

Bloque 1: Fundamentos de ORM y JPA

Contenido

- Objetivos de aprendizaje
- 1. El problema del mapeo objeto-relacional
- 2. Introducción a ORM
- 3. Arquitectura JPA e Hibernate
- 4. Configuración de Hibernate sin Spring
- 5. Definición de entidades JPA
- 6. Operaciones CRUD con Hibernate
- 7. Ejercicio práctico guiado
- 8. Resumen y conceptos clave
- 9. Ejercicios propuestos

Objetivos de aprendizaje

Al finalizar este bloque, el alumno será capaz de:

- Comprender el problema del desajuste de impedancia objeto-relacional
- Identificar las ventajas e inconvenientes de utilizar herramientas ORM
- Diferenciar entre la especificación JPA y sus implementaciones
- Instalar y configurar Hibernate como proveedor JPA
- Desarrollar aplicaciones básicas utilizando la API nativa de Hibernate
- Realizar operaciones CRUD mediante SessionFactory y Session

1. El problema del mapeo objeto-relacional

1.1. Paradigma orientado a objetos vs modelo relacional

En el desarrollo de aplicaciones empresariales nos encontramos con dos mundos fundamentalmente diferentes que deben coexistir: el paradigma orientado a objetos utilizado en lenguajes como Java, y el modelo relacional empleado en bases de datos SQL.

El paradigma orientado a objetos se caracteriza por:

- Encapsulación de datos y comportamiento en clases
- Herencia y polimorfismo
- Referencias directas entre objetos
- Identidad basada en la referencia de memoria
- Navegación mediante asociaciones

El modelo relacional se caracteriza por:

- Datos organizados en tablas con filas y columnas
- Sin concepto de herencia nativo
- Relaciones mediante claves foráneas
- Identidad basada en claves primarias
- Acceso mediante operaciones de conjunto (SQL)

1.2. El desajuste de impedancia (Impedance Mismatch)

El término «impedance mismatch» describe las diferencias fundamentales entre ambos paradigmas que dificultan la traducción directa de uno a otro. Los principales problemas son:

Problema de granularidad: En Java podemos crear clases con diferente nivel de detalle. Por ejemplo, una clase Dirección puede ser un atributo de Persona. En el modelo relacional, debemos decidir si crear una tabla separada o incluir los campos directamente.

```
// En Java: composición natural
public class Persona {
    private Long id;
    private String nombre;
    private Direccion direccion; // Objeto embebido
}

public class Direccion {
    private String calle;
    private String ciudad;
    private String codigoPostal;
}
```

En SQL tenemos dos opciones, cada una con sus implicaciones:

```
-- Opción 1: Campos en la misma tabla
CREATE TABLE persona (
    id BIGINT PRIMARY KEY,
    nombre VARCHAR(100),
    calle VARCHAR(200),
    ciudad VARCHAR(100),
    codigo_postal VARCHAR(10)
);

-- Opción 2: Tabla separada con FK
CREATE TABLE direccion (
    id BIGINT PRIMARY KEY,
    calle VARCHAR(200),
    ciudad VARCHAR(100),
    codigo_postal VARCHAR(10)
);

CREATE TABLE persona (
    id BIGINT PRIMARY KEY,
    nombre VARCHAR(100),
    direccion_id BIGINT REFERENCES direccion(id)
);
```

Problema de herencia: Java soporta herencia de forma nativa, pero las bases de datos relacionales no tienen un concepto equivalente directo.

```
// Herencia en Java
public abstract class Empleado {
    private Long id;
    private String nombre;
    private double salarioBase;
}

public class EmpleadoFijo extends Empleado {
    private int antiguedad;
    private double bonus;
}

public class EmpleadoTemporal extends Empleado {
    private LocalDate fechaFinContrato;
    private String empresaETT;
}
```

¿Cómo representamos esto en tablas? Existen varias estrategias, cada una con ventajas e inconvenientes que estudiaremos más adelante.

Problema de identidad: En Java, dos objetos pueden ser iguales de dos formas: identidad de referencia (==) e igualdad de contenido (equals). En bases de datos, la identidad se determina únicamente por la clave primaria.

```
Persona p1 = personaRepository.findById(1L);
Persona p2 = personaRepository.findById(1L);

// ¿Son el mismo objeto?
p1 == p2;      // Puede ser false (diferentes instancias)
p1.equals(p2); // Debería ser true (misma PK)
```

Problema de asociaciones: En Java navegamos entre objetos mediante referencias. Las bases de datos utilizan claves foráneas y requieren JOINs explícitos.

```
// En Java: navegación directa
Departamento dept = empleado.getDepartamento();
String nombreDept = dept.getNombre();
List<Empleado> companeros = dept.getEmpleados();

// En SQL: requiere JOIN
SELECT d.nombre
FROM empleado e
JOIN departamento d ON e.departamento_id = d.id
WHERE e.id = ?;
```

Problema de navegación: Los objetos se navegan uno a uno siguiendo referencias. SQL trabaja con conjuntos de datos, lo que puede generar el problema N+1.

```
// Código aparentemente inocente
List<Departamento> departamentos = obtenerTodosDepartamentos();
for (Departamento d : departamentos) {
    // Cada llamada puede generar una consulta SQL
    System.out.println(d.getNombre() + ": " + d.getEmpleados().size());
}
// Resultado: 1 consulta para departamentos + N consultas para empleados
```

1.3. Soluciones tradicionales al problema

Antes de la aparición de frameworks ORM, los desarrolladores utilizaban diferentes enfoques:

JDBC directo: Escribir manualmente todo el código SQL y el mapeo de resultados.

```
public Persona buscarPersonaPorId(Long id) {
    String sql = "SELECT id, nombre, email, fecha_nacimiento FROM persona WHERE id = ?";

    try (Connection conn = dataSource.getConnection();
        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setLong(1, id);

        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                Persona p = new Persona();
                p.setId(rs.getLong("id"));
                p.setNombre(rs.getString("nombre"));
                p.setEmail(rs.getString("email"));
                p.setFechaNacimiento(rs.getDate("fecha_nacimiento").toLocalDate());
                return p;
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Error al buscar persona", e);
    }
    return null;
}
```

Problemas de este enfoque: código repetitivo, propenso a errores, difícil de mantener, sin caché, gestión manual de conexiones y transacciones.

Patrón DAO con SQL embebido: Encapsular el acceso a datos en clases especializadas.

```
public class PersonaDAO {  
  
    public void insertar(Persona p) { /* SQL INSERT */ }  
    public void actualizar(Persona p) { /* SQL UPDATE */ }  
    public void eliminar(Long id) { /* SQL DELETE */ }  
    public Persona buscarPorId(Long id) { /* SQL SELECT */ }  
    public List<Persona> buscarTodos() { /* SQL SELECT */ }  
    public List<Persona> buscarPorNombre(String nombre) { /* SQL SELECT */ }  
}
```

Mejora la organización pero sigue requiriendo código SQL manual y mapeo repetitivo.

2. Introducción a ORM

2.1. Definición y propósito

ORM (Object-Relational Mapping) es una técnica de programación que permite convertir datos entre el sistema de tipos de un lenguaje de programación orientado a objetos y una base de datos relacional.

Un framework ORM actúa como una capa de abstracción que:

- Traduce automáticamente objetos Java a filas de tablas y viceversa
- Genera las sentencias SQL necesarias
- Gestiona las conexiones y transacciones
- Proporciona caché para mejorar el rendimiento
- Ofrece un lenguaje de consultas orientado a objetos

2.2. Funcionamiento básico de un ORM

El proceso de mapeo se define mediante metadatos que indican la correspondencia entre clases y tablas:

| Clase Java | | Tabla SQL |
|------------------|--------|----------------|
| Persona | <----> | persona |
| - id: Long | <----> | id BIGINT PK |
| - nombre: String | <----> | nombre VARCHAR |
| - email: String | <----> | email VARCHAR |

El ORM intercepta las operaciones sobre objetos y las traduce a SQL:

```
// Código Java
Persona p = new Persona("Juan", "juan@email.com");
entityManager.persist(p);
```

SQL generado automáticamente:

```
INSERT INTO persona (nombre, email) VALUES ('Juan', 'juan@email.com');
```

2.3. Ventajas de utilizar ORM

Productividad: Reduce drásticamente la cantidad de código necesario. No es necesario escribir SQL repetitivo ni código de mapeo manual.

