Hibernate: Framework ORM para Java

Contenido

Módulo 0486 - Acceso a Datos | 2º DAM

1. Introducción

1.1. ¿Qué problema resuelve Hibernate?

Cuando trabajamos con bases de datos relacionales desde Java, nos encontramos con un problema fundamental: **la diferencia entre el paradigma orientado a objetos y el paradigma relacional**.

En Java trabajamos con:

- Objetos con atributos y métodos
- Herencia entre clases
- Colecciones (List, Set, Map)
- Referencias entre objetos

En bases de datos relacionales trabajamos con:

- Tablas con filas y columnas
- Claves primarias y foráneas
- Relaciones mediante JOINs
- Tipos de datos SQL

Esta diferencia se conoce como **«Impedance Mismatch»** (desajuste de impedancia) y hace que el código JDBC tradicional sea tedioso y propenso a errores.

1.2. Ejemplo del problema: JDBC Tradicional

Veamos cuánto código necesitamos para una operación simple con JDBC puro:

```java
// JDBC TRADICIONAL - Obtener todos los productos
public List<Producto> findAll() {
    List<Producto> productos = new ArrayList<>();
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        // 1. Obtener conexión
        conn = DriverManager.getConnection(URL, USER, PASS);

        // 2. Crear sentencia SQL
        String sql = "SELECT id, nombre, descripcion, precio, stock, categoria FROM
        stmt = conn.prepareStatement(sql);

        // 3. Ejecutar consulta
        rs = stmt.executeQuery();

        // 4. Mapear resultados MANUALMENTE
        while (rs.next()) {
            Producto p = new Producto();
            p.setId(rs.getLong("id"));
            p.setNombre(rs.getString("nombre"));
            p.setDescripcion(rs.getString("descripcion"));
            p.setPrecio(rs.getBigDecimal("precio"));
            p.setStock(rs.getInt("stock"));
            p.setCategoria(rs.getString("categoria"));
            productos.add(p);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        // 5. Cerrar recursos (¡obligatorio!)
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (conn != null) conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return productos;
}
```

Problemas de este enfoque:

- Mucho código repetitivo (boilerplate)
- Gestión manual de conexiones
- Mapeo manual ResultSet → Objeto
- Propenso a errores (olvidar cerrar recursos)
- SQL mezclado con código Java

- Difícil de mantener

2. ¿Qué es Hibernate?

2.1. Definición

Hibernate es un framework de **mapeo objeto-relacional (ORM)** para Java que permite:

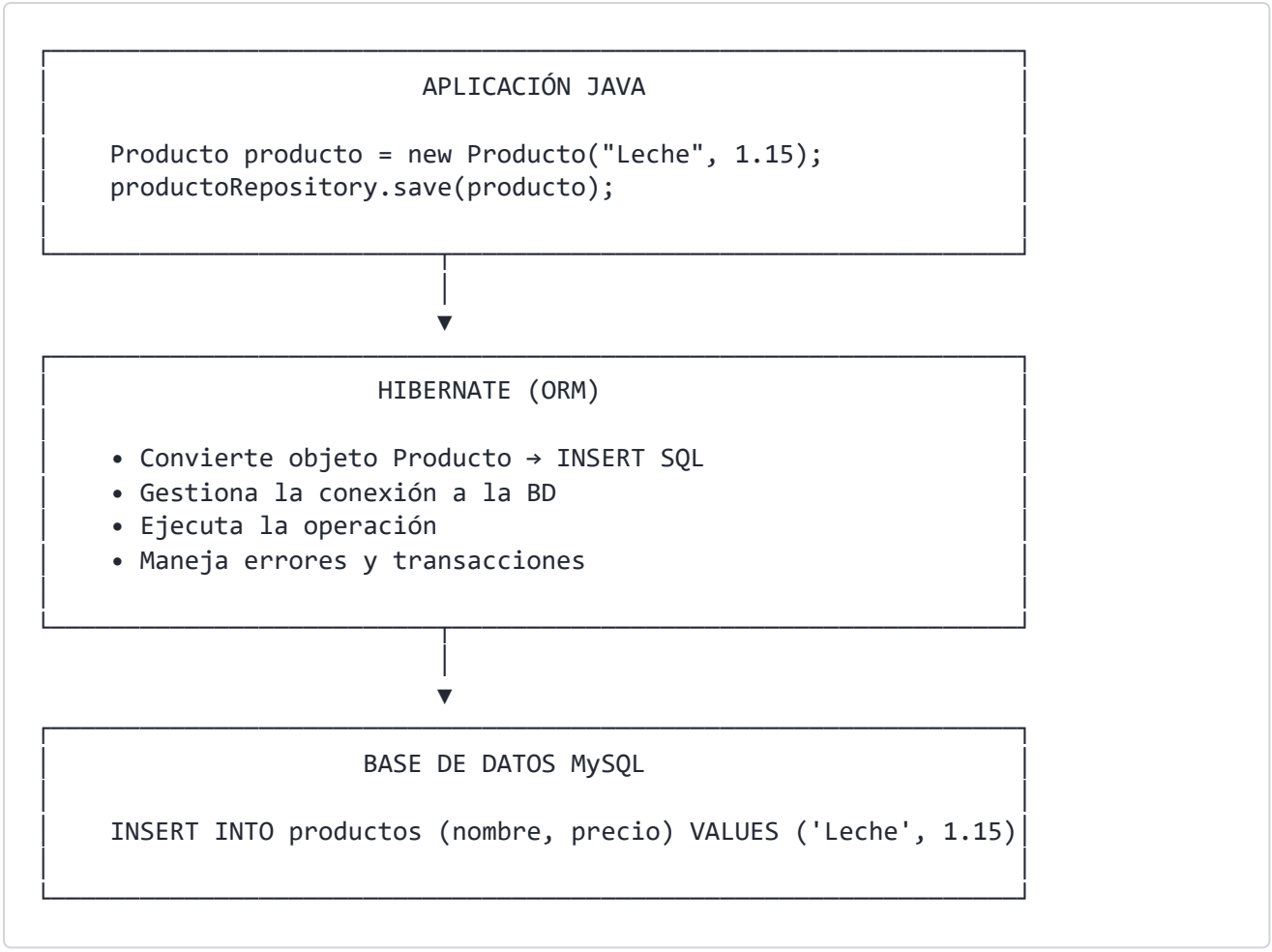
- Mapear clases Java a tablas de base de datos
- Convertir automáticamente entre objetos y registros
- Generar SQL automáticamente
- Gestionar conexiones y transacciones
- Cachear datos para mejorar rendimiento

> **Nota**
>
> Hibernate implementa la especificación **JPA (Java Persistence API)**, que es el estándar de Java para persistencia. Esto significa que puedes usar anotaciones estándar de JPA y Hibernate las interpreta.

2.2. ¿Qué es un ORM?

ORM significa **Object-Relational Mapping** (Mapeo Objeto-Relacional).

Es una técnica que permite convertir datos entre el sistema de tipos de un lenguaje orientado a objetos y una base de datos relacional.



2.3. Ventajas de usar Hibernate

Ventaja	Descripción
Productividad	Menos código que escribir y mantener
Portabilidad	Cambiar de BD sin modificar código Java
Mantenibilidad	Código más limpio y organizado
Rendimiento	Caché de primer y segundo nivel
Seguridad	Previene SQL Injection automáticamente
Transacciones	Gestión automática de transacciones

2.4. Hibernate vs JDBC: Comparación de código

Con JDBC (40+ líneas):

