

## 06 - JPA

Hasta ahora, hemos utilizado JdbcTemplate para conectarnos a la bbdd y tratar los datos, aunque no es la forma más habitual de hacerlo.

La persistencia de datos en aplicaciones es un aspecto fundamental para el desarrollo de sistemas robustos y escalables. En el entorno de Java, JPA (Java Persistence Api) (<https://www.oracle.com/java/technologies/persistence-jsp.html>) ha surgido como una solución estándar para el mapeo objeto-relacional (ORM), ofreciendo a los desarrolladores una forma eficiente y coherente de interactuar con bases de datos relacionales utilizando objetos Java.

JPA facilita la representación de entidades de persistencia como objetos en el código Java, permitiendo su almacenamiento y recuperación en una base de datos de manera transparente. Al abstraer la lógica de persistencia, JPA simplifica considerablemente el manejo de la capa de acceso a datos, promoviendo un código más limpio, mantenible y portable entre diferentes proveedores de bases de datos.

JPA define la gestión de datos en aplicaciones mediante la representación de objetos en una base de datos relacional. JPA proporciona un conjunto de interfaces y clases abstractas que permiten a los desarrolladores interactuar con la base de datos de una manera orientada a objetos, sin tener que preocuparse por detalles específicos de la implementación subyacente de la base de datos.

Existen diferentes implementaciones de JPA, entre las que destacan:

- Hibernate (<https://hibernate.org/>): Es una de las implementaciones JPA más populares. Ofrece funcionalidades avanzadas y flexibilidad en el mapeo objeto-relacional, además de herramientas adicionales que simplifican el desarrollo y la gestión de la base de datos.
- EclipseLink (<https://eclipse.dev/eclipselink/>): Otra implementación JPA robusta, potente y de alto rendimiento. Ofrece características de mapeo avanzadas, soporte para estándares JPA y herramientas de persistencia útiles.
- Apache OpenJPA (<https://openjpa.apache.org/>): Una implementación JPA que proviene del proyecto OpenJPA de Apache. Proporciona funcionalidades completas de JPA, cumpliendo con los estándares de la API().

Estas implementaciones ofrecen funcionalidades similares en términos de mapeo objeto-relacional y operaciones CRUD, pero pueden diferir en sus características específicas, herramientas adicionales y rendimiento en ciertos contextos.

Al utilizar JPA, los desarrolladores pueden escribir código independiente de la base de datos subyacente, lo que les permite cambiar de proveedor de bases de datos sin tener que modificar considerablemente el código de la aplicación, lo que resulta en sistemas más flexibles y mantenibles a largo plazo.

En Spring Boot tenemos opciones para trabajar con JPA. Si bien Spring Data JPA (<https://spring.io/projects/spring-data-jpa>) es una forma común y conveniente de comenzar, podemos personalizar el enfoque dependiendo de las necesidades específicas del proyecto.

Por ejemplo, además de Spring Data JPA, podríamos optar por configurar JPA manualmente agregando las dependencias necesarias de forma individual. Esto nos ofrece un mayor control sobre cada componente que utilizamos.

Además, podemos elegir implementaciones específicas de JPA según nuestras preferencias o requisitos del proyecto. Aunque *Hibernate* es la implementación JPA predeterminada en Spring Boot, podemos optar por otras implementaciones como *EclipseLink* o *Apache OpenJPA*.

## Añadiendo la dependencia y configurando la conexión

En nuestro caso, vamos a utilizar la configuración común de JPA en un proyecto Spring. Al incluir *Spring Data JPA* en un proyecto Spring Boot, obtenemos una configuración predeterminada que facilita la interacción con bases de datos utilizando JPA.

Este starter no solo proporciona *Spring Data JPA*, sino que también configura automáticamente otros componentes esenciales como *Spring JDBC*, *Spring Transactions*, *Spring AOP* y *Spring Aspects*, si se requieren para el contexto de persistencia de datos.

