. (Ver ejemplo en @Embeddable).
* **@AttributeOverride / @AttributeOverrides**: Se usa para sobreescribir el mapeo de columnas de un atributo de una clase @Embeddable cuando se incrusta en una entidad, útil para nombres de columna distintos.
  + Ubicación: Campo @Embedded.

Java  
import jakarta.persistence.AttributeOverride;  
import jakarta.persistence.AttributeOverrides;  
import jakarta.persistence.Column;  
import jakarta.persistence.Embedded;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
// Supongamos que ya existe la clase @Embeddable Direccion con 'calle' y 'ciudad'  
  
@Entity  
public class Empresa {  
 @Id  
 private Long id;  
 private String nombre;  
  
 @Embedded  
 @AttributeOverrides({ // Sobreescribimos los nombres de columna para esta instancia de Direccion  
 @AttributeOverride(name = "calle", column = @Column(name = "calle\_fiscal")),  
 @AttributeOverride(name = "ciudad", column = @Column(name = "ciudad\_fiscal"))  
 })  
 private Direccion direccionFiscal;  
  
 @Embedded  
 @AttributeOverrides({  
 @AttributeOverride(name = "calle", column = @Column(name = "calle\_envio")),  
 @AttributeOverride(name = "ciudad", column = @Column(name = "ciudad\_envio"))  
 })  
 private Direccion direccionEnvio;  
 // ...  
}

* **@Version (Control de Concurrencia Optimista)**: Marca un campo numérico o de fecha/hora que Hibernate usa para el control de concurrencia optimista, lanzando OptimisticLockException si el valor ha cambiado.
  + Ubicación: Campo int, long, short, java.sql.Timestamp, etc.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.Version;  
  
@Entity  
public class CuentaBancaria {  
 @Id  
 private Long id;  
 private double saldo;  
  
 @Version // Hibernate usará este campo para detectar modificaciones concurrentes  
 private Long version; // O puede ser int, Timestamp  
 // ...  
}

* **@Formula (Hibernate específica)**: Permite definir un campo calculado en la entidad basado en una expresión SQL, cuyo valor se calcula dinámicamente al cargar la entidad.
  + Ubicación: Campo.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import org.hibernate.annotations.Formula; // Anotación específica de Hibernate  
  
@Entity  
public class OrdenCompra {  
 @Id  
 private Long id;  
 private int cantidad;  
 private double precioUnitario;  
  
 // Este campo 'total' no se almacena en la BD, se calcula al cargar el objeto  
 @Formula("cantidad \* precioUnitario")  
 private double total;  
 // ...  
}

### III. Anotaciones de Relación (Asociaciones entre Entidades)

Definen las relaciones entre diferentes entidades (uno a uno, uno a muchos, muchos a uno, muchos a muchos).

* **@OneToOne**: Mapea una relación uno a uno.
  + Atributos comunes: mappedBy, cascade, fetch (LAZY por defecto), optional.
  + Ubicación: Campo de la relación.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.OneToOne;  
import jakarta.persistence.JoinColumn;  
  
@Entity  
public class Persona {  
 @Id  
 private Long id;  
 private String nombre;  
  
 @OneToOne // Una persona tiene un único pasaporte  
 @JoinColumn(name = "pasaporte\_id") // La FK 'pasaporte\_id' está en la tabla de Persona  
 private Pasaporte pasaporte;  
 // ...  
}  
  
@Entity  
public class Pasaporte {  
 @Id  
 private Long id;  
 private String numero;  
  
 @OneToOne(mappedBy = "pasaporte") // Mapea la relación inversa, el FK está en Persona  
 private Persona persona;  
 // ...  
}

* **@OneToMany**: Mapea una relación uno a muchos.
  + Atributos comunes: Igual que @OneToOne, con mappedBy crucial.
  + Ubicación: Campo Collection (ej. List, Set) en el lado "uno".

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.OneToMany;  
import jakarta.persistence.CascadeType;  
import java.util.List;  
import java.util.ArrayList;  
  
@Entity  
public class Departamento {  
 @Id  
 private Long id;  
 private String nombre;  
  
 @OneToMany(mappedBy = "departamento", cascade = CascadeType.ALL, orphanRemoval = true)  
 // Un departamento tiene muchos empleados.  
 // "mappedBy" indica que el lado "muchos" (Empleado) es el dueño de la relación (tiene la FK).  
 // CascadeType.ALL: Operaciones (persist, remove) en Departamento se propagan a Empleado.  
 // orphanRemoval: Si un Empleado se desvincula de un Departamento, se elimina.  
 private List<Empleado> empleados = new ArrayList<>();  
 // ...  
}  
  
@Entity  
public class Empleado {  
 @Id  
 private Long id;  
 private String nombre;  
  
