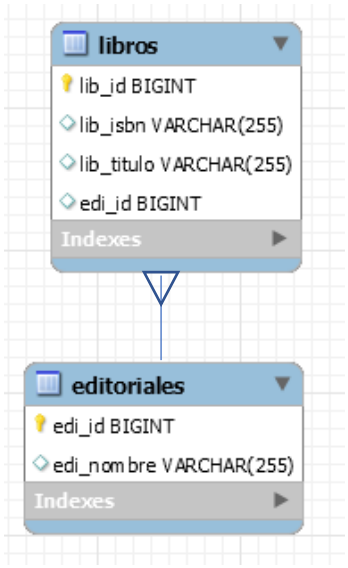
Relación @ManytoOne

Las relaciones Muchos a Uno (@ManyToOne) se caracterizan por entidades que están relacionadas con un solo elemento de un tipo determinado, pero esta relación no es excluyente, es decir, este último puede ser parte de varias, por lo tanto, tenemos una Entidad de tipo A, la cual puede estar relacionada con una Entidad de tipo B, pero al mismo tiempo, la Entidad tipo B puede pertenecer a otras instancias de la Entidad A.

En este tipo particular de relación solamente el lado muchos, sabe la existencia de la otra entidad. Un ejemplo clásico es el siguiente




```
18 @Entity
19 @Table(name = "libros")
20 public class Libro implements Serializable{
21     /**
24     private static final long serialVersionUID = 1L;
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     @Column(name = "lib_id")
28     private Long id;
29     @Column(name = "lib_isbn")
30     private String isbn;
31     @Column(name = "lib_titulo")
32     private String titulo;
33     @ManyToOne(fetch = FetchType.LAZY)
34     @JoinColumn(name = "edi_id")
35     private Editorial editorial;
```

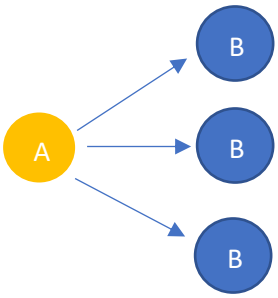
El de la izquierda es el modelo relacional de las tablas involucradas. Todo libro es publicado por alguna editorial. Esta editorial forma parte de los registros de la tabla editoriales. Por ese motivo no es posible que un libro tenga una editorial que no esté registrada en la tabla de editoriales. Por otro lado, una editorial puede publicar muchos libros diferentes. Este tipo de relación de tablas se representa con la clave foránea, que, para resumirlo de manera práctica, es la clave primaria de la tabla editoriales, presente en la tabla e libros como un campo más.

Para representar esta relación del modelo relacional en el mundo de los objetos, se establece que un atributo de la clase Libro es de tipo editorial (línea 35). En la línea 33 se establece la anotación @ManyToOne, para indicar que el objeto de este tipo es una entidad existente, la entidad Editorial. En la línea 34 se agrega la anotación @JoinColumn, que es fundamental para indicar que la unión de las dos tablas se hace precisamente por la clave primaria de la tabla editoriales (que es la clave foránea de la tabla libros). Para agregar fetch indica la estrategia por la cual se recuperará el estado del atributo editorial (es decir sus datos). La estrategia LAZY indica que se lo recuperará cuando se lo requiera únicamente. Esto ahorra procesos de consultas a la base de datos.

Finalmente observe que no se ha realizado ninguna operación sobre la entidad Editorial, respecto e Libro. Recuerde que se aclaró que, en este tipo de relaciones, solamente el lado muchos sabe de la existencia de la otra entidad.

Relación @OneToMany

En las relaciones Uno a Muchos, un objeto de la entidad "A" (el lado uno) está relacionado con muchos objetos de la entidad "B" (el lado muchos). Es una relación unidireccional, la instancia de la entidad "A" tiene una referencia a los objetos (observe que se pone en plural) de tipo "B" y esta relación está representada por una colección (un List o un Set). Así una instancia de la entidad "A" puede acceder a cada uno de los objetos de tipo "B" de esa colección; pero no de forma viceversa. Este tipo de relación se vería de forma gráfica más o menos así:



Donde un objeto de la entidad A conoce a varios objetos de la instancia B, es más en realidad se dice que esa instancia de la Entidad A “tiene” varios objetos de la instancia B.

