SAFA
NTRA. SRA. DE LOS REYES
SEVILLA

Curso 2026/2027
Luis Javier López López

MANUAL DE CLASE

Spring Boot de cero

Del proyecto vacío a las primeras consultas

Arrancar el proyecto, configurar las dependencias, conectar con PostgreSQL, entender las capas y escribir tus primeras entidades, repositorios y consultas. Válido para 2º DAM (Acceso a Datos) y 2º DAW (Desarrollo Web en Entorno Servidor).

Java 21	Spring Boot 3	PostgreSQL 16	Spring Data JPA
Lenguaje	Framework	Base de datos	Persistencia

SAFA Ntra. Sra. de los Reyes · Sevilla

Ejemplo guía: BiblioAula

Contenido

1 Antes de empezar 1

2 Qué es Spring Boot y cómo funciona 2

3 La arquitectura en capas 3

4 Crear el proyecto 4

5 El pom.xml por dentro 6

6 Preparar PostgreSQL 8

7 Configurar la conexión 10

8 El modelo del ejemplo 12

9 Las entidades 13

10 Las relaciones 15

11 Los repositorios 18

12 Las consultas 19

13 Probar la conexión y las consultas 23

14 Errores frecuentes 25

15 Chuleta 26

Cómo usar este manual Sigue las secciones en orden con el ejemplo BiblioAula y, cada vez que aparezca un bloque «Ahora tú», aplica lo aprendido a tu propio proyecto. Las secciones 14 y 15 son de consulta rápida: tenlas a mano mientras programas.

1 Antes de empezar

Este manual te lleva desde un proyecto vacío hasta consultar datos reales de PostgreSQL con Spring Data JPA. Todo lo que hagas aquí lo repetirás después en tu propio proyecto.

1.1 Qué vas a construir

Vamos a montar **BiblioAula**, la biblioteca de un centro educativo: libros, sus autores y los préstamos al alumnado. Es un ejemplo pequeño a propósito, con las tres situaciones que te vas a encontrar siempre: una relación de uno a muchos, una de muchos a muchos y un estado que cambia con el tiempo.

Al terminar tendrás una aplicación que arranca, se conecta a la base de datos, valida que tus entidades coinciden con las tablas y responde a consultas escritas por ti.

1.2 Qué necesitas instalado

Herramienta	Versión	Cómo comprobar que la tienes
JDK (Java)	21 o superior	`java -version` en la terminal
Maven	Incluido en el proyecto	`./mvnw -v` dentro de la carpeta del proyecto
IntelliJ IDEA o VS Code	Última estable	Abre el IDE y comprueba que reconoce proyectos Maven
PostgreSQL	16	`psql --version`
Postman o Bruno	Última estable	Para probar la API más adelante

> **Si algo falla** Antes de seguir, arregla la instalación. Un 90 % de los problemas del primer día son de versiones: un JDK antiguo o un PostgreSQL que no está arrancado.

1.3 Cómo usar el manual

- Los bloques oscuros **«Ahora tú»** son tareas sobre **tu propio proyecto**: cada vez que aparezca uno, aplica lo aprendido a tu caso.
- Los bloques de código son para escribirlos, no para copiarlos y pegarlos. Se aprende escribiendo y viendo los errores del IDE.
- Al final tienes una sección de errores frecuentes y una chuleta de anotaciones.

2 Qué es Spring Boot y cómo funciona

Antes de tocar código conviene entender qué hace el framework por ti, porque casi todo lo que parece magia tiene una explicación sencilla.

2.1 De Spring a Spring Boot

Spring es un framework que se encarga de crear y conectar los objetos de tu aplicación. Tú no haces `new` de tus servicios: los declaras y Spring los construye, los guarda en un contenedor y se los pasa a quien los necesite. Eso es la **inyección de dependencias**.

Spring Boot es Spring con las pilas puestas: te da configuración por defecto razonable, un servidor web integrado y los **starters**, que son paquetes de dependencias que ya encajan entre sí. Así pasas de una tarde de configuración a un proyecto que arranca en un minuto.

2.2 Las tres ideas que hay que tener claras

Idea	Qué significa	Dónde la ves
Contenedor de beans	Spring crea una única instancia de cada componente y la guarda para repartirla	`@Service`, `@Repository`, `@RestController`
Inyección de dependencias	Un componente recibe lo que necesita por el constructor, no lo crea él	El constructor de tus servicios, o `@RequiredArgsConstructor` de Lombok
Autoconfiguración	Según las dependencias del `pom.xml`, Spring configura lo que hace falta	Añades el driver de PostgreSQL y ya tienes conexión y `EntityManager`

2.3 El punto de arranque

```java
@SpringBootApplication
public class BiblioAulaApplication {

    public static void main(String[] args) {
        SpringApplication.run(BiblioAulaApplication.class, args);
    }
}
```

`@SpringBootApplication` hace tres cosas a la vez: marca la clase como configuración, activa la autoconfiguración y, la más importante para ti, **escanea el paquete donde está la clase y todos sus subpaquetes** buscando componentes. Por eso todo tu código debe colgar de ese paquete: si creas una clase fuera, Spring no la encontrará.

