**RELACIONES DE ENTIDAD EN JPA/HIBERNATE**

No hace falta decir que el concepto ORM (Mapeo relacional de objetos) ha facilitado la vida de los desarrolladores. Si necesitamos mantener una relación entre dos tablas, la utilización de una anotación en nuestra entidad/clase es más que suficiente. Además, no necesitamos diseñar las tablas de la base de datos. Sin embargo, los desarrolladores deben comprender mejor el uso de las anotaciones que crean una relación entre dos tablas.

**¿Qué es una relación unidireccional y bidireccional?**

Usando la técnica ORM, podemos configurar una relación como unidireccional o bidireccional. Una relación unidireccional significa que el flujo de datos es solo en una dirección. No hace falta decir que una relación bidireccional significa que el flujo de datos es en ambas direcciones. Además, una relación bidireccional tiene tanto un lado propietario como un lado inverso. Una relación unidireccional tiene sólo un lado propietario. El lado propietario de una relación determina cómo se actualiza la relación en la base de datos. Ahora entendámoslo programáticamente. En una relación unidireccional, solo una entidad tiene un campo de relación o una propiedad que hace referencia a la otra. En una relación bidireccional, cada entidad tiene un campo de relación o propiedad que hace referencia a la otra entidad.

**¿Qué es el lado propietario y el lado inverso/de referencia en un ORM?**

De hecho, el lado propietario y el lado inverso son las terminologías técnicas de la tecnología ORM. No son los conceptos de entidades participantes, son los dos lados de una relación bidireccional. El lado propietario inicia la creación de la relación con la base de datos. Generalmente, este es el lado donde reside la clave externa.

Sin embargo, desde el punto de vista de la base de datos, la entidad lateral propietaria (entidad propietaria) tendrá una columna de clave externa. Obviamente, el otro lado restante es el lado inverso o el lado de referencia. El lado inverso mapea el lado propietario usando el atributo mappedBy. Además, la gente de la industria también los llama roles de la entidad. En cada relación hay dos entidades que están relacionadas entre sí, cada entidad desempeña un papel que es Entidad propietaria o Entidad no propietaria.

**¿Cuáles son las reglas que se deben seguir para una relación bidireccional?**

Las relaciones bidireccionales deben seguir estas reglas, como se menciona a continuación:

1º. El lado inverso de una relación bidireccional debe hacer referencia a su lado propietario mediante el uso del elemento mappedBy de la anotación @OneToOne, @OneToMany o @ManyToMany. El atributo mappedBy proporciona el valor de propiedad o campo en la entidad propietaria de la relación.

2º. El lado muchos de las relaciones bidireccionales de muchos a uno no deben definir el elemento mappedBy. El lado de muchos es siempre el lado propietario de la relación. De hecho, ManyToOne es siempre el lado propietario de una asociación bidireccional. De manera similar, OneToMany es siempre el lado inverso de una asociación bidireccional.

3º. Para relaciones bidireccionales uno a uno, el lado propietario corresponde al lado que contiene la clave foránea correspondiente.

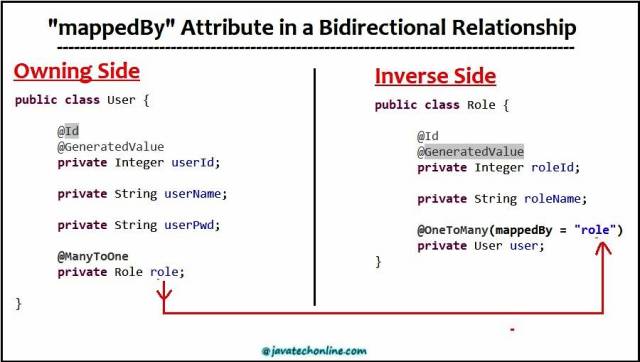
4º. Para relaciones bidireccionales de muchos a muchos, cualquiera de los lados puede ser el lado propietario. Por lo tanto, nosotros mismos podemos elegir el lado propietario de una asociación de muchos a muchos.

**¿Por qué usamos el atributo mappedBy?**

El atributo mappedBy contiene el nombre del campo de asociación en el lado propietario. Aplicamos el atributo mappedBy en el lado inverso/no propietario de la relación. El atributo indica a hibernate/JPA que no cree una columna adicional para este campo en la tabla, ya que ya se creó en el lado propietario.