Como siempre, lo primero que tenemos que hacer es añadir la dependencia de JPA a nuestro archivo pom.xml:

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-jpa</artifactId>
4 </dependency>
```

Ahora ya estamos listos para usar JPA.

En nuestro archivo *application.properties* ya tenemos la configuración de la conexión a la bbdd, con lo que no habría que tocar nada. En cualquier caso, vamos a añadir un par de opciones que nos serán útiles:

```
1 spring.datasource.url=jdbc:mariadb://localhost:3306/movies
2 spring.datasource.username=root
3 spring.datasource.password=root
4 spring.jpa.hibernate.ddl-auto=none
5 # Habilitar logs de consultas SQL
6 spring.jpa.show-sql=true
```

La propiedad **spring.jpa.hibernate.ddl-auto** es una configuración en aplicaciones Spring que determina cómo Hibernate maneja la creación y actualización de la estructura de la base de datos. Tiene varios valores que permiten definir cómo se realiza esta gestión:

- **none:** Este valor desactiva cualquier acción automática de creación o actualización de la base de datos por parte de Hibernate. Con este modo, Hibernate no hace ningún cambio en la estructura de la base de datos existente.
- **create:** Al utilizar este valor, Hibernate creará la estructura de la base de datos si no existe. Si la base de datos ya está presente, Hibernate la eliminará y la volverá a crear, lo que conlleva la pérdida de datos existentes.
- **create-drop:** Similar a create, Hibernate crea la estructura de la base de datos si no existe. Sin embargo, al cerrar la sesión de Hibernate o detener la aplicación, Hibernate eliminará la base de datos, lo que puede ser útil para entornos de desarrollo o pruebas.
- **update:** Este valor indica a Hibernate que actualice la estructura de la base de datos según los cambios en las entidades. No elimina los datos existentes, pero puede modificar o eliminar columnas, tablas, etc., para reflejar los cambios en el modelo de datos.

En nuestro caso, estamos conectándonos a una bbdd que ya existe. Si quisiésemos que JPA creara la bbdd de forma automática, tendríamos que cambiar el valor de `spring.jpa.hibernate.ddl-auto` a `create`, por ejemplo.

A veces, tu configuración del SGBD no permite crear la bbdd de forma automática. Una solución sencilla es crear la bbdd desde fuera y ejecutar Spring para crear las tablas.

Aunque Hibernate puede inferir el dialecto de la base de datos basándose en la `URL()` de conexión, especificar explícitamente el dialecto a través de `spring.jpa.properties.hibernate.dialect` es una práctica recomendada para asegurar una configuración precisa y completa. Por ejemplo, en nuestro caso podríamos añadir:

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect
spring.jpa.hibernate.ddl-auto=create
```

La última línea (`spring.jpa.show-sql=true`) es una configuración que se utiliza en aplicaciones Spring con Hibernate como implementación JPA. Cuando se establece a `true`, esta propiedad le indica a Hibernate que imprima las consultas SQL generadas por la aplicación en la consola, lo que nos puede ser útil mientras estamos en desarrollo.

## Estructura

Vamos a añadir un nuevo package **dao.db.jpa** con la implementación JPA de nuestros DAOs. Dentro, tendremos (además de las implementaciones concretas) otros 3 packages:

- **dao.db.jpa.entity:** Entidades JPA
- **dao.db.jpa.mapper:** Mapeadores entre las entidades JPA y nuestro modelo de dominio
- **dao.db.jpa.repository:** Repositorios JPA

## Entidades

JPA utiliza sus propias entidades que mapean una tabla de la bbdd. Para ello, sólo tenemos que añadir la anotación **@Entity** a nuestras entidades.

Para distinguir estas entidades de nuestros modelos de dominio les añadiremos *Entity* al nombre. Así, por ejemplo, la entidad de libros quedaría:

```
1 @Entity
2 @Data
3 @NoArgsConstructor
4 public class BookEntity {
```

JPA intentará conectar la entidad con la tabla correspondiente. El problema es que nuestras entidades llevan el sufijo *Entity*, con lo que no encontrará ninguna tabla llamada *BookEntity*. Por suerte, la solución es muy sencilla: indicarle el nombre de la tabla mediante la anotación **@Table**:

```
1 @Entity
2 @Table(name = "books")
3 @Data
4 @NoArgsConstructor
5 public class BookEntity {
```

Lo siguiente será añadir anotaciones a nuestros campos para indicar algunas propiedades para que JPA sea capaz de mapear directamente desde la bbdd. En primer lugar, añadiremos **@ID** y **@GeneratedValue**.

La anotación **@GeneratedValue** en JPA, junto con **@Id**, se utiliza para especificar cómo se generan los valores para una clave primaria en una entidad persistente. En particular, **@GeneratedValue** se usa para indicar la estrategia que se utilizará para generar los valores de las claves primarias de manera automática por parte de la base de datos.

La estrategia **GenerationType.IDENTITY** especifica que la generación de valores de clave primaria se realizará utilizando una columna de identidad de la base de datos, lo que significa que la base de datos se encargará de generar automáticamente valores únicos para la clave primaria cuando se inserten nuevas filas en la tabla asociada a la entidad:

```

1  @Entity
2  @Table(name = "books")
3  @Data
4  @NoArgsConstructor
5  public class BookEntity {
6
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10     private String isbn;
11     private String titleEs;
12     ...