Mantenibilidad: Los cambios en el modelo de datos se reflejan en un solo lugar. Si añadimos un campo a una entidad, el ORM actualiza automáticamente las operaciones CRUD.

Portabilidad: El código es independiente del motor de base de datos. Cambiar de MySQL a PostgreSQL requiere mínimos ajustes.

Rendimiento: Los ORM modernos incluyen optimizaciones como caché de primer y segundo nivel, lazy loading, y batch processing.

Seguridad: Previene inyección SQL al utilizar consultas parametrizadas internamente.

Consistencia: Garantiza que el modelo de objetos y el esquema de base de datos estén sincronizados.

2.4. Inconvenientes de utilizar ORM

Curva de aprendizaje: Requiere comprender conceptos como ciclo de vida de entidades, contexto de persistencia, fetch strategies, etc.

Pérdida de control: El SQL generado puede no ser óptimo para casos complejos. A veces es necesario recurrir a consultas nativas.

Overhead: Añade una capa de abstracción que consume recursos. Para operaciones masivas simples, JDBC directo puede ser más eficiente.

Problema N+1: Si no se configura correctamente, puede generar múltiples consultas innecesarias.

Complejidad en consultas avanzadas: Algunas consultas SQL complejas son difíciles de expresar con el lenguaje de consultas del ORM.

Abstracción permeable: En ocasiones es necesario entender el SQL subyacente para resolver problemas de rendimiento.

3. Arquitectura JPA e Hibernate

3.1. JPA: La especificación

JPA (Java Persistence API) es una especificación de Java que define un estándar para el mapeo objeto-relacional. Es importante entender que JPA no es una implementación, sino un conjunto de interfaces y anotaciones que definen cómo debe comportarse un proveedor de persistencia.

La especificación JPA define:

- Anotaciones para mapeo (`@Entity`, `@Table`, `@Column`, `@Id, etc.`)
- API de EntityManager para operaciones CRUD
- JPQL (Java Persistence Query Language) para consultas
- Criteria API para consultas programáticas
- Gestión del ciclo de vida de entidades
- Reglas para transacciones y caché

Versiones principales de JPA:

| Versión | Año | Novedades principales |
|---------|------|--|
| JPA 1.0 | 2006 | Especificación inicial |
| JPA 2.0 | 2009 | Criteria API, caché L2 estandarizado |
| JPA 2.1 | 2013 | Stored procedures, converters, entity graphs |
| JPA 2.2 | 2017 | Soporte para Java 8 Date/Time, streaming |
| JPA 3.0 | 2020 | Migración a Jakarta EE (javax -> jakarta) |
| JPA 3.1 | 2022 | UUID como tipo básico, mejoras JPQL |

3.2. Implementaciones de JPA

Al ser JPA una especificación, necesitamos una implementación concreta para utilizarla:

Hibernate: La implementación más popular y madura. Fue creada antes de JPA y posteriormente se adaptó al estándar. Ofrece características adicionales más allá de la especificación.

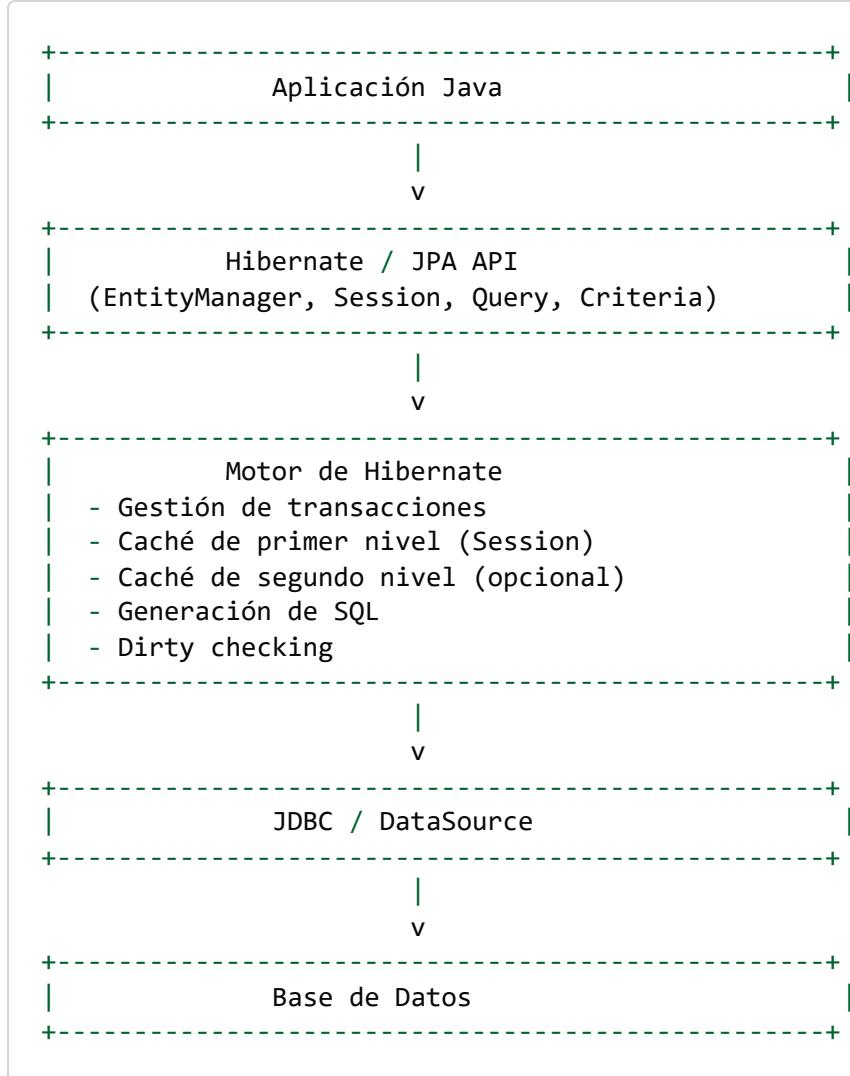
EclipseLink: Implementación de referencia de JPA. Desarrollada por la Eclipse Foundation, es la evolución de TopLink de Oracle.

OpenJPA: Implementación de Apache. Menos utilizada actualmente pero completamente funcional.

En este curso utilizaremos Hibernate por ser el estándar de facto en la industria y la implementación por defecto en Spring Boot.