> **Regla práctica** La clase con `@SpringBootApplication` vive en el paquete raíz, por ejemplo `es.safareyes.biblioaula`, y el resto de paquetes (`entity`, `repository`, `service`…) cuelgan de ahí.

3 La arquitectura en capas

Cada clase tiene un único trabajo. Respetar esto es lo que diferencia un proyecto que se puede mantener de uno que no.

Capa	Responsabilidad	Qué NO hace
Controlador	Recibe la petición HTTP, valida el formato de entrada y devuelve la respuesta	No contiene reglas de negocio ni habla con el repositorio
Servicio	Aplica las reglas de negocio, coordina repositorios y gestiona la transacción	No sabe nada de HTTP ni de códigos de estado
Repositorio	Consulta y guarda en la base de datos	No decide nada: solo consulta lo que le piden
Entidad	Representa una fila de una tabla	No lleva validaciones de entrada ni lógica de negocio
DTO	Transporta datos entre el exterior y la aplicación	No se persiste nunca

3.1 El recorrido de una petición

Cuando alguien pide `GET /api/v1/libros/3`, el camino es siempre el mismo:

1. El **controlador** recibe la petición, extrae el identificador y llama al servicio.
2. El **servicio** comprueba lo que haga falta y pide los datos al repositorio.
3. El **repositorio** ejecuta la consulta; Hibernate la traduce a SQL y PostgreSQL responde.
4. Hibernate convierte las filas en **entidades** y se las devuelve al servicio.
5. El servicio convierte la entidad en un **DTO** y el controlador lo devuelve como JSON.

> **El error más común del principio** Inyectar el repositorio directamente en el controlador porque «es más rápido». Funciona, pero rompe la separación de capas: la lógica acaba repartida por todas partes y no se puede reutilizar ni probar.

3.2 Estructura de paquetes

PAQUETES

```
es.safareyes.biblioaula
├── BiblioAulaApplication.java   → clase de arranque
├── config                       → configuración propia
├── controller                   → @RestController
├── dto                          → records de entrada y salida
├── entity                       → @Entity y enums
├── exception                    → excepciones propias
├── mapper                       → conversión entidad ↔ DTO
├── repository                   → interfaces JpaRepository
└── service                      → reglas de negocio
```

En este manual vamos a trabajar de dentro hacia fuera: primero `entity`, luego `repository` y, al final, una prueba rápida. Los controladores y los DTOs llegan en la siguiente unidad.

4 Crear el proyecto

Vamos a generar el esqueleto con Spring Initializr. Tarda menos de dos minutos, pero conviene entender cada campo.

4.1 Spring Initializr

Entra en `start.spring.io` (o usa el asistente del IDE) y rellena:

Campo	Valor	Por qué
Project	Maven	Es el gestor de dependencias que usamos en clase
Language	Java	—
Spring Boot	La última versión estable sin SNAPSHOT ni M1	Las versiones de prueba cambian sin avisar
Group	`es.safareyes`	Identifica a la organización; será el inicio del paquete
Artifact	`biblioaula`	Nombre del proyecto y del `jar` que se genera
Packaging	Jar	Spring Boot lleva su propio servidor dentro
Java	21	La versión que tienes instalada

4.2 Dependencias de partida

Dependencia en Initializr	Artefacto	Para qué
Spring Web	`spring-boot-starter-web`	Publicar endpoints REST y levantar el servidor
Spring Data JPA	`spring-boot-starter-data-jpa`	Entidades, repositorios y Hibernate
PostgreSQL Driver	`postgresql`	Hablar con PostgreSQL
Validation	`spring-boot-starter-validation`	Validar los datos que entran
Lombok	`lombok`	Escribir menos código repetitivo
Spring Boot DevTools	`spring-boot-devtools`	Reinicio automático al guardar cambios

> **¿Y si se me olvida una dependencia?** No pasa nada: se añaden a mano en el `pom.xml` cuando quieras. Initializr solo te ahorra escribirlas la primera vez.

4.3 Qué te encuentras al abrirlo

ESTRUCTURA

```
biblioaula
├── pom.xml                          → dependencias y configuración de Maven
├── mvnw / mvnw.cmd                  → Maven incluido, no hace falta instalarlo
└── src
    ├── main
    │   ├── java/es/safareyes/biblioaula/BiblioAulaApplication.java
    │   └── resources
    │       ├── application.properties   → configuración de la aplicación
    │       ├── static/                  → recursos estáticos
    │       └── templates/               → plantillas (no las usaremos)
    └── test/java/...                    → pruebas
```

4.4 Primer arranque

1. Ejecuta la clase `BiblioAulaApplication` desde el IDE, o `./mvnw spring-boot:run` en la terminal.
2. Lee la consola: verás el banner de Spring, el puerto (`8080` por defecto) y el tiempo de arranque.
3. Si la aplicación se para nada más arrancar quejándose de la base de datos, es normal: todavía no la hemos configurado.

AHORA TÚ · TU PROYECTO

Crea el proyecto de tu caso (`swapcloset`, `smartchef`, `schoolevents`, `coffeestation`…) con estas mismas dependencias y comprueba que arranca antes de seguir.