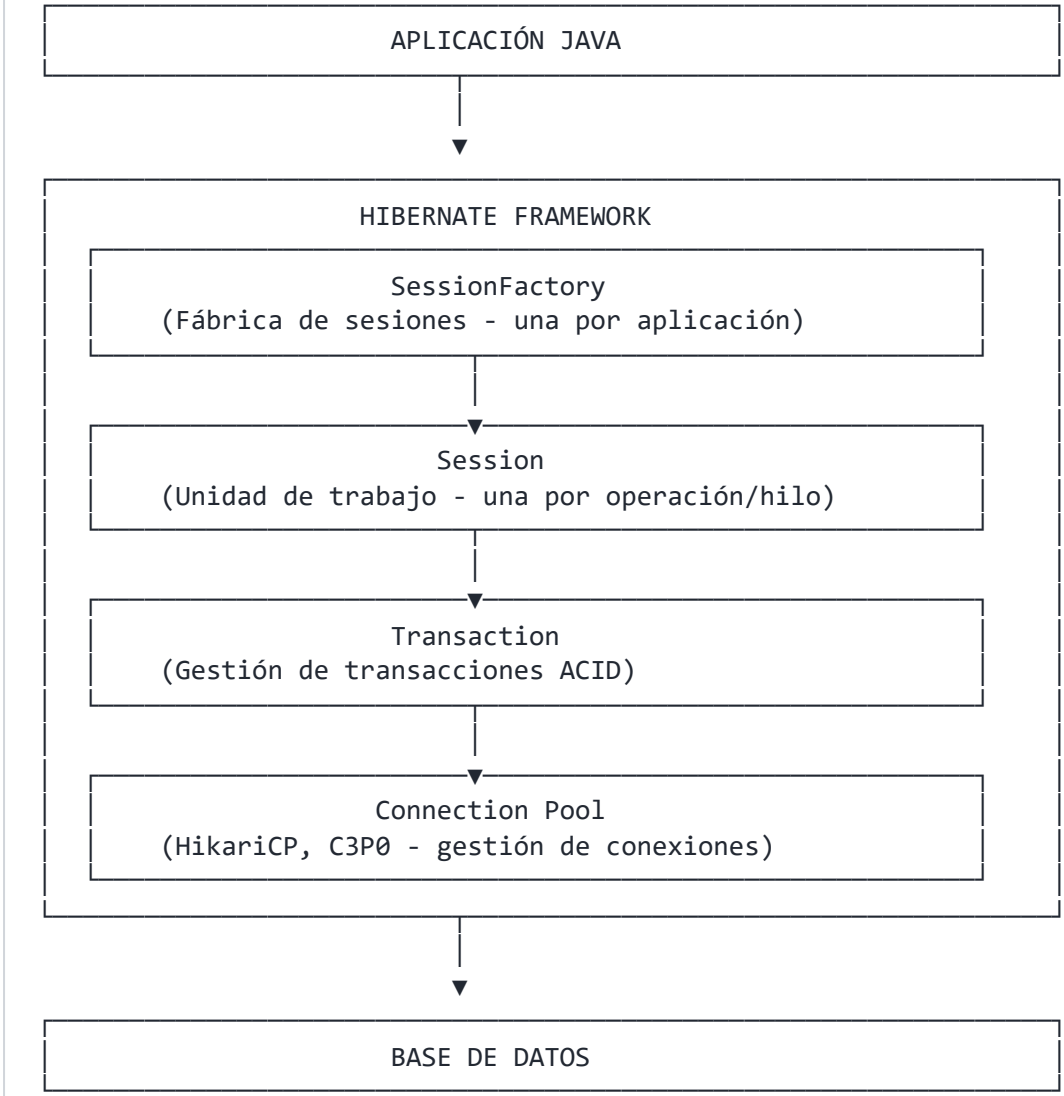
```java
public void guardarProducto(Producto p) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        String sql = "INSERT INTO productos (nombre, precio) VALUES (?, ?)";
        stmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        stmt.setString(1, p.getNombre());
        stmt.setBigDecimal(2, p.getPrecio());
        stmt.executeUpdate();
        ResultSet rs = stmt.getGeneratedKeys();
        if (rs.next()) {
            p.setId(rs.getLong(1));
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        // ... cerrar recursos
    }
}
```

Con Hibernate (1 línea):

```java
public void guardarProducto(Producto p) {
    entityManager.persist(p);
}
```

3. Arquitectura de Hibernate

3.1. Componentes principales



3.2. Conceptos clave

SessionFactory

- Se crea **una sola vez** al iniciar la aplicación
- Es **thread-safe** (seguro para múltiples hilos)
- Costoso de crear, por eso se reutiliza
- Contiene la configuración y los mapeos

Session

- Se crea para **cada unidad de trabajo**
- **NO es thread-safe** (un hilo = una sesión)
- Representa una conversación con la BD
- Gestiona los objetos persistentes

Transaction

- Agrupa operaciones que deben ejecutarse juntas
- Sigue las propiedades **ACID**:
  - **A**tomicidad: Todo o nada
  - **C**onsistencia: BD siempre válida
  - **I**solation: Transacciones aisladas
  - **D**urabilidad: Cambios permanentes

4. Configuración de Hibernate

4.1. Dependencias Maven

Para usar Hibernate en un proyecto Spring Boot, necesitamos estas dependencias:

```
<!-- pom.xml -->
<dependencies>
    <!-- Spring Boot Starter Data JPA (incluye Hibernate) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Driver de MySQL -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Importante

La dependencia `spring-boot-starter-data-jpa` incluye automáticamente:

- Hibernate ORM
- Spring Data JPA
- HikariCP (pool de conexiones)
- API de transacciones

4.2. Configuración en application.properties

```
# ═══════════════════════════════════════════════════════════
# CONFIGURACIÓN DE HIBERNATE EN SPRING BOOT
# ═══════════════════════════════════════════════════════════

# ─────────────────────────────────────────────────────────
# Conexión a la Base de Datos
# ─────────────────────────────────────────────────────────
spring.datasource.url=jdbc:mysql://localhost:3306/mi_base_datos
spring.datasource.username=root
spring.datasource.password=mi_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# ─────────────────────────────────────────────────────────
# Configuración de Hibernate
# ─────────────────────────────────────────────────────────

# Mostrar las consultas SQL generadas (útil para depurar)
spring.jpa.show-sql=true

# Formatear el SQL para mejor legibilidad
spring.jpa.properties.hibernate.format_sql=true

# Estrategia de generación del esquema
# Valores posibles:
#   none     - No hace nada (recomendado en producción)
#   validate - Valida que el esquema coincida con las entidades
#   update   - Actualiza el esquema automáticamente
#   create   - Crea el esquema (borra datos existentes)
#   create-drop - Crea al inicio, elimina al cerrar
spring.jpa.hibernate.ddl-auto=update

# ─────────────────────────────────────────────────────────
# Pool de Conexiones (HikariCP)
# ─────────────────────────────────────────────────────────
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.connection-timeout=30000
```

4.3. Opciones de ddl-auto explicadas

Valor	Comportamiento	Uso recomendado
`none`	No modifica el esquema	**Producción**
`validate`	Solo verifica que coincida	**Producción**
`update`	Añade columnas/tablas nuevas	**Desarrollo**
`create`	Crea tablas (borra datos)	**Testing**
`create-drop`	Crea y elimina al cerrar	**Testing**

> ⚠ Advertencia
>
> **NUNCA uses `create` o `create-drop` en producción.** Perderías todos los datos de la base de datos cada vez que se reinicie la aplicación.

5. Anotaciones JPA/Hibernate

5.1. Anotaciones de Entidad

Las anotaciones son la forma de indicar a Hibernate cómo mapear nuestras clases Java a tablas de la base de datos.

`@Entity`

Marca una clase como **entidad persistente**. Hibernate la mapeará a una tabla.

```
import jakarta.persistence.Entity;

@Entity  // Esta clase se mapeará a una tabla
public class Producto {
    // ...
}
```

`@Table`

Especifica el nombre de la tabla en la base de datos. Si no se indica, usa el nombre de la clase.

```
import jakarta.persistence.Entity;
import jakarta.persistence.Table;

@Entity
@Table(name = "productos")  // Nombre de la tabla en la BD
public class Producto {
    // ...
}
```

Atributos de `@Table`:

Atributo	Descripción	Ejemplo
`name`	Nombre de la tabla	`@Table(name = "productos")`
`schema`	Esquema de la BD	`@Table(schema = "ventas")`
`catalog`	Catálogo de la BD	`@Table(catalog = "tienda")`

5.2. Anotaciones de Clave Primaria

`@Id`

Marca un campo como **clave primaria** de la tabla.

```
import jakarta.persistence.Id;

@Entity
public class Producto {

    @Id  // Este campo es la clave primaria
    private Long id;
}
```

`@GeneratedValue`

Indica cómo se genera automáticamente el valor de la clave primaria.

