

Operamos con bases de datos

Java



Operamos con bases de datos

Trabajar con Java y SpringBoot facilita mucho el desarrollo de aplicaciones, pero ¿qué hacemos si necesitamos almacenar datos para que persistan una vez se apague la aplicación?

A lo largo de este fastbook, responderemos a esta importante pregunta y te explicaremos en qué consisten las operaciones con bases de datos en el contexto de Spring Boot y Java. Veremos cómo se aplican los principios de la programación orientada a objetos en el acceso y manipulación de la data, en una base de datos con Spring Boot. Además, aprenderéis sobre las funcionalidades y ventajas de utilizar el mapeo objeto-relacional de JPA con el objetivo de simplificar la persistencia de datos en una base de datos relacional.

Autor: Antonio Marchante

 **Tips sobre las bases de datos**

 **Liquibase**

 **Configuración de nuestras bases de datos**

 **Creación de entities**

 **Etiquetas**

 **Paginación de resultados**

 **Operaciones con bases de datos**

 **Conclusiones**

Tips sobre las bases de datos



A la hora de almacenar datos existen varias opciones para el sistema de estos. Las más populares son SQL y no-SQL. Las bases de datos SQL son las más comunes históricamente (han sido un estándar hasta hace bien poco y la mayoría de los proyectos están desarrolladas con ellas) y, por tanto, necesitamos aprender todo SQL. Por otra parte, tenemos las noSQL, que en los últimos años han tenido su auge gracias a grandes compañías como Facebook, Netflix, IBM, etc., que han apostado por ellas.

Lo interesante en este punto inicial es que veamos qué ventajas ofrece cada una.

SQL

Las bases de datos SQL se basan en tablas y sus datos en filas y columnas. Es decir, tienen una organización estructurada.

La forma de acceder a estos datos es por medio de consultas SQL (*Structured Query Language*) y, gracias a ellas, podremos interactuar con los datos (consultado, actualizando, insertando o eliminándolos).

La estructura en tablas es óptima para aplicaciones cuyo **conjunto de datos es rígido y está predefinido**, es decir, cuando los datos tienen una estructura inalterable a lo largo del uso de la aplicación. Gracias a esto, podemos garantizar la integridad y consistencia de los datos, por medio de relaciones y restricciones entre tablas y por campo.

Sin embargo, la rigidez y confianza que nos ofrece SQL puede volverse limitante cuando los datos con los que vamos a trabajar no están estructurados o pueden ser modificables, debido a que todos los registros tienen que seguir la misma estructura y no se pueden definir subconjuntos de objetos. En otras palabras: no podremos introducir un objeto con más campos de los definidos sin alterar el esquema de la base de datos. Algunas de las bases de datos SQL más populares son SQLServer, PostgreSQL o MySQL

No-SQL

Las bases de datos NoSQL, por su parte, se basan en la idea de que los datos pueden no estar estructurados y, por tanto, ser flexibles.

- Los datos en este tipo de bases de datos se almacenan en un **formato no tabular**: en documentos, gráficos o por pares de clave-valor. Este tipo de bases de datos, debido a su formato, se utilizan en aplicaciones con datos no relacionales como las de las redes sociales, en sistemas de entretenimiento, etc.
- La **principal ventaja de las bases de datos NoSQL es su flexibilidad**, gracias a que permiten agregar nuevos campos o estructuras (por ejemplo, objetos embebidos dentro de otros) sin necesidad de cambiar el esquema definido. Además de esto, son muy escalables, lo que nos permite trabajar con grandes volúmenes de datos de forma eficiente.
- Sin embargo, debido a que **carecen de relaciones entre los objetos y la falta de una estructura definida**, trabajar con estos datos y realizar consultas puede ser una tarea compleja en comparación a una base de datos SQL (donde los objetos están claramente definidos).
- Además de esto, al ser relativamente novedosas, están **menos establecidas en el mundo de la programación**, por lo que existe menos documentación sobre ellas, menos formación y, por ende, su curva de aprendizaje puede ser algo mayor que la de SQL.

Algunas de las bases de datos NoSQL más populares son MongoDB, Cassandra y Redis.