3.3. Arquitectura de Hibernate

Hibernate se estructura en varias capas:



Componentes principales:

- **Configuration**: Carga la configuración y los mapeos
- **SessionFactory**: Factoría de sesiones, thread-safe, una por base de datos
- **Session**: Unidad de trabajo, no thread-safe, contexto de persistencia
- **Transaction**: Gestiona los límites transaccionales
- **Query**: Ejecuta consultas HQL/JPQL
- **Criteria**: Construye consultas programáticamente

3.4. JPA API vs Hibernate API

Podemos trabajar con Hibernate de dos formas:

API estándar JPA (recomendada):

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory("mi-unidad");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
em.persist(entidad);
tx.commit();

em.close();
emf.close();

```

API nativa de Hibernate:

```

SessionFactory sf = new Configuration().configure().buildSessionFactory();
Session session = sf.openSession();
Transaction tx = session.beginTransaction();

session.persist(entidad);
tx.commit();

session.close();
sf.close();

```

La correspondencia entre ambas APIs es directa:

| JPA | Hibernate |
|----------------------|---------------------|
| EntityManagerFactory | SessionFactory |
| EntityManager | Session |
| EntityTransaction | Transaction |
| persist() | persist() / save() |
| merge() | merge() / update() |
| remove() | remove() / delete() |
| find() | get() |
| createQuery() | createQuery() |

4. Configuración de Hibernate sin Spring

Antes de utilizar Spring Data JPA, es fundamental comprender cómo funciona Hibernate de forma independiente. Esto permite entender qué hace Spring «por debajo» y facilita la resolución de problemas.

4.1. Dependencias Maven

Para un proyecto Hibernate puro necesitamos las siguientes dependencias:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ejemplo</groupId>
    <artifactId>hibernate-puro</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <hibernate.version>6.4.0.Final</hibernate.version>
        <mysql.version>8.0.33</mysql.version>
    </properties>

    <dependencies>
        <!-- Hibernate Core -->
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate.version}</version>
        </dependency>

        <!-- Driver MySQL -->
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>${mysql.version}</version>
        </dependency>

        <!-- Logging (opcional pero recomendado) -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.9</version>
        </dependency>
    </dependencies>

</project>

```

4.2. Archivo de configuración hibernate.cfg.xml

El archivo principal de configuración se ubica en `src/main/resources/hibernate.cfg.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Configuración de conexión a base de datos -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hiber
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

        <!-- Dialecto SQL específico para MySQL 8 -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop

        <!-- Pool de conexiones básico (solo para desarrollo) -->
        <property name="hibernate.connection.pool_size">5</property>

        <!-- Gestión del esquema de base de datos -->
        <!-- Opciones: validate, update, create, create-drop, none -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Mostrar SQL en consola (solo desarrollo) -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.highlight_sql">true</property>

        <!-- Comentarios en el SQL generado -->
        <property name="hibernate.use_sql_comments">true</property>

        <!-- Contexto de sesión actual -->
        <property name="hibernate.current_session_context_class">thread</property>

        <!-- Registro de entidades mapeadas -->
        <mapping class="com.ejemplo.modelo.Producto"/>
        <mapping class="com.ejemplo.modelo.Categoría"/>

    </session-factory>
</hibernate-configuration>
```

Explicación de las propiedades principales:

La propiedad `hibernate.hbm2ddl.auto` controla cómo Hibernate gestiona el esquema:

| Valor | Comportamiento |
|-------------|--|
| validate | Valida el esquema, no realiza cambios |
| update | Actualiza el esquema si hay diferencias |
| create | Crea el esquema, destruyendo datos previos |
| create-drop | Crea al iniciar, elimina al cerrar |
| none | No hace nada con el esquema |

En producción se recomienda `validate` o `none`, utilizando herramientas de migración como Flyway o Liquibase.

4.3. Configuración alternativa con persistence.xml (JPA estándar)

Si preferimos usar la API estándar JPA, el archivo de configuración es

`src/main/resources/META-INF/persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
              version="3.0">

    <persistence-unit name="ejemplo-pu" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>com.ejemplo.modelo.Producto</class>
        <class>com.ejemplo.modelo.Categoría</class>

        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/hibernate_ejemplo"/>
            <property name="jakarta.persistence.jdbc.user" value="root"/>
            <property name="jakarta.persistence.jdbc.password" value="root"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

4.4. Configuración programática (sin XML)

También es posible configurar Hibernate completamente mediante código Java:

```
package com.ejemplo.config;

import com.ejemplo.modelo.Producto;
import com.ejemplo.modelo.Categoría;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.cfg.Environment;
import org.hibernate.service.ServiceRegistry;

import java.util.Properties;

public class HibernateConfig {

    private static SessionFactory sessionFactory;

    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            try {
                Configuration configuration = new Configuration();

                Properties settings = new Properties();
                settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");
                settings.put(Environment.URL, "jdbc:mysql://localhost:3306/hibernat");
                settings.put(Environment.USER, "root");
                settings.put(Environment.PASS, "root");
                settings.put(Environment.DIALECT, "org.hibernate.dialect.MySQLDiale");
                settings.put(Environment.SHOW_SQL, "true");
                settings.put(Environment.FORMAT_SQL, "true");
                settings.put(Environment.HBM2DDL_AUTO, "update");
                settings.put(Environment.CURRENT_SESSION_CONTEXT_CLASS, "thread");

                configuration.setProperties(settings);

                // Registrar entidades
                configuration.addAnnotatedClass(Producto.class);
                configuration.addAnnotatedClass(Categoría.class);

                ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                    .applySettings(configuration.getProperties())
                    .build();

                sessionFactory = configuration.buildSessionFactory(serviceRegistry)

            } catch (Exception e) {
                throw new RuntimeException("Error al crear SessionFactory", e);
            }
        }
        return sessionFactory;
    }

    public static void shutdown() {
        if (sessionFactory != null) {
            sessionFactory.close();
        }
    }
}
```

5. Definición de entidades JPA

5.1. Anatomía de una entidad

Una entidad JPA es una clase Java que representa una tabla en la base de datos. Los requisitos mínimos son:

1. Anotación `@Entity` en la clase
2. Un campo identificador con `@Id`
3. Constructor sin argumentos (puede ser protected)
4. No puede ser final ni tener métodos final

```
package com.ejemplo.modelo;

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

@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", columnDefinition = "TEXT")
    private String descripcion;

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

    @Column(name = "stock")
    private Integer stock;

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

    @Column(name = "fecha_creacion")
    private LocalDateTime fechaCreacion;

    // Constructor sin argumentos requerido por JPA
    public Producto() {
    }

    // Constructor para crear nuevos productos
    public Producto(String nombre, BigDecimal precio) {
        this.nombre = nombre;
        this.precio = precio;
        this.activo = true;
        this.fechaCreacion = LocalDateTime.now();
    }

    // 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 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 Boolean getActivo() {
    return activo;
}

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

public LocalDateTime getFechaCreacion() {
    return fechaCreacion;
}

public void setFechaCreacion(LocalDateTime fechaCreacion) {
    this.fechaCreacion = fechaCreacion;
}

@Override
public String toString() {
    return "Producto{" +
        "id=" + id +
        ", nombre='" + nombre + '\'' +
        ", precio=" + precio +
        ", stock=" + stock +
        ", activo=" + activo +
        '}';
}
}
```

5.2. Anotaciones básicas de mapeo

@Entity: Marca la clase como una entidad JPA. Hibernate creará una tabla para esta clase.

@Table: Permite personalizar el nombre de la tabla y otras propiedades.

```
@Entity
@Table(name = "tbl_productos",
       schema = "inventario",
       uniqueConstraints = @UniqueConstraint(columnNames = {"codigo", "proveedor_id"})
public class Producto { }
```

@Id: Marca el campo como clave primaria.

@GeneratedValue: Define la estrategia de generación de valores para el ID.

```
// AUTO: Hibernate elige la estrategia según el dialecto
@GeneratedValue(strategy = GenerationType.AUTO)

// IDENTITY: Usa autoincremento de la BD (recomendado para MySQL)
@GeneratedValue(strategy = GenerationType.IDENTITY)

// SEQUENCE: Usa una secuencia (recomendado para PostgreSQL/Oracle)
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "producto_seq")
@SequenceGenerator(name = "producto_seq", sequenceName = "producto_sequence")

// TABLE: Usa una tabla auxiliar para generar IDs
@GeneratedValue(strategy = GenerationType.TABLE)
```

@Column: Personaliza el mapeo de un campo a una columna.