 @ManyToOne // Un empleado pertenece a un departamento  
 @JoinColumn(name = "departamento\_id") // La FK 'departamento\_id' está en la tabla de Empleado  
 private Departamento departamento;  
 // ...  
}

* **@ManyToOne**: Mapea una relación muchos a uno.
  + Atributos comunes: Igual que @OneToOne, con optional y fetch relevantes.
  + Ubicación: Campo de la relación en el lado "muchos". (Ver ejemplo en @OneToMany).
* **@ManyToMany**: Mapea una relación muchos a muchos, requiriendo una tabla de unión.
  + Atributos comunes: mappedBy, cascade, fetch.
  + Ubicación: Campo Collection en ambos lados.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.ManyToMany;  
import jakarta.persistence.JoinTable;  
import jakarta.persistence.JoinColumn;  
import java.util.Set;  
import java.util.HashSet;  
  
@Entity  
public class Curso {  
 @Id  
 private Long id;  
 private String titulo;  
  
 @ManyToMany // Un curso puede tener muchos estudiantes  
 @JoinTable(  
 name = "curso\_estudiante", // Nombre de la tabla de unión  
 joinColumns = @JoinColumn(name = "curso\_id"), // FK de Curso en la tabla de unión  
 inverseJoinColumns = @JoinColumn(name = "estudiante\_id") // FK de Estudiante en la tabla de unión  
 )  
 private Set<Estudiante> estudiantes = new HashSet<>();  
 // ...  
}  
  
@Entity  
public class Estudiante {  
 @Id  
 private Long id;  
 private String nombre;  
  
 @ManyToMany(mappedBy = "estudiantes") // Un estudiante puede estar en muchos cursos  
 private Set<Curso> cursos = new HashSet<>();  
 // ...  
}

* **@JoinColumn**: Se usa con @OneToOne, @ManyToOne para especificar la columna de clave foránea en la tabla.
  + Atributos comunes: name (nombre de la columna FK), referencedColumnName.
  + Ubicación: Campo de la relación. (Ver ejemplos en @OneToOne y @OneToMany).
* **@JoinTable**: Se usa con @ManyToMany para definir la tabla de unión.
  + Atributos comunes: name (nombre de la tabla de unión), joinColumns, inverseJoinColumns.
  + Ubicación: En uno de los campos @ManyToMany. (Ver ejemplo en @ManyToMany).

### IV. Anotaciones de Generación de ID Personalizada

Permiten mayor personalización en cómo se generan las claves primarias.

* **@SequenceGenerator**: Define un generador de secuencia de base de datos para @GeneratedValue.
  + Ubicación: Clase o paquete.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.SequenceGenerator;  
  
@Entity  
@SequenceGenerator(name = "mi\_secuencia\_gen", sequenceName = "mi\_secuencia", allocationSize = 1)  
public class Elemento {  
 @Id  
 @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "mi\_secuencia\_gen")  
 private Long id;  
 private String valor;  
 // ...  
}

* **@TableGenerator**: Define un generador basado en una tabla para @GeneratedValue.
  + Ubicación: Clase o paquete.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.TableGenerator;  
  
@Entity  
@TableGenerator(name = "id\_generador\_tabla", table = "generadores\_id",  
 pkColumnName = "nombre\_entidad", valueColumnName = "siguiente\_valor",  
 allocationSize = 1)  
public class Articulo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.TABLE, generator = "id\_generador\_tabla")  
 private Long id;  
 private String descripcion;  
 // ...  
}

* **@GenericGenerator (Hibernate específica)**: Anotación de Hibernate más flexible para definir estrategias de generación de ID personalizadas (UUIDs, Hi/Lo, etc.).
  + Ubicación: Clase o paquete.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import org.hibernate.annotations.GenericGenerator; // Anotación específica de Hibernate  
  
@Entity  
public class LogEntry {  
 @Id  
 @GeneratedValue(generator = "uuid2") // Usa una estrategia de generación UUID  
 @GenericGenerator(name = "uuid2", strategy = "uuid2")  
 private String id; // El ID será un String UUID  
 private String mensaje;  
 // ...  
}

### V. Anotaciones de Consultas (Queries)

Para definir consultas predefinidas o propiedades de consulta.

