



Mapeo de entidades

1. Introducción 🌐

📌 Anotaciones de Persistencia

Ejemplo básico:

2. Modos de Acceso a una Entidad 🔧

🔑 Tipos de Acceso

2.1 Acceso por Atributo

🚫 Características:

Ejemplo:

2.2 Acceso por Propiedad

🐚 Características:

Ejemplo:

2.3 Acceso Mixto

🌱 Características:

Ejemplo:

3. Mapeo a una Tabla concreta: @Table 📄

3.1. Nombre de la tabla

👤 Consejo

3.2. Esquemas

👤 Consejo

3.3. Catálogos

3.4. Nombres sensibles a mayúsculas de tablas y columnas

Ejemplos de nombres equivalentes en bases de datos estándar:

Ejemplo con identificadores delimitados:

4. Mapeo de Tipos Simples 🌍

- [4.1. Tipos primitivos de Java](#)
- [4.2. Clases envolventes de tipos primitivos](#)
- [4.3. Arrays de bytes y caracteres](#)
- [4.4. Tipos numéricos grandes](#)
- [4.5. Cadenas](#)
- [4.6. Tipos temporales de Java](#)
- [4.7. Tipos temporales de JDBC](#)
- [4.8. Tipos enumerados](#)
- [4.9. Objetos serializables](#)

 **Nota**

5. Mapeo de columnas: `@Column`

 Elementos de la anotación `@Column` :

Ejemplo de uso:

 **Nota**

6. Carga Perezosa (Lazy Fetching)

 ¿Qué es la carga perezosa?

 Ventajas de la carga perezosa

 Configuración

Ejemplo de Implementación

 Caso de Uso:

 Consideraciones importantes

7. Objetos Grandes (LOBs)

 ¿Qué son los LOBs?

 Configuración de `@Lob`

Ejemplo con CLOB y BLOB

 Uso de LOBs con carga perezosa

 Ventajas y desventajas 

 **Nota:**

8. Tipos Enumerados (enum): `@Enumerated`

8.1 Mapeo Ordinal de Enumerados

 Ejemplo de Enumerado y Entidad con Mapeo Ordinal

 Problemas con el Mapeo Ordinal

8.2 Mapeo de Enumerados como Cadenas

Ejemplo de Mapeo de Enumerado como Cadena

 Ventajas y Desventajas del Mapeo como Cadena

8.3 Uso de Métodos `@PostLoad` y `@PrePersist`

Ejemplo de Uso con Atributos Transitorios

8.4 Uso de Conversores Personalizados con `@Converter`

 Implementación de un Conversor Personalizado

 Ventajas del Uso de Conversores

8.5 Uso de Enumerados en JPQL

Ejemplo de Consulta JPQL con Enumerados

Conclusión

9. Tipos Temporales: `@Temporal`

Tipos Temporales Admitidos

Uso de `@Temporal`

Valores de `TemporalType`

Ejemplo de Uso

Deprecación en JPA 3.2 y Superior

Resumen Visual: Elección de Tipos Temporales

10. Atributos Transitorios: `@Transient`

Cómo Hacer un Campo Transitorio

Razones para Usar Campos Transitorios

Ejemplo de Campo Transitorio

Diferencia entre `transient` y `@Transient`

10. Mapeo de Clave Primaria: `@Id`

10.1 Sobrescritura de la Clave Primaria

10.2 Tipos de Claves Primarias Admitidos

10.3 Generación de Claves Primarias: `@GeneratedValue`

Diagrama Visual de Estrategias

10.3.1 Generación Automática: `GenerationType.AUTO`

10.3.2 Generación con Tablas: `GenerationType.TABLE`

10.3.3 Generación con Secuencia: `GenerationType.SEQUENCE`

10.3.4 Generación con Identidad: `GenerationType.IDENTITY`



10.3.5 Generación con UUID: `GenerationType.UUID`

1. Introducción



El mapeo objeto-relacional (ORM) es la base de JPA, permitiendo conectar el mundo de los objetos en Java con las bases de datos relacionales.

Esto incluye:

-  **Correspondencia de atributos del objeto con columnas de la base de datos.**
-  **Ejecución de consultas entre objetos.**




El mapeo de objetos se realiza a través de **anotaciones** que van precedidas por el símbolo `@`

Anotaciones de Persistencia



Las anotaciones en JPA proporcionan una manera sencilla y flexible de definir el comportamiento de persistencia.

- **Características clave de las anotaciones:**

-  **Ubicación:** Se colocan en clases, métodos o atributos.
-  **Sintaxis:** Antes de la definición de la clase, atributo o método correspondiente.
-  **Flexibilidad:** Las anotaciones pueden aplicarse a atributos o métodos dependiendo de tus preferencias.