5 El pom.xml por dentro

El `pom.xml` es la lista de la compra de tu proyecto. Conviene saber leerlo, porque es donde se resuelven la mitad de los problemas.

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.x.x</version>
</parent>

<properties>
    <java.version>21</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

- El **parent** fija las versiones de todas las dependencias de Spring: por eso las declaras **sin versión**.
- `scope runtime` en el driver significa que hace falta al ejecutar, no al compilar.
- Cuando añadas algo al `pom.xml`, recarga el proyecto Maven en el IDE para que descargue lo nuevo.

> **Lombok en el IDE** Además de la dependencia, el IDE necesita tener activado el procesamiento de anotaciones. Si ves errores del tipo «cannot find symbol: method getNombre()», es eso.

5.1 Comandos de Maven que vas a usar

Comando	Qué hace
`./mvnw spring-boot:run`	Arranca la aplicación
`./mvnw clean package`	Compila y genera el `jar` en `target/`
`./mvnw test`	Ejecuta las pruebas

Comando	Qué hace
`./mvnw dependency:tree`	Muestra el árbol de dependencias: útil cuando hay conflictos de versiones

6 Preparar PostgreSQL

Spring no crea la base de datos por ti: tiene que existir antes de arrancar.

6.1 Crear la base de datos y el usuario

```sql
-- Desde psql, conectado como el superusuario postgres
CREATE DATABASE biblioaula;
CREATE USER biblio_user WITH PASSWORD 'biblio_pass';
GRANT ALL PRIVILEGES ON DATABASE biblioaula TO biblio_user;

-- Ya dentro de la base de datos (\c biblioaula)
GRANT ALL ON SCHEMA public TO biblio_user;
```

También puedes hacerlo con pgAdmin si prefieres el entorno gráfico: botón derecho sobre *Databases* → *Create* → *Database*.

6.2 Comprobar que responde

```
psql -h localhost -p 5432 -U biblio_user -d biblioaula
# Si entra, escribe \dt para ver las tablas (todavía no habrá ninguna) y \q para salir
```

> **Errores típicos aquí** «Connection refused» significa que PostgreSQL no está arrancado. «password authentication failed» es usuario o contraseña mal escritos, no un problema de Spring.

6.3 Crear las tablas

En este manual creamos las tablas con un script y **le pedimos a Hibernate que solo las valide**. Es la forma de trabajar del proyecto: la base de datos la diseñas tú, no la genera el framework.

```sql
CREATE TABLE autores (
    id     BIGSERIAL PRIMARY KEY,
    nombre VARCHAR(120) NOT NULL,
    pais   VARCHAR(60)
);

CREATE TABLE libros (
    id            BIGSERIAL PRIMARY KEY,
    isbn          VARCHAR(20)  NOT NULL UNIQUE,
    titulo        VARCHAR(200) NOT NULL,
    anio          INTEGER      NOT NULL,
    ejemplares    INTEGER      NOT NULL DEFAULT 1,
    CONSTRAINT chk_ejemplares CHECK (ejemplares >= 0)
);

CREATE TABLE libro_autor (
    libro_id BIGINT NOT NULL REFERENCES libros(id)  ON DELETE CASCADE,
    autor_id BIGINT NOT NULL REFERENCES autores(id) ON DELETE RESTRICT,
    PRIMARY KEY (libro_id, autor_id)
);

CREATE TABLE prestamos (
    id               BIGSERIAL PRIMARY KEY,
    libro_id         BIGINT NOT NULL REFERENCES libros(id) ON DELETE RESTRICT,
    lector           VARCHAR(120) NOT NULL,
    fecha_prestamo   DATE NOT NULL,
    fecha_devolucion DATE,
    estado           VARCHAR(20) NOT NULL,
    CONSTRAINT chk_fechas CHECK (fecha_devolucion IS NULL OR fecha_devolucion >= fecha_prestamo)
);

CREATE INDEX idx_prestamos_estado ON prestamos (estado);
```

```sql
INSERT INTO autores (nombre, pais) VALUES
  ('Ana María Matute', 'España'), ('Haruki Murakami', 'Japón'), ('Mary Shelley', 'Reino Unido');

INSERT INTO libros (isbn, titulo, anio, ejemplares) VALUES
  ('978-84-1', 'Olvidado Rey Gudú', 1996, 3),
  ('978-84-2', 'Tokio Blues', 1987, 2),
  ('978-84-3', 'Frankenstein', 1818, 1);

INSERT INTO libro_autor (libro_id, autor_id) VALUES (1, 1), (2, 2), (3, 3);

INSERT INTO prestamos (libro_id, lector, fecha_prestamo, fecha_devolucion, estado) VALUES
  (1, 'lucia@centro.es', '2026-09-10', '2026-09-24', 'DEVUELTO'),
  (2, 'diego@centro.es', '2026-09-20', NULL,         'ACTIVO'),
  (3, 'sara@centro.es',  '2026-09-01', NULL,         'RETRASADO');
```

AHORA TÚ · TU PROYECTO

Crea la base de datos de tu caso y ejecuta tu `schema.sql` y tu `data.sql` de la fase 1. Comprueba con `\dt` que están todas tus tablas.