* **@NamedQuery / @NamedQueries**: Define consultas JPQL con nombre que pueden ser reutilizadas.
  + Atributos: name, query.
  + Ubicación: Declaración de la clase de la entidad.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.NamedQuery;  
import jakarta.persistence.NamedQueries;  
import jakarta.persistence.Id;  
  
@Entity  
@NamedQueries({  
 @NamedQuery(name = "Producto.findAll", query = "SELECT p FROM Producto p"),  
 @NamedQuery(name = "Producto.findByPrecioMayorQue",  
 query = "SELECT p FROM Producto p WHERE p.precio > :minPrecio")  
})  
public class Producto {  
 @Id  
 private Long id;  
 private String nombre;  
 private double precio;  
 // ...  
}

* **@NamedNativeQuery / @NamedNativeQueries**: Similar a NamedQuery, pero para definir consultas SQL nativas.
  + Ubicación: Declaración de la clase de la entidad.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.NamedNativeQuery;  
import jakarta.persistence.Id;  
  
@Entity  
@NamedNativeQuery(name = "Usuario.findActivosSQL",  
 query = "SELECT \* FROM Usuario WHERE esta\_activo = TRUE",  
 resultClass = Usuario.class)  
public class Usuario {  
 @Id  
 private Long id;  
 private String nombre;  
 private boolean estaActivo;  
 // ...  
}

### VI. Anotaciones de Auditoría y Timestamps (Hibernate específica)

* **@CreationTimestamp (Hibernate específica)**: Marca un campo de fecha/hora para ser automáticamente configurado con la marca de tiempo de creación de la entidad.
  + Ubicación: Campo Date o Calendar.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import org.hibernate.annotations.CreationTimestamp; // Anotación específica de Hibernate  
import java.time.LocalDateTime; // Se recomienda usar java.time en vez de java.util.Date  
  
@Entity  
public class Comentario {  
 @Id  
 private Long id;  
 private String texto;  
  
 @CreationTimestamp // Se asigna automáticamente la fecha y hora de creación  
 private LocalDateTime fechaCreacion;  
 // ...  
}

* **@UpdateTimestamp (Hibernate específica)**: Marca un campo de fecha/hora para ser automáticamente actualizado con la marca de tiempo de la última modificación de la entidad.
  + Ubicación: Campo Date o Calendar.

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import org.hibernate.annotations.UpdateTimestamp; // Anotación específica de Hibernate  
import java.time.LocalDateTime;  
  
@Entity  
public class Configuracion {  
 @Id  
 private Long id;  
 private String valor;  
  
 @UpdateTimestamp // Se actualiza automáticamente cada vez que la entidad se modifica  
 private LocalDateTime ultimaModificacion;  
 // ...  
}

## Herencia en JPA

JPA soporta la herencia en el modelo de objetos, mapeando la jerarquía de clases a tablas en la base de datos. Ofrece **tres estrategias de mapeo de herencia**.

### Estrategias de Mapeo de Herencia

* **SINGLE\_TABLE (Estrategia de Tabla Única):**
  + **Descripción:** Todas las clases de la jerarquía (clase padre y subclases) se mapean a **una única tabla** en la base de datos. Es la estrategia por defecto.
  + **Funcionamiento:** La tabla contiene columnas para todos los atributos y usa una **columna discriminadora** para identificar el tipo de entidad.
  + **Ventajas:** Simplicidad y eficiencia en consultas polimórficas.
  + **Desventajas:** Puede generar muchas columnas nulas para atributos específicos de subclases.
  + **Anotación:** @Inheritance(strategy = InheritanceType.SINGLE\_TABLE).

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Inheritance;  
import jakarta.persistence.InheritanceType;  
import jakarta.persistence.DiscriminatorColumn; // Para nombrar la columna discriminadora  
import jakarta.persistence.DiscriminatorType; // Tipo de dato de la columna discriminadora  
  
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE\_TABLE) // Todas las clases en una tabla  
@DiscriminatorColumn(name = "TIPO\_PERSONA", discriminatorType = DiscriminatorType.STRING) // Columna para diferenciar tipos  
public abstract class Persona {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private String nombre;  
 // ...  
}  
  
@Entity  
// @DiscriminatorValue("CLIENTE") // Valor para esta subclase en la columna TIPO\_PERSONA  
public class Cliente extends Persona {  
 private String email;  
 // ...  
}  
  
@Entity  
// @DiscriminatorValue("EMPLEADO") // Valor para esta subclase en la columna TIPO\_PERSONA  
public class Gerente extends Persona {  
 private double salario;  
 // ...  
}  
  
// Tabla resultante (ejemplo en SQL):  
// CREATE TABLE Persona (  
// id BIGINT PRIMARY KEY,  
// nombre VARCHAR(255),  
// TIPO\_PERSONA VARCHAR(255), -- Columna discriminadora  
// email VARCHAR(255), -- Columna para Cliente (será NULL para Gerente)  
// salario DOUBLE -- Columna para Gerente (será NULL para Cliente)  
// );

* **JOINED (Estrategia de Subclases Unida / Tabla por Subclase):**
  + **Descripción:** Cada clase en la jerarquía (padre y subclases) se mapea a **su propia tabla**.
  + **Funcionamiento:** La tabla del padre tiene sus atributos; las tablas de las subclases tienen sus atributos específicos y una **clave foránea (FK)** que referencia la clave primaria (PK) del padre. JPA realiza un JOIN para reconstruir un objeto completo.
  + **Ventajas:** Esquema de base de datos más normalizado, sin columnas nulas innecesarias.
  + **Desventajas:** Consultas polimórficas pueden ser menos eficientes por los JOINs.
  + **Anotación:** @Inheritance(strategy = InheritanceType.JOINED).