Tipo de Anotación	Descripción	Ejemplos
Lógicas	Modelan entidades desde la perspectiva del modelado de objetos.	<code>@Entity</code> , <code>@Id</code> , <code>@ManyToOne</code>
Físicas	Se enfocan en el modelo de datos de la base de datos	<code>@Table</code> , <code>@Column</code>

Ejemplo básico:

```
@Entity
public class Producto {
    @Id
    private Long id;
    private String nombre;
    private double precio;
}
```

Nota: También puedes definir metadatos equivalentes en XML, pero las anotaciones son el enfoque preferido para aplicaciones modernas.

2. Modos de Acceso a una Entidad

El modo de acceso indica cómo el proveedor de persistencia interactúa con el estado de las entidades.

Tipos de Acceso

1. **Acceso por atributo:** Usando reflexión en los atributos directamente.
2. **Acceso por propiedad:** A través de métodos `getter` y `setter`.
3. **Acceso mixto:** Combina ambos enfoques según las necesidades.

2.1 Acceso por Atributo

Las anotaciones se colocan directamente sobre los atributos.



Todos los **atributos deben declararse como** `protected` o `private`. **Se prohíben los atributos** `public`.

Características:

- **Anotaciones en los atributos:** Las anotaciones, como `@Id`, se colocan directamente sobre los atributos.
- Los métodos `getter` y `setter` pueden estar presentes, pero no son utilizados por el proveedor.
- Los atributos deben tener visibilidad **private**, **protected** o **de paquete (default)**, pero no **public**.
- Este método es preferido cuando se desea mayor control sobre los datos o cuando no se quiere exponer directamente el estado persistente.

Ejemplo:

```
@Entity
public class Employee {
    @Id
    private Long id;    // Anotación directamente en el atributo.
    private String name; // Nombre persistente, mapeado por defecto.
    private long salary; // Salario persistente, mapeado por defecto.
```

```

public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public long getSalary() { return salary; }
public void setSalary(long salary) { this.salary = salary; }
}

```

Ventajas	Desventajas
🚀 Simplicidad.	Menor control sobre getters y setters.
🕶️ Fácil de leer.	No se pueden aplicar validaciones en los métodos.

2.2 Acceso por Propiedad

Las anotaciones se colocan en los métodos `getter`.



Cuando se utiliza el modo de acceso por propiedad **debe haber métodos `getter` y `setter` para las propiedades persistentes.**



Características:

- El **tipo de propiedad** se determina por el **tipo devuelto del método `getter`** y **debe ser el mismo que el tipo del único parámetro pasado al método `setter`.**
- Ambos métodos **deben tener visibilidad** `public` o `protected`
- Útil cuando se necesita lógica adicional al acceder o modificar los datos.

Ejemplo:

```

@Entity
public class Employee {
    private long id;    // No anotado directamente, usado solo internamente.
    private String name; // Nombre persistente accesible por getter y setter.
    private long wage;  // Respalda la propiedad `salary`.
}

```

@Id

```
public long getId() { return id; } // Anotación en el getter.
```

```
public void setId(long id) { this.id = id; }
```

```
public String getName() { return name; }
```

```
public void setName(String name) { this.name = name; }
```

```
public long getSalary() { return wage; } // Getter y setter conectan con `wage`.
```

```
public void setSalary(long salary) { this.wage = salary; }  
}
```

Ventajas	Desventajas
 Mayor control sobre los datos.	 Mayor complejidad.
 Permite validaciones.	

2.3 Acceso Mixto

El acceso mixto combina los modos de acceso **por atributo** y **por propiedad** dentro de una misma entidad o en una jerarquía de entidades.

Características:

- El modo predeterminado se especifica a nivel de clase con la anotación `@Access`.
- Se puede anular el modo de acceso predeterminado en atributos o propiedades individuales.
- Ideal para casos donde se requiere una **transformación de datos** o cuando se heredan entidades con diferentes requisitos de acceso.
- Permite usar diferentes enfoques dentro de una misma entidad.

Ejemplo:

Una entidad que usa acceso por atributo como predeterminado, pero incluye una propiedad con acceso por getter para realizar una transformación:

```

@Entity
@Access(AccessType.FIELD) // Acceso predeterminado por atributo.
public class Employee {
    @Id
    private long id;    // Acceso directo a través del atributo.
    @Transient
    private String phoneNum; // No se persiste directamente.

    public static final String LOCAL_AREA_CODE = "613";

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getPhoneNumber() { return phoneNum; }
    public void setPhoneNumber(String num) { this.phoneNum = num; }

    @Access(AccessType.PROPERTY)
    @Column(name = "PHONE")
    protected String getPhoneNumberForDb() {
        if (phoneNum.length() == 10)
            return phoneNum; // Sin transformación.
        else
            return LOCAL_AREA_CODE + phoneNum; // Añade código de área.
    }

    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3); // Elimina el código de área.
        else
            phoneNum = num;
    }
}

```

3. Mapeo a una Tabla concreta: **@Table**



La anotación `@Table` en JPA permite configurar cómo una entidad Java se asocia con una tabla específica en la base de datos. Aunque el mapeo básico solo requiere `@Entity` e `@Id`, `@Table` ofrece más control sobre los nombres de tablas, esquemas y catálogos.