7 Configurar la conexión

Toda la configuración de la aplicación vive en `src/main/resources/application.properties`.

```properties
# --- Conexion a la base de datos ----------------------------
spring.datasource.url=jdbc:postgresql://localhost:5432/biblioaula
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}

# --- JPA / Hibernate ----------------------------------------
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false

# --- Servidor -----------------------------------------------
server.port=8080
```

PROPERTIES

7.1 Qué hace cada línea

Propiedad	Para qué sirve
`spring.datasource.url`	Dónde está la base de datos: servidor, puerto y nombre
`spring.datasource.username` / `password`	Credenciales. Se leen de variables de entorno para no subirlas a GitHub
`spring.jpa.hibernate.ddl-auto`	Qué hace Hibernate con el esquema al arrancar. Ver la tabla siguiente
`spring.jpa.show-sql`	Muestra en consola el SQL que genera Hibernate: imprescindible mientras aprendes
`spring.jpa.open-in-view`	Se pone a `false` para que la sesión no siga abierta al devolver la respuesta
`server.port`	Puerto del servidor; cámbialo si el 8080 está ocupado

7.2 Los valores de ddl-auto

Valor	Qué hace	Cuándo usarlo
`validate`	Comprueba que tus entidades coinciden con las tablas y falla si no	**Nuestro caso**: la base de datos la diseñas tú
`update`	Modifica el esquema para encajar con las entidades	Nunca en este módulo: oculta errores de diseño
`create` / `create-drop`	Borra y recrea las tablas en cada arranque	Solo en pruebas desechables
`none`	No hace nada	En producción, con migraciones gestionadas aparte

7.3 Variables de entorno

Las credenciales no se escriben en el fichero. Se definen fuera y Spring las sustituye al arrancar:

TERMINAL

```
# Linux o macOS
export DB_USER=biblio_user
export DB_PASSWORD=biblio_pass

# Windows (PowerShell)
$env:DB_USER="biblio_user"
$env:DB_PASSWORD="biblio_pass"
```

En IntelliJ también puedes definirlas en *Run → Edit Configurations → Environment variables*.

Comprobación rápida Arranca la aplicación. Si en la consola aparece `HikariPool-1 - Start completed`, la conexión funciona. Hikari es el pool de conexiones que Spring Boot configura por ti.

AHORA TÚ · TU PROYECTO

Configura tu `application.properties` con tu base de datos, tus variables de entorno y `ddl-auto=validate`. Añade `application.properties` al `.gitignore` solo si guardas secretos en él; lo normal es dejarlo y no escribir nunca las contraseñas dentro.

8 El modelo del ejemplo

Antes de escribir entidades hay que tener claro el modelo. Estas son las cuatro tablas de BiblioAula.

Tabla	Contenido	Relaciones
`autores`	Quién escribe los libros	N:M con libros
`libros`	Título, ISBN, año y ejemplares disponibles	N:M con autores · 1:N con préstamos
`libro_autor`	Tabla intermedia de la relación N:M	Clave primaria compuesta
`prestamos`	Quién se lleva qué libro y cuándo lo devuelve	N:1 con libros

Tres situaciones que vas a repetir en tu proyecto:

- **Uno a muchos:** un libro tiene muchos préstamos y cada préstamo es de un libro.
- **Muchos a muchos:** un libro puede tener varios autores y un autor, varios libros.
- **Estado:** un préstamo está `ACTIVO`, `DEVUELTO` o `RETRASADO`. Eso se mapea con un enum.

9 Las entidades

Una entidad es una clase Java que representa una tabla. Cada instancia es una fila.

9.1 La entidad más simple

```java
@Entity
@Table(name = "autores")
@Getter @Setter
@NoArgsConstructor @AllArgsConstructor @Builder
public class Autor {

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

    @Column(nullable = false, length = 120)
    private String nombre;

    @Column(length = 60)
    private String pais;
}
```

Anotación	Qué hace
`@Entity`	Marca la clase como entidad gestionada por JPA
`@Table(name = …)`	Indica la tabla. Sin ella, Hibernate usa el nombre de la clase
`@Id`	Señala la clave primaria
`@GeneratedValue(IDENTITY)`	El valor lo genera la base de datos: en PostgreSQL, el `BIGSERIAL`
`@Column`	Ajusta nombre, longitud y obligatoriedad de la columna

Lombok en entidades Usa `@Getter`, `@Setter`, `@NoArgsConstructor`, `@AllArgsConstructor` y `@Builder`. **Nunca `@Data` en una entidad**: genera `equals`, `hashCode` y `toString` que recorren las relaciones y acaban provocando bucles infinitos y consultas que no esperabas.

9.2 Enums, fechas y decimales

```java
public enum EstadoPrestamo { ACTIVO, DEVUELTO, RETRASADO }

@Enumerated(EnumType.STRING)
@Column(nullable = false, length = 20)
private EstadoPrestamo estado;

private LocalDate fechaPrestamo;      // DATE
private LocalDateTime creadoEn;       // TIMESTAMP
private BigDecimal precio;            // NUMERIC(10,2), nunca double
```

`EnumType.STRING` guarda el nombre del estado. Si usas `ORDINAL`, se guarda un número y el día que añadas un valor en medio del enum, todos los datos antiguos pasan a significar otra cosa.