```

Fijate en el campo *titleEs*. JPA no es capaz de asociar el atributo a un campo de la tabla. Normal, ya que en la bbdd el campo se llama *title\_es*. Por suerte, contamos con la anotación **@Column** para indicarle el nombre de la columna en la bbdd:

```

1  @Entity
2  @Table(name = "books")
3  @Data
4  @NoArgsConstructor
5  public class BookEntity {
6
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10     private String isbn;
11     @Column(name = "title_es")
12     private String titleEs;
13     ...

```

De esta forma, nuestra entidad BookEntity quedaría:

```

1  @Entity
2  @Table(name = "books")
3  @Data
4  @NoArgsConstructor
5  public class BookEntity {
6
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10     private String isbn;
11     @Column(name = "title_es")
12     private String titleEs;
13     @Column(name = "title_en")
14     private String titleEn;
15     @Column(name = "synopsis_es")
16     private String synopsisEs;
17     @Column(name = "synopsis_en")
18     private String synopsisEn;
19     private BigDecimal price;
20     private float discount;
21     private String cover;
22
23     private PublisherEntity publisher;
24     private CategoryEntity category;
25     private List<AuthorEntity> authors;
26     private List<GenreEntity> genres;
27 }

```

Y, por ejemplo, CategoryEntity:

```

1  @Entity
2  @Table(name = "categories")
3  @Data
4  @NoArgsConstructor
5  public class CategoryEntity {
6
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10     @Column(name = "name_es")
11     private String nameEs;
12     @Column(name = "name_en")
13     private String nameEn;
14     private String slug;
15 }

```

## Relaciones entre entidades

En JPA podemos representar relaciones entre entidades para facilitar el tratamiento de datos. Para hacerlo, contamos con una serie de anotaciones según el tipo de relación que queramos representar:

- **@OneToOne**: Define una relación uno a uno entre dos entidades.
- **@OneToMany**: Define una relación de uno a muchos, donde una entidad tiene una asociación con múltiples entidades de otro tipo.
- **@ManyToOne**: Establece una relación muchos a uno, indicando que varias entidades de una clase están relacionadas con una única entidad de otra clase.

- **@ManyToMany:** Define una relación muchos a muchos entre dos entidades, lo que implica que una entidad puede estar asociada con múltiples entidades de otro tipo y viceversa.

En general, no queremos relaciones bidireccionales, con lo que añadiremos las relaciones sólo un sentido. En nuestro caso, será BookEntity la que definirá las relaciones con el resto de entidades.

Empecemos con las relaciones Publisher y Category. Ambas serán de tipo @ManyToOne y tendremos que indicar el campo de enganche (la clave ajena en la tabla books). Para eso, utilizaremos **@JoinColumn**;

```

1 @Entity
2 @Table(name = "books")
3 @Data
4 @NoArgsConstructor
5 public class BookEntity {
6
7     @Id
8     @GeneratedValue(strategy = GenerationType.IDENTITY)
9     private Long id;
10    ...
11
12    @ManyToOne(fetch = FetchType.LAZY)
13    @JoinColumn(name = "publisher_id")
14    private PublisherEntity publisher;
15    @ManyToOne(fetch = FetchType.LAZY)
16    @JoinColumn(name = "category_id")
17    private CategoryEntity category;
18    private List<AuthorEntity> authors;
19    private List<GenreEntity> genres;
20 }

```

Fijate en el código de arriba. Hemos añadido la opción **fetch = FetchType.LAZY** para indicar que queremos que ese campo se recupere con carga perezosa (lazy loading)

Más adelante profundizaremos en el lazy loading.

Vamos ahora con las relaciones con autores y géneros. En este caso, el tipo es @ManyToMany. En este tipo de relaciones tenemos que indicarle la tabla de enganche y las claves ajenas de dicha tabla. Para eso, utilizamos la anotación @JoinTable:

```

1  @Entity
2  @Table(name = "books")
3  @Data
4  @NoArgsConstructor
5  public class BookEntity {
6
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10     ...
11
12     @ManyToOne(fetch = FetchType.LAZY)
13     @JoinColumn(name = "publisher_id")
14     private PublisherEntity publisher;
15     @ManyToOne(fetch = FetchType.LAZY)
16     @JoinColumn(name = "category_id")
17     private CategoryEntity category;
18     @ManyToMany(fetch = FetchType.LAZY)
19     @JoinTable(
20         name = "books_authors",
21         joinColumns = @JoinColumn(name = "book_id"),
22         inverseJoinColumns = @JoinColumn(name = "author_id")
23     )
24     private List<AuthorEntity> authors;
25     @ManyToMany(fetch = FetchType.LAZY)
26     @JoinTable(
27         name = "books_genres",
28         joinColumns = @JoinColumn(name = "book_id"),
29         inverseJoinColumns = @JoinColumn(name = "genre_id")
30     )
31     private List<GenreEntity> genres;
32 }