Escenarios de uso

En base a lo visto, podemos discernir **en qué escenarios será más apropiada una u otra base de datos** en función de las características de la aplicación que vamos a desarrollar.

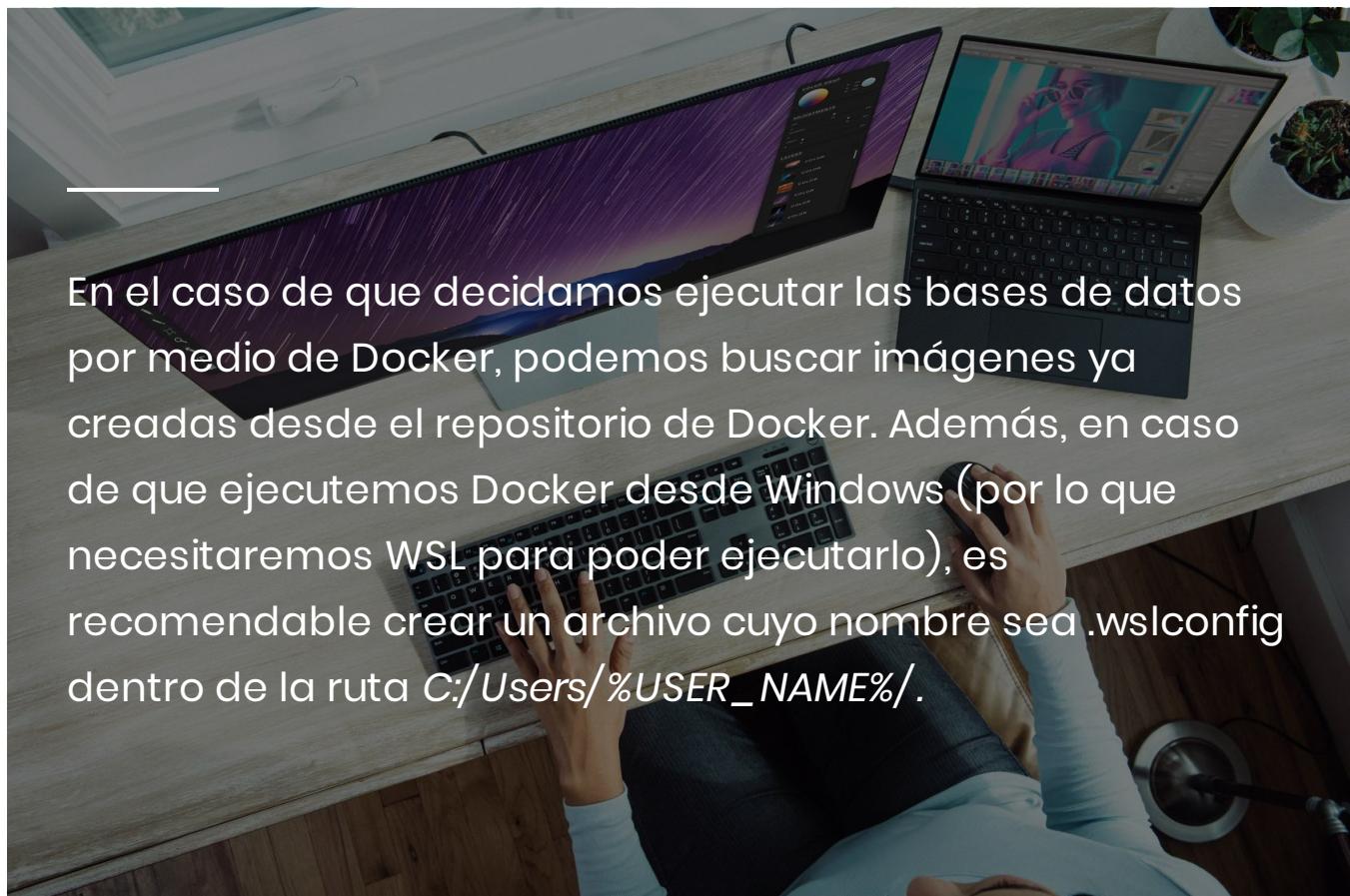
- Por un lado, las bases de datos SQL son **óptimas cuando tenemos una estructura de los datos claramente definida y establecida** y necesitamos de consultas complejas que hagan un filtrado de los datos por unos campos concretos que deben estar presente en todos nuestros objetos. Debido a esto, su uso más común es en aplicaciones empresariales, bancos, gestión inventario o logístico, etc.
- Por otro lado, **las bases de datos NoSQL están más pensadas para aplicaciones web, redes sociales y aplicaciones** que manejan grandes volúmenes de datos no estructurados. Son las mejores cuando buscamos escalabilidad y flexibilidad.