En otras palabras, cuando usamos el atributo mappedBy, significa que la relación entre las entidades ya se ha mapeado en el lado propietario, así que no lo haga dos veces. Además, mappedBy indica que la relación entre entidades es bidireccional.

Si no usamos el atributo mappedBy, representa dos relaciones unidireccionales. Además, habrá una columna de mapeo adicional en cada dirección. Por ejemplo, la siguiente captura de pantalla representa el uso del atributo mappedBy.

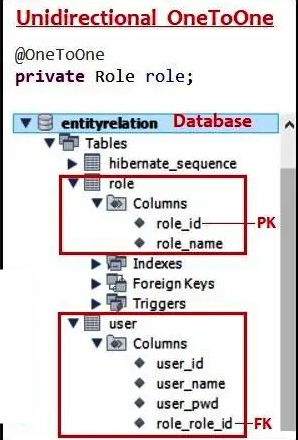


**Caso de uso:** Supongamos que tenemos que mantener una relación entre la tabla Usuario y Rol. Para satisfacer la relación uno a uno entre el usuario y la tabla de roles, un usuario tendrá solo un rol.

**Uno a uno: Unidireccional**

Podemos obtener una relación unidireccional entre la entidad Usuario y Rol aplicando @OneToOne en el campo relacional en cualquier lado. Por ejemplo, si queremos tener una relación uno a uno de Usuario a Rol, necesitamos agregar un campo con tipo Rol en la entidad Usuario.

Significa que un usuario tendrá un rol. Por lo tanto, debemos aplicar @OneToOne en el campo con un tipo Rol en la entidad Usuario.



**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToOne;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

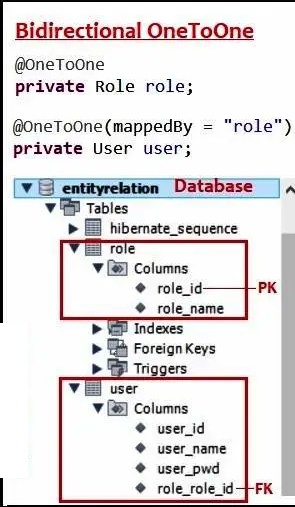
**@OneToOne**

**private Role role;**

**}**

**Uno a uno: bidireccional**

Para satisfacer la relación bidireccional, debemos aplicar @OneToOne en ambos lados, es decir. en el campo de tipo Rol en la entidad Usuario y también en el campo de tipo Usuario en la entidad Rol. Además, necesitamos tener el atributo mappedBy en cualquier lado de @OneToOne para decirle a JPA/Hibernate que el mapeo ya se realizó en el otro lado y no crear una columna adicional. Por ejemplo, el siguiente código demuestra cómo crearemos esta relación entre dos tablas usando la anotación @OneToOne.



**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToOne;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@OneToOne**

**private Role role;**

**}**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToOne;**

**@Entity**

**public class Role {**

**@Id**

**@GeneratedValue**

**private Integer roleId;**

**private String roleName;**

**@OneToOne(mappedBy = "role")**

**private User user;**

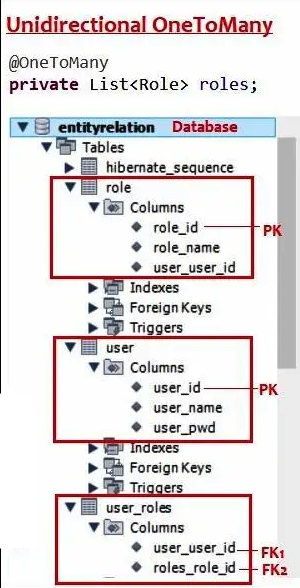
**}**

**Uno a muchos**

**Caso de uso:** Supongamos que tenemos que mantener una relación de uno a muchos entre la tabla Usuario y Rol. Para satisfacer la relación de uno a muchos entre el usuario y la tabla de roles, un usuario tendrá múltiples roles.

**Uno a muchos: unidireccional**

Podemos obtener una relación unidireccional entre la entidad Usuario y Rol aplicando @OneToMany en el campo relacional en cualquier lado. Por ejemplo, si queremos tener una relación de uno a muchos de Usuario a Rol, necesitamos agregar un campo con el tipo Lista<Rol> en la entidad Usuario. Significa que un usuario tendrá muchos roles. Por lo tanto, debemos aplicar @OneToMany en el campo con un tipo List<Role> en la entidad Usuario. Por ejemplo, el siguiente código demuestra el concepto.