```

## Repositorios JPA

En *Spring Data JPA*, para poder operar con los datos necesitamos definir **Repositorios JPA** (no confundir con nuestros repositorios). Básicamente se tratan de interfaces que heredan de `JpaRepository` (<https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>), dónde tenemos que indicar la entidad del repositorio y el tipo de la clave principal:

```
1 public interface BookJpaRepository extends JpaRepository<BookEntity, Long> {
2
3 }
```

*Spring Data JPA* simplifica enormemente la interacción con la base de datos al proporcionar una implementación predeterminada de métodos CRUD en la interfaz *JpaRepository*. Cuando extiendes esta interfaz para crear tu repositorio, obtienes métodos como *save*, *findById*, *findAll*, *delete*, *count*, entre otros, listos para ser usados sin necesidad de escribir la implementación de cada uno.

Además de estos métodos predefinidos, puedes definir métodos en tu interfaz de repositorio siguiendo una convención de nombres específica. *Spring Data JPA* analiza el nombre del método y genera consultas SQL correspondientes automáticamente.

Por ejemplo, nuestro método para buscar libros por ISBN sería:

```

1 public interface BookJpaRepository extends JpaRepository<BookEntity, Long> {
2     BookEntity findByIsbn(String isbn);
3 }
4

```

Spring Data JPA interpretará este método y generará una consulta SQL de forma automática para buscar libros por ISBN. La convención de nombres juega un papel crucial aquí: el prefijo *findBy* indica que quieres buscar entidades por un campo específico (isbn en este caso). Spring Data JPA analiza el nombre del método, separa *findBy*, interpreta *Isbn* como el nombre del campo y genera la consulta correspondiente.

Esta abstracción ahorra tiempo y esfuerzo al eliminar la necesidad de escribir consultas SQL repetitivas. En lugar de preocuparte por la sintaxis SQL, puedes centrarte en definir métodos descriptivos en tu interfaz de repositorio para manejar las consultas de manera más intuitiva y enfocarte en la lógica de tu aplicación.

Ten en cuenta que para que esto funcione tienes que seguir la nomenclatura de Spring Data JPA para los métodos. Aquí (<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>) puedes ver una referencia de esa nomenclatura.

Otra cosa importante es que los métodos *findBy* en *Spring Data JPA* devuelven un *Optional* en lugar de una instancia directa de la entidad.

El siguiente paso será crear una nueva implementación de nuestro DAO para que utilice ese repositorio. Pero antes, necesitamos mapeadores para mapear entidades JPA a nuestros modelos y viceversa:

```

1 @Mapper(uses = {PublisherJpaMapper.class, AuthorJpaMapper.class, GenreJpaMapper.class, CategoryJpaMapper.class})
2 public interface BookJpaMapper {
3
4     BookJpaMapper INSTANCE = Mappers.getMapper(BookJpaMapper.class);
5
6     Book toBookWithDetails(BookEntity bookEntity);
7
8     @Mapping(target = "authors", ignore = true)
9     @Mapping(target = "genres", ignore = true)
10    @Mapping(target = "publisher", ignore = true)
11    @Mapping(target = "category", ignore = true)
12    Book toBook(BookEntity bookEntity);
13
14    BookEntity toBookEntity(Book book);
15
16 }

```

En este caso, utilizamos MapStruct para simplificar el código (aunque podríamos también hacerlo a mano o utilizar cualquier otro mapeador). Si te fijas, esta vez tenemos dos mapeadores de BookEntity. Uno con los campos asociados (autores, géneros, editorial y categoría) y otro que los ignora. Esto evitará que JPA intente cargar las entidades asociadas debido al lazy loading.

Ahora ya podemos crear nuestra nueva implementación del DAO, utilizando en las respuestas el mapeador que nos interese:

```

1 @Component
2 @RequiredArgsConstructor
3 public class BookDaoJpa implements BookDaoDb {
4
5     private final BookJpaRepository bookJpaRepository;
6
7     @Override
8     public Optional<Book> findByIsbn(String isbn) {
9         return Optional.ofNullable(bookJpaRepository.findByIsbn(isbn))
10            .map(BookJpaMapper.INSTANCE::toBookWithDetails);
11    }
12    ...
13    @Override
14    public List<Book> getAll() {
15        return bookJpaRepository
16            .findAll()
17            .stream()
18            .map(BookJpaMapper.INSTANCE::toBook)
19            .toList();
20    }
21    ...