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Inheritance;  
import jakarta.persistence.InheritanceType;  
import jakarta.persistence.PrimaryKeyJoinColumn; // Para especificar la FK en las subclases  
  
@Entity  
@Inheritance(strategy = InheritanceType.JOINED) // Cada clase en su propia tabla, unidas por PK/FK  
public abstract class Empleado {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private String nombre;  
 // ...  
}  
  
@Entity  
@PrimaryKeyJoinColumn(name = "empleado\_id") // La columna FK en la tabla 'Desarrollador' que apunta a 'Empleado.id'  
public class Desarrollador extends Empleado {  
 private String lenguajePrincipal;  
 // ...  
}  
  
@Entity  
@PrimaryKeyJoinColumn(name = "empleado\_id") // La columna FK en la tabla 'Gerente' que apunta a 'Empleado.id'  
public class Gerente extends Empleado {  
 private double bonus;  
 // ...  
}  
  
// Tablas resultantes (ejemplo en SQL):  
// CREATE TABLE Empleado (  
// id BIGINT PRIMARY KEY,  
// nombre VARCHAR(255)  
// );  
//  
// CREATE TABLE Desarrollador (  
// empleado\_id BIGINT PRIMARY KEY, -- Esta es la FK a Empleado.id  
// lenguajePrincipal VARCHAR(255),  
// FOREIGN KEY (empleado\_id) REFERENCES Empleado(id)  
// );  
//  
// CREATE TABLE Gerente (  
// empleado\_id BIGINT PRIMARY KEY, -- Esta es la FK a Empleado.id  
// bonus DOUBLE,  
// FOREIGN KEY (empleado\_id) REFERENCES Empleado(id)  
// );

* **TABLE\_PER\_CLASS (Estrategia de Tabla por Clase Concreta):**
  + **Descripción:** Cada **clase concreta** en la jerarquía se mapea a **su propia tabla**.
  + **Funcionamiento:** La tabla de cada subclase contiene **todas las columnas**, incluyendo las heredadas. No hay JOINs, pero hay duplicación de columnas.
  + **Ventajas:** Puede ser eficiente para consultar un tipo específico.
  + **Desventajas:** Duplicación de datos y esquema, dificultad para consultas polimórficas (requieren UNIONs). **Generalmente no se recomienda**.
  + **Anotación:** @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS).

Java  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Inheritance;  
import jakarta.persistence.InheritanceType;  
  
@Entity  
@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS) // Cada clase concreta tiene su propia tabla con todos sus datos  
public abstract class Vehiculo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO) // Necesita estrategia de generación que funcione con esta herencia  
 private Long id;  
 private String marca;  
 // ...  
}  
  
@Entity  
public class Coche extends Vehiculo {  
 private int numeroPuertas;  
 // ...  
}  
  
@Entity  
public class Moto extends Vehiculo {  
 private boolean tieneSidecar;  
 // ...  
}  
  
// Tablas resultantes (ejemplo en SQL):  
// CREATE TABLE Coche (  
// id BIGINT PRIMARY KEY,  
// marca VARCHAR(255), -- Atributo heredado  
// numeroPuertas INT  
// );  
//  
// CREATE TABLE Moto (  
// id BIGINT PRIMARY KEY,  
// marca VARCHAR(255), -- Atributo heredado (duplicado)  
// tieneSidecar BOOLEAN  
// );

## Persistencia en JPA: El Contexto y el Estado de las Entidades

El concepto de **Contexto de Persistencia** es fundamental en JPA.

### Contexto de Persistencia

* **Creación:** Un EntityManager está asociado a un **Contexto de Persistencia** cuando se crea u obtiene.
* **Función:** Actúa como una caché de primer nivel y un área de trabajo donde el EntityManager gestiona el **ciclo de vida de las entidades**, controlando y monitorizando los objetos que manipula.
* **Conexión a la Base de Datos:** El EntityManager y el Contexto de Persistencia mantienen una referencia a la conexión subyacente a la base de datos.
* **Transacciones:** Las operaciones de persistencia se realizan dentro de una **transacción**. Al realizar un *commit* de una transacción, el EntityManager detecta los cambios en las entidades dentro de su Contexto de Persistencia y los sincroniza con la base de datos. Los cambios en los objetos gestionados se propagan cuando la transacción se completa exitosamente; los objetos no gestionados no son afectados.  
  Java  
  import jakarta.persistence.EntityManager;  
  import jakarta.persistence.EntityManagerFactory;  
  import jakarta.persistence.Persistence;  
    
  public class EjemploPersistencia {  
    
   public static void main(String[] args) {  
   // 1. Crear EntityManagerFactory (costoso, se hace una vez por aplicación)  
   // "miUnidadPersistencia" es el nombre definido en persistence.xml  
   EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadPersistencia");  
    