```java
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;

@Entity
public class Producto {

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

Estrategias de generación:

Estrategia	Descripción	Base de datos
`IDENTITY`	Auto-increment de la BD	MySQL, SQL Server
`SEQUENCE`	Secuencia de la BD	Oracle, PostgreSQL
`TABLE`	Tabla auxiliar para IDs	Todas
`AUTO`	Hibernate elige	Todas

```java
// Ejemplo con IDENTITY (MySQL)
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

// Ejemplo con SEQUENCE (Oracle/PostgreSQL)
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "producto_seq")
@SequenceGenerator(name = "producto_seq", sequenceName = "PRODUCTO_SEQ")
private Long id;
```

5.3. Anotaciones de Columnas

`@Column`

Configura el mapeo de un atributo a una columna.

```java
import jakarta.persistence.Column;

@Entity
@Table(name = "productos")
public class Producto {

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

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

    @Column(name = "descripcion", length = 500)
    private String descripcion;

    @Column(name = "precio", nullable = false, precision = 10, scale = 2)
    private BigDecimal precio;

    @Column(name = "stock", columnDefinition = "INT DEFAULT 0")
    private Integer stock;

    @Column(name = "activo")
    private Boolean activo;
}
```

Atributos de `@Column`:

Atributo	Descripción	Valor por defecto
`name`	Nombre de la columna	Nombre del atributo
`nullable`	¿Permite NULL?	`true`
`length`	Longitud (para String)	255
`precision`	Dígitos totales (decimal)	0
`scale`	Dígitos decimales	0
`unique`	¿Valor único?	`false`
`insertable`	¿Se incluye en INSERT?	`true`
`updatable`	¿Se incluye en UPDATE?	`true`
`columnDefinition`	SQL personalizado	-

5.4. Anotaciones Temporales

`@Temporal` (Java 7 y anteriores)

```
import jakarta.persistence.Temporal;
import jakarta.persistence.TemporalType;
import java.util.Date;

@Entity
public class Pedido {

    @Temporal(TemporalType.DATE)      // Solo fecha: 2024-12-08
    private Date fechaPedido;

    @Temporal(TemporalType.TIME)      // Solo hora: 14:30:00
    private Date horaPedido;

    @Temporal(TemporalType.TIMESTAMP) // Fecha y hora: 2024-12-08 14:30:00
    private Date fechaHoraCompleta;
}
```

Java 8+ (LocalDate, LocalDateTime)

Con Java 8 o superior, es preferible usar las clases del paquete `java.time`:

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

@Entity
public class Pedido {

    // No necesita @Temporal - Hibernate lo detecta automáticamente
    private LocalDate fechaPedido;        // DATE
    private LocalTime horaPedido;         // TIME
    private LocalDateTime fechaCreacion;  // DATETIME/TIMESTAMP
}
```

5.5. Anotaciones de Enumerados

`@Enumerated`

Mapea un enum de Java a una columna de la BD.

```java
// Definición del enum
public enum EstadoPedido {
    PENDIENTE,
    PROCESANDO,
    ENVIADO,
    ENTREGADO,
    CANCELADO
}

@Entity
public class Pedido {

    // Guarda el ORDINAL (0, 1, 2, 3, 4)
    @Enumerated(EnumType.ORDINAL)
    private EstadoPedido estado;

    // Guarda el STRING ("PENDIENTE", "PROCESANDO", etc.)
    @Enumerated(EnumType.STRING)
    private EstadoPedido estadoTexto;
}
```

Truco

Recomendación: Usa siempre `EnumType.STRING`. Si usas `ORDINAL` y añades un nuevo valor al enum en medio, se desincronizarán los datos existentes.

5.6. Campos Transitorios

`@Transient`

Indica que un campo **NO debe persistirse** en la base de datos.

```java
@Entity
public class Producto {

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

    private String nombre;
    private BigDecimal precio;
    private Integer stock;

    // Este campo NO se guarda en la BD
    // Es un cálculo que hacemos en memoria
    @Transient
    private BigDecimal precioConIva;

    public BigDecimal getPrecioConIva() {
        if (precio != null) {
            return precio.multiply(new BigDecimal("1.21"));
        }
        return BigDecimal.ZERO;
    }
}
```

6. Entidad Completa: Ejemplo Detallado

6.1. Estructura de una entidad JPA

Veamos un ejemplo completo de una entidad bien documentada:

```java
package com.dam.tienda.model;

import jakarta.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDateTime;

/**
 * ═══════════════════════════════════════════════════════════════
 * ENTIDAD: Producto
 * ═══════════════════════════════════════════════════════════════
 * Representa un producto en el sistema de inventario.
 *
 * Tabla en BD: productos
 *
 * Mapeo de columnas:
 *
 * ┌──────────────────┬──────────────────┬──────────────────────┐
 * │ Atributo Java    │ Columna BD       │ Tipo SQL             │
 * ├──────────────────┼──────────────────┼──────────────────────┤
 * │ id               │ id               │ BIGINT AUTO_INCREMENT │
 * │ codigo           │ codigo           │ VARCHAR(20) UNIQUE   │
 * │ nombre           │ nombre           │ VARCHAR(100) NOT NULL │
 * │ descripcion      │ descripcion      │ TEXT                 │
 * │ precio           │ precio           │ DECIMAL(10,2) NOT NULL│
 * │ stock            │ stock            │ INT DEFAULT 0        │
 * │ categoria        │ categoria        │ VARCHAR(50)          │
 * │ activo           │ activo           │ BOOLEAN DEFAULT TRUE │
 * │ fechaCreacion    │ fecha_creacion   │ DATETIME             │
 * │ fechaModificacion│ fecha_modificacion│ DATETIME            │
 * └──────────────────┴──────────────────┴──────────────────────┘
 *
 * @author Departamento de Informática
 * @version 1.0
 */
@Entity
@Table(
    name = "productos",
    indexes = {
        @Index(name = "idx_producto_codigo", columnList = "codigo"),
        @Index(name = "idx_producto_categoria", columnList = "categoria")
    }
)
public class Producto {

    // ═══════════════════════════════════════════════════════════
    // CLAVE PRIMARIA
    // ═══════════════════════════════════════════════════════════

    /**
     * Identificador único del producto.
     * Generado automáticamente por MySQL (AUTO_INCREMENT).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    // ═══════════════════════════════════════════════════════════
    // ATRIBUTOS DE NEGOCIO
```

```
    // ═══════════════════════════════════════════════════════════

    /**
     * Código único del producto (SKU).
     * Ejemplo: "PROD-001", "LAC-0023"
     */
    @Column(name = "codigo", length = 20, unique = true, nullable = false)
    private String codigo;

    /**
     * Nombre comercial del producto.
     */
    @Column(name = "nombre", length = 100, nullable = false)
    private String nombre;

    /**
     * Descripción detallada del producto.
     * Usa TEXT en MySQL para permitir textos largos.
     */
    @Column(name = "descripcion", columnDefinition = "TEXT")
    private String descripcion;

    /**
     * Precio de venta al público (sin IVA).
     * Usamos BigDecimal para precisión en cálculos monetarios.
     */
    @Column(name = "precio", nullable = false, precision = 10, scale = 2)
    private BigDecimal precio;

    /**
     * Cantidad disponible en almacén.
     */
    @Column(name = "stock", columnDefinition = "INT DEFAULT 0")
    private Integer stock = 0;

    /**
     * Categoría del producto.
     * Ejemplos: "Electrónica", "Hogar", "Alimentación"
     */
    @Column(name = "categoria", length = 50)
    private String categoria;

    /**
     * Indica si el producto está activo (visible en catálogo).
     */
    @Column(name = "activo", columnDefinition = "BOOLEAN DEFAULT TRUE")
    private Boolean activo = true;

    // ═══════════════════════════════════════════════════════════
    // CAMPOS DE AUDITORÍA
    // ═══════════════════════════════════════════════════════════

    /**
     * Fecha y hora de creación del registro.
     */
    @Column(name = "fecha_creacion", updatable = false)
    private LocalDateTime fechaCreacion;

    /**
```

```
     * Fecha y hora de última modificación.
     */
    @Column(name = "fecha_modificacion")
    private LocalDateTime fechaModificacion;

    // ═══════════════════════════════════════════════════════════
    // CAMPOS TRANSITORIOS (no se persisten)
    // ═══════════════════════════════════════════════════════════

    /**
     * Precio con IVA incluido (21%).
     * Calculado en memoria, no se guarda en BD.
     */
    @Transient
    private BigDecimal precioConIva;