En resumen, cada una tiene un enfoque diferente a la hora de gestionar los datos con los que vamos a trabajar dado que cada una estructurará la información de una forma y nos obligará a nosotros, como desarrolladores, a manejar de una u otra forma los datos que recibiremos.

Configuración de nuestras bases de datos



Para poder **configurar una base de datos** con nuestra aplicación, o bien deberemos tener ejecutando una base de datos local, o bien acceder a una remota.



En el caso de que decidamos ejecutar las bases de datos por medio de Docker, podemos buscar imágenes ya creadas desde el repositorio de Docker. Además, en caso de que ejecutemos Docker desde Windows (por lo que necesitaremos WSL para poder ejecutarlo), es recomendable crear un archivo cuyo nombre sea `.wslconfig` dentro de la ruta `C:/Users/%USER_NAME%/.wslconfig`.

El archivo lo editaremos con un editor de textos (como Notepad++ o Visual Studio Code) y añadiremos las siguientes líneas:

```
[wsl2]  
  
memory=3GB    # Limits VM memory in WSL 2 up to 3GB  
  
processors=4 # Makes the WSL 2 VM use two virtual processors
```

De esta forma, podremos **limitar la cantidad de recursos que WSL** (en su versión 2, la cual os recomiendo usar en lugar de la primera) hace de nuestro equipo.

Después de esto, deberemos **importar las dependencias necesarias en Maven** para la base de datos que hayamos elegido usar.

 Imaginemos ahora que queremos utilizar una base de datos Postgre. Para ello, podremos dirigirnos a [Maven Repository](#) o [Maven Central](#) y elegir la dependencia para esta base de datos.

```
<dependency>  
  
    <groupId>org.postgresql</groupId>  
  
    <artifactId>postgresql</artifactId>  
  
    <scope>runtime</scope>  
  
</dependency>
```

También deberemos **añadir la dependencia a JPA** para poder utilizar consultas dentro de la aplicación.

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

Ya continuación, deberemos **definir las properties** (en el archivo *application.properties*) relativos a la conexión con la base de datos (URL, puerto, usuario y contraseña).

```
spring.datasource.url=jdbc:postgresql://localhost:5432/demo
spring.datasource.username=postgres
spring.datasource.password=mysecretpassword
```

 Como guía, os dejo este enlace de Baeldung en el que trabajar con una base de datos H2, la cual es una base de datos para SpringBoot que se utiliza en memoria. Puedes practicar con ella y, así, conocer un nuevo concepto [aquí](#).

Etiquetas



SpringBoot nos facilita el manejo de consultas y entidades gracias a etiquetas propias que nos ayudan a estos propósitos.

Por una parte, tenemos la definición de las entidades que están asociadas a las tablas de la base de datos por medio de un ORM (*Object-Relational Mapping*) como Hibernate, el cual nos ayuda a mapear objetos desde la BD a nuestros objetos Java y viceversa. Para ello, deberemos utilizar conjuntamente las etiquetas `@Table`, `@Column`, `@Id`, etc.

Aquí te muestro un ejemplo de **entidad con las etiquetas**:

```
@Entity  
@Table(name="user")  
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @Column(name="name")  
    private String name;  
  
    @Column(name="age")  
    private int age;  
}
```

Analizando el código, podemos ver cómo definimos, por un lado, la entidad y la relación con la tabla por medio de las etiquetas `@Entity` y `@Table` y, por otro lado, cómo definimos las columnas por medio de la etiqueta `@Column`; y, para el caso de los id's, podremos **definir tanto su campo como la estrategia de generación**, es decir, cuando insertemos un nuevo registro, qué estrategia seguirá para asignarle un valor automáticamente a este ID.