```
@Column(
    name = "nombre_producto",           // Nombre de la columna
    nullable = false,                   // NOT NULL
    unique = true,                     // UNIQUE
    length = 150,                      // VARCHAR(150)
    insertable = true,                 // Incluir en INSERT
    updatable = true,                  // Incluir en UPDATE
    columnDefinition = "TEXT"          // Definición SQL exacta
)
private String nombre;

@Column(precision = 10, scale = 2) // DECIMAL(10,2)
private BigDecimal precio;
```

5.3. Entidad con relación básica

Veamos cómo definir una entidad Categoría que tendrá una relación con Producto:

```
package com.ejemplo.modelo;

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

@Entity
@Table(name = "categorias")
public class Categoria {

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

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

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

    @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Producto> productos = new ArrayList<>();

    public Categoria() {
    }

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    // Métodos de conveniencia para gestionar la relación bidireccional
    public void addProducto(Producto producto) {
        productos.add(producto);
        producto.setCategoria(this);
    }

    public void removeProducto(Producto producto) {
        productos.remove(producto);
        producto.setCategoria(null);
    }

    // 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 getDescripcion() {
    return descripcion;
}

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

public List<Producto> getProductos() {
    return productos;
}

public void setProductos(List<Producto> productos) {
    this.productos = productos;
}

@Override
public String toString() {
    return "Categoria{" +
        "id=" + id +
        ", nombre='" + nombre + '\'' +
        ", numProductos=" + productos.size() +
        '}';
}
}

```

Y actualizamos Producto para incluir la relación inversa:

```

// Añadir en la clase Producto

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "categoria_id")
private Categoria categoria;

public Categoria getCategoria() {
    return categoria;
}

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

```

6. Operaciones CRUD con Hibernate

6.1. Clase utilitaria HibernateUtil

Creamos una clase para gestionar la SessionFactory de forma centralizada:

```
package com.ejemplo.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Carga hibernate.cfg.xml del classpath
            sessionFactory = new Configuration()
                .configure()
                .buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Error al crear SessionFactory: " + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

6.2. Operación CREATE (persist)

```

package com.example.operaciones;

import com.example.modelo.Producto;
import com.example.modelo.Categoría;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import java.math.BigDecimal;

public class CrearEntidades {

    public static void main(String[] args) {

        // Crear una categoría
        Categoría categoria = crearCategoria("Electrónica", "Productos electrónicos");
        System.out.println("Categoría creada: " + categoria);

        // Crear productos asociados a la categoría
        Producto p1 = crearProducto("Laptop HP", new BigDecimal("899.99"), categoria);
        Producto p2 = crearProducto("Mouse Logitech", new BigDecimal("29.99"), categoria);

        System.out.println("Producto creado: " + p1);
        System.out.println("Producto creado: " + p2);

        HibernateUtil.shutdown();
    }

    public static Categoría crearCategoria(String nombre, String descripción) {
        Transaction transaction = null;
        Categoría categoria = new Categoría(nombre);
        categoria.setDescripción(descripción);

        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            session.persist(categoría);

            transaction.commit();
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            throw new RuntimeException("Error al crear categoría", e);
        }

        return categoria;
    }

    public static Producto crearProducto(String nombre, BigDecimal precio, Categoría categoria) {
        Transaction transaction = null;
        Producto producto = new Producto(nombre, precio);

        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            // Reasociar la categoría a esta sesión

```

```
Categoría categoriaManaged = session.get(Categoría.class, categoría.getId());
producto.setCategoria(categoríaManaged);
producto.setStock(100);

session.persist(producto);

transaction.commit();
} catch (Exception e) {
    if (transaction != null) {
        transaction.rollback();
    }
    throw new RuntimeException("Error al crear producto", e);
}

return producto;
}
}
```

6.3. Operación READ (get, find, queries)

```

package com.example.operaciones;

import com.example.modelo.Producto;
import com.example.modelo.Categoría;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.query.Query;
import java.util.List;

public class LeerEntidades {

    public static void main(String[] args) {

        // Buscar por ID
        Producto producto = buscarProductoPorId(1L);
        System.out.println("Producto encontrado: " + producto);

        // Buscar todos
        List<Producto> todos = buscarTodosProductos();
        System.out.println("Total productos: " + todos.size());

        // Buscar con condiciones
        List<Producto> activos = buscarProductosActivos();
        System.out.println("Productos activos: " + activos.size());

        // Buscar por categoría
        List<Producto> porCategoria = buscarProductosPorCategoria("Electrónica");
        System.out.println("Productos en Electrónica: " + porCategoria.size());

        HibernateUtil.shutdown();
    }

    public static Producto buscarProductoPorId(Long id) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            // get() devuelve null si no existe
            return session.get(Producto.class, id);

            // find() es el equivalente JPA
            // return session.find(Producto.class, id);
        }
    }

    public static List<Producto> buscarTodosProductos() {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Producto> query = session.createQuery(
                "FROM Producto", Producto.class);
            return query.list();
        }
    }

    public static List<Producto> buscarProductosActivos() {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Producto> query = session.createQuery(
                "FROM Producto p WHERE p.activo = :activo", Producto.class);
            query.setParameter("activo", true);
            return query.list();
        }
    }
}

```

```
        }

    }

    public static List<Producto> buscarProductosPorCategoria(String nombreCategoria) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Producto> query = session.createQuery(
                "FROM Producto p WHERE p.categoría.nombre = :categoria",
                Producto.class);
            query.setParameter("categoria", nombreCategoria);
            return query.list();
        }
    }

    public static Producto buscarProductoPorNombre(String nombre) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Producto> query = session.createQuery(
                "FROM Producto p WHERE p.nombre = :nombre", Producto.class);
            query.setParameter("nombre", nombre);
            // getSingleResult() lanza excepción si no hay resultados
            // uniqueResult() devuelve null si no hay resultados
            return query.uniqueResult();
        }
    }
}
```

6.4. Operación UPDATE (merge, dirty checking)

```

package com.ejemplo.operaciones;

import com.ejemplo.modelo.Producto;
import com.ejemplo.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import java.math.BigDecimal;

public class ActualizarEntidades {
    public static void main(String[] args) {
        // Actualización dentro de la sesión (dirty checking)
        actualizarPrecioConDirtyChecking(1L, new BigDecimal("999.99"));

        // Actualización con merge (entidad detached)
        Producto productoDetached = obtenerProductoDetached(1L);
        productoDetached.setStock(50);
        actualizarProductoConMerge(productoDetached);

        // Actualización con HQL
        int actualizados = actualizarPreciosCategoria("Electrónica", 1.10);
        System.out.println("Productos actualizados: " + actualizados);
    }

    // Dirty Checking - cambios automáticos en entidades managed
    public static void actualizarPrecioConDirtyChecking(Long id, BigDecimal nuevoPrecio) {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            Producto producto = session.get(Producto.class, id);
            if (producto != null) {
                // Al modificar una entidad managed, Hibernate detecta los cambios
                producto.setPrecio(nuevoPrecio);
                // NO es necesario llamar a update() - dirty checking lo hace automática
            }

            transaction.commit();
            System.out.println("Precio actualizado con dirty checking");
        } catch (Exception e) {
            if (transaction != null) transaction.rollback();
            e.printStackTrace();
        }
    }

    // Obtener producto que quedará detached
    public static Producto obtenerProductoDetached(Long id) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            return session.get(Producto.class, id);
        }
        // Al cerrar la sesión, el producto queda detached
    }

    // Merge - para entidades detached
    public static void actualizarProductoConMerge(Producto productoDetached) {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {

```

```
transaction = session.beginTransaction();

// merge() copia el estado del objeto detached a una entidad managed
Producto productoManaged = session.merge(productoDetached);

transaction.commit();
System.out.println("Producto actualizado con merge");
} catch (Exception e) {
    if (transaction != null) transaction.rollback();
    e.printStackTrace();
}
}

// UPDATE masivo con HQL
public static int actualizarPreciosCategoria(String categoria, double factor) {
    Transaction transaction = null;
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        transaction = session.beginTransaction();

        String hql = "UPDATE Producto p SET p.precio = p.precio * :factor " +
                    "WHERE p.categoría.nombre = :categoria";

        int actualizados = session.createMutationQuery(hql)
            .setParameter("factor", factor)
            .setParameter("categoria", categoria)
            .executeUpdate();

        transaction.commit();
        return actualizados;
    } catch (Exception e) {
        if (transaction != null) transaction.rollback();
        e.printStackTrace();
        return 0;
    }
}
}
```

