Ejercicio 14 – Relación ManyToMany

En un contexto de bases de datos, una relación muchos a muchos (Many-to-Many) significa que un conjunto de registros en una tabla está relacionado con un conjunto de registros en otra tabla, y viceversa. Esta relación se modela mediante una tabla intermedia que contiene las claves primarias de ambas tablas y puede, en algunos casos, incluir atributos adicionales específicos de esa relación.

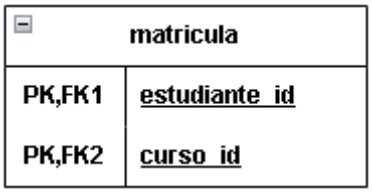
Cuando trabajamos con Spring Data JPA, hay varias maneras de mapear estas relaciones muchos a muchos, y la elección dependerá, principalmente, de la existencia de estos atributos adicionales.

Caso 1: Sin atributos adicionales

En el caso de que la tabla intermedia no requiera de atributos adicionales, Spring Data JPA la manejará de forma automática.

Imagina el siguiente ejemplo donde un estudiante puede matricularse en muchos cursos y cada curso puede tener matriculados a muchos estudiantes.

La tabla intermedia, a la que llamaremos matricula, tendrá como clave primaria una combinación de las claves ajenas:



La implementación en Spring Data JPA sería la siguiente:

```
@Entity
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "matricula",
        joinColumns = @JoinColumn(name = "estudiante_id"),
        inverseJoinColumns = @JoinColumn(name = "curso_id")
    )
    private Set<Curso> cursos;

    // Otros atributos, constructores, getters y setters
}
```

```
@Entity
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany(mappedBy = "cursos")
    private Set<Estudiante> estudiantes;

    // Otros atributos, constructores, getters y setters
}
```

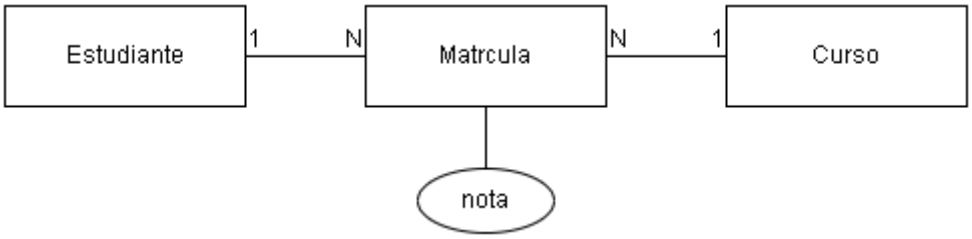
Como podemos ver, hemos utilizado la clase Estudiante como lado propietario de la relación, y hemos definido en ella la tabla de unión con la anotación ***@JoinTable***. A la anotación @JoinTable le proporcionamos el nombre de la tabla de unión (matricula) con el atributo ***name***, así como las claves ajenas con las anotaciones ***@JoinColumn***. El atributo ***joinColumns*** se conecta con el lado propietario de la relación y el ***inverseJoinColumns*** con el otro.

Ten en cuenta que no es necesario utilizar @JoinTable o incluso @JoinColumn. JPA generará los nombres de tablas y columnas por nosotros. Sin embargo, la estrategia que utiliza JPA no siempre coincidirá con las convenciones de nomenclatura que utilizamos.

Caso 2: Con atributos adicionales y clave primaria compuesta

En caso de que la tabla intermedia tenga atributos adicionales, debemos modelarla explícitamente como una entidad. Ahora bien, tenemos la opción crear una clave primaria compuesta o dejar que tenga su propia clave primaria.

Imagina que la relación tiene ahora el atributo nota, con la nota de un alumno en un curso.