Una vez definidas las entidades, podremos utilizarlas para **definir consultas sobre las mismas tablas** y que los resultados se mapeen automáticamente a nuestro objeto en Java, gracias a la definición que hemos hecho previamente. Estas consultas estarán definidas en nuestro repositorio, el cual contendrá toda la lógica relacionada con las consultas. La más utilizada será la etiqueta `@Query`, la cual se utiliza para definir consultas personalizadas en lenguaje SQL o JPQL (*Java Persistence Query Language*) dentro del contexto de Spring Data.

Un ejemplo de **definición de una consulta** dentro de un repositorio podría ser el siguiente.

```
@Repository  
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("SELECT u FROM User u WHERE u.age > :age")  
    List<User> findByAgeGreaterThan(@Param("age") int age);  
}
```

En este ejemplo vemos cómo, gracias a la etiqueta `@Query`, podemos definir, de forma sencilla, **una consulta para buscar un usuario cuya edad sea mayor** que la que pasamos como parámetro. Esta consulta está escrita con formato JPQL dentro de la etiqueta, la cual nos permite utilizar los propios objetos definidos en Java.

**Es importante que destaquemos varios aspectos de
esta clase, ¡toma nota!**

El primero es que se trata de una interfaz dada, y que no será necesario definir los métodos específicos para la realización de las consultas, ya que SpringBoot lo hará automáticamente gracias a la definición que le damos con `@Query`. El segundo es que esta interfaz heredará directamente de `JpaRepository`, por lo tanto, será el encargado de gestionar estas consultas pasándole en primer lugar el objeto que utilizará para mapear los resultados (`User`) y el valor de la clave primaria que está definida en la tabla (`Long`).

Operaciones con bases de datos



Qualentum Lab

Llegados a este punto, es importante para que puedas trabajar con BD que conozcas los principales operadores.

¡Empezamos!

CRUD

CRUD (*create, read, update, delete*) son las siglas de las operaciones básicas que podemos realizar con una base de datos y que casi cualquier *endpoint* que trabaje con objetos debería poseer (a excepción de los objetos que sean finales y no admitan, por ejemplo, modificarse, añadir nuevos, eliminar, etc.).

Native Queries

Las Native Queries son un tipo de queries que podemos **escribir utilizando lenguaje JPQL** para, gracias a unas entidades ya definidas por nosotros en base a unas tablas, generar consultas que nos devuelvan datos en función de unos parámetros que nosotros definamos.

Por ejemplo, imaginemos que tenemos definida una entidad (`@Entity`) en función de una tabla 'User'. Podremos generar un Repository para definir la consulta que podremos llamar desde, por ejemplo, un service.

```
public interface UserRepository extends JpaRepository<User, Integer> {  
    @Query("SELECT u FROM User u WHERE u.status = 'online'")  
    List<User> findAllActiveUsers();  
}
```

Como vemos en nuestro ejemplo, deberemos extender de `JpaRepository`, donde indicaremos el tipo de objeto y el tipo de campo que conformará el ID de la tabla (en nuestro ejemplo, el id será de tipo `Integer`).

Después, gracias a `@Query`, definiremos la *query* que queremos hacer sobre la tabla. Con esta simple definición, **devolverá todos los registros de usuario** cuyo estado sea online y lo almacenará automáticamente en una lista de usuarios, la que podremos usar más adelante en la aplicación.



Te comarto [la página oficial de Baeldung](#) donde se trata en más detalle este tema.

Derived Queries

Las *Derived Queries*, por su parte, son consultas que podremos definir sin necesidad de escribir la *query* con JPQL.

Esto es debido a que JPA será capaz de **mapear automáticamente los campos** por los que estamos realizando la búsqueda y gestionarlo para, pasado un argumento, poder filtrar por el mismo.