3.1. Nombre de la tabla

Por defecto, el nombre de la tabla será el nombre de la clase de la entidad sin cualificar. Sin embargo, puedes personalizarlo usando el atributo `name` de `@Table`.

```
@Entity
@Table(name = "EMP")
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

- **Predeterminado:** Si no se especifica, el nombre de la tabla será `Employee`.
- **Personalizado:** Con `@Table(name = "EMP")`, el nombre será `EMP`.

Consejo

Los nombres predeterminados no se especifican como sensibles a mayúsculas o minúsculas. Sin embargo, algunas bases de datos, como MySQL y H2, manejan esta característica de forma diferente:

- **MySQL:** Insensible por defecto.
- **H2:** Sensible a mayúsculas/minúsculas, salvo que se configure `IGNORECASE=TRUE` en la URL de conexión.

```
jdbc:h2:~/test;IGNORECASE=TRUE
```

3.2. Esquemas

Puedes especificar el esquema en el atributo `schema` de `@Table`. Esto permite organizar tablas dentro de diferentes contextos lógicos en la base de datos.

```
@Entity
@Table(name = "EMP", schema = "HR")
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

- En este caso, el proveedor de persistencia buscará la tabla como `HR.EMP`.

Consejo

Algunos proveedores permiten incluir el esquema directamente en el atributo `name` como `@Table(name = "HR.EMP")`. Sin embargo, esto **no es estándar** y puede limitar la portabilidad de la aplicación.

3.3. Catálogos

Para bases de datos que admiten catálogos, el atributo `catalog` en `@Table` permite especificarlos.

```
@Entity
@Table(name = "EMP", catalog = "HR")
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

- Aquí, la tabla `EMP` estará contenida en el catálogo `HR`.

3.4. Nombres sensibles a mayúsculas de tablas y columnas

El estándar SQL establece que los identificadores de base de datos no delimitados no son sensibles a mayúsculas o minúsculas. Sin embargo, **si es**

necesario distinguir entre identificadores, deben delimitarse con comillas dobles (").

Ejemplos de nombres equivalentes en bases de datos estándar:

```
@Table(name = "autor")
@Table(name = "Autor")
@Table(name = "AUTOR")
```

- Todos los nombres anteriores hacen referencia a la misma tabla.

Ejemplo con identificadores delimitados:

En una base de datos sensible a mayúsculas, las siguientes anotaciones **se referirían a tablas distintas:**

```
@Table(name = "\"Autor\"")
@Table(name = "\"AUTOR\"")
```

4. Mapeo de Tipos Simples



Los tipos simples de Java se pueden mapear directamente a columnas en la base de datos. A continuación, se enumeran los tipos compatibles que se pueden usar en las entidades JPA:

4.1. Tipos primitivos de Java

Estos se asignan automáticamente a tipos compatibles en la base de datos:

- **Números enteros y decimales:** `byte` , `short` , `int` , `long` , `float` , `double` .
- **Otros:** `char` , `boolean` .

4.2. Clases envolventes de tipos primitivos

Estas clases proporcionan un mapeo similar a sus contrapartes primitivas:

- **Enteros y decimales:** `Byte` , `Short` , `Integer` , `Long` , `Float` , `Double` .
 - **Otros:** `Character` , `Boolean` .
-

4.3. Arrays de bytes y caracteres

Se utilizan para almacenar datos como blobs o secuencias de caracteres:

- `byte[]` , `Byte[]` , `char[]` , `Character[]` .
-

4.4. Tipos numéricos grandes

Para datos numéricos que necesitan una precisión adicional:

- `java.math.BigInteger` .
 - `java.math.BigDecimal` .
-

4.5. Cadenas

Para almacenar textos, puedes usar:

- `java.lang.String` .
-

4.6. Tipos temporales de Java

JPA permite el uso de una variedad de tipos temporales para fechas y horas:

- **Java Util:**
 - `java.util.Date` .
 - `java.util.Calendar` .
- **API de Java 8 (`java.time`):**
 - `java.time.LocalDate` .
 - `java.time.LocalDateTime` .
 - `java.time.LocalDateTime` .
 - `java.time.OffsetTime` .
 - `java.time.OffsetDateTime` .

 Nota: Para tipos como `java.time.Instant`, se necesita un `AttributeConverter`.

4.7. Tipos temporales de JDBC

Compatibles con las funciones específicas de JDBC:

- `java.sql.Date` .
- `java.sql.Time` .
- `java.sql.Timestamp` .

4.8. Tipos enumerados

Cualquier tipo enumerado definido por el sistema o por el usuario.

4.9. Objetos serializables

Puedes mapear cualquier tipo que sea serializable en Java.

Nota

Si el tipo de Java no coincide directamente con el tipo JDBC, algunos proveedores de persistencia pueden realizar conversiones automáticas. Si esto no es posible, será necesario implementar un `AttributeConverter`.

5. Mapeo de columnas: `@Column`

Aunque los campos básicos son persistentes por defecto, la anotación `@Column` permite definir atributos específicos de las columnas asociadas.

Elementos de la anotación `@Column` :

Elemento	Descripción	Valor Predeterminado
<code>name</code>	Nombre de la columna en la base de datos.	Nombre del atributo.

length	Longitud máxima de cadenas o arrays de caracteres.	255
unique	Define si los valores deben ser únicos.	false
nullable	Permite valores nulos en la columna.	true
insertable	Si la columna se incluye en operaciones de inserción.	true
updatable	Si la columna se incluye en operaciones de actualización.	true
precision	Precisión numérica para valores decimales.	0
scale	Escala numérica para valores decimales.	0
columnDefinition	Definición SQL directa para la columna (menos portable).	Ninguna.

Ejemplo de uso:

```

java
CopiarEditar
@Entity
public class Employee {
    @Id
    @Column(name = "EMP_ID", nullable = false, unique = true)
    private long id;

    @Column(name = "NAME", length = 100, nullable = false)
    private String name;

    @Column(name = "SALARY", precision = 10, scale = 2)
    private BigDecimal salary;

    @Column(name = "COMMENTS", columnDefinition = "TEXT")
    private String comments;
}

```

Nota

La combinación de `@Column` y `@Table` permite un control total sobre el mapeo entre las clases y la base de datos, garantizando una mayor flexibilidad y portabilidad.

6. Carga Perezosa (Lazy Fetching) 🦥

? ¿Qué es la carga perezosa?

La carga perezosa es una estrategia de recuperación de datos utilizada por JPA (Java Persistence API) para optimizar el rendimiento de las consultas a la base de datos.



Cuando se utiliza la carga perezosa, los datos de un atributo o relación no se recuperan de inmediato al cargar la entidad, sino que se obtienen solo cuando el atributo es accedido por el código.

✅ Ventajas de la carga perezosa

1. **Eficiencia:** Reduce la cantidad de datos cargados desde la base de datos, lo que disminuye el consumo de memoria y mejora el rendimiento general.
2. **Control de datos cargados:** Permite cargar solo los datos estrictamente necesarios en cada momento.
3. **Aptitud para relaciones complejas:** En relaciones uno a muchos o muchos a muchos, evita cargar listas o colecciones innecesarias.



Configuración

Para habilitar la carga perezosa, se utiliza la anotación `@Basic` con el atributo `fetch`:

```
@Basic(fetch = FetchType.LAZY)
```

Esto también es aplicable en asociaciones con `@OneToMany`, `@ManyToOne`, `@OneToOne`, y `@ManyToMany` mediante la configuración de `fetch = FetchType.LAZY`.

Por defecto:

- Relaciones **@OneToOne** y **@ManyToOne**: `FetchType.EAGER` .
- Relaciones **@OneToMany** y **@ManyToMany**: `FetchType.LAZY` .
- Entidades: `FetchType.EAGER` .

Ejemplo de Implementación

Entidad `Employee` con carga perezosa en un atributo grande:

```
@Entity
public class Employee {

    @Id
    private Long id;

    @Column(name = "NAME")
    private String name;

    @Basic(fetch = FetchType.LAZY)
    @Lob
    @Column(name = "COMMENTS")
    private String comments;

    // Getters y setters
}
```

Caso de Uso:

1. Si se accede solo al nombre, el campo `comments` no se cargará.
2. Cuando se acceda al atributo `comments` , se realizará una consulta adicional a la base de datos para cargar su contenido.