   // 2. Crear EntityManager (asociado a un Contexto de Persistencia, por cada "conversación" con la BD)  
   EntityManager em = emf.createEntityManager();  
    
   // 3. Iniciar una transacción  
   em.getTransaction().begin();  
    
   try {  
   // Crear una nueva entidad (estado NEW)  
   Producto productoNuevo = new Producto("Laptop", 1200.0);  
   System.out.println("Estado de productoNuevo: NEW");  
    
   // Persistir la entidad (pasa a estado MANAGED)  
   em.persist(productoNuevo);  
   System.out.println("Estado de productoNuevo después de persist(): MANAGED");  
   System.out.println("ID asignado: " + productoNuevo.getId()); // ID podría ser asignado aquí si es IDENTITY  
    
   // Modificar una entidad gestionada  
   productoNuevo.setPrecio(1150.0); // El cambio es rastreado por el EntityManager  
   System.out.println("Precio modificado en el objeto. Pendiente de sincronizar.");  
    
   // Confirmar la transacción (los cambios se sincronizan con la BD)  
   em.getTransaction().commit();  
   System.out.println("Transacción confirmada. Cambios persistidos.");  
    
   // La entidad sigue en estado MANAGED hasta que el EM se cierre o se desasocie explícitamente  
   System.out.println("Estado de productoNuevo después de commit: MANAGED");  
    
   } catch (Exception e) {  
   // Si algo falla, revertir la transacción  
   if (em.getTransaction().isActive()) {  
   em.getTransaction().rollback();  
   }  
   System.err.println("Error durante la transacción: " + e.getMessage());  
   } finally {  
   // Cerrar el EntityManager (desasocia todas las entidades de este contexto)  
   if (em.isOpen()) {  
   em.close();  
   System.out.println("EntityManager cerrado. Estado de productoNuevo: DETACHED");  
   }  
   // Cerrar EntityManagerFactory (al final de la aplicación)  
   if (emf.isOpen()) {  
   emf.close();  
   }  
   }  
   }  
  }

### Estado de las Entidades

Una entidad JPA puede estar en cuatro estados diferentes en relación con un Contexto de Persistencia:

* **New (Nuevo / Transient):**
  + **Descripción:** Instancia creada en memoria con new, pero **no asociada a ningún PersistenceContext**.
  + **Características:** No tiene representación en la base de datos y no está gestionada por JPA. Los cambios no se sincronizarán hasta que sea persistida.

Java  
// Producto aún no guardado ni conocido por JPA  
Producto productoEnMemoria = new Producto("Teclado", 75.0);  
// Estado: NEW (Transitorio)

* **Managed (Gestionado / Persistent):**
  + **Descripción:** Entidad **asociada a un PersistenceContext**. Esto ocurre al llamar a entityManager.persist(), entityManager.merge(), o al cargarla desde la base de datos (ej., con entityManager.find()).
  + **Características:** El EntityManager rastrea todos los cambios, que se sincronizarán con la base de datos al hacer *commit* de la transacción.

Java  
EntityManager em = ...; // Obtener EntityManager  
em.getTransaction().begin();  
  
Producto nuevoProducto = new Producto("Ratón", 30.0);  
em.persist(nuevoProducto); // Ahora 'nuevoProducto' está en estado MANAGED  
// Los cambios en 'nuevoProducto' serán rastreados.  
nuevoProducto.setPrecio(35.0); // Este cambio se guardará al hacer commit.  
  
Producto productoExistente = em.find(Producto.class, 1L); // 'productoExistente' también está en estado MANAGED  
productoExistente.setNombre("Monitor UltraWide"); // Este cambio también se guardará.  
  
em.getTransaction().commit();

* **Detached (Desasociado):**
  + **Descripción:** Una entidad que **ha estado asociada a un PersistenceContext pero deja de estarlo**. Puede ocurrir al cerrar el EntityManager (entityManager.close()), usar entityManager.detach(), o por expulsión de la caché de segundo nivel.
  + **Características:** Existe en memoria, pero los cambios **no se sincronizarán automáticamente**. Para persistir cambios, debe ser "adjuntada" de nuevo al PersistenceContext (normalmente con entityManager.merge()).