    // ═══════════════════════════════════════════════════════════
    // CALLBACKS DEL CICLO DE VIDA
    // ═══════════════════════════════════════════════════════════

    /**
     * Se ejecuta automáticamente ANTES de insertar en la BD.
     */
    @PrePersist
    protected void onCreate() {
        this.fechaCreacion = LocalDateTime.now();
        this.fechaModificacion = LocalDateTime.now();
    }

    /**
     * Se ejecuta automáticamente ANTES de actualizar en la BD.
     */
    @PreUpdate
    protected void onUpdate() {
        this.fechaModificacion = LocalDateTime.now();
    }

    // ═══════════════════════════════════════════════════════════
    // CONSTRUCTORES
    // ═══════════════════════════════════════════════════════════

    /**
     * Constructor vacío requerido por JPA/Hibernate.
     */
    public Producto() {
    }

    /**
     * Constructor con campos obligatorios.
     */
    public Producto(String codigo, String nombre, BigDecimal precio) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.precio = precio;
    }

    /**
     * Constructor completo.
     */
```

```
    public Producto(String codigo, String nombre, String descripcion,
                    BigDecimal precio, Integer stock, String categoria) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.descripcion = descripcion;
        this.precio = precio;
        this.stock = stock;
        this.categoria = categoria;
    }

    // ========================================================
    // GETTERS Y SETTERS
    // ========================================================

    public Long getId() {
        return id;
    }

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

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public String getNombre() {
        return nombre;
    }

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

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public BigDecimal getPrecio() {
        return precio;
    }

    public void setPrecio(BigDecimal precio) {
        this.precio = precio;
    }

    public Integer getStock() {
        return stock;
    }

    public void setStock(Integer stock) {
```

```
        this.stock = stock;
    }

    public String getCategoria() {
        return categoria;
    }

    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }

    public Boolean getActivo() {
        return activo;
    }

    public void setActivo(Boolean activo) {
        this.activo = activo;
    }

    public LocalDateTime getFechaCreacion() {
        return fechaCreacion;
    }

    public LocalDateTime getFechaModificacion() {
        return fechaModificacion;
    }

    /**
     * Calcula el precio con IVA (21%).
     */
    public BigDecimal getPrecioConIva() {
        if (precio != null) {
            return precio.multiply(new BigDecimal("1.21"))
                         .setScale(2, java.math.RoundingMode.HALF_UP);
        }
        return BigDecimal.ZERO;
    }

    // ═══════════════════════════════════════════════════════
    // MÉTODOS DE UTILIDAD
    // ═══════════════════════════════════════════════════════

    /**
     * Verifica si hay stock disponible.
     */
    public boolean hayStock() {
        return stock != null && stock > 0;
    }

    /**
     * Reduce el stock en la cantidad indicada.
     * @throws IllegalArgumentException si no hay suficiente stock
     */
    public void reducirStock(int cantidad) {
        if (stock == null || stock < cantidad) {
            throw new IllegalArgumentException("Stock insuficiente");
        }
        this.stock -= cantidad;
    }
```

```
    @Override
    public String toString() {
        return String.format(
            "Producto[id=%d, codigo='%s', nombre='%s', precio=%.2f€, stock=%d]",
            id, codigo, nombre, precio, stock
        );
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Producto producto = (Producto) o;
        return id != null && id.equals(producto.id);
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }
}
```

7. Relaciones entre Entidades

7.1. Tipos de relaciones

En bases de datos relacionales existen cuatro tipos de relaciones principales:

Relación	Descripción	Ejemplo
`@OneToOne`	Uno a uno	Usuario ↔ Perfil
`@OneToMany`	Uno a muchos	Categoría → Productos
`@ManyToOne`	Muchos a uno	Productos → Categoría
`@ManyToMany`	Muchos a muchos	Productos ↔ Etiquetas

7.2. Relación `@ManyToOne` / `@OneToMany`

La relación más común. Por ejemplo: **muchos productos pertenecen a una categoría**.

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Categoria (lado "uno")
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "categorias")
public class Categoria {

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

    @Column(name = "nombre", nullable = false, length = 50)
    private String nombre;

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

    // Relación inversa: Una categoría tiene MUCHOS productos
    // mappedBy indica que la relación está mapeada en el atributo "categoria" de P
    @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL, fetch = FetchType
    private List<Producto> productos = new ArrayList<>();

    // Constructor, getters y setters...
}
```

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Producto (lado "muchos")
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "productos")
public class Producto {

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

    @Column(name = "nombre", nullable = false)
    private String nombre;

    @Column(name = "precio", nullable = false)
    private BigDecimal precio;

    // Relación: Muchos productos pertenecen a UNA categoría
    // @JoinColumn especifica la columna de clave foránea
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "categoria_id", nullable = false)
    private Categoria categoria;

    // Constructor, getters y setters...
}
```

Tabla resultante en BD:

```sql
CREATE TABLE categorias (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    descripcion VARCHAR(255)
);

CREATE TABLE productos (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    precio DECIMAL(10,2) NOT NULL,
    categoria_id BIGINT NOT NULL,
    FOREIGN KEY (categoria_id) REFERENCES categorias(id)
);
```

7.3. Relación `@OneToOne`

Cuando dos entidades tienen una relación uno a uno. Por ejemplo: **un usuario tiene un perfil**.

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Usuario
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "usuarios")
public class Usuario {

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

    @Column(name = "email", nullable = false, unique = true)
    private String email;

    @Column(name = "password", nullable = false)
    private String password;

    // Relación uno a uno con Perfil
    // cascade = ALL: las operaciones se propagan al perfil
    // orphanRemoval = true: si se elimina el usuario, se elimina el perfil
    @OneToOne(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true
    private Perfil perfil;
}
```

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Perfil
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "perfiles")
public class Perfil {

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

    @Column(name = "nombre_completo")
    private String nombreCompleto;

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

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

    // Lado propietario de la relación
    @OneToOne
    @JoinColumn(name = "usuario_id", unique = true)
    private Usuario usuario;
}
```

7.4. Relación `@ManyToMany`

Cuando ambas entidades pueden tener múltiples instancias relacionadas. Por ejemplo: **un producto puede tener múltiples etiquetas y una etiqueta puede estar en múltiples productos**.

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Producto
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "productos")
public class Producto {

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

    private String nombre;

    // Relación muchos a muchos con Etiqueta
    // @JoinTable define la tabla intermedia
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "productos_etiquetas",  // Nombre de la tabla intermedia
        joinColumns = @JoinColumn(name = "producto_id"),
        inverseJoinColumns = @JoinColumn(name = "etiqueta_id")
    )
    private Set<Etiqueta> etiquetas = new HashSet<>();

    // Métodos de utilidad para mantener la sincronización
    public void addEtiqueta(Etiqueta etiqueta) {
        this.etiquetas.add(etiqueta);
        etiqueta.getProductos().add(this);
    }

    public void removeEtiqueta(Etiqueta etiqueta) {
        this.etiquetas.remove(etiqueta);
        etiqueta.getProductos().remove(this);
    }
}
```

```java
// ═══════════════════════════════════════════════════════════
// ENTIDAD: Etiqueta
// ═══════════════════════════════════════════════════════════
@Entity
@Table(name = "etiquetas")
public class Etiqueta {

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

    @Column(name = "nombre", unique = true)
    private String nombre;

    // Lado inverso de la relación
    @ManyToMany(mappedBy = "etiquetas")
    private Set<Producto> productos = new HashSet<>();
}
```

Tablas resultantes:

```sql
CREATE TABLE productos (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100)
);

CREATE TABLE etiquetas (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50) UNIQUE
);

-- Tabla intermedia (generada automáticamente)
CREATE TABLE productos_etiquetas (
    producto_id BIGINT,
    etiqueta_id BIGINT,
    PRIMARY KEY (producto_id, etiqueta_id),
    FOREIGN KEY (producto_id) REFERENCES productos(id),
    FOREIGN KEY (etiqueta_id) REFERENCES etiquetas(id)
);
```

7.5. Fetch Types: LAZY vs EAGER

Cuando cargamos una entidad, ¿debemos cargar también sus relaciones?