Por ejemplo, imaginemos que queremos buscar un usuario dado un nombre. En nuestro Repository, podremos simplemente definir la siguiente consulta:

```
List<User> findByName(String name)
```

Y él, automáticamente, nos devolverá **los mismos resultados** que si hiciésemos lo siguiente:

```
@Query("SELECT u FROM User u WHERE u.name = :name")
List<User> findUserByName(@Param("name") String name);
```

Donde :name será el nombre que pasamos como parámetro para la Query.

 De nuevo, te comarto la página de Baeldung [aquí](#), donde se ahonda más en este tema y podrás ver ejemplos útiles que te servirán de referencia para practicar.

Liquibase



Liquibase es un plugin o herramienta que nos ayuda con la administración, creación y mantenimiento de bases de datos de forma programática y automatizada.

Este [plugin](#) se asegura de que, a la hora de arrancar la aplicación, se ejecuten **los scripts definidos en su clase de configuración** asegurándonos así que, en el momento en que se reinicia la aplicación, no existe ningún cambio en las tablas de la BD.

También permite la carga de datos desde script, lo cual también nos resultará útil si tenemos definidas las tablas de maestros o enumerados, ya que nos aseguraremos de que el contenido de estas siempre estará disponible, incluso **pese a que se haya visto alterado o eliminado por algún medio externo**.

La configuración de Liquibase para SpringBoot requiere los siguientes pasos, veámoslos en detalle.

1

Añadir las dependencias

Deberemos añadir las dependencias a nuestro fichero `pom.xml`, tanto las relativas a Liquibase como las del driver para la base de datos que estemos utilizando. Por ejemplo, si en nuestra aplicación utilizásemos PostgreSQL, deberíamos añadir las siguientes dependencias:

```
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
    <version>3.6.2</version>
</dependency>
```

2

Crear la estructura de los archivos de configuración

En la ruta `/src/main/resources` deberemos **crear una nueva carpeta o directorio** llamado `db/changelog` y, dentro de este directorio, deberemos crear un archivo que contenga el `<databaseChangeLog>` *inicial*, con el que se creará la base de datos.

```
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.4.xsd
        http://www.liquibase.org/xml/ns/dbchangelog-ext
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">
```

```
<changeSet id="table_car_creation" author="user">

    <createTable tableName="car">

        <column name="id" type="bigint"
autoIncrement="${autoIncrement}">
            <constraints primaryKey="true" nullable="false" />
        </column>

        <column name="brand" type="varchar(255)">
            <constraints nullable="true" />
        </column>

        <column name="price" type="double">
            <constraints nullable="true" />
        </column>
    </createTable>
</changeSet>
</databaseChangeLog>
```

3

Configurar el archivo de propiedades

```
liquibase.change-log=classpath:liquibase-changeLog.xml
```

4

Añadir el plugin al *pom.xml*

```
<plugin>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-maven-plugin</artifactId>
    <version>${liquibase.version}</version>
    ...
    <configuration>
        ...
        <driver>org.h2.Driver</driver>
        <url>jdbc:h2:file:./target/h2db/db/carapp</url>
        <username>carapp</username>
        <password> />
        <outputChangeLogFile>src/main/resources/liquibase-
outputChangeLog.xml</outputChangeLogFile>
    </configuration>
</plugin>
```

Una vez hecho esto, al arrancar la aplicación debería ejecutar nuestros datos introducidos en Liquibase para modificar la base de datos.



Te aconsejo estudiar Tutorial Liquibase: cómo utilizar Liquibase de [Refactorizando.blog](#) que contiene un caso práctico.

Creación de entities



Qualentum Lab



Las entidades o *entities* son la representación en nuestra aplicación de una entidad o tabla de la base de datos.

Consiste fundamentalmente en crear un objeto en Java basándonos en los campos de una tabla, teniendo en cuenta sus relaciones con otras tablas, en caso de que las tuviera.

Dicho esto, **vamos a estudiar la relación entre tablas**.