**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToMany;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@OneToMany**

**private List<Role> roles;**

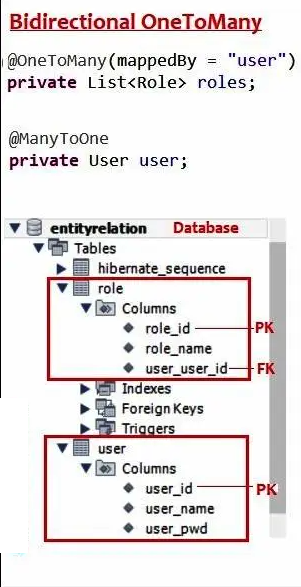
**}**

**Uno a muchos: bidireccional**

Para satisfacer la relación bidireccional, debemos aplicar @OneToMany en un lado, es decir. en el campo con tipo List<Role> en la entidad Usuario y @ManyToOne en el otro lado también, es decir. en el campo de tipo Usuario en la entidad Rol.

Además, necesitamos tener el atributo mappedBy en @OneToMany para decirle a JPA/Hibernate que el mapeo ya está hecho por el otro lado y no crear una columna adicional.

Por ejemplo, el siguiente código demuestra cómo crearemos esta relación entre dos tablas usando la anotación @ManyToOne y @OneToMany.



**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToMany;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@OneToMany(mappedBy = "user")**

**private List<Role> roles;**

**}**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToOne;**

**@Entity**

**public class Role {**

**@Id**

**@GeneratedValue**

**private Integer roleId;**

**private String roleName;**

**@ManyToOne**

**private User user;**

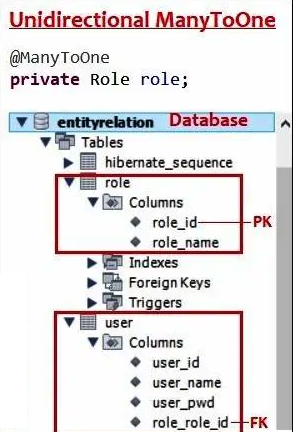
**}**

**muchos a uno**

**Caso de uso:** Supongamos que tenemos que mantener una relación entre la tabla Usuario y Rol. Para satisfacer la relación muchos a uno entre el usuario y la tabla de funciones, varios usuarios tendrán una sola función.

**Muchos a uno: unidireccional**

Podemos obtener una relación unidireccional entre la entidad Usuario y Rol aplicando @ManyToOne en el campo relacional en cualquier lado. Por ejemplo, si queremos tener una relación de muchos a uno de Usuario a Rol, necesitamos agregar un campo con tipo Rol en la entidad Usuario. Significa que muchos usuarios tendrán un rol. Por lo tanto, debemos aplicar @ManyToOne en el campo con un tipo Rol en la entidad Usuario. Por ejemplo, el siguiente código demuestra el concepto.



**import javax.persistence.Entity;**

**Import Java.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**Import Java.persistence.ManyToOne;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

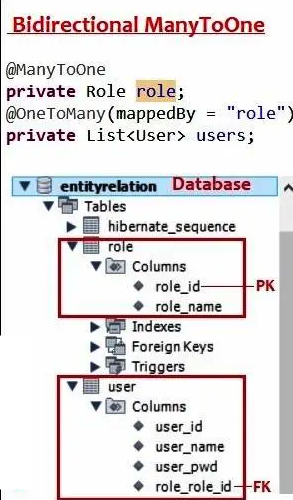
**@ManyToOne**

**private Role role;**

**}**

**Muchos a uno: bidireccional**

Para satisfacer la relación bidireccional, debemos aplicar @ManyToOne en un lado, es decir, en el campo con tipo Rol en la entidad Usuario y @OneToMany en el otro lado también, es decir, en el campo con tipo List<User> en la entidad Rol. Además, necesitamos tener el atributo mappedBy en @OneToMany para decirle a JPA/Hibernate que el mapeo ya está hecho por el otro lado y no crear una columna adicional. Por ejemplo, el siguiente código demuestra cómo crearemos esta relación entre dos tablas usando la anotación @ManyToOne y @OneToMany.