Tipo	Descripción	Cuándo se carga
`LAZY`	Carga diferida	Solo cuando se accede
`EAGER`	Carga inmediata	Siempre, junto con la entidad

```java
// LAZY (por defecto en @OneToMany y @ManyToMany)
// Los productos NO se cargan hasta que accedemos a ellos
@OneToMany(mappedBy = "categoria", fetch = FetchType.LAZY)
private List<Producto> productos;

// EAGER (por defecto en @ManyToOne y @OneToOne)
// La categoría se carga siempre junto con el producto
@ManyToOne(fetch = FetchType.EAGER)
private Categoria categoria;
```

Importante

Recomendación: Usa siempre `FetchType.LAZY` y carga los datos relacionados solo cuando los necesites. Esto evita problemas de rendimiento con muchos datos.

7.6. Cascade Types

Define qué operaciones se propagan a las entidades relacionadas.

Tipo	Descripción
`PERSIST`	Propaga la operación de guardado
`MERGE`	Propaga la operación de actualización
`REMOVE`	Propaga la operación de eliminación
`REFRESH`	Propaga la operación de recarga
`DETACH`	Propaga la operación de separación
`ALL`	Propaga TODAS las operaciones

```
// Si guardamos una categoría, también se guardan sus productos
@OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)
private List<Producto> productos;

// Solo propaga guardado y actualización
@OneToMany(mappedBy = "categoria", cascade = {CascadeType.PERSIST, CascadeType.MERG
private List<Producto> productos;
```

8. Repositorios con Spring Data JPA

8.1. Creación de un Repositorio

Spring Data JPA simplifica enormemente el acceso a datos. Solo necesitas crear una **interfaz** que extienda `JpaRepository`:

```java
package com.dam.tienda.repository;

import com.dam.tienda.model.Producto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

/**
 * Repositorio de Productos.
 *
 * Al extender JpaRepository<Producto, Long>:
 * - Producto: tipo de entidad
 * - Long: tipo de la clave primaria
 *
 * Heredamos automáticamente todos los métodos CRUD.
 */
@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {
    // ¡Los métodos CRUD ya están implementados!
}
```

8.2. Métodos heredados de JpaRepository

Sin escribir ningún código, tienes disponibles estos métodos:

```java
// ═══════════════════════════════════════════════════════════
// MÉTODOS CRUD HEREDADOS
// ═══════════════════════════════════════════════════════════

// Guardar/Actualizar
Producto save(Producto producto);
List<Producto> saveAll(List<Producto> productos);

// Buscar
Optional<Producto> findById(Long id);
List<Producto> findAll();
List<Producto> findAllById(List<Long> ids);

// Existencia
boolean existsById(Long id);
long count();

// Eliminar
void deleteById(Long id);
void delete(Producto producto);
void deleteAll();
void deleteAllById(List<Long> ids);

// Paginación y ordenación
List<Producto> findAll(Sort sort);
Page<Producto> findAll(Pageable pageable);
```

8.3. Query Methods: Consultas Automáticas

Spring Data JPA puede **generar consultas automáticamente** basándose en el nombre del método:

```java
@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {

    // ═══════════════════════════════════════════════════════
    // CONSULTAS GENERADAS AUTOMÁTICAMENTE POR EL NOMBRE DEL MÉTODO
    // ═══════════════════════════════════════════════════════

    // SELECT * FROM productos WHERE nombre = ?
    List<Producto> findByNombre(String nombre);

    // SELECT * FROM productos WHERE categoria = ?
    List<Producto> findByCategoria(String categoria);

    // SELECT * FROM productos WHERE precio < ?
    List<Producto> findByPrecioLessThan(BigDecimal precio);

    // SELECT * FROM productos WHERE precio > ?
    List<Producto> findByPrecioGreaterThan(BigDecimal precio);

    // SELECT * FROM productos WHERE precio BETWEEN ? AND ?
    List<Producto> findByPrecioBetween(BigDecimal min, BigDecimal max);

    // SELECT * FROM productos WHERE nombre LIKE '%?%' (ignorando mayúsculas)
    List<Producto> findByNombreContainingIgnoreCase(String nombre);

    // SELECT * FROM productos WHERE nombre LIKE '?%'
    List<Producto> findByNombreStartingWith(String prefijo);

    // SELECT * FROM productos WHERE nombre LIKE '%?'
    List<Producto> findByNombreEndingWith(String sufijo);

    // SELECT * FROM productos WHERE activo = true
    List<Producto> findByActivoTrue();

    // SELECT * FROM productos WHERE activo = false
    List<Producto> findByActivoFalse();

    // SELECT * FROM productos WHERE stock > 0 AND activo = true
    List<Producto> findByStockGreaterThanAndActivoTrue(Integer stock);

    // SELECT * FROM productos WHERE categoria = ? OR precio < ?
    List<Producto> findByCategoriaOrPrecioLessThan(String categoria, BigDecimal pre

    // SELECT * FROM productos WHERE categoria = ? ORDER BY precio ASC
    List<Producto> findByCategoriaOrderByPrecioAsc(String categoria);

    // SELECT * FROM productos WHERE categoria = ? ORDER BY precio DESC
    List<Producto> findByCategoriaOrderByPrecioDesc(String categoria);

    // SELECT COUNT(*) FROM productos WHERE categoria = ?
    long countByCategoria(String categoria);

    // SELECT * FROM productos WHERE categoria = ? (solo el primero)
    Optional<Producto> findFirstByCategoria(String categoria);

    // SELECT * FROM productos ORDER BY precio DESC LIMIT 5
    List<Producto> findTop5ByOrderByPrecioDesc();
```

```
    // DELETE FROM productos WHERE activo = false
    void deleteByActivoFalse();

    // SELECT CASE WHEN COUNT(*) > 0 THEN true ELSE false END FROM productos WHERE
    boolean existsByCodigo(String codigo);
}
```

8.4. Palabras clave para Query Methods

Palabra clave	SQL equivalente	Ejemplo
`And`	AND	`findByNombreAndCategoria`
`Or`	OR	`findByNombreOrCategoria`
`Between`	BETWEEN	`findByPrecioBetween`
`LessThan`	<	`findByPrecioLessThan`
`LessThanEqual`	<=	`findByPrecioLessThanEqual`
`GreaterThan`	>	`findByPrecioGreaterThan`
`GreaterThanEqual`	>=	`findByPrecioGreaterThanEqual`
`IsNull`	IS NULL	`findByDescripcionIsNull`
`IsNotNull`	IS NOT NULL	`findByDescripcionIsNotNull`
`Like`	LIKE	`findByNombreLike`
`NotLike`	NOT LIKE	`findByNombreNotLike`
`Containing`	LIKE %...%	`findByNombreContaining`
`StartingWith`	LIKE ...%	`findByNombreStartingWith`
`EndingWith`	LIKE %...	`findByNombreEndingWith`
`Not`	<>	`findByCategoriaNo`
`In`	IN	`findByCategoriaIn(List)`
`NotIn`	NOT IN	`findByCategoriaNotIn(List)`
`True`	= true	`findByActivoTrue`
`False`	= false	`findByActivoFalse`
`OrderBy`	ORDER BY	`findByOrderByPrecioDesc`
`IgnoreCase`	LOWER()	`findByNombreIgnoreCase`

Palabra clave	SQL equivalente	Ejemplo
`First` / `Top`	LIMIT 1	`findFirstByCategoria`
`Distinct`	DISTINCT	`findDistinctByCategoria`