```
Employee emp = entityManager.find(Employee.class, 1L);
System.out.println(emp.getName()); // Solo se carga el nombre.
System.out.println(emp.getComments()); // En este momento, se realiza la c
onsulta adicional.
```


🤔 Consideraciones importantes

Carga ansiosa (EAGER):

Aunque puede ser conveniente para simplificar el acceso a datos, puede causar problemas de rendimiento al cargar datos innecesarios.

LazyInitializationException:

Al trabajar con carga perezosa en entornos fuera de un contexto de persistencia (como vistas web), asegúrate de inicializar los datos antes de cerrar la sesión de Hibernate.

7. Objetos Grandes (LOBs)

🐾 ¿Qué son los LOBs?



En JPA, los **LOBs** (Large Objects) son campos utilizados para almacenar grandes cantidades de datos binarios o de texto. Estos datos pueden ser:

- **CLOB** (Character Large Object): Diseñado para almacenar grandes cantidades de texto.
- **BLOB** (Binary Large Object): Diseñado para almacenar datos binarios, como imágenes o videos.

🔧 Configuración de @Lob

La anotación `@Lob` se utiliza para marcar un atributo como LOB en JPA. Dependiendo del tipo de dato Java, se mapeará automáticamente como CLOB o BLOB:

- **CLOB**: Usado para `String`, `char[]`, `Character[]`.
- **BLOB**: Usado para `byte[]`, `Byte[]`, o cualquier clase que implemente `Serializable`.

Ejemplo con CLOB y BLOB

Entidad `Documento` con un archivo binario e información textual:

```
@Entity
public class Documento {

    @Id
    private Long id;

    @Lob
    private String contenido; // CLOB

    @Lob
    private byte[] archivo; // BLOB

    // Getters y setters
}
```

🧐 Uso de LOBs con carga perezosa

Cuando los LOBs son datos grandes que no siempre se necesitan, combínalos con `FetchType.LAZY`:

```
@Basic(fetch = FetchType.LAZY)
@Lob
private byte[] archivo;
```

✅ Ventajas y desventajas ❌

- **Ventajas:**
 - Capacidad para almacenar datos muy grandes.
 - Integración directa con bases de datos modernas.
- **Desventajas:**
 - Impacto en el rendimiento si no se configuran correctamente (por ejemplo, sin carga perezosa).

 **Nota:**

Consejo: Los LOB son útiles para almacenar datos grandes, pero no se deben abusar de ellos. Los LOB pueden ser ineficientes para recuperar y almacenar. Siempre que sea posible, se deben evitar los LOB. Si se necesita almacenar datos grandes, se debe considerar el uso de un sistema de archivos o un sistema de almacenamiento de objetos.

8. Tipos Enumerados (enum): **@Enumerated**

```
public enum EmployeeType {  
    FULL_TIME_EMPLOYEE, // ordinal 0  
    PART_TIME_EMPLOYEE, // ordinal 1  
    CONTRACT_EMPLOYEE // ordinal 2  
}
```

8.1 Mapeo Ordinal de Enumerados

Los valores de un tipo enumerado en Java tienen una **asignación ordinal** implícita que se **determina por el orden en que se declararon**.



El **ordinal** se usa de modo predeterminado para representar y almacenar los valores del tipo enumerado en la base de datos.

El proveedor asumirá que la **columna de la base de datos es de tipo entero**.



Ejemplo de Enumerado y Entidad con Mapeo Ordinal

```
@Entity  
public class Employee {  
    @Id  
    private long id;
```

```
@Enumerated(EnumType.ORDINAL) // Mapeo ordinal
private EmployeeType type;
// ...
}
```

🚫 Problemas con el Mapeo Ordinal



Si se introduce un nuevo valor en medio del enumerado o se reorganizan las constantes, el mapeo ordinal puede provocar inconsistencias en los datos ya almacenados. Por ejemplo, agregar `PART_TIME_BENEFITS_EMPLOYEE` después de `PART_TIME_EMPLOYEE` cambiaría los valores ordinales de los elementos siguientes, causando errores en los registros existentes.

8.2 Mapeo de Enumerados como Cadenas

Para evitar los problemas asociados al mapeo ordinal, se puede utilizar la anotación `@Enumerated` con el valor `EnumType.STRING`. Esto almacena los nombres de las constantes en la base de datos como cadenas, lo que hace que el mapeo sea más robusto frente a cambios en el enumerado.

Ejemplo de Mapeo de Enumerado como Cadena

```
@Entity
public class Employee {
    @Id
    private long id;

    @Enumerated(EnumType.STRING) // Mapeo como cadena
    private EmployeeType type;
    // ...
}
```

👤 Ventajas y Desventajas del Mapeo como Cadena

-  **Ventajas:**

- Evita problemas de consistencia al modificar el orden de las constantes o al agregar nuevas.
- Hace que los datos en la base de datos sean más legibles.
- **❌ Desventajas:**
 - Requiere más espacio de almacenamiento que el mapeo ordinal.
 - Es más sensible a cambios en los nombres de las constantes.

8.3 Uso de Métodos @PostLoad y @PrePersist



Para obtener un mayor control sobre cómo se mapean y persisten los valores enumerados, se pueden emplear los métodos de ciclo de vida

`@PostLoad` y `@PrePersist`.

- `@PostLoad` : se invoca después de que se cargue una entidad de la base de datos. `PostLoad`.
- `@PrePersist` : se invoca antes de que se persista una entidad en la base de datos. `PrePersist`.

Ejemplo de Uso con Atributos Transitorios

En este ejemplo, un enumerado `Prioridad` utiliza un valor entero para representar su estado, mientras que el código de negocio trabaja con la instancia del enumerado.