```

## Seleccionando el DAO correspondiente

Si ejecutamos la aplicación ahora, veremos que nos salta un error. Esto se debe a que tenemos dos implementaciones de nuestra interfaz *BookDaoJdbc* y Spring no sabe cuál de las dos inyectar en el repositorio:

```

1 public class BookDaoJdbc implements BookDaoDb {
2     ...
3 }
4
5 public class BookDaoJpa implements BookDaoDb {
6     ...
7 }
8
9 public class BookRepositoryImpl implements BookRepository {
10
11     private final BookDaoDb bookDaoDb;
12     ...
13 }

```

Debemos indicar a Spring qué implementación concreta queremos utilizar. Lo podemos hacer de varias formas, por ejemplo usando `Qualifier` (<https://docs.spring.io/spring-framework/reference/core/beans/annotation-config/autowired-qualifiers.html>). En nuestro caso, vamos a utilizar la anotación `Primary` (<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Primary.html>) en una de las implementaciones (en este caso la de JPA) para indicarle a Spring que tendrá preferencia sobre el resto de implementaciones a la hora de inyectarla:

```

1 @Component
2 @Primary
3 @RequiredArgsConstructor
4 public class BookDaoJpa implements BookDaoDb {
5     ...

```

Ahora debería funcionar de nuevo la aplicación usando *Spring Data Jpa*.

## Carga de datos relacionados

Cuando hemos definido nuestra entidad *BookEntity*, hemos indicado que queremos hacer lazy loading en la carga de entidades relacionadas. Las dos opciones que tenemos son:

- **LAZY** (carga perezosa): La carga de la relación se realiza cuando se accede a ella, es decir, los datos relacionados no se cargan inmediatamente cuando se consulta la entidad principal. Solo se cargan cuando se hace explícitamente referencia a la relación (por ejemplo, accediendo a una propiedad de la entidad relacionada). Esto puede mejorar el rendimiento si la relación no se necesita inmediatamente, pero puede llevar a consultas adicionales si se accede a la relación más tarde.
- **EAGER** (carga ansiosa o impaciente): La carga de la relación se realiza de forma inmediata cuando se consulta la entidad principal, es decir, los datos relacionados se cargan junto con la entidad principal. Esto puede ser útil cuando se sabe que siempre se necesitarán los datos relacionados, pero puede afectar el rendimiento si no se manejan adecuadamente, especialmente con relaciones complejas o grandes volúmenes de datos.

En nuestro caso, cuando recuperamos un listado de libros sólo queremos los datos básicos, sin las entidades relacionadas, con lo que usamos la primera opción:

```

1 @ManyToOne(fetch = FetchType.LAZY)
2 @JoinColumn(name = "publisher_id")
3 private PublisherEntity publisher;
4 @ManyToOne(fetch = FetchType.LAZY)
5 @JoinColumn(name = "category_id")
6 private CategoryEntity category;
7 @ManyToMany(fetch = FetchType.LAZY)
8 @JoinTable(
9     name = "books_authors",
10     joinColumns = @JoinColumn(name = "book_id"),
11     inverseJoinColumns = @JoinColumn(name = "author_id")
12 )
13 private List<AuthorEntity> authors;
14 @ManyToMany(fetch = FetchType.LAZY)
15 @JoinTable(
16     name = "books_genres",
17     joinColumns = @JoinColumn(name = "book_id"),
18     inverseJoinColumns = @JoinColumn(name = "genre_id")
19 )
20 private List<GenreEntity> genres;

```

Puede que pienses que tampoco perderíamos mucho rendimiento si recuperáramos los datos del editor y la categoría mediante dos JOINS. En ese caso, podríamos optar por hacer la carga impaciente:

```

1 @ManyToOne(fetch = FetchType.EAGER)
2 @JoinColumn(name = "publisher_id")
3 private PublisherEntity publisher;
4 @ManyToOne(fetch = FetchType.EAGER)
5 @JoinColumn(name = "category_id")

```

Esto presenta un par de problemas. Primero, siempre recuperará los datos de la editorial y la categoría, aunque no los necesitemos (por ejemplo, cuando devolvamos un listado de libros).