8.5. Consultas personalizadas con `@Query`

Cuando los Query Methods no son suficientes, puedes escribir consultas JPQL o SQL nativo:

```java
@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {

    // ═══════════════════════════════════════════════════════
    // CONSULTAS JPQL (Java Persistence Query Language)
    // ═══════════════════════════════════════════════════════

    // JPQL usa nombres de ENTIDADES y ATRIBUTOS (no tablas y columnas)
    @Query("SELECT p FROM Producto p WHERE p.precio > :precio")
    List<Producto> buscarPorPrecioMayorQue(@Param("precio") BigDecimal precio);

    // Con múltiples parámetros
    @Query("SELECT p FROM Producto p WHERE p.categoria = :cat AND p.precio BETWEEN
    List<Producto> buscarPorCategoriaYRangoPrecio(
        @Param("cat") String categoria,
        @Param("min") BigDecimal precioMin,
        @Param("max") BigDecimal precioMax
    );

    // Consulta de agregación
    @Query("SELECT p.categoria, COUNT(p), AVG(p.precio) FROM Producto p GROUP BY p.
    List<Object[]> obtenerEstadisticasPorCategoria();

    // Obtener solo ciertos campos
    @Query("SELECT p.nombre, p.precio FROM Producto p WHERE p.activo = true")
    List<Object[]> obtenerNombreYPrecioActivos();

    // Con JOIN en relaciones
    @Query("SELECT p FROM Producto p JOIN p.categoria c WHERE c.nombre = :nombreCat
    List<Producto> buscarPorNombreCategoria(@Param("nombreCategoria") String nombre

    // Consultas de actualización
    @Modifying  // Necesario para UPDATE/DELETE
    @Query("UPDATE Producto p SET p.activo = false WHERE p.stock = 0")
    int desactivarProductosSinStock();

    @Modifying
    @Query("UPDATE Producto p SET p.precio = p.precio * :factor WHERE p.categoria =
    int actualizarPreciosPorCategoria(
        @Param("cat") String categoria,
        @Param("factor") BigDecimal factor
    );

    // ═══════════════════════════════════════════════════════
    // CONSULTAS SQL NATIVAS
    // ═══════════════════════════════════════════════════════

    // Cuando necesitas SQL específico de tu BD
    @Query(value = "SELECT * FROM productos WHERE MATCH(nombre, descripcion) AGAINS
           nativeQuery = true)
    List<Producto> busquedaFullText(@Param("texto") String texto);

    // Procedimiento almacenado (SQL nativo)
    @Query(value = "CALL actualizar_precios(:porcentaje)", nativeQuery = true)
    void ejecutarActualizacionPrecios(@Param("porcentaje") BigDecimal porcentaje);
}
```

9. Servicio: Capa de Lógica de Negocio

9.1. Estructura de un Servicio

El servicio contiene la lógica de negocio y actúa como intermediario entre el controlador y el repositorio:

```java
package com.dam.tienda.service;

import com.dam.tienda.model.Producto;
import com.dam.tienda.repository.ProductoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

/**
 * Servicio de Productos.
 *
 * @Service: Marca la clase como componente de servicio de Spring.
 *
 * Responsabilidades:
 * - Implementar la lógica de negocio
 * - Gestionar transacciones
 * - Validaciones de negocio
 * - Orquestar operaciones entre repositorios
 */
@Service
public class ProductoService {

    private final ProductoRepository productoRepository;

    /**
     * Inyección de dependencias por constructor (recomendado).
     */
    @Autowired
    public ProductoService(ProductoRepository productoRepository) {
        this.productoRepository = productoRepository;
    }

    // ========================================================================
    // OPERACIONES DE LECTURA
    // ========================================================================

    /**
     * Obtiene todos los productos.
     *
     * @Transactional(readOnly = true): Optimiza la transacción para lectura.
     */
    @Transactional(readOnly = true)
    public List<Producto> obtenerTodos() {
        return productoRepository.findAll();
    }

    /**
     * Busca un producto por ID.
     */
    @Transactional(readOnly = true)
    public Optional<Producto> obtenerPorId(Long id) {
        return productoRepository.findById(id);
    }
```

```java
    /**
     * Busca productos por categoría.
     */
    @Transactional(readOnly = true)
    public List<Producto> obtenerPorCategoria(String categoria) {
        return productoRepository.findByCategoria(categoria);
    }

    /**
     * Busca productos disponibles (con stock).
     */
    @Transactional(readOnly = true)
    public List<Producto> obtenerDisponibles() {
        return productoRepository.findByStockGreaterThanAndActivoTrue(0);
    }

    // ========================================================
    // OPERACIONES DE ESCRITURA
    // ========================================================

    /**
     * Guarda un nuevo producto o actualiza uno existente.
     *
     * @Transactional: Asegura que la operación sea atómica.
     */
    @Transactional
    public Producto guardar(Producto producto) {
        // Validaciones de negocio
        validarProducto(producto);

        return productoRepository.save(producto);
    }

    /**
     * Actualiza el stock de un producto.
     */
    @Transactional
    public Producto actualizarStock(Long id, Integer nuevoStock) {
        Producto producto = productoRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Producto no encontrado: " + id

        if (nuevoStock < 0) {
            throw new IllegalArgumentException("El stock no puede ser negativo");
        }

        producto.setStock(nuevoStock);
        return productoRepository.save(producto);
    }

    /**
     * Procesa una venta: reduce el stock.
     */
    @Transactional
    public Producto procesarVenta(Long productoId, Integer cantidad) {
        Producto producto = productoRepository.findById(productoId)
            .orElseThrow(() -> new RuntimeException("Producto no encontrado: " + pr

        if (producto.getStock() < cantidad) {
            throw new RuntimeException("Stock insuficiente. Disponible: " + product
```

```
        }

        producto.setStock(producto.getStock() - cantidad);
        return productoRepository.save(producto);
    }

    /**
     * Aplica un descuento a todos los productos de una categoría.
     */
    @Transactional
    public int aplicarDescuento(String categoria, BigDecimal porcentajeDescuento) {
        List<Producto> productos = productoRepository.findByCategoria(categoria);

        BigDecimal factor = BigDecimal.ONE.subtract(
            porcentajeDescuento.divide(new BigDecimal("100"))
        );

        for (Producto p : productos) {
            BigDecimal nuevoPrecio = p.getPrecio().multiply(factor)
                .setScale(2, java.math.RoundingMode.HALF_UP);
            p.setPrecio(nuevoPrecio);
        }

        productoRepository.saveAll(productos);
        return productos.size();
    }

    /**
     * Elimina un producto por ID.
     */
    @Transactional
    public void eliminar(Long id) {
        if (!productoRepository.existsById(id)) {
            throw new RuntimeException("Producto no encontrado: " + id);
        }
        productoRepository.deleteById(id);
    }

    // ═══════════════════════════════════════════════════════
    // MÉTODOS DE VALIDACIÓN PRIVADOS
    // ═══════════════════════════════════════════════════════

    private void validarProducto(Producto producto) {
        if (producto.getNombre() == null || producto.getNombre().trim().isEmpty())
            throw new IllegalArgumentException("El nombre es obligatorio");
        }

        if (producto.getPrecio() == null || producto.getPrecio().compareTo(BigDecim
            throw new IllegalArgumentException("El precio debe ser mayor que cero")
        }

        if (producto.getStock() != null && producto.getStock() < 0) {
            throw new IllegalArgumentException("El stock no puede ser negativo");
        }
    }
}
```

9.2. Gestión de Transacciones

La anotación `@Transactional` es fundamental para garantizar la integridad de los datos:

```
/**
 * ═══════════════════════════════════════════════════════════
 * @Transactional - Gestión de transacciones
 * ═══════════════════════════════════════════════════════════
 *
 * Propiedades principales:
 *
 * readOnly = true    → Optimiza para operaciones de solo lectura
 * rollbackFor        → Excepciones que provocan rollback
 * noRollbackFor      → Excepciones que NO provocan rollback
 * propagation        → Comportamiento con transacciones existentes
 * isolation          → Nivel de aislamiento
 * timeout            → Tiempo máximo en segundos
 */

// Transacción de solo lectura (más eficiente)
@Transactional(readOnly = true)
public List<Producto> obtenerTodos() {
    return productoRepository.findAll();
}

// Transacción de escritura (por defecto)
@Transactional
public Producto guardar(Producto producto) {
    return productoRepository.save(producto);
}

// Rollback en cualquier excepción
@Transactional(rollbackFor = Exception.class)
public void operacionCritica() {
    // Si ocurre cualquier excepción, se hace rollback
}

// Timeout de 30 segundos
@Transactional(timeout = 30)
public void operacionLarga() {
    // Si tarda más de 30 segundos, falla
}
```

10. Ejemplo Práctico Completo

10.1. Aplicación de ejemplo: Gestión de Biblioteca

Vamos a crear un ejemplo completo de una aplicación para gestionar una biblioteca con libros y autores.

