Editorial con Spring Boot, H2 y Swagger

1. Objetivo del ejercicio

El objetivo es construir una API REST con Spring Boot que:

1. Trabaje con una base de datos relacional H2.
2. La base de datos contiene **tres tablas** (ya creadas desde `schema.sql`):
   - `author` (autor).
   - `book` (libro). *Autor y libro se relacionan.*
   - `publisher` (editorial).
3. El modelo contempla la relación **1:N** entre `author` y `book`:
   - Un autor tiene muchos libros.
   - Un libro pertenece a un único autor.
4. El modelo contempla a `publisher` como una tabla **independiente**, sin relaciones con las anteriores.
5. Debes exponer *endpoints REST* para ***gestionar*** estas entidades.
6. Debes documentar la API con **Swagger/OpenAPI** (springdoc-openapi).

El paquete raíz será:

```
com.docencia.aed
```

La estructura por capas que debes crear será:

```
com.docencia.aed
 ├─ entity        # Entidades JPA (Author, Book, Publisher)
 ├─ repository    # Repositorios Spring Data JPA
 ├─ service       # Interfaces de servicio
 │   └─ impl      # Implementaciones de los servicios
 └─ controller    # Controladores REST
```

Vas a trabajar con un proyecto Maven con Spring Boot ya configurado con:

- Spring Web
- Spring Data JPA
- H2
- springdoc-openapi (Swagger UI)

Y en `src/main/resources` ya tienes:

- `application.properties` con la configuración de H2 y de inicialización.
- `schema.sql` con la definición de las tablas.
- `data.sql` con datos de ejemplo.

Configuración actual en `application.properties`:

```
spring.application.name=editorial

# Configuración de H2 en memoria
spring.datasource.url=jdbc:h2:mem:editorialdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON
_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

# Spring define e inserta los datos en la bbdd
spring.jpa.hibernate.ddl-auto=none
#spring.jpa.hibernate.ddl-auto=create-drop
#spring.sql.init.mode=never
spring.jpa.show-sql=true

# Consola H2 activa en /h2-console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

---

1. ¿Quién crea las tablas y carga los datos?

En este proyecto:

- Las **tablas** se crean con `schema.sql`.
- Los **datos iniciales** se cargan con `data.sql`.

Spring Boot, al detectar que usamos una base de datos **embebida** (H2) y que existen estos ficheros en `src/main/resources`, los ejecuta automáticamente al arrancar:

1. Ejecuta `schema.sql` → crea las tablas.
2. Ejecuta `data.sql` → inserta datos de ejemplo.

Esto sucede **aunque** tengamos:

```
spring.jpa.hibernate.ddl-auto=none
```

Porque esta propiedad afecta solo a **Hibernate/JPA**, no a la ejecución de scripts SQL.

---

2. ¿Qué hace `spring.jpa.hibernate.ddl-auto`?

Esta propiedad controla cómo Hibernate/JPA gestiona el esquema de la base de datos **a partir de las entidades**:

- `none`
  Hibernate **no toca** el esquema (no crea ni modifica tablas).
  → El esquema debe existir ya (por ejemplo, porque lo crea `schema.sql`).
- `create`
  Borra (si puede) y crea de nuevo todas las tablas al arrancar, a partir de las entidades.
- `create-drop`
  Igual que `create`, pero además **borra las tablas al parar** la aplicación.
- `update`
  Intenta adaptar el esquema a las entidades (añadiendo columnas/tablas si hace falta) sin borrar los datos existentes.

En nuestra configuración actual usamos:

```
spring.jpa.hibernate.ddl-auto=none
```

Esto significa:

> "La BBDD la define `schema.sql`. Hibernate solo la usa, no la crea ni la modifica".

La línea comentada:

```
#spring.jpa.hibernate.ddl-auto=create-drop
```

es un ejemplo de alternativa: si la descomentas y quitas `schema.sql`, Hibernate crearía y borraría las tablas automáticamente según las entidades.

---

3. ¿Qué hace `spring.sql.init.mode`?

Spring Boot tiene otro mecanismo de inicialización, independiente de JPA/Hibernate: la ejecución de scripts SQL (`schema.sql`, `data.sql`, etc.).

Por defecto:

- Si la base de datos es **embebida** (H2, HSQL, Derby), Spring Boot asume:

```
spring.sql.init.mode=embedded
```

y busca `schema.sql` y `data.sql` en el classpath.

Si quisieras **evitar** que se ejecuten esos scripts, podrías poner:

```
spring.sql.init.mode=never
```

En el `application.properties` actual lo tienes comentado:

```
#spring.sql.init.mode=never
```

Eso significa que estamos usando el comportamiento por defecto:
→ H2 + `schema.sql` + `data.sql` = se ejecutan al arrancar.

---

4. ¿Qué hace `spring.jpa.show-sql=true`?

Esta propiedad sirve para **ver las consultas SQL** que ejecuta Hibernate en la consola:

```
spring.jpa.show-sql=true
```

Gracias a esto, cuando uses los repositorios o los servicios verás en el log sentencias del tipo:

```
select a1_0.id, a1_0.country, a1_0.name from author a1_0
```

> **Importante**: la base de datos se crea y se rellena automáticamente con `schema.sql` y `data.sql` al arrancar la aplicación. No necesitas crear la BBDD a mano, y **NO SE VA A ACTUALIZAR CON LAS ENTIDDADES DADO QUE NO ESTA INDICADO EN EL FICHERO aplications.properties**.

Consulta la `bbdd` su estado ejecutando:

```
mvn clean spring-boot:run
```

Accede a `http://localhost:8080/h2-console/`, con las credenciales:

- JDBC URL: `jdbc:h2:mem:editorialdb`
- User Name: `sa`
- Password: `no lleva contraseña`

2. Estructura que vamos a crear

El proyecto **NO contiene** todavía:

- Entidades JPA
- Repositorios
- Servicios
- Controladores

Tu trabajo consiste en:

1. Crear las entidades JPA (`Author`, `Book`, `Publisher`) con las relaciones.
2. Crear los repositorios.
3. Implementar los servicios (interfaces + implementación).
4. Implementar los controladores REST.
5. Añadir documentación Swagger a los controladores (anotaciones).

---

3. Esquema de la base de datos.

El esquema de la base de datos esta definido en `schema.sql` e inicializado con `data.sql`.

3.1. Tabla `author`

Tabla `author`:

- `id` (PK)
- `name` (obligatorio)
- `country` (opcional)

Definición:

```
CREATE TABLE author (
    id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    country VARCHAR(100)
);
```

Ejemplo de datos de prueba:

```
INSERT INTO author (name, country) VALUES ('Gabriel García Márquez',
'Colombia');
INSERT INTO author (name, country) VALUES ('Isabel Allende', 'Chile');
```

3.2. Tabla `book`

Tabla `book`:

- `id` (PK)
- `title` (obligatorio)
- `publication_year` (opcional)
- `author_id` (FK → `author.id`)

Relación:

- `Author` 1 — N `Book`

Definición:

```
CREATE TABLE book (
    id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    publication_year INT,
    author_id BIGINT NOT NULL,
    CONSTRAINT fk_book_author FOREIGN KEY (author_id) REFERENCES author(id)
);
```

Ejemplo de datos de prueba:

```
INSERT INTO book (title, publication_year, author_id) VALUES ('Cien años de soledad', 1967, 1);
INSERT INTO book (title, publication_year, author_id) VALUES ('El coronel no tiene quien le escriba', 1961, 1);
INSERT INTO book (title, publication_year, author_id) VALUES ('La casa de los espíritus', 1982, 2);
```

> **Nota**: Puedes llamar a la propiedad `publicationYear` y mapearla a la columna `publication_year` usando `@Column(name = "publication_year")`.

3.3. Tabla `publisher`

Tabla `publisher`:

- `id` (PK)
- `name` (obligatorio)
- `city` (opcional)

Definición:

```
CREATE TABLE publisher (
    id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name VARCHAR(150) NOT NULL,
    city VARCHAR(100)
);
```

Ejemplo de datos de prueba:

```
INSERT INTO publisher (name, city) VALUES ('Editorial Sur', 'Madrid');
INSERT INTO publisher (name, city) VALUES ('Norte Libros', 'Barcelona');
```

En esta versión, `Publisher` es **independiente** (no tiene relaciones con `Author` ni `Book`).

> **Nota**: Usa `@GeneratedValue`, en los `id` para hacer las `Entitys` autoincrementales.

---

4. Endpoints REST

4.1. Autores (`/authors`)

- `GET /authors`
  Devuelve la lista de autores.
- `GET /authors/{id}`
  Devuelve un autor por id.
- `POST /authors`
  Crea un nuevo autor.
- `GET /authors/{id}/books`
  Devuelve todos los libros del autor con id dado.

4.2. Libros (`/books`)

- `GET /books`
  Devuelve la lista de libros.
- `GET /books/{id}`
  Devuelve un libro por id.
- `POST /books?authorId={idAutor}`
  Crea un libro asociado al autor indicado por `authorId`.

4.3. Editoriales (`/publishers`)

- `GET /publishers`
  Devuelve la lista de editoriales.
- `POST /publishers`
  Crea una nueva editorial.

---

5. Construye la solución paso a paso

La base del proyecto ya está creada. Sigue estos pasos dentro del proyecto que se te ha proporcionado.

Paso 1: Revisar la estructura de paquetes

En `src/main/java/com/docencia/aed` ya tienes creados los paquetes vacíos:

- `entity`
- `repository`
- `service`
- `service/impl`
- `controller`

Si no existieran, créalos.

Paso 2: Crear las entidades JPA

1. **`Author`** (`com.docencia.aed.entity.Author`)
    - Campos: `id`, `name`, `country`.
    - Anotaciones: `@Entity`, `@Table(name = "author")`, `@Id`.
    - Relación 1:N con `Book`:

```
@OneToMany(mappedBy = "author")
private List<Book> books;
```

2. **`Book`** (`com.docencia.aed.entity.Book`)
    - Campos: `id`, `title`, `publicationYear`, `author`.
    - Mapeo de la columna del año:

```
@Column(name = "publication_year")
private Integer publicationYear;
```

    - Relación N:1 con `Author`:

```
@ManyToOne(optional = false)
@JoinColumn(name = "author_id")
private Author author;
```

3. **`Publisher`** (`com.docencia.aed.entity.Publisher`)
    - Campos: `id`, `name`, `city`.
    - Sin relaciones (tabla independiente).

> Recuerda correctamente las propiedades: `@GeneratedValue`, `@Column`, etc.

Paso 3: Crear los repositorios

En `com.docencia.aed.repository`:

- `AuthorRepository extends JpaRepository<Author, IDENTIFICA_TIPO_COLUMNA>`
- `BookRepository extends JpaRepository<Book, IDENTIFICA_TIPO_COLUMNA>`
    - Añadir método para buscar por autor:

```
List<Book> findByAuthorId(IDENTIFICA_TIPO_COLUMNA authorId);
```

- `PublisherRepository extends JpaRepository<Publisher, IDENTIFICA_TIPO_COLUMNA>`

Paso 4: Crear los servicios

En `com.docencia.aed.service` define las **interfaces** (puedes seguir esta convención):

- `IAuthorService`
- `IBookService`
- `IPublisherService`

Cada interfaz debe declarar los métodos necesarios, por ejemplo para `IAuthorService`:

```
List<Author> findAll();
Author findById(IDENTIFICA_TIPO_COLUMNA id);
Author create(Author author);
List<Book> findBooksByAuthor(IDENTIFICA_TIPO_COLUMNA authorId);
```

En `com.docencia.aed.service.impl` crea las **implementaciones** usando los repositorios.

Paso 5: Crear los controladores REST

En `com.docencia.aed.controller`:

- `AuthorController` con rutas `/authors`.
- `BookController` con rutas `/books`.
- `PublisherController` con rutas `/publishers`.

> **Nota**: Cada controlador deberá usar **la interfaz de servicio** correspondiente (`IAuthorService`, `IBookService`, etc.). de la capa de servicios. Recuerda las anotaciones que debes de colocar para que se inyecte de forma correcta.

Ejemplo de mapeo de un método:

```
@GetMapping("/authors")
public List<Author> getAllAuthors() {
    return authorService.findAll();
}
```

Paso 6: Añadir Swagger/OpenAPI

1. La dependencia `springdoc-openapi-starter-webmvc-ui` ya está en el `pom.xml`.
2. Añade anotaciones en la clase principal (`EditorialApplication`):

```
@OpenAPIDefinition(
    info = @Info(
        title = "API de Autores, Libros y Editoriales",
        version = "1.0",
        description = "Ejercicio guiado en Navidad"
    )
)
@SpringBootApplication
public class EditorialApplication { ... }
```

3. En los controladores, añade:
   - `@Tag` en la clase.
   - `@Operation` en cada método.
   - `@ApiResponses` (opcional) para documentar códigos de respuesta.
4. Comprueba Swagger UI en:

- `http://localhost:8080/swagger-ui.html`
  o
- `http://localhost:8080/swagger-ui/index.html`

Mucha suerte, y recuerda que *sólo debes de tener claro lo que tienes que construir*.