Para brindar suponga el siguiente ejemplo:

Está creando una aplicación donde se ingresa por medio de la autenticación de usuarios. Este usuario puede tendrá como mínimo un nombre de usuario y su password. En muchas aplicaciones además se le asigna un rol. Este rol permite organizar entre otras cosas, los permisos de acceso del usuario (algunos usuarios solo pueden entrar a páginas de consultas, otros pueden crear, modificar o eliminar registros en diversas tablas). Muchas veces es suficiente con que un usuario tenga un único rol, es decir se conecta con su nombre y contraseña y ya tiene predeterminado su rol.

En otras situaciones, es probable que un mismo usuario tenga en la aplicación diversos roles. Supongamos que ese es el caso que queremos aplicar ahora: Tenemos una entidad Usuario que tiene uno o más roles, donde cada rol es una instancia de la entidad Rol, así quedarían definidos en el mundo de los objetos

```java
@Entity
@Component
@Table(name = "roles")
public class Rol implements Serializable{

    /**
     * se asocia a la clave primaria de un registro
     */
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "rol_id")
    private Long id;
    /**
     * Representa un tipo específico de rol
     */
    @Column(name = "rol_tipo", length = 20, nullable = true)
    private String tipo;
    /**
     * Representa una descripción del rol especificado
     */
    @Column(name = "rol_descripcion",length = 220, nullable = true)
    private String descripcion;
```

```java
@Entity
@Table(name = "usuarios")
public class Usuario implements Serializable{
    /**
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "usu_id")
    private Long id;
    @Column(name = "usu_nombre_u")
    private String nombreUsuario;
    @Column(name = "usu_password")
    private String password;
    @OneToMany(cascade = CascadeType.ALL,
               fetch = FetchType.EAGER)
    private List<Rol> roles;
```

La definición de la izquierda es la Entidad Rol. Observe que no hay nada particular con respecto a anotaciones de relaciones entre entidades. Como dijimos anteriormente en las relaciones @OneToMany solamente el lado muchos, conoce a las instancias de la otra entidad.

Por otro lado, en la definición de la clase Usuario (el de la derecha) se ha declarado un atributo que es una colección de objetos de tipo Rol (línea 34). Es decir, un usuario puede tener más de un rol. Para establecer esa relación en la línea 32 se establece la relación @OneToMany, indicando solamente [y de manera opcional] las estrategias de actualización y recuperación de información.

La pregunta sería, como genera la implementación JPA las tablas necesarias para establecer esta relación:

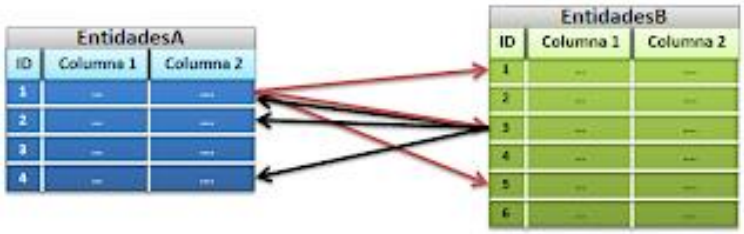
Se han creado 3 tablas:

- La tabla usuarios, permite almacenar los datos de los atributos de la entidad Usuario con excepción de la lista de roles
- La tabla roles, permite almacenar todos los datos de los atributos de la entidad Rol
- La tabla usuarios_roles permite guardar la relación establecida por los roles que posee un usuario en particular. Esta tabla se denomina Tabla de Unión. Tanto la tabla usuarios como roles poseen un campo que es la clave primaria (indicadas en las entidades respectivas mediante la anotación @Id) Estos campos permiten indicar que cada registro en las tablas es único. Entonces en esta tabla intermedia (usuarios_roles) se colocan las claves primarias de ambas, así es posible saber por medio de la clave primaria de un usuario, cuáles son sus roles. En el mundo de las tablas es la única manera de establecer esta necesidad de relación que requiere @OneToMany