Además, JPA no garantiza hacer la sentencia con JOINS. De hecho, si miras en la terminal las SQL que ejecuta, verás que realiza 3 consultas. Una para recuperar los libros y otras dos para recuperar la editorial y la categoría.

Ésto se conoce como **n+1 Queries**, y es típico al desarrollar aplicaciones con frameworks de persistencia, como Hibernate, Spring Data JPA... Para evitarlo tenemos varias opciones. Lo primero, vamos a dejar las relaciones con Publisher y Category con lazy loading de nuevo:

```

1  @ManyToOne(fetch = FetchType.LAZY)
2  @JoinColumn(name = "publisher_id")
3  private PublisherEntity publisher;
4  @ManyToOne(fetch = FetchType.LAZY)
5  @JoinColumn(name = "category_id")

```

De esta forma, sólo cargará las entidades asociadas cuando sea necesario. Además, para forzar al framework a usar JOINS en las consultas, añadiremos una anotación a los métodos del repositorio JPA que devuelvan las entidades con sus relaciones, como *findByIsbn*:

```

1  public interface BookJpaRepository extends JpaRepository<BookEntity, Long> {
2
3      @EntityGraph(attributePaths = {"publisher", "category"})
4      BookEntity findByIsbn(String isbn);
5  }

```

La anotación *EntityGraph* (<https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/EntityGraph.html>) usada sobre un método de un repositorio JPA, indica que tiene que recuperar los datos de las entidades relacionadas indicadas en *attributePaths* aunque en la entidad esté definido como *fetch.LAZY*. Además, lo hará mediante JOINS.

Otra opción sería crear nuestro propio SQL con JOINS:

```

1  public interface BookJpaRepository extends JpaRepository<BookEntity, Long> {
2
3      @Query("SELECT b FROM BookEntity b " +
4             "JOIN FETCH b.publisher p " +
5             "JOIN FETCH b.category c " +
6             "WHERE b.isbn = :isbn")
7      BookEntity findByIsbn(String isbn);
8  }

```

Ten en cuenta, que si optamos por esta estrategia debemos indicar en cada método qué entidades queremos recuperar (incluso los métodos que nos proporciona Spring Data JPA). En nuestro caso, también usamos el método *findById*, con lo que deberíamos también modificarlo:

```

1  public interface BookJpaRepository extends JpaRepository<BookEntity, Long> {
2
3      @EntityGraph(attributePaths = {"publisher", "category"})
4      BookEntity findByIsbn(String isbn);
5
6      @Override
7      @EntityGraph(attributePaths = {"publisher", "category"})
8      Optional<BookEntity> findById(Long id);
9  }
10 }

```

Podemos usar la anotación *@Override* en el método *findById*, ya que estamos sobrescribiendo un método de una clase antecesora proporcionada por Spring Data JPA con la implementación de dicho método.

## Paginación

Paginar resultados con Spring Data JPA es tan sencillo como utilizar el método *findAll* ([https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html#findAll\(org.springframework.data.domain.Pageable\)](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html#findAll(org.springframework.data.domain.Pageable))) pasándole un objeto *Pageable* (<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html>). Este método nos devolverá un objeto de tipo *Page* (<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Page.html>), con un conjunto de datos y otra información asociada.

Para crear el objeto, usamos su implementación *PageRequest* (<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/PageRequest.html>) pasándole el número de página y el tamaño de página:

```

1  @Override
2  public List<Book> getAll(int page, int size) {
3      Pageable pageable = PageRequest.of(page, size);
4      Page<BookEntity> bookPage= bookJpaRepository.findAll(pageable);
5      ...
6  }

```

Antes hemos dicho que el objeto *Page* que devuelve el método *findAll* contiene más información que el listado de objetos paginados. Entre otra, tenemos el número de entidades totales. En nuestra aplicación, creamos un caso de uso donde devolvíamos ese total:

```

1  public interface BookCountUseCase {
2
3      int execute();
4  }

```

Y lo utilizábamos en el controlador para recuperar ese dato y montar nuestra respuesta:

```

1  @GetMapping
2  public ResponseEntity<PaginatedResponse<BookCollection>> getAll(
3      @RequestParam(defaultValue = "1") int page,
4      @RequestParam(required = false) Integer size) {
5
6      int pageSize = (size != null) ? size : Integer.parseInt(defaultPageSize);
7      List<BookCollection> bookCollections = bookGetAllUseCase
8          .execute(page - 1, pageSize)
9          .stream()
10         .map(BookMapper.INSTANCE::toBookCollection)
11         .toList();
12     int total = bookCountUseCase.execute();
13
14     PaginatedResponse<BookCollection> response = new PaginatedResponse<>(bookCollections, total, page, page
15     return new ResponseEntity<>(response, HttpStatus.OK);
16 }

```

Al utilizar Spring Data JPA, si dejáramos el código así haría dos veces la sentencia "SELECT count(\*) FROM books", ya que Spring Data JPA la realiza para montar el objeto *Page* (puedes comprobarlo mirando en la terminal las sentencias SQL que se ejecutan).