La tabla intermedia, tendrá un diseño físico como el siguiente:

matricula	
PK,FK1	estudiante_id
PK,FK2	curso_id
	nota

Debido a que nuestra clave principal es una clave compuesta, tenemos que crear una nueva clase que contendrá las diferentes partes de la clave:

```
@Embeddable
class MatriculaKey implements Serializable {

    @Column(name = "estudiante_id")
    Long estudianteId;

    @Column(name = "curso_id")
    Long cursoId;

    // Constructores, getters y setters
    // hashcode y equals
}
```

Ten en cuenta que una clase de clave compuesta debe cumplir algunos requisitos clave:

- Tenemos que marcarla con ***@Embeddable***.
- Tiene que implementar ***java.io.Serializable***.
- Necesitamos proporcionar una implementación de los métodos ***hashcode()*** y ***equals().***

Con esta clase, podemos crear la entidad que modela la tabla de unión:

```
@Entity
class Matricula {

    @EmbeddedId
    MatriculaKey id;

    @ManyToOne
    @MapsId("estudianteId")
    @JoinColumn(name = "estudiante_id")
    Estudiante estudiante;

    @ManyToOne
    @MapsId("cursoId")
    @JoinColumn(name = "curso_id")
    Curso curso;

    float nota;

    // Constructores, getters y setters
}
```

Este código es muy similar a la implementación de una entidad normal. Sin embargo, tenemos algunas diferencias clave:

- Usamos ***@EmbeddedId*** para marcar la clave primaria, que es una instancia de la clase *MatriculaKey*.
- Marcamos los campos de estudiantes y cursos con ***@MapsId***.

@MapsId significa que vinculamos esos campos a una parte de la clave y son las claves externas de una relación de muchos a uno.

Después de esto, podemos configurar las referencias inversas en las entidades Estudiante y Curso como antes:

```
@Entity
public class Estudiante {
    // ...
    @OneToMany(mappedBy = "estudiante")
    private Set<Matricula> matriculas;
    // ...
}

@Entity
public class Curso {
    // ...
    @OneToMany(mappedBy = "curso")
    private Set<Matricula> matriculas;
    // ...
}
```

Básicamente, hemos descompuesto estructuralmente la relación de muchos a muchos en dos relaciones de muchos a uno. ¿Por qué pudimos hacer esto? No existe una relación de muchos a muchos en bases de datos. Las modelamos, pero no existe tal implementación.

Además, esta solución tiene una característica adicional que aún no hemos mencionado. La solución simple de muchos a muchos crea una relación entre dos entidades. Por tanto, no podemos ampliar la relación a más entidades. Pero no tenemos este límite en esta solución: podemos modelar relaciones entre cualquier cantidad de tipos de entidades.

Por ejemplo, cuando los estudiantes se pueden matricular con un profesor específico que imparte un curso específico. De esa forma, una nota sería una relación entre tres entidades: estudiante, curso y profesor.

Caso 3: Con atributos adicionales y clave primaria simple

En casos donde un estudiante pueda registrar diversas notas en un curso, la clave primaria compuesta anterior no funciona. En este tipo de situaciones (o si queremos simplificarnos la solución del caso anterior), podemos introducir una entidad con una clave primaria separada. El diseño de dicha tabla sería el siguiente:

matricula	
PK	id
FK	estudiante_id
FK	curso_id
	nota

La implementación sería la siguiente:

```
@Entity
public class Matricula {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;

    @ManyToOne
    @JoinColumn(name = "estudiante_id")
    Estudiante estudiante;

    @ManyToOne
    @JoinColumn(name = "curso_id")
    Curso curso;

    float nota;

    // Constructores, getters y setters
}
```

Las entidades Estudiante y Curso se mantendrían igual:

```
@Entity
public class Estudiante {
    // ...
    @OneToMany(mappedBy = "estudiante")
    private Set<Matricula> matriculas;
    // ...
}

@Entity
public class Curso {
    // ...
    @OneToMany(mappedBy = "curso")
    private Set<Matricula> matriculas;
    // ...
}
```

Aunque esta solución puede abordar el problema, parece extraño crear una clave principal dedicada a menos que sea necesario.

Además, desde la perspectiva de las bases de datos, no tiene mucho sentido cuando la combinación de dos claves ajenas produce una clave compuesta perfecta.

De todas maneras, la elección entre las implementaciones de la solución 2 y la solución 3 suele ser simplemente una preferencia personal.

Tarea a realizar

Con esta información, y viendo la implementación de la base de datos de fórmula 1, modela la relación muchos a muchos entre ***races*** y ***drivers***, con su tabla de unión ***results***.