Entidad Autor

```java
package com.dam.biblioteca.model;

import jakarta.persistence.*;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "autores")
public class Autor {

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

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

    @Column(name = "nacionalidad", length = 50)
    private String nacionalidad;

    @Column(name = "fecha_nacimiento")
    private LocalDate fechaNacimiento;

    // Un autor puede tener muchos libros
    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL, fetch = FetchType.LAZ
    private List<Libro> libros = new ArrayList<>();

    // Constructores
    public Autor() {}

    public Autor(String nombre, String nacionalidad) {
        this.nombre = nombre;
        this.nacionalidad = nacionalidad;
    }

    // Método de utilidad para añadir libros
    public void addLibro(Libro libro) {
        libros.add(libro);
        libro.setAutor(this);
    }

    // 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 String getNacionalidad() { return nacionalidad; }
    public void setNacionalidad(String nacionalidad) { this.nacionalidad = nacional
    public LocalDate getFechaNacimiento() { return fechaNacimiento; }
    public void setFechaNacimiento(LocalDate fechaNacimiento) { this.fechaNacimient
    public List<Libro> getLibros() { return libros; }
    public void setLibros(List<Libro> libros) { this.libros = libros; }

    @Override
    public String toString() {
```

```
        return String.format("Autor[id=%d, nombre='%s', nacionalidad='%s']", id, no
    }
}
```

Entidad Libro

```java
package com.dam.biblioteca.model;

import jakarta.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDate;

@Entity
@Table(name = "libros")
public class Libro {

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

    @Column(name = "isbn", unique = true, nullable = false, length = 20)
    private String isbn;

    @Column(name = "titulo", nullable = false, length = 200)
    private String titulo;

    @Column(name = "genero", length = 50)
    private String genero;

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

    @Column(name = "fecha_publicacion")
    private LocalDate fechaPublicacion;

    @Column(name = "disponible")
    private Boolean disponible = true;

    // Muchos libros pertenecen a un autor
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "autor_id", nullable = false)
    private Autor autor;

    // Constructores
    public Libro() {}

    public Libro(String isbn, String titulo, String genero, BigDecimal precio) {
        this.isbn = isbn;
        this.titulo = titulo;
        this.genero = genero;
        this.precio = precio;
    }

    // Getters y Setters...
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }
    public String getTitulo() { return titulo; }
    public void setTitulo(String titulo) { this.titulo = titulo; }
    public String getGenero() { return genero; }
    public void setGenero(String genero) { this.genero = genero; }
```

```
    public BigDecimal getPrecio() { return precio; }
    public void setPrecio(BigDecimal precio) { this.precio = precio; }
    public LocalDate getFechaPublicacion() { return fechaPublicacion; }
    public void setFechaPublicacion(LocalDate fechaPublicacion) { this.fechaPublica
    public Boolean getDisponible() { return disponible; }
    public void setDisponible(Boolean disponible) { this.disponible = disponible; }
    public Autor getAutor() { return autor; }
    public void setAutor(Autor autor) { this.autor = autor; }

    @Override
    public String toString() {
        return String.format("Libro[id=%d, isbn='%s', titulo='%s', autor='%s']",
            id, isbn, titulo, autor != null ? autor.getNombre() : "Sin autor");
    }
}
```

Repositorios

```
// AutorRepository.java
package com.dam.biblioteca.repository;

import com.dam.biblioteca.model.Autor;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface AutorRepository extends JpaRepository<Autor, Long> {

    List<Autor> findByNacionalidad(String nacionalidad);

    List<Autor> findByNombreContainingIgnoreCase(String nombre);

    @Query("SELECT a FROM Autor a WHERE SIZE(a.libros) > :cantidad")
    List<Autor> findAutoresConMasDeNLibros(int cantidad);
}
```

```java
// LibroRepository.java
package com.dam.biblioteca.repository;

import com.dam.biblioteca.model.Libro;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {

    Optional<Libro> findByIsbn(String isbn);

    List<Libro> findByGenero(String genero);

    List<Libro> findByDisponibleTrue();

    List<Libro> findByAutorId(Long autorId);

    List<Libro> findByPrecioBetween(BigDecimal min, BigDecimal max);

    @Query("SELECT l FROM Libro l WHERE l.autor.nombre = :nombreAutor")
    List<Libro> findByNombreAutor(@Param("nombreAutor") String nombreAutor);

    @Query("SELECT l.genero, COUNT(l), AVG(l.precio) FROM Libro l GROUP BY l.genero
    List<Object[]> obtenerEstadisticasPorGenero();

    long countByGenero(String genero);
}
```

Servicio

```
package com.dam.biblioteca.service;

import com.dam.biblioteca.model.Autor;
import com.dam.biblioteca.model.Libro;
import com.dam.biblioteca.repository.AutorRepository;
import com.dam.biblioteca.repository.LibroRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

@Service
public class BibliotecaService {

    private final AutorRepository autorRepository;
    private final LibroRepository libroRepository;

    @Autowired
    public BibliotecaService(AutorRepository autorRepository, LibroRepository libro
        this.autorRepository = autorRepository;
        this.libroRepository = libroRepository;
    }

    // ========================================================
    // OPERACIONES CON AUTORES
    // ========================================================

    @Transactional(readOnly = true)
    public List<Autor> obtenerTodosLosAutores() {
        return autorRepository.findAll();
    }

    @Transactional
    public Autor guardarAutor(Autor autor) {
        return autorRepository.save(autor);
    }

    @Transactional
    public Autor guardarAutorConLibros(Autor autor, List<Libro> libros) {
        for (Libro libro : libros) {
            autor.addLibro(libro);
        }
        return autorRepository.save(autor);
    }

    // ========================================================
    // OPERACIONES CON LIBROS
    // ========================================================

    @Transactional(readOnly = true)
    public List<Libro> obtenerTodosLosLibros() {
        return libroRepository.findAll();
    }
```

```
    @Transactional(readOnly = true)
    public Optional<Libro> buscarPorIsbn(String isbn) {
        return libroRepository.findByIsbn(isbn);
    }

    @Transactional(readOnly = true)
    public List<Libro> obtenerLibrosDisponibles() {
        return libroRepository.findByDisponibleTrue();
    }

    @Transactional
    public Libro prestarLibro(String isbn) {
        Libro libro = libroRepository.findByIsbn(isbn)
            .orElseThrow(() -> new RuntimeException("Libro no encontrado: " + isbn)

        if (!libro.getDisponible()) {
            throw new RuntimeException("El libro no está disponible para préstamo")
        }

        libro.setDisponible(false);
        return libroRepository.save(libro);
    }

    @Transactional
    public Libro devolverLibro(String isbn) {
        Libro libro = libroRepository.findByIsbn(isbn)
            .orElseThrow(() -> new RuntimeException("Libro no encontrado: " + isbn)

        libro.setDisponible(true);
        return libroRepository.save(libro);
    }

    @Transactional(readOnly = true)
    public List<Libro> buscarPorRangoPrecio(BigDecimal min, BigDecimal max) {
        return libroRepository.findByPrecioBetween(min, max);
    }

    @Transactional(readOnly = true)
    public void mostrarEstadisticas() {
        System.out.println("\n=== ESTADÍSTICAS DE LA BIBLIOTECA ===\n");

        List<Object[]> stats = libroRepository.obtenerEstadisticasPorGenero();

        for (Object[] row : stats) {
            String genero = (String) row[0];
            Long cantidad = (Long) row[1];
            Double precioMedio = (Double) row[2];

            System.out.printf("Género: %-15s | Cantidad: %3d | Precio medio: %.2f€%
                genero, cantidad, precioMedio);
        }
    }
}
```