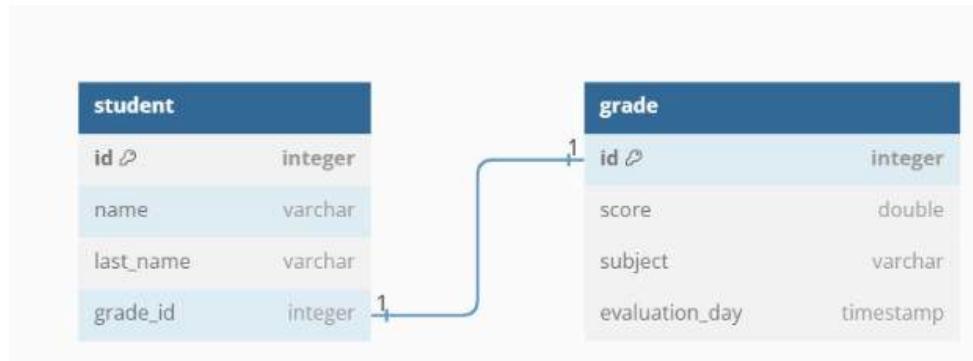
Relación entre tablas

1

Relación uno a uno

La relación 1 a 1 se da cuando solo un elemento de la tabla A estará presente en la tabla B, es decir, un objeto de la tabla A solo podrá contener una relación de la tabla B y viceversa. A través de un ejemplo, lo vamos a entender mejor.

Supongamos que tenemos dos tablas, 'Estudiante' y 'Nota' (imaginemos que será la nota media de un estudiante a lo largo de un curso). Un estudiante solo puede tener una nota (para una asignatura, por ejemplo) y una nota solo estará vinculada a un estudiante.



Fuente: propia.

Vamos a ver ahora cómo sería **la relación entre estas dos entidades en Java**.

```

@Entity
@Table(name = "student")
public class Student {
    @Id
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "last_name")
    private String lastName;
    @OneToOne(mappedBy = "student", cascade = CascadeType.ALL)
    private Grade grade;
}

```

```
@Entity  
 @Table(name = "grade")  
 public class Grade {  
     @Id  
     private Long id;  
  
     @Column(name = "score")  
     private Double score;  
  
     @Column(name = "subject")  
     private String subject;  
  
     @OneToOne  
     @JoinColumn(name = "grade_id")  
     private Student student;  
 }
```

Vemos cómo se crea **la relación con la etiqueta** `@OneToOne` (que especifica el tipo de relación) y la etiqueta `@JoinColumn` (que especificará la columna mediante la cual se relacionan ambas entidades).

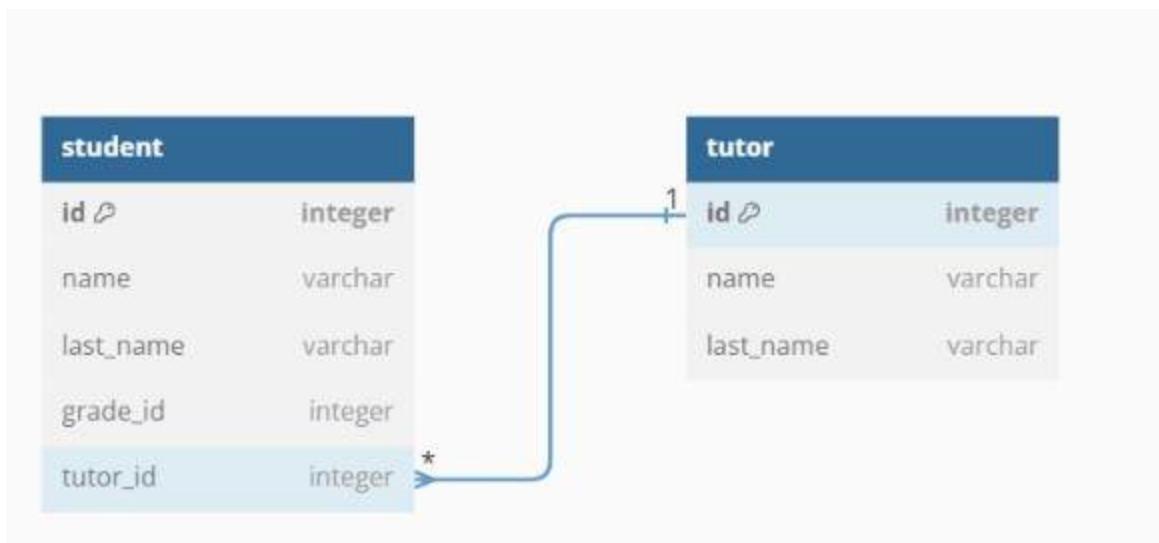
Después de esto, bastará con crear el objeto `Grade` dentro de la entidad de `Student`, debido a que consultaremos por usuario, y de ahí podremos obtener su nota.