Java  
EntityManager em1 = ...;  
em1.getTransaction().begin();  
Producto p1 = em1.find(Producto.class, 1L); // p1 es MANAGED en em1  
em1.getTransaction().commit();  
em1.close(); // p1 ahora está DETACHED (el EM se cerró)  
  
p1.setNombre("Nuevo Nombre Detached"); // Este cambio NO se guarda automáticamente  
  
EntityManager em2 = ...; // Nuevo EntityManager, nuevo Contexto de Persistencia  
em2.getTransaction().begin();  
Producto p1Reattached = em2.merge(p1); // 'p1' se adjunta de nuevo a em2, 'p1Reattached' es MANAGED  
 // Los cambios de 'p1' (Nuevo Nombre Detached) ahora están en 'p1Reattached'  
em2.getTransaction().commit();  
em2.close();

* **Removed (Eliminado):**
  + **Descripción:** Entidad gestionada por un PersistenceContext y marcada para ser eliminada de la base de datos con entityManager.remove().
  + **Características:** Aunque sigue en el Contexto de Persistencia temporalmente, será borrada de la base de datos al hacer *commit*. Después del *commit*, puede considerarse en estado detached.

Java  
EntityManager em = ...;  
em.getTransaction().begin();  
  
Producto productoAEliminar = em.find(Producto.class, 2L); // productoAEliminar es MANAGED  
em.remove(productoAEliminar); // productoAEliminar pasa a estado REMOVED  
 // Todavía está en memoria, pero marcada para eliminación.  
  
em.getTransaction().commit(); // Se ejecuta el DELETE en la BD.  
 // Después del commit, 'productoAEliminar' está efectivamente DETACHED  
 // (ya no tiene una representación en la BD).

## Mapeo de Relaciones en Hibernate

Hibernate (con JPA) es una herramienta poderosa que nos permite conectar objetos Java con tablas de bases de datos. Un aspecto fundamental de esto es cómo manejamos las **relaciones** entre nuestros datos.

### 1. Relación OneToOne

Una relación **One-to-One** significa que **una entidad está directamente ligada a otra, y viceversa**. Piensa en una **Persona** y su **Pasaporte**: cada persona tiene un solo pasaporte, y cada pasaporte pertenece a una sola persona.

* **¿Cómo funciona?** Normalmente, una de las tablas almacena una **clave foránea (FK)** que apunta a la clave primaria (PK) de la otra tabla.
* **En tu código:**
  + Usas @OneToOne en el campo que representa la relación en ambas entidades (si la relación es bidireccional).
  + La entidad que "posee" la clave foránea (la que la tiene en su tabla) usará @JoinColumn para especificar el nombre de esa columna FK.
  + La otra entidad, en el lado mappedBy, indica que la relación la maneja el campo de la primera entidad.

**Ejemplo:**

Java

* // Entidad Persona
* @Entity
* public class Persona {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombre;
* @OneToOne(mappedBy = "persona", cascade = CascadeType.ALL, orphanRemoval = true)
* // 'mappedBy' dice que el campo 'persona' en la clase Pasaporte es quien gestiona la FK.
* // 'cascade = CascadeType.ALL' propaga operaciones (guardar, borrar).
* // 'orphanRemoval = true' borra el pasaporte si se desvincula de la persona.
* private Pasaporte pasaporte;
* // Getters y Setters
* 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 Pasaporte getPasaporte() { return pasaporte; }
* public void setPasaporte(Pasaporte pasaporte) {
* this.pasaporte = pasaporte;
* if (pasaporte != null) {
* pasaporte.setPersona(this); // Importante para la bidireccionalidad
* }
* }
* }
* // Entidad Pasaporte
* @Entity
* public class Pasaporte {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String numeroPasaporte;
* @OneToOne(fetch = FetchType.LAZY) // Es buena práctica usar LAZY, aunque EAGER es el default aquí.
* @JoinColumn(name = "persona\_id") // Esta es la FK en la tabla 'Pasaporte' que apunta a 'Persona'.
* private Persona persona;
* // Getters y Setters
* public Long getId() { return id; }
* public void setId(Long id) { this.id = id; }
* public String getNumeroPasaporte() { return numeroPasaporte; }
* public void setNumeroPasaporte(String numeroPasaporte) { this.numeroPasaporte = numeroPasaporte; }
* public Persona getPersona() { return persona; }
* public void setPersona(Persona persona) { this.persona = persona; }
* }

### 2. Relación ManyToOne

La relación **Many-to-One** es muy común: **muchas instancias de una entidad se relacionan con una única instancia de otra**. Imagina **Libros** y un **Autor**: muchos libros pueden ser escritos por un solo autor, pero cada libro tiene un único autor.

* **¿Cómo funciona?** La tabla del lado "muchos" contiene una **clave foránea (FK)** que referencia la clave primaria (PK) de la tabla del lado "uno".
* **En tu código:**
  + En la entidad del lado "muchos" (la que tiene la FK), usas @ManyToOne y @JoinColumn para la FK.
  + En la entidad del lado "uno", usas @OneToMany para mapear la colección de entidades "muchas". En este lado usarás mappedBy.