**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToOne;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@ManyToOne**

**private Role role;**

**}**

**import java.util.List;**

**import javax.persistence.CascadeType;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.OneToMany;**

**@Entity**

**public class Role {**

**@Id**

**@GeneratedValue**

**private Integer roleId;**

**private String roleName;**

**@OneToMany(mappedBy = "role", cascade = CascadeType.ALL)**

**private List<User> users;**

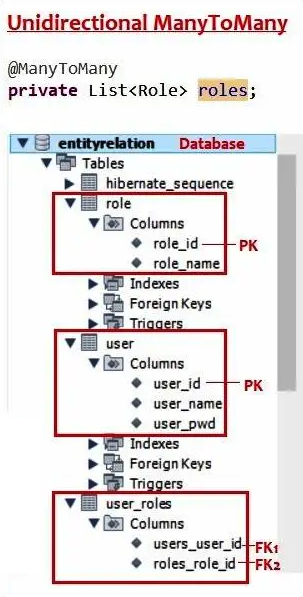
**}**

**Muchos a muchos**

**Caso de uso:** Supongamos que tenemos que mantener una relación de muchos a muchos entre la tabla Usuario y Rol. Para satisfacer la relación de muchos a muchos entre el usuario y la tabla de roles, varios usuarios tendrán múltiples roles.

**Muchos a muchos: unidireccional**

Podemos obtener una relación unidireccional entre la entidad Usuario y Rol aplicando @ManyToMany en el campo relacional en cualquier lado. Por ejemplo, si queremos tener una relación de muchos a muchos de Usuario a Rol, necesitamos agregar un campo con el tipo Lista<Rol> en la entidad Usuario. Significa que muchos usuarios tendrán muchos roles. Por lo tanto, debemos aplicar @ManyToMany en el campo con un tipo List<Role> en la entidad Usuario. Por ejemplo, el siguiente código demuestra el concepto.



**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToMany;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

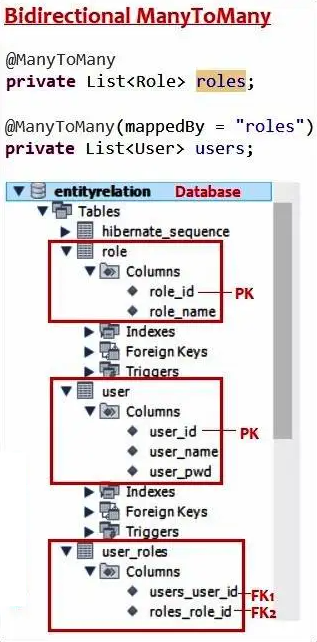
**@ManyToMany**

**private List<Role> roles;**

**}**

**Muchos a muchos: bidireccional**

Para satisfacer la relación bidireccional, debemos aplicar @ManyToMany en ambos lados, es decir. en el campo con tipo List<Role> en la entidad Usuario y @ManyToMany en el otro lado también, es decir. en el campo con tipo List<User> en la entidad Rol. Además, necesitamos tener el atributo mappedBy en cualquier lado para decirle a JPA/Hibernate que el mapeo ya se realizó en el otro lado y no crear una columna adicional. Por ejemplo, el siguiente código demuestra cómo crearemos esta relación entre dos tablas usando la anotación @ManyToMany.



**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToMany;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@ManyToMany**

**private List<Role> roles;**

**}**

**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToMany;**

**@Entity**

**public class Role {**

**@Id**

**@GeneratedValue**

**private Integer roleId;**

**private String roleName;**

**@ManyToMany(mappedBy = "roles")**

**private List<User> users;**

**}**

**¿Qué es el tipo de búsqueda en la relación de entidad?**

Fetch Type representa la estrategia para obtener datos de la base de datos. Tenemos dos tipos de estrategias; LAZY Y EAGER.

**FetchType. *LAZY*** indica que los datos deben obtenerse de forma diferida cuando se llama por primera vez. El proveedor de persistencia (JPA/Hibernate, etc.) recupera los datos de la entidad principal primero y luego recupera los datos de la entidad secundaria cuando se piden.

**FetchType. *EAGER*** indica que los datos deben buscarse cuando se llama por primera vez. El proveedor de persistencia (JPA/Hibernate, etc.) recupera los datos de la entidad principal y secundaria juntos a la vez.

**¿Qué es la carga lazy y eager?**

Supongamos que tenemos dos entidades y existe una relación entre ellas. Por ejemplo, podemos tener una entidad como Profesor y otra entidad como Alumno y un Profesor puede tener muchos alumnos. La entidad Profesor puede tener algunas propiedades básicas, como identificación, nombre, dirección, etc., así como una propiedad de colección como estudiantes que devuelve la lista de alumnos de un profesor determinado. Ahora, cuando cargamos un profesor de la base de datos, JPA carga sus campos de identificación, nombre y dirección por nosotros.