6.5. Operación DELETE (remove)

```

package com.ejemplo.operaciones;

import com.ejemplo.modelo.Producto;
import com.ejemplo.modelo.Categoría;
import com.ejemplo.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class EliminarEntidades {
    public static void main(String[] args) {
        // Eliminar producto por ID
        eliminarProducto(5L);

        // Eliminar con HQL
        int eliminados = eliminarProductosSinStock();
        System.out.println("Productos eliminados: " + eliminados);

        // Eliminar categoría (con cascade)
        eliminarCategoria(3L);
    }

    // Eliminar entidad individual
    public static void eliminarProducto(Long id) {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            Producto producto = session.get(Producto.class, id);
            if (producto != null) {
                session.remove(producto);
                System.out.println("Producto eliminado: " + producto.getNombre());
            } else {
                System.out.println("Producto no encontrado");
            }

            transaction.commit();
        } catch (Exception e) {
            if (transaction != null) transaction.rollback();
            e.printStackTrace();
        }
    }

    // DELETE masivo con HQL
    public static int eliminarProductosSinStock() {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            String hql = "DELETE FROM Producto p WHERE p.stock <= 0";
            int eliminados = session.createMutationQuery(hql).executeUpdate();

            transaction.commit();
            return eliminados;
        } catch (Exception e) {
            if (transaction != null) transaction.rollback();
            e.printStackTrace();
        }
    }
}

```

```

        return 0;
    }

    // Eliminar con cascade (elimina productos relacionados)
    public static void eliminarCategoria(Long id) {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            transaction = session.beginTransaction();

            Categoria categoria = session.get(Categoria.class, id);
            if (categoria != null) {
                // Si cascade está configurado, eliminará los productos también
                session.remove(categoria);
                System.out.println("Categoría eliminada: " + categoria.getNombre())
            }

            transaction.commit();
        } catch (Exception e) {
            if (transaction != null) transaction.rollback();
            e.printStackTrace();
        }
    }
}

```

7. Ejercicio práctico guiado

7.1. Descripción del ejercicio

Vamos a crear una aplicación completa de gestión de una biblioteca utilizando Hibernate puro. El sistema gestionará libros, autores y préstamos.

Requisitos:

1. Un libro tiene título, ISBN, año de publicación y puede tener múltiples autores
2. Un autor tiene nombre, nacionalidad y puede haber escrito múltiples libros
3. Un préstamo registra qué libro se prestó, a quién, la fecha de préstamo y la fecha de devolución

7.2. Paso 1: Crear el proyecto Maven

Crear la estructura del proyecto:

```
biblioteca-hibernate/
├── pom.xml
└── src/
    └── main/
        ├── java/
        │   └── com/
        │       └── biblioteca/
        │           ├── modelo/
        │           │   ├── Libro.java
        │           │   └── Autor.java
        │           ├── dao/
        │           │   ├── LibroDAO.java
        │           │   └── AutorDAO.java
        │           ├── util/
        │           │   └── HibernateUtil.java
        │           └── App.java
    └── resources/
        └── hibernate.cfg.xml
```

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.biblioteca</groupId>
    <artifactId>biblioteca-hibernate</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <name>Biblioteca Hibernate</name>
    <description>Sistema de gestión de biblioteca con Hibernate puro</description>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <hibernate.version>6.4.0.Final</hibernate.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate.version}</version>
        </dependency>

        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>8.0.33</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.9</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.11.0</version>
                <configuration>
                    <source>17</source>
                    <target>17</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

7.3. Paso 2: Configurar Hibernate

src/main/resources/hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/biblioteca
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>

    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>

    <mapping class="com.biblioteca.modelo.Libro"/>
    <mapping class="com.biblioteca.modelo.Autor"/>
    <mapping class="com.biblioteca.modelo.Prestamo"/>
  </session-factory>
</hibernate-configuration>
```

7.4. Paso 3: Crear las entidades

Autor.java:

```
package com.biblioteca.modelo;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@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 = "biografia", columnDefinition = "TEXT")
    private String biografia;

    @ManyToMany(mappedBy = "autores")
    private Set<Libro> libros = new HashSet<>();

    public Autor() {
    }

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

    // 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 = nacionalidad;
    }
}
```

```
public String getBiografia() {
    return biografia;
}

public void setBiografia(String biografia) {
    this.biografia = biografia;
}

public Set<Libro> getLibros() {
    return libros;
}

public void setLibros(Set<Libro> libros) {
    this.libros = libros;
}

@Override
public String toString() {
    return "Autor{id=" + id + ", nombre='" + nombre + "', nacionalidad='" + nac
}
}
```

Libro.java:

```

package com.biblioteca.modelo;

import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

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

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

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

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

    @Column(name = "anio_publicacion")
    private Integer anioPublicacion;

    @Column(name = "num_paginas")
    private Integer numPaginas;

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

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "libro_autor",
        joinColumns = @JoinColumn(name = "libro_id"),
        inverseJoinColumns = @JoinColumn(name = "autor_id")
    )
    private Set<Autor> autores = new HashSet<>();

    @OneToMany(mappedBy = "libro", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Prestamo> prestamos = new ArrayList<>();

    public Libro() {
    }

    public Libro(String titulo, String isbn, Integer anioPublicacion) {
        this.titulo = titulo;
        this.isbn = isbn;
        this.anioPublicacion = anioPublicacion;
    }

    // Métodos de conveniencia para gestionar relaciones
    public void addAutor(Autor autor) {
        autores.add(autor);
        autor.getLibros().add(this);
    }

    public void removeAutor(Autor autor) {
}

```

```
        autores.remove(autor);
        autor.getLibros().remove(this);
    }

    public void addPrestamo(Prestamo prestamo) {
        prestamos.add(prestamo);
        prestamo.setLibro(this);
    }

    // Getters y Setters
    public Long getId() {
        return id;
    }

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

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public Integer getAnioPublicacion() {
        return anioPublicacion;
    }

    public void setAnioPublicacion(Integer anioPublicacion) {
        this.anioPublicacion = anioPublicacion;
    }

    public Integer getNumPaginas() {
        return numPaginas;
    }

    public void setNumPaginas(Integer numPaginas) {
        this.numPaginas = numPaginas;
    }

    public Boolean getDisponible() {
        return disponible;
    }

    public void setDisponible(Boolean disponible) {
        this.disponible = disponible;
    }

    public Set<Autor> getAutores() {
```

```
        return autores;
    }

    public void setAutores(Set<Autor> autores) {
        this.autores = autores;
    }

    public List<Prestamo> getPrestamos() {
        return prestamos;
    }

    public void setPrestamos(List<Prestamo> prestamos) {
        this.prestamos = prestamos;
    }

    @Override
    public String toString() {
        return "Libro{id=" + id + ", titulo='" + titulo + "', isbn='" + isbn +
               "', año=" + anioPublicacion + ", disponible=" + disponible + "}";
    }
}
```

Prestamo.java:

```

package com.biblioteca.modelo;

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

@Entity
@Table(name = "prestamos")
public class Prestamo {

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "libro_id", nullable = false)
    private Libro libro;

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

    @Column(name = "email_usuario", length = 100)
    private String emailUsuario;

    @Column(name = "fecha_prestamo", nullable = false)
    private LocalDate fechaPrestamo;

    @Column(name = "fecha_devolucion_prevista", nullable = false)
    private LocalDate fechaDevolucionPrevista;

    @Column(name = "fecha_devolucion_real")
    private LocalDate fechaDevolucionReal;