Clase Principal

```
package com.dam.biblioteca;

import com.dam.biblioteca.model.Autor;
import com.dam.biblioteca.model.Libro;
import com.dam.biblioteca.service.BibliotecaService;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.Arrays;

@SpringBootApplication
public class BibliotecaApplication {

    public static void main(String[] args) {
        System.out.println("╔═══════════════════════════════════════════════════════
        System.out.println("║          SISTEMA DE GESTIÓN DE BIBLIOTECA
        System.out.println("║          Ejemplo de Hibernate con Spring Boot
        System.out.println("╚═══════════════════════════════════════════════════════

        ApplicationContext contexto = SpringApplication.run(BibliotecaApplication.c
        BibliotecaService service = contexto.getBean(BibliotecaService.class);

        // ═══════════════════════════════════════════════════════
        // CREAR DATOS DE EJEMPLO
        // ═══════════════════════════════════════════════════════

        // Crear autor con libros
        Autor cervantes = new Autor("Miguel de Cervantes", "Española");
        cervantes.setFechaNacimiento(LocalDate.of(1547, 9, 29));

        Libro quijote1 = new Libro("978-84-376-0494-7", "Don Quijote de la Mancha -
            "Novela", new BigDecimal("19.95"));
        Libro quijote2 = new Libro("978-84-376-0495-4", "Don Quijote de la Mancha -
            "Novela", new BigDecimal("19.95"));

        service.guardarAutorConLibros(cervantes, Arrays.asList(quijote1, quijote2))

        // Crear otro autor
        Autor garcia = new Autor("Gabriel García Márquez", "Colombiana");

        Libro cienAnios = new Libro("978-84-376-0553-1", "Cien años de soledad",
            "Realismo mágico", new BigDecimal("15.50"));
        Libro cronicaMuerte = new Libro("978-84-376-0554-8", "Crónica de una muerte
            "Novela", new BigDecimal("12.95"));

        service.guardarAutorConLibros(garcia, Arrays.asList(cienAnios, cronicaMuert

        // ═══════════════════════════════════════════════════════
        // DEMOSTRAR CONSULTAS
        // ═══════════════════════════════════════════════════════

        System.out.println("\n──── TODOS LOS LIBROS ────");
        service.obtenerTodosLosLibros().forEach(System.out::println);
```

```
        System.out.println("\n────── LIBROS DISPONIBLES ──────");
        service.obtenerLibrosDisponibles().forEach(System.out::println);

        System.out.println("\n────── PRÉSTAMO DE LIBRO ──────");
        Libro prestado = service.prestarLibro("978-84-376-0494-7");
        System.out.println("Libro prestado: " + prestado.getTitulo());
        System.out.println("Disponible: " + prestado.getDisponible());

        System.out.println("\n────── LIBROS DISPONIBLES DESPUÉS DEL PRÉSTAMO ──────")
        service.obtenerLibrosDisponibles().forEach(System.out::println);

        System.out.println("\n────── DEVOLUCIÓN DE LIBRO ──────");
        Libro devuelto = service.devolverLibro("978-84-376-0494-7");
        System.out.println("Libro devuelto: " + devuelto.getTitulo());

        // Estadísticas
        service.mostrarEstadisticas();

        System.out.println("\n═══════════════════════════════════════════════════════════");
        System.out.println("                    FIN DEL PROGRAMA                       ");
        System.out.println("═══════════════════════════════════════════════════════════");
    }
}
```

11. Resumen y Buenas Prácticas

11.1. Checklist de Hibernate

✅ **Checklist para usar Hibernate correctamente**

Entidades:

- [ ] Usar @Entity y @Table en todas las entidades
- [ ] Definir @Id y @GeneratedValue para la clave primaria
- [ ] Usar @Column para configurar columnas específicas
- [ ] Implementar constructor vacío (requerido por JPA)
- [ ] Usar `BigDecimal` para valores monetarios (no Double)
- [ ] Usar `LocalDate` / `LocalDateTime` para fechas (no Date)

Relaciones:

- [ ] Usar `FetchType.LAZY` por defecto
- [ ] Definir el lado propietario con @JoinColumn
- [ ] Usar `mappedBy` en el lado inverso
- [ ] Configurar `cascade` apropiadamente

Repositorios:

- [ ] Extender `JpaRepository<Entidad, TipoId>`
- [ ] Usar Query Methods cuando sea posible
- [ ] Usar @Query para consultas complejas

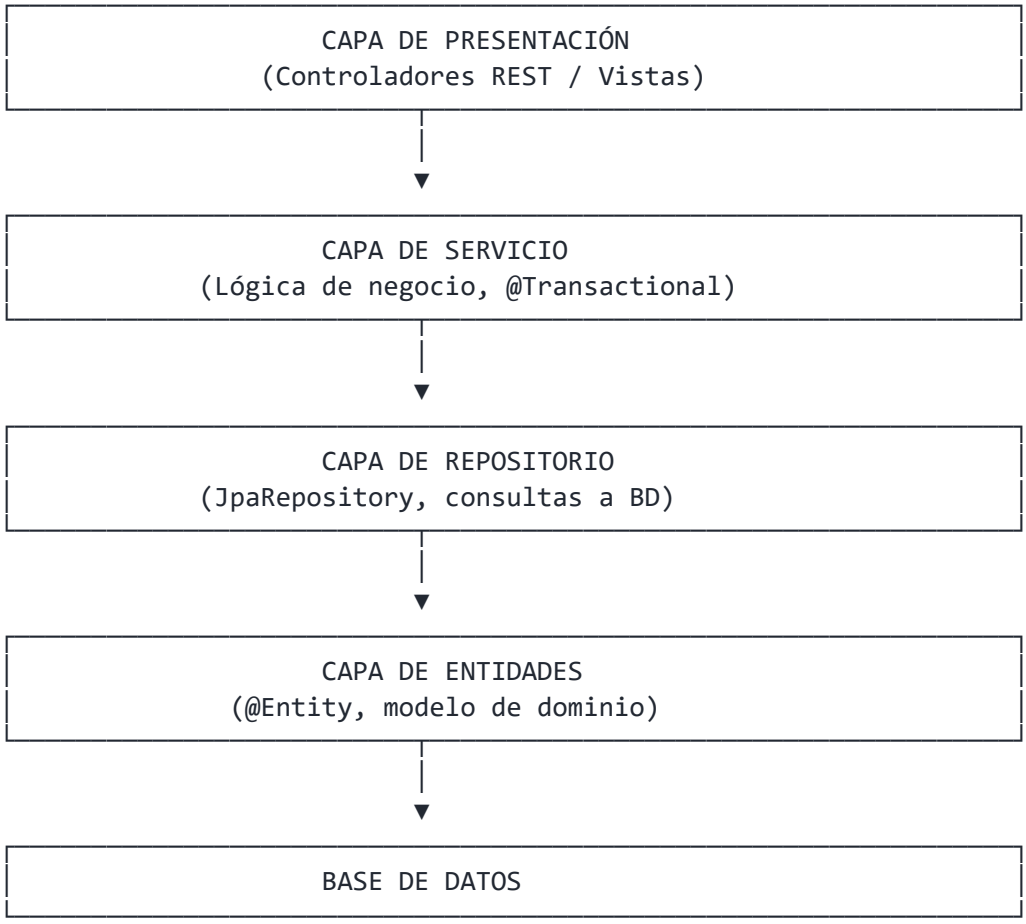
Servicios:

- [ ] Usar @Service y @Transactional
- [ ] Usar `readOnly = true` para operaciones de lectura
- [ ] Implementar validaciones de negocio

11.2. Errores comunes a evitar

> ⚠ Advertencia
>
> **Errores frecuentes con Hibernate:**
>
> 1. **LazyInitializationException**: Acceder a datos LAZY fuera de una transacción
> 2. **N+1 Problem**: Hacer una consulta por cada elemento de una colección
> 3. **Usar EAGER en todas partes**: Carga datos innecesarios
> 4. **Olvidar @Transactional**: Las operaciones de escritura pueden fallar
> 5. **No implementar equals/hashCode**: Problemas con colecciones y Set
> 6. **Usar Double para precios**: Errores de precisión decimal

11.3. Arquitectura de capas recomendada



12. Ejercicios Propuestos

Ejercicio 1: Entidad básica

Crea una entidad `Cliente` con los campos: id, nombre, email (único), telefono, fechaRegistro. Implementa el repositorio con métodos para buscar por email y por nombre.

Ejercicio 2: Relación `@ManyToOne`

Amplía el ejercicio anterior creando una entidad `Pedido` que pertenezca a un `Cliente`. Un cliente puede tener muchos pedidos.

Ejercicio 3: Query Methods

En el repositorio de `Pedido`, implementa métodos para:

- Buscar pedidos de un cliente
- Buscar pedidos entre dos fechas
- Contar pedidos por estado
- Obtener los 10 pedidos más recientes

Ejercicio 4: Consultas JPQL

Implementa las siguientes consultas usando `@Query`:

- Obtener el total gastado por cada cliente
- Listar clientes que no han realizado ningún pedido
- Obtener los productos más vendidos

13. Referencias

- Documentación oficial de Hibernate
- Spring Data JPA Reference
- Jakarta Persistence API (JPA) Specification
- Baeldung - Hibernate Tutorials

Documento elaborado para el módulo 0486 - Acceso a Datos 2º Desarrollo de Aplicaciones Multiplataforma (DAM)