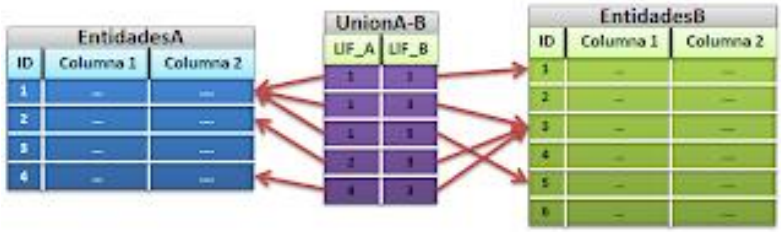
Relación @ManyToMany

Las relaciones Mucho a Muchos (@ManyToMany) se caracterizan por entidades que están relacionadas con muchos elementos de un tipo determinado, pero al mismo tiempo, cada uno de estos últimos no son exclusivos de esa entidad en particular, si no que pueden ser parte de varios, por lo tanto, tenemos una Entidad de tipo A, la cual puede estar relacionada con muchas Entidades de tipo B, pero al mismo tiempo, la Entidad tipo B puede pertenecer a varias instancias de la Entidad A. Este tipo de relaciones es muy común cuando trabajamos con bases de datos relacionales, por eso es importante saber cómo trabajar con ellas. Normalmente los registros en base de datos se relacionan usando una clave foránea de una tabla con el identificador de otra tabla, como en la siguiente imagen:

Algo muy importante a tomar en cuenta cuando trabajamos con relaciones @ManyToMany, es que en realidad este tipo de relaciones no existen físicamente en la base de datos, y en su lugar, es necesario crear una tabla intermedia que relacione las dos Entidades del modelo de datos porque para relacionar muchos registros de la entidad A de la base de datos, con muchos registros de la entidad B (también de la base de datos), NO basta con una clave foránea en una de las tablas para poder establecer la relación existente

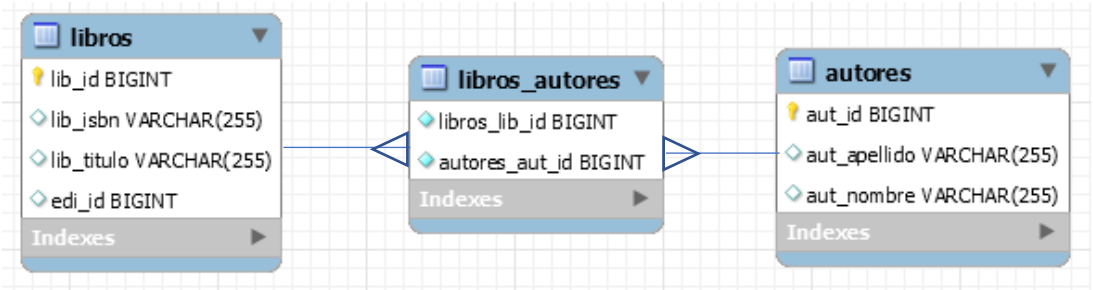


Por este motivo, la tercera tabla es conocida como join table, o tabla de enlace, o tabla de unión:



Esta tercera tabla lo único que hace es mantener las relaciones de las entidades A que están relacionadas con las entidades B, y las entidades B que están relacionadas con las entidades A. Para lograrlo, mantiene los identificadores de ambas tablas (sus respectivas claves primarias) como claves foráneas, esto es lo que mantiene esta tabla de unión.

Un ejemplo clásico de estas relaciones son los libros con sus autores, de esta forma, un libro puede tener varios autores, y a su vez, los autores pueden haber escrito muchos libros. Pero para que quede más claro, veamos como quedarían las tablas Libros, Autores y su tabla de enlace libros_autores



Para lograr este mapeo se procede la siguiente manera que indica la Figura 1:

En la Entidad Libro se agrega una colección de objetos de tipo Autor (línea 37) como atributo, esto es, un Libro puede tener varios autores. En la línea 36 se puede observar que se coloca la anotación @ManyToMany. Esta relación sería suficiente, para que se cree la tabla de enlace, logrando guardar en el mundo de las tablas relacionales cuales son los autores de una instancia de la entidad Libro, entonces se dice que esta es una relación @ManyToMany unidireccional.

```
20 public class Libro implements Serializable{
21     /**..
24     private static final long serialVersionUID = 1L;
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     @Column(name = "lib_id")
28     private Long id;
29     @Column(name = "lib_isbn")
30     private String isbn;
31     @Column(name = "lib_titulo")
32     private String titulo;
33     @ManyToOne(fetch = FetchType.LAZY)
34     @JoinColumn(name = "edi_id")
35     private Editorial editorial;
36     @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
37     private List<Autor> autores;
```