    @Column(name = "observaciones", columnDefinition = "TEXT")
    private String observaciones;

    public Prestamo() {
    }

    public Prestamo(Libro libro, String nombreUsuario, LocalDate fechaPrestamo, int diasPrestamo) {
        this.libro = libro;
        this.nombreUsuario = nombreUsuario;
        this.fechaPrestamo = fechaPrestamo;
        this.fechaDevolucionPrevista = fechaPrestamo.plusDays(diasPrestamo);
    }

    public boolean estaActivo() {
        return fechaDevolucionReal == null;
    }

    public boolean estaRetrasado() {
        if (fechaDevolucionReal != null) {
            return fechaDevolucionReal.isAfter(fechaDevolucionPrevista);
        }
        return LocalDate.now().isAfter(fechaDevolucionPrevista);
    }

    // Getters y Setters
    public Long getId() {
}

```

```
        return id;
    }

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

    public Libro getLibro() {
        return libro;
    }

    public void setLibro(Libro libro) {
        this.libro = libro;
    }

    public String getNombreUsuario() {
        return nombreUsuario;
    }

    public void setNombreUsuario(String nombreUsuario) {
        this.nombreUsuario = nombreUsuario;
    }

    public String getEmailUsuario() {
        return emailUsuario;
    }

    public void setEmailUsuario(String emailUsuario) {
        this.emailUsuario = emailUsuario;
    }

    public LocalDate getFechaPrestamo() {
        return fechaPrestamo;
    }

    public void setFechaPrestamo(LocalDate fechaPrestamo) {
        this.fechaPrestamo = fechaPrestamo;
    }

    public LocalDate getFechaDevolucionPrevista() {
        return fechaDevolucionPrevista;
    }

    public void setFechaDevolucionPrevista(LocalDate fechaDevolucionPrevista) {
        this.fechaDevolucionPrevista = fechaDevolucionPrevista;
    }

    public LocalDate getFechaDevolucionReal() {
        return fechaDevolucionReal;
    }

    public void setFechaDevolucionReal(LocalDate fechaDevolucionReal) {
        this.fechaDevolucionReal = fechaDevolucionReal;
    }

    public String getObservaciones() {
        return observaciones;
    }
```

```

public void setObservaciones(String observaciones) {
    this.observaciones = observaciones;
}

@Override
public String toString() {
    return "Prestamo{id=" + id + ", libro=" + (libro != null ? libro.getTitulo()
        ", usuario='" + nombreUsuario + "'", fechaPrestamo=" + fechaPrestamo
        ", activo=" + estaActivo() + ", retrasado=" + estaRetrasado() + "}";
}
}

```

7.5. Paso 4: Crear HibernateUtil

HibernateUtil.java:

```

package com.biblioteca.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration()
                .configure("hibernate.cfg.xml")
                .buildSessionFactory();

            System.out.println("SessionFactory creada correctamente");

        } catch (Throwable ex) {
            System.err.println("Error al crear SessionFactory: " + ex.getMessage())
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        if (sessionFactory != null && !sessionFactory.isClosed()) {
            sessionFactory.close();
            System.out.println("SessionFactory cerrada");
        }
    }
}

```

7.6. Paso 5: Crear las clases DAO

AutorDAO.java:

```
package com.biblioteca.dao;

import com.biblioteca.modelo.Autor;
import com.biblioteca.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

import java.util.List;

public class AutorDAO {

    public Autor guardar(Autor autor) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            tx = session.beginTransaction();
            session.persist(autor);
            tx.commit();
            return autor;
        } catch (Exception e) {
            if (tx != null) tx.rollback();
            throw new RuntimeException("Error al guardar autor", e);
        }
    }

    public Autor buscarPorId(Long id) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            return session.get(Autor.class, id);
        }
    }

    public List<Autor> buscarTodos() {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            return session.createQuery("FROM Autor ORDER BY nombre", Autor.class).list();
        }
    }

    public List<Autor> buscarPorNacionalidad(String nacionalidad) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Autor> query = session.createQuery(
                "FROM Autor WHERE nacionalidad = :nacionalidad ORDER BY nombre"
            );
            query.setParameter("nacionalidad", nacionalidad);
            return query.list();
        }
    }

    public List<Autor> buscarPorNombre(String nombre) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Autor> query = session.createQuery(
                "FROM Autor WHERE LOWER(nombre) LIKE LOWER(:nombre) ORDER BY no"
            );
            query.setParameter("nombre", "%" + nombre + "%");
            return query.list();
        }
    }

    public Autor actualizar(Autor autor) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
```

```
tx = session.beginTransaction();
Autor autorActualizado = session.merge(autor);
tx.commit();
return autorActualizado;
} catch (Exception e) {
if (tx != null) tx.rollback();
throw new RuntimeException("Error al actualizar autor", e);
}
}

public void eliminar(Long id) {
Transaction tx = null;
try (Session session = HibernateUtil.getSessionFactory().openSession()) {
tx = session.beginTransaction();
Autor autor = session.get(Autor.class, id);
if (autor != null) {
session.remove(autor);
}
tx.commit();
} catch (Exception e) {
if (tx != null) tx.rollback();
throw new RuntimeException("Error al eliminar autor", e);
}
}
}
```

LibroDAO.java:

```
package com.biblioteca.dao;

import com.biblioteca.modelo.Libro;
import com.biblioteca.modelo.Autor;
import com.biblioteca.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

import java.util.List;

public class LibroDAO {

    public Libro guardar(Libro libro) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            tx = session.beginTransaction();
            session.persist(libro);
            tx.commit();
            return libro;
        } catch (Exception e) {
            if (tx != null) tx.rollback();
            throw new RuntimeException("Error al guardar libro", e);
        }
    }

    public Libro guardarConAutores(Libro libro, List<Autor> autores) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            tx = session.beginTransaction();

            // Asociar autores existentes al libro
            for (Autor autor : autores) {
                Autor autorManaged = session.get(Autor.class, autor.getId());
                if (autorManaged != null) {
                    libro.addAutor(autorManaged);
                }
            }

            session.persist(libro);
            tx.commit();
            return libro;
        } catch (Exception e) {
            if (tx != null) tx.rollback();
            throw new RuntimeException("Error al guardar libro con autores", e);
        }
    }

    public Libro buscarPorId(Long id) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            return session.get(Libro.class, id);
        }
    }

    public Libro buscarPorIdConAutores(Long id) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query<Libro> query = session.createQuery(
                "SELECT DISTINCT l FROM Libro l LEFT JOIN FETCH l.autores WHERE"
            );
        }
    }
}
```

```

        Libro.class);
    query.setParameter("id", id);
    return query.uniqueResult();
}

public Libro buscarPorIsbn(String isbn) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Libro> query = session.createQuery(
            "FROM Libro WHERE isbn = :isbn", Libro.class);
        query.setParameter("isbn", isbn);
        return query.uniqueResult();
    }
}

public List<Libro> buscarTodos() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery("FROM Libro ORDER BY titulo", Libro.class).list();
    }
}

public List<Libro> buscarDisponibles() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery(
            "FROM Libro WHERE disponible = true ORDER BY titulo", Libro.class).list();
    }
}

public List<Libro> buscarPorTitulo(String titulo) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Libro> query = session.createQuery(
            "FROM Libro WHERE LOWER(titulo) LIKE LOWER(:titulo) ORDER BY titulo", Libro.class);
        query.setParameter("titulo", "%" + titulo + "%");
        return query.list();
    }
}

public List<Libro> buscarPorAutor(Long autorId) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Libro> query = session.createQuery(
            "SELECT l FROM Libro l JOIN l.autores a WHERE a.id = :autorId ORDER BY l.titulo", Libro.class);
        query.setParameter("autorId", autorId);
        return query.list();
    }
}

public List<Libro> buscarPorAnioPublicacion(int anio) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Libro> query = session.createQuery(
            "FROM Libro WHERE anioPublicacion = :anio ORDER BY titulo", Libro.class);
        query.setParameter("anio", anio);
        return query.list();
    }
}

public Libro actualizar(Libro libro) {
    Transaction tx = null;
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        tx = session.beginTransaction();
        session.update(libro);
        tx.commit();
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        e.printStackTrace();
    }
}

```