9.3 Campos de auditoría

JAVA

```java
@Column(updatable = false)
private LocalDateTime creadoEn;
private LocalDateTime actualizadoEn;

@PrePersist
void alCrear() {
    creadoEn = LocalDateTime.now();
    actualizadoEn = creadoEn;
}

@PreUpdate
void alActualizar() {
    actualizadoEn = LocalDateTime.now();
}
```

AHORA TÚ · TU PROYECTO

Escribe la entidad más sencilla de tu proyecto, la que no tiene relaciones todavía (una categoría, un plan, una sala, una plaza, un ingrediente, un lugar…) y arranca la aplicación. Si `ddl-auto=validate` no protesta, tu mapeo coincide con tu tabla.

10 Las relaciones

Aquí es donde se complica y donde se cometen casi todos los errores. Ve despacio.

10.1 Muchos a uno: el lado que manda

```java
@Entity
@Table(name = "prestamos")
@Getter @Setter @NoArgsConstructor
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(nullable = false, length = 120)
    private String lector;

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

    @Column(name = "fecha_devolucion")
    private LocalDate fechaDevolucion;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 20)
    private EstadoPrestamo estado;
}
```

- El lado `@ManyToOne` **es el dueño** de la relación: es el que tiene la columna con la clave ajena.
- `@JoinColumn(name = "libro_id")` indica esa columna. Si no la pones, Hibernate se inventa un nombre y `validate` falla.
- **Siempre** `fetch = FetchType.LAZY`: sin él, cada préstamo que cargues arrastra su libro aunque no lo necesites.

10.2 Uno a muchos: el lado inverso

```java
@Entity
@Table(name = "libros")
@Getter @Setter @NoArgsConstructor
public class Libro {

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

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

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

    private Integer anio;
    private Integer ejemplares;

    @OneToMany(mappedBy = "libro", fetch = FetchType.LAZY)
    private List<Prestamo> prestamos = new ArrayList<>();
}
```

`mappedBy = "libro"` significa: «esta relación ya está mapeada por el atributo `libro` de la entidad `Prestamo`». Sin `mappedBy`, Hibernate entiende que son dos relaciones distintas y busca una tabla intermedia que no existe.

> **¿Necesito el lado inverso?** Solo si vas a navegar de libro a préstamos. Una relación bidireccional que no se usa es una fuente de problemas gratis. Empieza por el `@ManyToOne` y añade el `@OneToMany` cuando te haga falta.

10.3 Muchos a muchos

```java
// En Libro
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(
    name = "libro_autor",
    joinColumns = @JoinColumn(name = "libro_id"),
    inverseJoinColumns = @JoinColumn(name = "autor_id")
)
private Set<Autor> autores = new HashSet<>();

// En Autor (lado inverso, opcional)
@ManyToMany(mappedBy = "autores", fetch = FetchType.LAZY)
private Set<Libro> libros = new HashSet<>();
```

> **Cuando la tabla intermedia tiene datos propios** Si la relación necesita guardar algo más (una cantidad, una fecha, un orden), deja de ser un `@ManyToMany` y pasa a ser **una entidad propia** con dos `@ManyToOne`. Es lo que ocurre con las líneas de un pedido, las reservas o las inscripciones.

10.4 Resumen rápido

Situación	Mapeo	Dónde está la clave ajena
Muchos préstamos, un libro	`@ManyToOne` + `@JoinColumn` en `Prestamo`	En la tabla `prestamos`
Un libro, muchos préstamos	`@OneToMany(mappedBy = "libro")` en `Libro`	No añade columna: es el lado inverso
Libros y autores, sin datos extra	`@ManyToMany` + `@JoinTable`	En la tabla intermedia
Libros y autores, con datos extra	Entidad propia con dos `@ManyToOne`	En la tabla intermedia, que ya es una entidad

AHORA TÚ · TU PROYECTO

Mapea las relaciones de tu proyecto y arranca de nuevo. Recuerda: todas `LAZY`, el `@JoinColumn` con el nombre exacto de tu columna y `mappedBy` en el lado inverso.

11 Los repositorios

Un repositorio es una interfaz. No se escribe su implementación: Spring Data la genera al arrancar.

```java
public interface LibroRepository extends JpaRepository<Libro, Long> {
}
```

Con esa línea ya tienes, sin escribir nada más:

Método heredado	Qué hace
`save(entidad)`	Inserta o actualiza
`findById(id)`	Devuelve un `Optional<Libro>`
`findAll()` / `findAll(Pageable)`	Todos los registros, con o sin paginación
`existsById(id)`	Comprueba si existe, sin traer la fila entera
`count()`	Número de filas
`deleteById(id)`	Borra

Optional, no null `findById` devuelve `Optional`. No uses `.get()` a pelo: usa `.orElseThrow(() -> new RecursoNoEncontradoException("No existe el libro " + id))`. Así el error es claro en lugar de un `NoSuchElementException` a medianoche.