Figura 1

Por otro lado, si en la clase Autor ud quiere recuperar los libros que este autor ha escrito, proceda de manera similar

```
14 @Entity
15 @Table(name = "autores")
16 public class Autor implements Serializable{
17     /**..
20     private static final long serialVersionUID = 1L;
21     @Id
22     @GeneratedValue(strategy = GenerationType.IDENTITY)
23     @Column(name = "aut_id")
24     private Long id;
25     @Column(name = "aut_nombre")
26     private String nombre;
27     @Column(name = "aut_apellido")
28     private String apellido;
29     @ManyToMany(mappedBy = "autores")
30     private List<Libro> libros;
```

Se ha creado la lista de objetos tipo Libro, y se ha agregado la relación @ManyToMany, pero debido a que la misma relación existe en Libro, aquí se agrega la propiedad mappedBy, donde se indica el nombre del atributo que está listado en la otra entidad.

Entonces, no se creará una nueva tabla de enlace, sino que se utilizará la preexistente. Se dice entonces que esta es una relación @ManyToMany bidireccional.

Convirtiendo la relación @OneToMany en bidireccional

Observe nuevamente la relación existente entre Libro y Editorial:

```
12 @Entity
13 @Table(name = "editoriales")
14 public class Editorial implements Serializable{
15
16     /**..
19     private static final long serialVersionUID = 1L;
20     @Id
21     @GeneratedValue(strategy = GenerationType.IDENTITY)
22     @Column(name = "edi_id")
23     private Long id;
24     @Column(name = "edi_nombre")
25     private String nombreEditorial;
```

```
18 @Entity
19 @Table(name = "libros")
20 public class Libro implements Serializable{
21     /**..
24     private static final long serialVersionUID = 1L;
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     @Column(name = "lib_id")
28     private Long id;
29     @Column(name = "lib_isbn")
30     private String isbn;
31     @Column(name = "lib_titulo")
32     private String titulo;
33     @ManyToOne(fetch = FetchType.LAZY)
34     @JoinColumn(name = "edi_id")
35     private Editorial editorial;
```

Probablemente ud se plantee lo siguiente: Puedo saber cuál es la editorial de un libro. ¿Puedo saber cuales son los libros publicados por una editorial, desde la editorial? La respuesta es no.

Podría crear una búsqueda de libros por una editorial específica. Particularmente creo que esta es un estrategia simple y funcional.

Sin embargo, también puede utilizar la anotación @OneToMany de tal manera que sea bidireccional, para lo cual se necesita la anotación @ManyToOne. Observe que ya tenemos esto, en Libro tenemos definida esta relación (líneas 33 a 35), por lo tanto, en la entidad Editorial se puede agregar lo siguiente:

```
16  @Entity
17  @Table(name = "usuarios")
18  public class Usuario implements Serializable{
19      /**..
22      private static final long serialVersionUID = 1L;
23
24      @Id
25      @GeneratedValue(strategy = GenerationType.IDENTITY)
26      @Column(name = "usu_id")
27      private Long id;
28      @Column(name = "usu_nombre_u")
29      private String nombreUsuario;
30      @Column(name = "usu_password")
31      private String password;
32      @OneToMany(cascade = CascadeType.ALL,
33                 fetch = FetchType.EAGER)
34      private List<Rol> roles;
```

Esto es válido, aunque generará una nueva tabla intermedia, debido a que como se mencionó anteriormente es la única forma de crear este tipo de relaciones en el mundo de las tablas relacionales.

Entonces ¿Cuál es la diferencia entre @ManyToMany y la combinación @OneToMany - @ManyToOne?

Pues que queda establecida que el libro solo posee una editorial, mientras que la editorial puede tener muchos libros en el modelo de objetos.

Nuevamente, considere cuando es conveniente utilizar la relación @OneToMany bidireccional, en el caso particular de este ejemplo, parece razonable únicamente mantener la relación @ManyToOne en Libro y no agregar la @OneToMany en Editorial, ya que se crea innecesariamente una tabla de enlace (recuerde que, para este caso puede obtener el mismo resultado haciendo una búsqueda de libros por una entidad).