```

        tx = session.beginTransaction();
        Libro libroActualizado = session.merge(libro);
        tx.commit();
        return libroActualizado;
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        throw new RuntimeException("Error al actualizar libro", e);
    }
}

public void cambiarDisponibilidad(Long id, boolean disponible) {
    Transaction tx = null;
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        tx = session.beginTransaction();
        Libro libro = session.get(Libro.class, id);
        if (libro != null) {
            libro.setDisponible(disponible);
        }
        tx.commit();
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        throw new RuntimeException("Error al cambiar disponibilidad", e);
    }
}

public void eliminar(Long id) {
    Transaction tx = null;
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        tx = session.beginTransaction();
        Libro libro = session.get(Libro.class, id);
        if (libro != null) {
            // Eliminar relaciones con autores
            libro.getAutores().clear();
            session.remove(libro);
        }
        tx.commit();
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        throw new RuntimeException("Error al eliminar libro", e);
    }
}

public long contarLibros() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery("SELECT COUNT(1) FROM Libro l", Long.class).getSingleResult();
    }
}

public long contarLibrosDisponibles() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery(
            "SELECT COUNT(1) FROM Libro l WHERE l.disponible = true", Long.class).getSingleResult();
    }
}
}

```

PrestamoDAO.java:

```
package com.biblioteca.dao;

import com.biblioteca.modelo.Prestamo;
import com.biblioteca.modelo.Libro;
import com.biblioteca.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

import java.time.LocalDate;
import java.util.List;

public class PrestamoDAO {

    public Prestamo realizarPrestamo(Long libroId, String nombreUsuario, String emailUsuario) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            tx = session.beginTransaction();

            Libro libro = session.get(Libro.class, libroId);
            if (libro == null) {
                throw new RuntimeException("Libro no encontrado");
            }
            if (!libro.getDisponible()) {
                throw new RuntimeException("Libro no disponible para préstamo");
            }

            Prestamo prestamo = new Prestamo(libro, nombreUsuario, LocalDate.now());
            prestamo.setEmailUsuario(emailUsuario);

            // Marcar libro como no disponible
            libro.setDisponible(false);
            libro.addPrestamo(prestamo);

            session.persist(prestamo);
            tx.commit();

            return prestamo;
        } catch (Exception e) {
            if (tx != null) tx.rollback();
            throw new RuntimeException("Error al realizar préstamo: " + e.getMessage());
        }
    }

    public Prestamo registrarDevolucion(Long prestamoId, String observaciones) {
        Transaction tx = null;
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            tx = session.beginTransaction();

            Prestamo prestamo = session.get(Prestamo.class, prestamoId);
            if (prestamo == null) {
                throw new RuntimeException("Préstamo no encontrado");
            }
            if (!prestamo.estaActivo()) {
                throw new RuntimeException("El préstamo ya fue devuelto");
            }

            prestamo.setFechaDevolucionReal(LocalDate.now());
        }
    }
}
```

```

        prestamo.setObservaciones(observaciones);

        // Marcar libro como disponible
        prestamo.getLibro().setDisponible(true);

        tx.commit();
        return prestamo;
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        throw new RuntimeException("Error al registrar devolución", e);
    }
}

public Prestamo buscarPorId(Long id) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.get(Prestamo.class, id);
    }
}

public List<Prestamo> buscarTodos() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery(
            "FROM Prestamo p JOIN FETCH p.libro ORDER BY p.fechaPrestamo DE
            Prestamo.class).list();
    }
}

public List<Prestamo> buscarActivos() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        return session.createQuery(
            "FROM Prestamo p JOIN FETCH p.libro WHERE p.fechaDevolucionReal
            "ORDER BY p.fechaDevolucionPrevista", Prestamo.class).list();
    }
}

public List<Prestamo> buscarRetrasados() {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Prestamo> query = session.createQuery(
            "FROM Prestamo p JOIN FETCH p.libro " +
            "WHERE p.fechaDevolucionReal IS NULL AND p.fechaDevolucionPrevista
            "ORDER BY p.fechaDevolucionPrevista", Prestamo.class);
        query.setParameter("hoy", LocalDate.now());
        return query.list();
    }
}

public List<Prestamo> buscarPorUsuario(String nombreUsuario) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Query<Prestamo> query = session.createQuery(
            "FROM Prestamo p JOIN FETCH p.libro " +
            "WHERE LOWER(p.nombreUsuario) LIKE LOWER(:nombre) " +
            "ORDER BY p.fechaPrestamo DESC", Prestamo.class);
        query.setParameter("nombre", "%" + nombreUsuario + "%");
        return query.list();
    }
}

public List<Prestamo> buscarPorLibro(Long libroId) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {

```

```

Query<Prestamo> query = session.createQuery(
    "FROM Prestamo p WHERE p.libro.id = :libroId ORDER BY p.fechaPr
    Prestamo.class);
query.setParameter("libroId", libroId);
return query.list();
}

public List<Prestamo> buscarPorRangoFechas(LocalDate desde, LocalDate hasta) {
try (Session session = HibernateUtil.getSessionFactory().openSession()) {
    Query<Prestamo> query = session.createQuery(
        "FROM Prestamo p JOIN FETCH p.libro " +
        "WHERE p.fechaPrestamo BETWEEN :desde AND :hasta " +
        "ORDER BY p.fechaPrestamo", Prestamo.class);
    query.setParameter("desde", desde);
    query.setParameter("hasta", hasta);
    return query.list();
}
}

public long contarPrestamosActivos() {
try (Session session = HibernateUtil.getSessionFactory().openSession()) {
    return session.createQuery(
        "SELECT COUNT(p) FROM Prestamo p WHERE p.fechaDevolucionReal IS
Long.class).uniqueResult();
}
}

public long contarPrestamosRetrasados() {
try (Session session = HibernateUtil.getSessionFactory().openSession()) {
    Query<Long> query = session.createQuery(
        "SELECT COUNT(p) FROM Prestamo p " +
        "WHERE p.fechaDevolucionReal IS NULL AND p.fechaDevolucionPrevi
Long.class);
    query.setParameter("hoy", LocalDate.now());
    return query.uniqueResult();
}
}
}

```

7.7. Paso 6: Crear la aplicación principal

App.java:

```

package com.biblioteca;

import com.biblioteca.dao.AutorDAO;
import com.biblioteca.dao.LibroDAO;
import com.biblioteca.dao.PrestamoDAO;
import com.biblioteca.modelo.Autor;
import com.biblioteca.modelo.Libro;
import com.biblioteca.modelo.Prestamo;
import com.biblioteca.util.HibernateUtil;

import java.util.Arrays;
import java.util.List;

public class App {

    private static final AutorDAO autorDAO = new AutorDAO();
    private static final LibroDAO libroDAO = new LibroDAO();
    private static final PrestamoDAO prestamoDAO = new PrestamoDAO();

    public static void main(String[] args) {
        try {
            System.out.println("==> SISTEMA DE GESTIÓN DE BIBLIOTECA ==>\n");

            // 1. Crear autores
            System.out.println("--- Creando autores ---");
            Autor cervantes = autorDAO.guardar(new Autor("Miguel de Cervantes", "Es"));
            Autor garcia = autorDAO.guardar(new Autor("Gabriel García Márquez", "Co"));
            Autor borges = autorDAO.guardar(new Autor("Jorge Luis Borges", "Argenti"));

            System.out.println("Autor creado: " + cervantes);
            System.out.println("Autor creado: " + garcia);
            System.out.println("Autor creado: " + borges);

            // 2. Crear libros con autores
            System.out.println("\n--- Creando libros ---");
            Libro quijote = new Libro("Don Quijote de la Mancha", "978-84-376-0494-");
            quijote.setNumPaginas(1345);
            quijote = libroDAO.guardarConAutores(quijote, Arrays.asList(cervantes));
            System.out.println("Libro creado: " + quijote);