Relación uno a muchos

En el caso de las relaciones uno a muchos (o 1 - *), lo que indicamos es que un objeto puede tener una lista de objetos relacionados, es decir, un objeto estará vinculado a muchos objetos.

Siguiendo con la analogía estudiantil, imaginemos que tenemos una tabla Estudiante y una tabla Tutor. Un estudiante solo tendrá un único tutor, sin embargo, un tutor tendrá a varios estudiantes bajo su responsabilidad.

Vamos a definir primero **la estructura de las tablas en la base de datos** como se muestra en la siguiente imagen.



Fuente: propia.

Ahora que tenemos la estructura, tenemos que saber cómo transcribimos esto a código Java.

```
@Entity  
 @Table(name = "student")  
 public class Student {  
     @Id  
     private Long id;  
  
     @Column(name = "name")  
     private String name;  
  
     @Column(name = "last_name")  
     private String lastName;  
  
     @ManyToOne  
     @JoinColumn(name = "tutor_id")  
     private Tutor tutor;  
 }
```

```
@Entity  
 @Table(name = "tutor")  
  
public class Tutor {  
  
    @Id  
    private Long id;  
  
    @Column(name = "name")  
    private String name;  
  
    @Column(name = "last_name")  
    private String lastName;  
  
    @OneToMany(mappedBy = "tutor", cascade = CascadeType.ALL)  
    private List<Student> student;  
}
```

Como puedes observar, gracias al `@MappedBy` podremos definir la propiedad de la clase `Tutor` que enlazará con nuestra clase `'Student'`. Es por esto por lo que deberemos definir la propiedad `'Student'` en la clase `'Tutor'`. Cabe destacar también cómo, para cada una de los objetos, deberemos indicar correctamente `@OneToMany` o `@ManyToOne`, en función de qué objeto será el que se repita en el otro.

3

Relación muchos a muchos

En este caso, ambos miembros pueden tener varias relaciones entre sí. Para explicar la relación muchos a muchos, utilicemos el ejemplo de un estudiante y una asignatura, donde un estudiante puede estar cursando varias asignaturas el mismo semestre, y una asignatura puede tener inscritos a varios alumnos.

A continuación, veamos **la relación entre las tablas** en la siguiente imagen.



Fuente: propia.

Si te fijas, en este caso, será necesario añadir una tabla intermedia. ¿Por qué? Debido a que, si no lo hiciésemos, tendríamos diversos registros de 'Student' y 'Subject' duplicados para crear relaciones entre ellos.

En lugar de esto, tendremos un registro 'Student' y un registro 'Subject' y con la tabla intermedia podremos relacionar varios 'Student' con 'Subject', gracias a los id's de cada tabla.

Una vez aclarado esto, a continuación, te muestro **cómo sería en código**:

```
@Table(name = "Student")  
  
public class Student {  
  
    // ...  
  
    @ManyToMany(cascade = { CascadeType.ALL })  
    @JoinTable(  
        name = "Student_Subject",  
        joinColumns = { @JoinColumn(name = "student_id") },  
        inverseJoinColumns = { @JoinColumn(name = "subject_id") }  
    )  
  
    List<Subject> subjects = new ArrayList<>();  
  
    // ...  
}
```

```
@Entity  
  
@Table(name = "Subject")  
  
public class Subject {  
  
    // ...  
  
    @ManyToMany(mappedBy = "subjects")  
    private List<Student> employees = new ArrayList<>();  
  
    // ...  
}
```

En este caso, en **la tabla sobre la que se realizarán las consultas principales** definiremos la relación con la tabla intermedia. Gracias a la propiedad *inverseJoinColumn*, podremos definir la columna que hace referencia al otro objeto.