**Ejemplo:**

Java

* // Entidad Autor (lado "uno")
* @Entity
* public class Autor {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombre;
* @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
* // 'mappedBy' indica que el campo 'autor' en la clase Libro es quien gestiona la FK.
* private List<Libro> libros = new ArrayList<>();
* // Getters y Setters, y métodos de ayuda para mantener la colección sincronizada
* 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 List<Libro> getLibros() { return libros; }
* public void setLibros(List<Libro> libros) { this.libros = libros; }
* public void addLibro(Libro libro) {
* libros.add(libro);
* libro.setAutor(this); // Asegura la bidireccionalidad
* }
* public void removeLibro(Libro libro) {
* libros.remove(libro);
* libro.setAutor(null);
* }
* }
* // Entidad Libro (lado "muchos")
* @Entity
* public class Libro {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String titulo;
* @ManyToOne(fetch = FetchType.LAZY) // Por defecto es EAGER, pero LAZY es generalmente mejor.
* @JoinColumn(name = "autor\_id") // Esta es la FK en la tabla 'Libro' que apunta a 'Autor'.
* private Autor autor;
* // Getters y Setters
* public Long getId() { return id; }
* public void setId(Long id) { this.id = id; }
* public String getTitulo() { return titulo; }
* public void setTitulo(String titulo) { this.titulo = titulo; }
* public Autor getAutor() { return autor; }
* public void setAutor(Autor autor) { this.autor = autor; }
* }

### 3. Relación ManyToMany

Una relación **Many-to-Many** significa que **muchas instancias de una entidad pueden asociarse con muchas instancias de otra**. Por ejemplo, muchos **Estudiantes** pueden estar en muchos **Cursos**, y muchos Cursos pueden tener muchos Estudiantes.

* **¿Cómo funciona?** En la base de datos, estas relaciones se resuelven con una **tabla intermedia**(también llamada tabla de unión o asociación). Esta tabla solo contiene claves foráneas que apuntan a las claves primarias de las dos tablas originales.
* **En tu código:**
  + Ambas entidades usan @ManyToMany.
  + Uno de los lados es el "propietario" de la relación y usa @JoinTable para definir la tabla intermedia y sus columnas de unión (joinColumns y inverseJoinColumns).
  + El otro lado usa mappedBy para indicar que la relación es gestionada por el campo del lado propietario.

**Ejemplo (sin atributos en la tabla intermedia):**

Java

* // Entidad Estudiante
* @Entity
* public class Estudiante {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombre;
* @ManyToMany(fetch = FetchType.LAZY) // LAZY por defecto para colecciones
* @JoinTable(
* name = "estudiante\_curso", // Nombre de la tabla intermedia
* joinColumns = @JoinColumn(name = "estudiante\_id"), // FK en 'estudiante\_curso' que apunta a 'Estudiante'.
* inverseJoinColumns = @JoinColumn(name = "curso\_id") // FK en 'estudiante\_curso' que apunta a 'Curso'.
* )
* private Set<Curso> cursos = new HashSet<>();
* // Getters y Setters, y métodos de ayuda
* 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 Set<Curso> getCursos() { return cursos; }
* public void setCursos(Set<Curso> cursos) { this.cursos = cursos; }
* public void addCurso(Curso curso) {
* cursos.add(curso);
* curso.getEstudiantes().add(this); // Sincroniza el otro lado
* }
* public void removeCurso(Curso curso) {
* cursos.remove(curso);
* curso.getEstudiantes().remove(this);
* }
* }
* // Entidad Curso
* @Entity
* public class Curso {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombreCurso;
* @ManyToMany(mappedBy = "cursos", fetch = FetchType.LAZY) // 'mappedBy' apunta al campo 'cursos' en Estudiante.
* private Set<Estudiante> estudiantes = new HashSet<>();
* // Getters y Setters
* public Long getId() { return id; }
* public void setId(Long id) { this.id = id; }
* public String getNombreCurso() { return nombreCurso; }
* public void setNombreCurso(String nombreCurso) { this.nombreCurso = nombreCurso; }
* public Set<Estudiante> getEstudiantes() { return estudiantes; }
* public void setEstudiantes(Set<Estudiante> estudiantes) { this.estudiantes = estudiantes; }
* }

#### Caso Especial: Tabla Intermedia ManyToMany con Atributos Adicionales

Si la tabla intermedia de una relación Many-to-Many necesita almacenar **atributos adicionales** (que no son solo las FKs), entonces ya no puede ser gestionada automáticamente por @ManyToMany. En este escenario, la tabla intermedia debe modelarse como una **entidad JPA propia**.

**Ejemplo:**

Si un estudiante se inscribe en un curso, y queremos saber la fechaInscripcion o la calificacion para *esa*inscripción específica, estos son atributos de la relación, no de la entidad Estudiante o Curso por sí sola.