11.1 Dónde se usa el repositorio

```java
@Service
@RequiredArgsConstructor
public class LibroService {

    private final LibroRepository libroRepository;   // inyección por constructor

    @Transactional(readOnly = true)
    public Libro buscarPorId(Long id) {
        return libroRepository.findById(id)
                .orElseThrow(() -> new RecursoNoEncontradoException("No existe el libro " + id));
    }
}
```

`@RequiredArgsConstructor` de Lombok genera el constructor con los campos `final`, y Spring lo usa para inyectar el repositorio. No hace falta `@Autowired`.

12 Las consultas

Aquí está el trabajo de verdad. Vamos a ver los formatos que se piden en el proyecto, del más simple al más elaborado.

12.1 Métodos derivados del nombre

Spring Data lee el nombre del método y escribe la consulta por ti. La estructura es `find | exists | count | delete` + `By` + campos + condiciones + `OrderBy`.

```java
public interface PrestamoRepository extends JpaRepository<Prestamo, Long> {

    List<Prestamo> findByEstado(EstadoPrestamo estado);

    List<Prestamo> findByLibroIdAndEstado(Long libroId, EstadoPrestamo estado);

    List<Prestamo> findByFechaPrestamoBetweenOrderByFechaPrestamoDesc(LocalDate desde, LocalDate
hasta);

    List<Prestamo> findByLectorContainingIgnoreCase(String texto);

    boolean existsByLibroIdAndEstado(Long libroId, EstadoPrestamo estado);

    long countByEstado(EstadoPrestamo estado);

    Optional<Prestamo> findFirstByLibroIdOrderByFechaPrestamoDesc(Long libroId);
}
```

Palabra clave	Ejemplo	SQL equivalente
`And` / `Or`	`findByEstadoAndLector`	`WHERE estado = ? AND lector = ?`
`Between`	`findByFechaPrestamoBetween`	`WHERE fecha_prestamo BETWEEN ? AND ?`
`LessThan` / `GreaterThan`	`findByAnioGreaterThan`	`WHERE anio > ?`
`Containing` + `IgnoreCase`	`findByTituloContainingIgnoreCase`	`WHERE LOWER(titulo) LIKE LOWER('%?%')`
`IsNull` / `IsNotNull`	`findByFechaDevolucionIsNull`	`WHERE fecha_devolucion IS NULL`
`In`	`findByEstadoIn`	`WHERE estado IN (?, ?)`
`OrderBy…Desc`	`findByEstadoOrderByFechaPrestamoDesc`	`ORDER BY fecha_prestamo DESC`
`First` / `Top3`	`findTop3ByOrderByAnioDesc`	`LIMIT 3`

El límite de los derivados Si el nombre del método empieza a parecer una frase de tres líneas, cambia a `@Query`. Son cómodos para dos o tres condiciones, no para diez.

12.2 JPQL con @Query

JPQL se parece a SQL pero trabaja con **entidades y atributos**, no con tablas y columnas. Fíjate: `Prestamo p`, no `prestamos`.

```java
@Query("""
    SELECT p FROM Prestamo p
    WHERE p.estado = :estado AND p.fechaPrestamo < :limite
    ORDER BY p.fechaPrestamo ASC
""")
List<Prestamo> buscarRetrasados(@Param("estado") EstadoPrestamo estado,
                                @Param("limite") LocalDate limite);

@Query("""
    SELECT l.titulo, COUNT(p)
    FROM Prestamo p JOIN p.libro l
    WHERE p.fechaPrestamo BETWEEN :desde AND :hasta
    GROUP BY l.titulo
    ORDER BY COUNT(p) DESC
""")
List<Object[]> rankingDeLibros(@Param("desde") LocalDate desde,
                               @Param("hasta") LocalDate hasta);
```

- `JOIN p.libro l` navega por la **relación** de la entidad, no por una clave ajena.
- Los parámetros van con `:nombre` y se enlazan con `@Param`.
- Los bloques de texto de Java (`"""`) hacen que las consultas largas se lean bien.

12.3 SQL nativo

Cuando necesitas algo propio de PostgreSQL (`DATE_TRUNC`, `EXTRACT`, `INTERVAL`), usas SQL nativo. Aquí sí hablas de tablas y columnas.

```sql
@Query(value = """
    SELECT DATE_TRUNC('month', fecha_prestamo) AS mes, COUNT(*) AS total
    FROM prestamos
    WHERE fecha_prestamo BETWEEN :desde AND :hasta
    GROUP BY mes
    ORDER BY mes
""", nativeQuery = true)
List<Object[]> prestamosPorMes(@Param("desde") LocalDate desde,
                               @Param("hasta") LocalDate hasta);
```

12.4 Consultas que modifican

```java
@Modifying(clearAutomatically = true)
@Transactional
@Query("""
    UPDATE Prestamo p SET p.estado = :nuevo
    WHERE p.estado = :actual AND p.fechaPrestamo < :limite
""")
int marcarRetrasados(@Param("actual") EstadoPrestamo actual,
                     @Param("nuevo") EstadoPrestamo nuevo,
                     @Param("limite") LocalDate limite);
```

- `@Modifying` avisa de que la consulta no devuelve filas, sino el número de registros afectados.
- `clearAutomatically = true` limpia el contexto de persistencia para que no te quedes con entidades desactualizadas en memoria.
- Necesita una transacción: `@Transactional` aquí o en el servicio que la llama.