Para solucionarlo, vamos a crear un nuevo modelo de dominio que contendrá un listado de entidades y el total:

```

1  @Data
2  @AllArgsConstructor
3  public class ListWithCount<T> {
4
5      private List<T> list;
6      private long count;
7  }

```

De esta forma, cambiaremos la firma del método *getAll* con paginación de nuestro DAO para devolver ese nuevo modelo:

```

1  public interface GenericDaoDb<T> {
2
3      List<T> getAll();
4      ListWithCount<T> getAll(int page, int size);
5      Optional<T> findById(long id);
6      long insert(T t);
7      void update(T t);
8      void delete(long id);
9      long count();
10     T save(T t);
11 }

```

Así en nuestras implementaciones devolveremos ambos datos, el listado de resultados y el total de elementos. Por ejemplo, nuestra implementación **BookDaoJdbc** quedaría:

```

1  @Override
2  public ListWithCount<Book> getAll(int page, int size) {
3      String sql = ""
4          SELECT * FROM books
5          LIMIT ? OFFSET ?
6          "";
7      List<Book> books = jdbcTemplate.query(sql, new BookRowMapper(), size, page * size);
8      int total = (int) this.count();
9      return new ListWithCount<Book>(books, total);
10 }
11
12 @Override
13 public long count() {
14     String sql = ""
15         SELECT COUNT(*) FROM books
16         "";
17     return jdbcTemplate.queryForObject(sql, Long.class);
18 }

```

Mientras que **BookDaoJpa** sería:

```

1  @Override
2  public ListWithCount<Book> getAll(int page, int size) {
3      Pageable pageable = PageRequest.of(page, size);
4      Page<BookEntity> bookPage = bookJpaRepository.findAll(pageable);
5      return new ListWithCount<Book>(
6          bookPage.stream()
7              .map(BookJpaMapper.INSTANCE::toBook)
8              .toList(),
9          bookPage.getTotalElements()
10 );
11 }

```

En la implementación JPA Utilizamos el método `getTotalElements` ([https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Page.html#getTotalElements\(\)](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Page.html#getTotalElements())) de *Page* para obtener el total de elementos

Con este cambio, ya no necesitamos llamar al caso de uso que nos devuelve el total de elementos, con lo que no repetiríamos la sentencia SQL al utilizar JPA:



```

1 public ResponseEntity<PaginatedResponse<BookCollection>> getAll(
2     @RequestParam(defaultValue = "1") int page,
3     @RequestParam(required = false) Integer size) {
4
5     int pageSize = (size != null) ? size : Integer.parseInt(defaultPageSize);
6     String baseUrl = PropertiesConfig.getSetting("app.base.url") + URL;
7     ListWithCount<Book> bookList = bookGetAllUseCase.execute(page - 1, pageSize);
8     PaginatedResponse<BookCollection> response = new PaginatedResponse<>(
9         bookList
10            .getList()
11            .stream()
12            .map(BookMapper.INSTANCE::toBookCollection)
13            .toList(),
14         bookList.getCount(), page, pageSize, baseUrl);
15     return new ResponseEntity<>(response, HttpStatus.OK);
16 }

```

## Actualizaciones, inserciones y borrados

Para actualizar o insertar recursos, JPA utiliza el método **save()**. Lo único que tenemos que hacer en la implementación del DAO es mapear nuestros modelos de datos de la capa de dominio a entidades de JPA y guardar el recurso:

```

1 @Override
2 public Book save(Book book) {
3     BookEntity bookEntity = BookJpaMapper.INSTANCE.toBookEntity(book);
4     return BookJpaMapper.INSTANCE.toBook(bookJpaRepository.save(bookEntity));
5 }

```

JPA devuelve el recurso recién creado/actualizado, con lo que en ambos casos podríamos devolver en el controlador dicho recurso en formato JSON (probablemente te tocará modificar el servicio y el controlador para indicar el tipo de recurso que devuelve cada método).