```
public enum Prioridad {  
    BAJA(100), MEDIA(200), ALTA(300);  
  
    private int valor;  
  
    private Prioridad(int valor) {  
        this.valor = valor;  
    }  
  
    public int getValor() {  
        return valor;  
    }  
}
```

```

public static Prioridad of(int valor) {
    if (codigo!=null||!codigo.equalsIgnoreCase(" ")){
        for (Prioridad p: Prioridad.values()){
            if (p.getValor().equalsIgnoreCase(valor)) return c;
        }
    }
    return null;
}

@Entity
public class Artículo {
    @Id
    private long id;

    @Basic
    private int valorPrioridad; // Mapeado en la base de datos

    @Transient
    private Prioridad prioridad; // Usado en el negocio

    @PostLoad
    private void cargarPrioridad() {
        this.prioridad = Prioridad.of(valorPrioridad);
    }

    @PrePersist
    private void persistirPrioridad() {
        this.valorPrioridad = prioridad.getValor();
    }
}

```

8.4 Uso de Conversores Personalizados con @Converter

Desde JPA 2.1, es posible definir conversores personalizados implementando la interfaz `AttributeConverter`. Esto proporciona una solución flexible y reutilizable para

mapear enumerados.

Implementación de un Conversor Personalizado

```
public enum Categoria {
    DEPORTE("D"), MUSICA("M"), TECNOLOGIA("T");

    private String codigo;

    private Categoria(String codigo) {
        this.codigo = codigo;
    }

    public String getCodigo() {
        return codigo;
    }

    public static Categoria fromCodigo(String codigo) {
        if (codigo != null || !codigo.equalsIgnoreCase(" ")) {
            for (Categoria c: Categoria.values()) {
                if (c.getCodigo().equalsIgnoreCase(codigo)) return c;
            }
        }
        return null;
    }
}

@Converter(autoApply = true) //No es una buena práctica
public class CategoriaConverter implements AttributeConverter<Categoria, String> {

    @Override
    public String convertToDatabaseColumn(Categoria categoria) {
        return (categoria != null) ? categoria.getCodigo() : null;
    }

    @Override
    public Categoria convertToEntityAttribute(String codigo) {
        return (codigo != null) ? Categoria.fromCodigo(codigo) : null;
    }
}
```

```

}

@Entity
public class Artículo {

    // ...

    @Convert(converter = ConvertidorCategoria.class) //Forma correcta de regi
    private Categoria categoria;
}

```

El uso de `autoApply = true` permite aplicar automáticamente este conversor a todos los atributos del tipo `Categoria`. (No es una buena práctica)

✓ Ventajas del Uso de Conversores

- Permite manejar enumerados con estructuras complejas o datos adicionales.
- Proporciona una mayor flexibilidad y control sobre el mapeo.
- Facilita la integración de enumerados en aplicaciones con reglas de negocio específicas.

8.5 Uso de Enumerados en JPQL

Los enumerados pueden utilizarse directamente en consultas JPQL. En este caso, es posible emplear el valor completo del enumerado o parámetros con nombre para mayor flexibilidad.

Ejemplo de Consulta JPQL con Enumerados

```

// Consulta estática
String jpql = "SELECT a FROM Artículo a WHERE a.categoria = com.exempl
e.Categoria.DEPORTE";
List<Articulo> articulos = em.createQuery(jpql, Articulo.class).getResultList
();

```