12.5 Proyecciones: traer solo lo necesario

```java
// Por interfaz: Spring Data la implementa sola
public interface LibroResumen {
    Long getId();
    String getTitulo();
    Integer getEjemplares();
}

List<LibroResumen> findByAnioGreaterThan(Integer anio);

// Por clase, con un record y SELECT new
public record RankingLibroDto(String titulo, Long prestamos) {}

@Query("""
    SELECT new es.safareyes.biblioaula.dto.RankingLibroDto(l.titulo, COUNT(p))
    FROM Prestamo p JOIN p.libro l
    GROUP BY l.titulo
""")
List<RankingLibroDto> ranking();
```

Una proyección evita traer la fila entera cuando solo necesitas tres campos, y te ahorra convertir `Object[]` a mano.

12.6 Paginación y ordenación

```java
Page<Libro> findByTituloContainingIgnoreCase(String texto, Pageable pageable);

// En el servicio
Pageable pageable = PageRequest.of(0, 10, Sort.by("titulo").ascending());
Page<Libro> pagina = libroRepository.findByTituloContainingIgnoreCase("el", pageable);

pagina.getContent();        // los 10 libros de esta página
pagina.getTotalElements();  // cuántos hay en total
pagina.getTotalPages();
```

Cuando lo expongas en un endpoint, Spring construye el `Pageable` solo a partir de `?page=0&size=10&sort=titulo,asc`.

12.7 Filtros que cambian: Specification

Cuando el usuario puede combinar filtros opcionales, no escribas un método por combinación. Extiende `JpaSpecificationExecutor` y construye la consulta según lo que llegue.

```java
public interface LibroRepository
        extends JpaRepository<Libro, Long>, JpaSpecificationExecutor<Libro> { }

public static Specification<Libro> conFiltros(String texto, Integer anioMinimo) {
    return (root, query, cb) -> {
        List<Predicate> p = new ArrayList<>();
        if (texto != null)
            p.add(cb.like(cb.lower(root.get("titulo")), "%" + texto.toLowerCase() + "%"));
        if (anioMinimo != null)
            p.add(cb.greaterThanOrEqualTo(root.get("anio"), anioMinimo));
        return cb.and(p.toArray(new Predicate[0]));
    };
}
```

12.8 El problema N+1

Si pides 50 préstamos y luego, en un bucle, accedes a `prestamo.getLibro().getTitulo()`, Hibernate lanza **una consulta por cada préstamo**: 51 en total. Se ve en la consola con `show-sql` activado.

```java
// Solución 1: traerlo todo de una vez
@Query("SELECT p FROM Prestamo p JOIN FETCH p.libro WHERE p.estado = :estado")
List<Prestamo> buscarConLibro(@Param("estado") EstadoPrestamo estado);

// Solución 2: el mismo efecto sin tocar la consulta
@EntityGraph(attributePaths = "libro")
List<Prestamo> findByEstado(EstadoPrestamo estado);
```

Cuidado con dos colecciones `JOIN FETCH` de dos listas a la vez provoca `MultipleBagFetchException`. Usa `Set` en una de ellas o resuélvelo en dos consultas.

AHORA TÚ · TU PROYECTO

Escribe en tus repositorios, como mínimo, una consulta de cada tipo: derivada, JPQL con `JOIN`, nativa de informe, una de modificación, una proyección y una paginada. Son las que te piden en la entrega.

13 Probar la conexión y las consultas

Nunca des por buena una consulta que no has ejecutado. Tienes tres formas de comprobarlo, de menos a más formal.

13.1 Leer la consola

Con `spring.jpa.show-sql=true` y `format_sql=true`, cada consulta aparece en la consola. Busca estas señales al arrancar:

Mensaje	Qué significa
`HikariPool-1 - Start completed`	La conexión con PostgreSQL funciona
`Tomcat started on port 8080`	El servidor web está escuchando
`Started BiblioAulaApplication in 2.1 seconds`	Arranque completo y correcto
`Schema-validation: missing table [libros]`	Hibernate no encuentra una tabla: revisa el script o el nombre en `@Table`

13.2 Un CommandLineRunner para probar rápido

Es la forma más directa de ejecutar consultas mientras desarrollas. Se borra o se desactiva cuando ya no hace falta.

```java
@Component
@RequiredArgsConstructor
@Slf4j
public class PruebaConsultas implements CommandLineRunner {

    private final LibroRepository libroRepository;
    private final PrestamoRepository prestamoRepository;

    @Override
    public void run(String... args) {
        log.info("Libros en la base de datos: {}", libroRepository.count());

        libroRepository.findAll()
                .forEach(l -> log.info("  {} ({})", l.getTitulo(), l.getAnio()));

        List<Prestamo> activos = prestamoRepository.findByEstado(EstadoPrestamo.ACTIVO);
        log.info("Préstamos activos: {}", activos.size());

        prestamoRepository.rankingDeLibros(LocalDate.of(2026, 1, 1), LocalDate.now())
                .forEach(fila -> log.info("  {} -> {} préstamos", fila[0], fila[1]));
    }
}
```

> **Actívalo solo cuando lo necesites** Añade `@Profile("dev")` a la clase y arranca con `spring.profiles.active=dev`. Así no se ejecuta en el resto de entornos.