* **¿Cómo funciona?**
  1. Creas una **nueva entidad JPA** que represente la tabla intermedia (ej., InscripcionCurso).
  2. Esta nueva entidad tendrá:
     + Su propia clave primaria (puede ser compuesta, o una PK artificial).
     + Dos relaciones @ManyToOne (una a Estudiante y otra a Curso).
     + Los campos para los atributos adicionales (ej., fechaInscripcion, calificacion).
  3. Las entidades Estudiante y Curso ya no tienen un @ManyToMany directo. En su lugar, tendrán una relación @OneToMany con la nueva entidad intermedia.

**Ejemplo (con atributos en la tabla intermedia):**

Java

* // Entidad Estudiante
* @Entity
* public class Estudiante {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombre;
* @OneToMany(mappedBy = "estudiante", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
* private Set<InscripcionCurso> inscripciones = new HashSet<>();
* // Getters y Setters, y métodos de ayuda
* 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 Set<InscripcionCurso> getInscripciones() { return inscripciones; }
* public void setInscripciones(Set<InscripcionCurso> inscripciones) { this.inscripciones = inscripciones; }
* public void addInscripcion(InscripcionCurso inscripcion) {
* inscripciones.add(inscripcion);
* inscripcion.setEstudiante(this);
* }
* public void removeInscripcion(InscripcionCurso inscripcion) {
* inscripciones.remove(inscripcion);
* inscripcion.setEstudiante(null);
* }
* }
* // Entidad Curso
* @Entity
* public class Curso {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id;
* private String nombreCurso;
* @OneToMany(mappedBy = "curso", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
* private Set<InscripcionCurso> inscripciones = new HashSet<>();
* // Getters y Setters
* public Long getId() { return id; }
* public void setId(Long id) { this.id = id; }
* public String getNombreCurso() { return nombreCurso; }
* public void setNombreCurso(String nombreCurso) { this.nombreCurso = nombreCurso; }
* public Set<InscripcionCurso> getInscripciones() { return inscripciones; }
* public void setInscripciones(Set<InscripcionCurso> inscripciones) { this.inscripciones = inscripciones; }
* public void addInscripcion(InscripcionCurso inscripcion) {
* inscripciones.add(inscripcion);
* inscripcion.setCurso(this);
* }
* public void removeInscripcion(InscripcionCurso inscripcion) {
* inscripciones.remove(inscripcion);
* inscripcion.setCurso(null);
* }
* }
* // Nueva Entidad Intermedia: InscripcionCurso
* @Entity
* public class InscripcionCurso {
* @Id
* @GeneratedValue(strategy = GenerationType.IDENTITY)
* private Long id; // Puede ser una PK simple o compuesta (con @EmbeddedId)
* @ManyToOne(fetch = FetchType.LAZY)
* @JoinColumn(name = "estudiante\_id") // FK al estudiante
* private Estudiante estudiante;
* @ManyToOne(fetch = FetchType.LAZY)
* @JoinColumn(name = "curso\_id") // FK al curso
* private Curso curso;
* private LocalDate fechaInscripcion; // Atributo adicional
* private Double calificacion; // Otro atributo adicional
* // Getters y Setters
* public Long getId() { return id; }
* public void setId(Long id) { this.id = id; }
* public Estudiante getEstudiante() { return estudiante; }
* public void setEstudiante(Estudiante estudiante) { this.estudiante = estudiante; }
* public Curso getCurso() { return curso; }
* public void setCurso(Curso curso) { this.curso = curso; }
* public LocalDate getFechaInscripcion() { return fechaInscripcion; }
* public void setFechaInscripcion(LocalDate fechaInscripcion) { this.fechaInscripcion = fechaInscripcion; }
* public Double getCalificacion() { return calificacion; }
* public void setCalificacion(Double calificacion) { this.calificacion = calificacion; }
* }

### Puntos Clave para Recordar:

* **@JoinColumn**: Siempre se usa en el lado que *posee* la clave foránea en la base de datos (o la que se considera propietaria de la relación).
* **mappedBy**: Se usa en el lado "inverso" de una relación bidireccional, indicando que la propiedad está mapeada por el campo en la otra entidad.
* **FetchType.LAZY vs EAGER**: LAZY es la estrategia por defecto para colecciones (@OneToMany, @ManyToMany) y se recomienda para OneToOne y ManyToOne en la mayoría de los casos para optimizar el rendimiento. EAGER carga la relación inmediatamente.
* **CascadeType**: Útil para propagar operaciones (persistencia, actualización, eliminación) a las entidades relacionadas. ALL es potente, úsalo con precaución.
* **Bidireccionalidad**: Para mantener la consistencia del modelo de objetos en memoria, es fundamental que, al modificar un lado de una relación bidireccional, también se actualice el otro lado (ej., usando métodos add/remove o setters).