```
// Consulta dinámica
String jpql = "SELECT a FROM Artículo a WHERE a.categoria = :categoria";
TypedQuery<Artículo> query = em.createQuery(jpql, Artículo.class);
query.setParameter("categoria", Categoria.TECNOLOGIA);
List<Artículo> articulos = query.getResultList();
```

Usar parámetros con nombre es preferible, ya que mejora la legibilidad y evita dependencias de nombres de clases y constantes.

Conclusión

Caso de Uso	Tipo de Mapeo	Justificación
Enumerados simples con pocos valores que no cambiarán en el tiempo	<code>@Enumerated(EnumType.ORDINAL)</code>	Uso eficiente de almacenamiento (ocupa menos espacio al usar números). Útil si el orden y los valores del enumerado no cambiarán jamás.
Enumerados con posibles modificaciones en el orden o la cantidad de valores	<code>@Enumerated(EnumType.STRING)</code>	Evita problemas de consistencia al modificar el orden o agregar nuevos valores. Además, mejora la legibilidad en la base de datos al almacenar cadenas en lugar de números.
Enumerados con valores complejos o datos adicionales	Conversores personalizados (<code>@Converter</code>)	Proporciona flexibilidad para mapear enumerados con lógica de negocio específica o estructuras complejas. Permite personalizar completamente cómo se almacenan y recuperan los valores.

Caso de Uso	Tipo de Mapeo	Justificación
Enumerados con valores transitorios o calculados	Métodos <code>@PostLoad</code> y <code>@PrePersist</code>	Útil cuando se necesita transformar un valor intermedio (como un número o una cadena) a un objeto enumerado antes de usarlo, o viceversa, para desacoplar la lógica interna del modelo y el almacenamiento en la base de datos.
Consultas JPQL que incluyen enumerados	Dependiendo del caso: <code>EnumType.STRING</code> o <code>@Converter</code>	Si las consultas JPQL se basan en enumerados, el mapeo como cadena (<code>EnumType.STRING</code>) es preferible para evitar problemas de compatibilidad. Los conversores también son útiles si el enumerado incluye lógica personalizada.

9. Tipos Temporales: `@Temporal`



Los **tipos temporales** en JPA son utilizados para trabajar con atributos basados en fechas y tiempos.

Tipos Temporales Admitidos

Categoría	Clases Admitidas
Java SQL	<code>java.sql.Date</code> , <code>java.sql.Time</code> , <code>java.sql.Timestamp</code>
Java Util	<code>java.util.Date</code> , <code>java.util.Calendar</code>
Java Time (Java 8)	<code>java.time</code> API

Uso de `@Temporal`

- La anotación `@Temporal` es necesaria para especificar cómo mapear las fechas y tiempos en los tipos de **Java Util** (`Date` y `Calendar`) hacia **tipos JDBC**.

🌟 Valores de TemporalType

`@Temporal` usa la enumeración `TemporalType` para especificar el mapeo:

Valor	Descripción
<code>DATE</code>	Mapea solo la fecha (sin tiempo).
<code>TIME</code>	Mapea solo la hora (sin fecha).
<code>TIMESTAMP</code>	Mapea fecha y hora (similar a <code>java.sql.Timestamp</code>).

📖 Ejemplo de Uso

```
@Entity
public class Employee {
    @Id
    private long id;

    @Temporal(TemporalType.DATE) // Solo almacena la fecha
    private Calendar dob;

    @Temporal(TemporalType.DATE) // También almacena fecha
    @Column(name="S_DATE")
    private Date startDate;

    // ...
}
```

Nota 📝: En este ejemplo:

- `dob` almacena la fecha de nacimiento.
- `startDate` almacena la fecha de inicio con un nombre de columna personalizado.

🚩 Deprecación en JPA 3.2 y Superior

La anotación `@Temporal` está **desaprobada** a partir de **JPA 3.2**.

Se recomienda usar los tipos de **Java Time API** (`java.time.LocalDate` , `java.time.LocalDateTime` , etc.) para trabajar con fechas.

Si es necesario, usa `@Convert` con un conversor de atributos.

Resumen Visual: Elección de Tipos Temporales

```
java.util.Date ↔ @Temporal  
java.util.Calendar ↔ @Temporal  
java.time (Java 8 y superior) ↔ SIN @Temporal
```

10. Atributos Transitorios: `@Transient`



Los atributos transitorios son aquellos **que no deben persistir en la base de datos**. Son útiles para almacenar información temporal o de cálculo.

Cómo Hacer un Campo Transitorio

1. Usa el **modificador** `transient` de Java.
 - No persiste en la base de datos ni durante la serialización.
2. Usa la **anotación** `@Transient`.
 - Se conserva durante la serialización, pero no se guarda en la base de datos.



Razones para Usar Campos Transitorios

- **Evitar persistencia duplicada:** Cuando mezclamos accesos por campos y getters.
- **Almacenamiento temporal:** Para guardar estados calculados o datos en caché.
- **Separación de lógica:** Manejar valores dependientes del entorno (como el idioma).



Ejemplo de Campo Transitorio

```

@Entity
public class Employee {
    @Id
    private long id;

    private String name;
    @Transient //No persiste solo en BD
    private long salary;

    transient private String translatedName; // No persistente ni en BD ni en serializaci3n