El borrado es igual de sencillo, simplemente tendremos que llamar al método `deleteById` ([https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteById\(ID\)](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#deleteById(ID))):

```

1 @Override
2 public void delete(long id) {
3     bookJpaRepository.deleteById(id);
4 }

```

Para borrar un recurso, también podríamos utilizar el método `delete` ([https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#delete\(T\)](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#delete(T))), aunque en este caso deberíamos pasarle la entidad completa

Con todo, nuestra implementación `BookDaoJpa` quedaría:

```



1  @Component
2  @Primary
3  @RequiredArgsConstructor
4  public class BookDaoJpa implements BookDaoDb {
5
6      private final BookJpaRepository bookJpaRepository;
7
8      @Override
9      public Optional<Book> findByIsbn(String isbn) {
10         return Optional.ofNullable(bookJpaRepository.findByIsbn(isbn))
11             .map(BookJpaMapper.INSTANCE::toBookWithDetails);
12     }
13
14     @Override
15     public void deleteAuthors(long id) {
16         bookJpaRepository.findById(id)
17             .ifPresent(bookEntity -> bookEntity.getAuthors().clear());
18     }
19
20     @Override
21     public void insertAuthors(long id, List<Author> authors) {
22         bookJpaRepository.findById(id)
23             .ifPresent(bookEntity -> bookEntity.setAuthors(
24                 authors.stream()
25                     .map(AuthorJpaMapper.INSTANCE::toAuthorEntity)
26                     .toList()
27             ));
28     }
29
30     @Override
31     public void deleteGenres(long id) {
32         bookJpaRepository.findById(id)
33             .ifPresent(bookEntity -> bookEntity.getGenres().clear());
34     }
35
36     @Override
37     public void insertGenres(long id, List<Genre> genres) {
38         bookJpaRepository.findById(id)
39             .ifPresent(bookEntity -> bookEntity.setGenres(
40                 genres.stream()
41                     .map(GenreJpaMapper.INSTANCE::toGenreEntity)
42                     .toList()
43             ));
44     }
45
46     @Override
47     public List<Book> getAll() {
48         return bookJpaRepository
49             .findAll()
50             .stream()
51             .map(BookJpaMapper.INSTANCE::toBook)
52             .toList();
53     }
54
55     @Override
56     public ListWithCount<Book> getAll(int page, int size) {
57         Pageable pageable = PageRequest.of(page, size);
58         Page<BookEntity> bookPage = bookJpaRepository.findAll(pageable);
59         return new ListWithCount<Book>(
60             bookPage.stream()
61                 .map(BookJpaMapper.INSTANCE::toBook)
62                 .toList(),
63             bookPage.getTotalElements()
64         );
65     }
66
67     @Override
68     public Optional<Book> findById(long id) {
69         return bookJpaRepository.findById(id)
70             .map(BookJpaMapper.INSTANCE::toBookWithDetails);
71     }
72
73     @Override
74     public long insert(Book book) {
75         return bookJpaRepository.save(BookJpaMapper.INSTANCE.toBookEntity(book)).getId();
76     }
77
78     @Override
79     public void update(Book book) {
80         bookJpaRepository.save(BookJpaMapper.INSTANCE.toBookEntity(book));
81     }
82
83     @Override
84     public void delete(long id) {
85         bookJpaRepository.deleteById(id);
86     }
87
88     @Override
89     public long count() {
90         return bookJpaRepository.count();
91     }
92
93     @Override
94     public Book save(Book book) {
95         BookEntity bookEntity = BookJpaMapper.INSTANCE.toBookEntity(book);

```

```
96         return BookJpaMapper.INSTANCE.toBook(bookJpaRepository.save(bookEntity));  
97     }  
98 }
```

Puede que pienses que con tanto mapeo de entidades de JPA a nuestros modelos de dominio, realicemos sentencias SQL de más. Por ejemplo, cuando actualizamos un libro, primero lo buscamos para ver si existe. Eso implica que obtenemos un `BookEntity`, lo mapeamos a `Book`, modificamos los campos necesarios y volvemos a mapearlo a `BookEntity` para guardar los cambios, con lo que JPA debería, antes de hacer el `save` volver a recuperar el recurso para asegurarse que existe.

En realidad, eso no ocurre, ya que, si recuerdas el tema de la capa dominio, ya hablamos del mecanismo que tiene para manejar entidades: `EntityManager` como gestor de entidades y `dirty checking` como mecanismo para marcar los cambios de éstas.

 clase/daw/dws/2eval/jpa.txt  Última modificación: 2024/12/18 13:01 por cesguiro

cesguiro



Excepto donde se indique lo contrario, el contenido de este wiki esta bajo la siguiente licencia:  
CC Attribution-Noncommercial-Share Alike 4.0 International