13.3 Tests de repositorio

La forma seria. `@DataJpaTest` levanta solo la capa de persistencia y deshace los cambios al terminar cada prueba.

JAVA

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class PrestamoRepositoryTest {

    @Autowired
    private PrestamoRepository prestamoRepository;

    @Test
    void encuentraLosPrestamosActivos() {
        List<Prestamo> activos = prestamoRepository.findByEstado(EstadoPrestamo.ACTIVO);

        assertThat(activos).isNotEmpty();
        assertThat(activos).allMatch(p -> p.getFechaDevolucion() == null);
    }
}
```

`@AutoConfigureTestDatabase(replace = NONE)` es importante: sin esa línea, Spring intentaría sustituir tu PostgreSQL por una base de datos en memoria y las consultas nativas fallarían.

13.4 Un endpoint de comprobación

Si quieres verlo desde el navegador o desde Postman antes de montar la capa REST completa:

JAVA

```
@RestController
@RequestMapping("/api/v1/pruebas")
@RequiredArgsConstructor
public class PruebaController {

    private final LibroRepository libroRepository;

    @GetMapping("/libros")
    public List<Libro> listar() {
        return libroRepository.findAll();
    }
}
```

Esto es temporal Un controlador que inyecta el repositorio y devuelve entidades rompe la separación de capas. Sirve para comprobar que todo funciona el primer día; en la entrega, entre medias van el servicio y los DTOs.

AHORA TÚ · TU PROYECTO

Prueba tus consultas con un `CommandLineRunner` y deja escrito al menos un test con `@DataJpaTest` de la consulta más importante de tu caso.

14 Errores frecuentes

Casi todos los errores del primer día están en esta tabla. Antes de preguntar, búscalo aquí.

Mensaje	Causa	Solución
`Connection refused`	PostgreSQL no está arrancado o el puerto no es el 5432	Arranca el servicio y comprueba la URL del `application.properties`
`password authentication failed`	Usuario o contraseña incorrectos	Revisa las variables de entorno; comprueba que entras con `psql` con esas credenciales
`Schema-validation: missing table`	La tabla no existe o el nombre no coincide	Ejecuta el script; revisa `@Table(name = …)`
`missing column [fecha_prestamo]`	El atributo no coincide con la columna	Añade `@Column(name = "fecha_prestamo")` o renombra
`No qualifying bean of type`	La clase está fuera del paquete que escanea Spring, o le falta la anotación	Muévela bajo el paquete raíz y marca `@Service`, `@Repository` o `@Component`
`LazyInitializationException`	Accedes a una relación perezosa fuera de la transacción	Trae el dato con `JOIN FETCH` o `@EntityGraph`, o resuélvelo dentro del servicio
`InvalidDataAccessApiUsage: No EntityManager with actual transaction`	Una consulta `@Modifying` sin transacción	Añade `@Transactional` en el repositorio o en el servicio
`MultipleBagFetchException`	Dos `JOIN FETCH` de listas en la misma consulta	Usa `Set` en una de las colecciones o parte la consulta en dos
`StackOverflowError` al imprimir	`@Data` o `toString` en entidades con relaciones bidireccionales	Quita `@Data`; usa `@ToString(exclude = …)`
`Port 8080 was already in use`	Otra aplicación ocupa el puerto	Cambia `server.port` o cierra el proceso anterior

15 Chuleta

Todo lo importante del manual en una página.

Anotaciones de entidad

Anotación	Para qué
`@Entity` · `@Table(name)`	Clase que representa una tabla
`@Id` · `@GeneratedValue(IDENTITY)`	Clave primaria autonumérica
`@Column(name, nullable, length, unique)`	Ajustes de la columna
`@Enumerated(EnumType.STRING)`	Guardar el nombre del enum, no su número
`@ManyToOne(fetch = LAZY)` · `@JoinColumn(name)`	Lado dueño: lleva la clave ajena
`@OneToMany(mappedBy)`	Lado inverso: no añade columna
`@ManyToMany` · `@JoinTable`	Relación con tabla intermedia sin datos propios
`@PrePersist` · `@PreUpdate`	Rellenar campos de auditoría

Consultas

Formato	Cuándo usarlo
Método derivado	Dos o tres condiciones simples
`@Query` con JPQL	`JOIN` entre entidades, agregados, consultas largas
`@Query(nativeQuery = true)`	Funciones propias de PostgreSQL, informes
`@Modifying`	`UPDATE` o `DELETE` masivos
Proyección por interfaz o `SELECT new`	Traer solo algunos campos
`Pageable` y `Page<T>`	Listados largos
`Specification`	Filtros opcionales combinables
`@EntityGraph` o `JOIN FETCH`	Evitar el problema N+1

Configuración imprescindible

PROPERTIES

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mi_bd
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.open-in-view=false
```

Reglas que no se negocian

- La base de datos la diseñas tú: `ddl-auto=validate`, nunca `update`.
- Todas las relaciones `LAZY`.
- Nada de `@Data` en entidades.
- Inyección por constructor, nunca `@Autowired` en atributos.
- Las contraseñas y secretos, en variables de entorno.
- Una consulta que no has ejecutado no está terminada.