    public String toString() {
        if (translatedName == null) {
            translatedName = ResourceBundle.getBundle("EmpResources").getStri
        }
        return translatedName + ": " + id + " " + name;
    }
}

```

🔑 Diferencia entre `transient` y `@Transient`

Modificador/Anotaci3n	Persistente	Serializable	Uso Com3n
<code>transient</code> (Java)	✗	✗	Para datos temporales en memoria.
<code>@Transient</code> (JPA)	✗	✓	Para datos que no deben guardarse en la base de datos.

10. Mapeo de Clave Primaria: `@Id`

📌 Definici3n

- Toda entidad debe tener una clave primaria asignada a una tabla.
- La anotaci3n `@Id` indica el identificador de la entidad.

💡 Nota:

Si el identificador está compuesto por un único atributo, se denomina **identificador simple**.

10.1 Sobrescritura de la Clave Primaria

Uso de @Column

- Se puede sobrescribir el nombre de la columna con la anotación `@Column`.
- Restricciones:
 - Claves primarias **no nulas** y **no actualizables**.
 - Si varios campos usan la misma columna, define `insertable=false`.

10.2 Tipos de Claves Primarias Admitidos

Tabla de Tipos Permitidos

Categoría	Ejemplos
Primitivos	<code>byte</code> , <code>int</code> , <code>short</code> , <code>long</code> , <code>char</code>
Clases Envolventes	<code>Byte</code> , <code>Integer</code> , <code>Short</code> , <code>Long</code> , <code>Character</code>
Texto	<code>java.lang.String</code>
Numéricos Grandes	<code>java.math.BigInteger</code> , <code>java.math.BigDecimal</code>
Tipos Temporales	<code>java.util.Date</code> , <code>java.sql.Date</code> , <code>java.util.Calendar</code> , <code>java.time.LocalDate</code> , etc.

Advertencia:

- Evitar `float` y `double` debido a problemas de precisión y confiabilidad.

10.3 Generación de Claves Primarias: @GeneratedValue

Definición

- `@GeneratedValue` permite que el proveedor de persistencia genere automáticamente los identificadores.
- Estrategias disponibles:

Estrategia	Descripción
AUTO	Estrategia predeterminada, seleccionada por el proveedor.
IDENTITY	Usa una columna de identidad de la base de datos.
SEQUENCE	Usa una secuencia de base de datos.
TABLE	Usa una tabla para garantizar unicidad en los IDs.
UUID	Genera un identificador único universal basado en <code>java.util.UUID</code> .

Ejemplo básico:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
```

Diagrama Visual de Estrategias

```
@GENERATEDVALUE
|
|-- AUTO
|   |-- Elige estrategia según la base de datos
|   |-- Ejemplo: prototipos/desarrollo
|
|-- IDENTITY
|   |-- Utiliza una columna de identidad de base de datos
|   |-- No se puede compartir entre entidades
|   |-- ID no disponible hasta la inserción
|
|-- SEQUENCE
|   |-- Utiliza una secuencia de base de datos
|   |-- Altamente eficiente y adaptable
|   |-- Permite compartir entre entidades
|   |-- Asignación en bloques para mejorar rendimiento
|
|-- TABLE
|   |-- Utiliza una tabla para almacenar identificadores
|   |-- Alta portabilidad entre bases de datos
```

```
| |-- Admite múltiples generadores en una tabla
| |-- Define generadores con @TableGenerator
|
|-- UUID
| |-- Genera identificadores únicos universales
| |-- Atributo debe ser de tipo java.util.UUID
```

10.3.1 Generación Automática: GenerationType.AUTO

Características

- Estrategia flexible para desarrollo/prototipos.
- Ejemplo:

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
}
```

10.3.2 Generación con Tablas: GenerationType.TABLE

Esquema de Tabla

Columna	Descripción
GEN_NAME	Identifica el generador específico
GEN_VALUE	Almacena el último valor asignado

Ejemplo:

```
CREATE TABLE ID_GEN (
    GEN_NAME VARCHAR(80),
    GEN_VALUE INTEGER,
    PRIMARY KEY (GEN_NAME)
);
```

Código Ejemplo


```
@TableGenerator(name = "Emp_Gen", table = "ID_GEN", pkColumnName = "GEN_NAME", valueColumnName = "GEN_VALUE")
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "Emp_Gen")
private long id;
```

10.3.3 Generación con Secuencia: GenerationType.SEQUENCE

Ventajas de las Secuencias

- Eficientes y rápidas.
- Pueden asignar bloques de IDs para minimizar consultas a la base de datos.

Ejemplo de Código

```
@SequenceGenerator(name = "Emp_Gen", sequenceName = "Emp_Seq")
@Id
@GeneratedValue(generator = "Emp_Gen")
private long id;
```

Definición SQL de la Secuencia

```
CREATE SEQUENCE Emp_Seq
MINVALUE 1
START WITH 1
INCREMENT BY 50;
```

10.3.4 Generación con Identidad: GenerationType.IDENTITY

Características

- Usa columnas autonuméricas como claves primarias.
- El ID no está disponible hasta que se realiza la inserción.

Ejemplo de Uso

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private long id;
```

10.3.5 Generación con UUID: GenerationType.UUID

Definición

- Genera un identificador único universal (UUID).
- Tipo de atributo requerido: `java.util.UUID`.

Ejemplo de Código

```
@Id  
@GeneratedValue(strategy = GenerationType.UUID)  
private UUID id;
```