Recuerda: debemos tener claro sobre qué objeto se realizarán las consultas y, en base a eso, definiremos un objeto u otro para que sea la referencia.

Paginación de resultados



La paginación de resultados es un método que se utiliza comúnmente en las API REST cuando existe un volumen alto de elementos a buscar. Esta paginación se basa en dividir el total de resultados en subdivisiones más pequeñas.

Cada una de estas divisiones será una página, el número de páginas dependerá del número de elementos total y el número de elementos que queramos incluir por página.

Por lo general, **el número de elementos que se incluye por página lo define el usuario**. Esto es así ya que, por ejemplo, a la hora de dirigirnos a una tienda online donde aparecen listados todos los objetos que tienen a la venta, podremos elegir la cantidad de objetos que queremos ver en cada página. De igual modo, y teniendo ya definidos el número total de elementos y el número de elementos por página que desea el usuario, podremos elegir una página concreta.

Siguiendo con el ejemplo anterior de un ecommerce, si quisiéramos ver los resultados de la página 2, y sabemos que cada página contiene 50 elementos, solo tendríamos que hacer una subdivisión del total de elementos que englobe aquellos que vayan desde el 51 al 101.

Cabe la posibilidad de aplicar también ciertos filtros al total de resultados antes de realizar la paginación.

Imaginemos que, del total de camisetas, solo queremos mostrar aquellas cuyo color sea el rojo. Simplemente, tendremos que definir la consulta para que nos devuelva el total de resultados cuyo color sea rojo y, una vez tengamos esa lista, podremos aplicar la paginación.

Una **llamada a una API REST** que ya esté desarrollada para admitir paginación sería la siguiente:

`/api/users?page=3&limit=10`

Es decir, haremos una petición de tipo GET a un recurso que devuelve usuarios, y le pediremos la página 3, donde cada página tendrá 10 elementos cada una.

Es una buena práctica que, cuando desarrollemos un método de paginación, incluyamos en la respuesta no solo la lista con los elementos solicitados sino también el número total de los resultados, el número de la página actual y el límite de registros por página que hemos definido.

A nivel de código, JPA se encargará de gestionar esta paginación e, incluso, de aplicar filtros para campos concretos ya definidos en nuestra entidad.

A continuación, te voy a mostrar un ejemplo de cómo formaríamos el método en el repositorio.

- Por un lado, deberemos **definir el método** que solicitará la información paginada a la base de datos. Ten en cuenta que hay que crear el objeto *Pageable*.

```
Page<UserEntity> userList =  
serviceJpaRepository.findAllPageable(PageRequest.of(pageNumber, pageSize,  
Sort.Direction.ASC));
```

- Por otro lado, definiremos el propio método que hará la solicitud a la base de datos, el cual recibirá, como parámetro, el objeto *Pageable* definido anteriormente y devolverá un objeto *Page* que incluirá los resultados.

```
Page<UserEntity> findAllPageable(Pageable pageable);
```

La paginación, sin lugar a duda, es **un recurso muy útil** (y muy usado) para mostrar una cantidad concreta de resultados, sin hacer que el objeto de respuesta a nuestra aplicación sea extremadamente largo y complejo de manejar. Además, **facilita a las aplicaciones web o móviles la forma de mostrar los resultados**. La manera en la que trabajan con grandes conjuntos de datos y la solicitud de recursos más livianos ayudarán a mejorar tanto la experiencia de usuario (evitando grandes tiempos de espera) como el rendimiento de la aplicación.

Conclusiones



En este tema, hemos visto qué son y cómo se pueden gestionar las operaciones en base de datos extrapoladas de una aplicación SpringBoot. También hemos aprendido una tarea fundamental en nuestro trabajo como desarrolladores: cómo transformar las relaciones de una estructura orientada a objetos para traducir las relaciones que se crean, dentro de una base de datos, para relacionar conceptos y elementos.

Con esto, podemos almacenar información que no será volátil, que perdurará en la base de datos para poder consultarla en un futuro o desde otra aplicación.

¡Enhорabuena! Fastbook superado



Qualentum.com