Tenemos dos opciones para cargar alumnos.

1. Cargarlos junto con el resto de los campos, es decir, eager.
2. Cargarlos a demanda (es decir, lazy) cuando llama al método getStudents() del profesor.

Cuando un profesor tiene muchos alumnos, no es eficiente cargar a todos sus alumnos junto con él, especialmente cuando no se necesitan y, en tales casos, podemos declarar que queremos que los alumnos se carguen cuando realmente se necesitan. Esto se llama carga diferida.

**¿Qué es la operación en cascada en la relación de entidad?**

Cada vez que manipulamos (insertar, eliminar, actualizar) filas en la tabla principal, también se manipularán las filas respectivas de la tabla secundaria con una columna clave coincidente. En la terminología de las bases de datos, lo llamamos efecto en cascada. Por ejemplo, un rol es parte de un usuario. Si intentamos eliminar el usuario, el rol también debe eliminarse. Esto se denomina relación de eliminación en cascada.

El tipo enumerado javax.persistence.CascadeType define las operaciones en cascada que se aplican en el elemento en cascada de las anotaciones de relación.

| **Operación en cascada** | **Descripción** |
| --- | --- |
| **CascadeType.ALL** | cascada. *ALL* es equivalente a cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}  Significa que estamos aplicando todas las operaciones en cascada a la entidad relacionada de la entidad matriz. Por ejemplo:  **@OneToMany(cascada = TipoCascada.**  **ALL )** |
| **CascadeType.DETACH** | Si separamos la entidad principal del contexto de persistencia, la entidad relacionada también se separará. |
| **CascadeType.MERGE** | Si fusionamos la entidad principal en el contexto de persistencia, la entidad relacionada también se fusionará. |
| **CascadeType.PERSIST** | Si persistimos la entidad principal en el contexto de persistencia, la entidad relacionada también persistirá. |
| **CascadeType.REFRESH** | Si actualizamos la entidad principal en el contexto de persistencia actual, la entidad relacionada también se actualizará. |
| **CascadeType.REMOVE** | Si eliminamos la entidad principal del contexto de persistencia actual, la entidad relacionada también se eliminará. |

**¿Qué es @JoinColumn y cuándo usarlo?**

La anotación @JoinColumn nos ofrece especificar la columna que usaremos para unir una asociación de entidades o una colección de elementos. Además, podemos usar esta anotación en cualquier entidad, ya sea principal o secundaria.

**¿Es obligatorio usar @JoinColumn?**

No es obligatorio proporcionar @JoinColumn al definir la relación. Si no lo proporcionamos en nuestro código, entonces el proveedor de persistencia (como Hibernate) generará automáticamente uno para usted, es decir, el nombre predeterminado para su columna. De forma predeterminada, utiliza los nombres de entidad y variable e incluye guiones bajos entre dos palabras. Sin embargo, si queremos asignar nombres de columna de nuestra elección, podemos hacerlo usando la anotación @JoinColumn.

**¿Qué es @JoinTable?**

Por lo general, cuando un mapeo crea una tabla separada para acomodar dos claves externas, lo llamamos Tabla de Unión. En esta tabla, una columna se llama Columna de unión y otra se llama Columna de unión inversa. Una tabla Join generalmente se crea cuando usamos @ManyToMany o @OneToMany unidireccional. Podemos especificar esta tabla usando la anotación @JoinTable. Si no especifica esta anotación, el proveedor de persistencia la creará con el nombre predeterminado.

**¿Qué es la columna de unión inversa?**

Como se mencionó anteriormente, las columnas de clave externa de la tabla de combinación que hacen referencia a la tabla principal de la entidad que no posee la relación. Ese es el lado inverso de la relación.

A continuación, se muestra un ejemplo del uso de @JoinTable, @JoinColumn e InverseJoinColumn.

**import java.util.List;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.Id;**

**import javax.persistence.JoinColumn;**

**import javax.persistence.JoinTable;**

**import javax.persistence.ManyToMany;**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer userId;**

**private String userName;**

**private String userPwd;**

**@ManyToMany**

**@JoinTable(name ="user\_role",**

**joinColumns = @JoinColumn(name = "user\_fk"),**

**inverseJoinColumns = @JoinColumn(name = "role\_fk")**

**)**

**private List<Role> roles;**

**}**