            Libro cienAnios = new Libro("Cien años de soledad", "978-84-376-0495-4");
            cienAnios.setNumPaginas(471);
            cienAnios = libroDAO.guardarConAutores(cienAnios, Arrays.asList(garcia));
            System.out.println("Libro creado: " + cienAnios);

            Libro ficciones = new Libro("Ficciones", "978-84-376-0496-1", 1944);
            ficciones.setNumPaginas(174);
            ficciones = libroDAO.guardarConAutores(ficciones, Arrays.asList(borges));
            System.out.println("Libro creado: " + ficciones);

            // Libro con múltiples autores
            Autor autorColaborador = autorDAO.guardar(new Autor("Adolfo Bioy Casares"));
            Libro cronicas = new Libro("Crónicas de Bustos Domecq", "978-84-376-0497-");
            cronicas = libroDAO.guardarConAutores(cronicas, Arrays.asList(borges, a));
            System.out.println("Libro creado (múltiples autores): " + cronicas);

            // 3. Consultar libros
            System.out.println("\n--- Consultando libros ---");
        }
    }
}

```

```

System.out.println("Total de libros: " + libroDAO.contarLibros());
System.out.println("Libros disponibles: " + libroDAO.contarLibrosDisponibles());

List<Libro> todosLibros = libroDAO.buscarTodos();
System.out.println("\nListado de todos los libros:");
todosLibros.forEach(l -> System.out.println(" - " + l));

// 4. Buscar libro con autores
System.out.println("\n--- Buscando libro con autores ---");
Libro libroConAutores = libroDAO.buscarPorIdConAutores(cronicas.getId());
System.out.println("Libro: " + libroConAutores.getTitulo());
System.out.println("Autores:");
libroConAutores.getAutores().forEach(a -> System.out.println(" - " + a));

// 5. Realizar préstamos
System.out.println("\n--- Realizando préstamos ---");
Prestamo prestamo1 = prestamoDAO.realizarPrestamo(
    quijote.getId(), "Juan García", "juan@email.com", 14);
System.out.println("Préstamo realizado: " + prestamo1);

Prestamo prestamo2 = prestamoDAO.realizarPrestamo(
    cienAnios.getId(), "María López", "maria@email.com", 21);
System.out.println("Préstamo realizado: " + prestamo2);

// 6. Consultar préstamos
System.out.println("\n--- Consultando préstamos ---");
System.out.println("Préstamos activos: " + prestamoDAO.contarPrestamosActivos());

List<Prestamo> prestamosActivos = prestamoDAO.buscarActivos();
System.out.println("\nListado de préstamos activos:");
prestamosActivos.forEach(p -> System.out.println(" - " + p));

// 7. Verificar disponibilidad
System.out.println("\n--- Verificando disponibilidad ---");
System.out.println("Libros disponibles después de préstamos: " +
    libroDAO.contarLibrosDisponibles());

List<Libro> disponibles = libroDAO.buscarDisponibles();
System.out.println("Libros disponibles:");
disponibles.forEach(l -> System.out.println(" - " + l.getTitulo()));

// 8. Registrar devolución
System.out.println("\n--- Registrando devolución ---");
Prestamo prestamoDevuelto = prestamoDAO.registrarDevolucion(
    prestamo1.getId(), "Libro en buen estado");
System.out.println("Devolución registrada: " + prestamoDevuelto);
System.out.println("¿Retrasado?: " + prestamoDevuelto.estaRetrasado());

// 9. Estadísticas finales
System.out.println("\n--- Estadísticas finales ---");
System.out.println("Total libros: " + libroDAO.contarLibros());
System.out.println("Libros disponibles: " + libroDAO.contarLibrosDisponibles());
System.out.println("Préstamos activos: " + prestamoDAO.contarPrestamosActivos());
System.out.println("Préstamos retrasados: " + prestamoDAO.contarPrestamosRetrasados());

// 10. Buscar por autor
System.out.println("\n--- Libros de Borges ---");
List<Libro> librosBorges = libroDAO.buscarPorAutor(borges.getId());
librosBorges.forEach(l -> System.out.println(" - " + l.getTitulo()));

```

```
System.out.println("\n==> APLICACIÓN FINALIZADA CORRECTAMENTE ==>");

} catch (Exception e) {
    System.err.println("Error en la aplicación: " + e.getMessage());
    e.printStackTrace();
} finally {
    HibernateUtil.shutdown();
}
}
```

7.8. Paso 7: Ejecutar y verificar

Para ejecutar el proyecto:

```
# Crear la base de datos (MySQL)
mysql -u root -p -e "CREATE DATABASE IF NOT EXISTS biblioteca;" 

# Compilar el proyecto
mvn clean compile

# Ejecutar la aplicación
mvn exec:java -Dexec.mainClass="com.biblioteca.App"
```

Salida esperada:

```

==== SISTEMA DE GESTIÓN DE BIBLIOTECA ====

--- Creando autores ---
Hibernate:
    insert
    into
        autores
        (biografia, nacionalidad, nombre)
    values
        (?, ?, ?)
Autor creado: Autor{id=1, nombre='Miguel de Cervantes', nacionalidad='Española'}
...

--- Creando libros ---
Hibernate:
    insert
    into
        libros
        (anio_publicacion, disponible, isbn, num_paginas, titulo)
    values
        (?, ?, ?, ?, ?)
...
...

--- Estadísticas finales ---
Total libros: 4
Libros disponibles: 3
Préstamos activos: 1
Préstamos retrasados: 0

==== APPLICACIÓN FINALIZADA CORRECTAMENTE ====

```

8. Resumen y conceptos clave

En este bloque hemos aprendido:

Conceptos teóricos:

1. El problema del desajuste de impedancia entre el paradigma orientado a objetos y el modelo relacional
2. Las diferencias en granularidad, herencia, identidad, asociaciones y navegación
3. Qué es un ORM y cómo resuelve estos problemas
4. Ventajas: productividad, mantenibilidad, portabilidad, seguridad
5. Inconvenientes: curva de aprendizaje, pérdida de control, overhead

Arquitectura JPA e Hibernate:

1. JPA es una especificación; Hibernate es la implementación más popular
2. Los componentes principales: SessionFactory, Session, Transaction

3. Equivalencia entre API JPA (EntityManager) y API Hibernate (Session)

Configuración:

1. Archivo hibernate.cfg.xml para configuración basada en XML
2. Archivo persistence.xml para configuración JPA estándar
3. Propiedades principales: conexión, dialecto, hbm2ddl.auto, show_sql

Entidades JPA:

1. Anotaciones básicas: `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`
2. Relaciones: `@ManyToOne`, `@OneToMany`, `@ManyToMany`
3. Métodos de conveniencia para gestionar relaciones bidireccionales

Operaciones CRUD:

1. CREATE: persist() para nuevas entidades
2. READ: get()/find() para búsqueda por ID, createQuery() para HQL
3. UPDATE: dirty checking automático, merge() para entidades detached
4. DELETE: remove() para eliminar entidades

9. Ejercicios propuestos

Ejercicio 1: Ampliar el sistema de biblioteca añadiendo una entidad Editorial con los campos: id, nombre, país, sitioWeb. Establecer una relación ManyToOne desde Libro hacia Editorial.

Ejercicio 2: Crear un método en PrestamoDAO que devuelva un informe con el número de préstamos por mes del año actual.

Ejercicio 3: Implementar un sistema de multas. Crear una entidad Multa que se genere automáticamente cuando un préstamo se devuelve con retraso. La multa debe calcular el importe en función de los días de retraso.

Ejercicio 4: Añadir un método de búsqueda avanzada que permita buscar libros por múltiples criterios opcionales: título, autor, año mínimo, año máximo, disponibilidad.

Ejercicio 5: Implementar paginación en los métodos de búsqueda. Modificar los DAOs para aceptar parámetros de página y tamaño de página.