# Relaciones en Hibernate

Vamos a ver como implementar y manejar las relaciones de la base de datos en nuestras clases Hibernate.

Para el ejemplo sigo la base de datos de compras que vimos en un apartado anterior. Los archivos DDL están aquí:

https://github.com/juanpablofuentes/Java/blob/master/CRUD_BD_Compras/ddl.sql

## Uno a muchos

Vamos a ver en primer lugar la relación 1-N. Para ello voy a implementar la clase **Producto** que tiene ese tipo de relación con categorías, la clase que ya tengo. Procedo del mismo modo que en el ejercicio anterior, creo la clase, pongo los campos y los constructores:

```
package com.trifulcas.hibernate.entidades;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "productos")
public class Productos {

	@Id
	@Column(name = "idproducto")
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int idproducto;

	@Column(name = "nombre")
	private String nombre;

	@Column(name = "descripcion")
	private String descripcion;

	@Column(name = "stock")
	private int stock;


	public Productos() {

	}

	public Productos(String nombre, String descripcion, int stock) {
		super();
		this.nombre = nombre;
		this.descripcion = descripcion;
		this.stock = stock;
	}
```

```
	public int getIdproducto() {
		return idproducto;
	}

	public void setIdproducto(int idproducto) {
		this.idproducto = idproducto;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getDescripcion() {
		return descripcion;
	}

	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

	public int getStock() {
		return stock;
	}

	public void setStock(int stock) {
		this.stock = stock;
	}


	@Override
	public String toString() {
		return "Productos [idproducto=" + idproducto + ", nombre=" +
nombre + ", descripcion=" + descripcion
				+ ", stock=" + stock +"]";
	}


}
```

He hecho todo igual aunque me he dejado a propósito la categoría. En mi tabla es una id ¿Qué debería ser en la clase Producto? En realidad un Producto no tiene una id, sino que tiene una categoría. Así que para indicar que nuestro producto tiene una categoría no ponemos el id, sino un objeto de la clase categoría. Tal que así:

```
private Categorias categoria;
```

Añadiremos los getters y setters pertinentes:

```
public Categorias getCategoria() {
		return categoria;
```

```
	}

	public void setCategoria(Categorias categoria) {
		this.categoria = categoria;
	}
```

Y ahora viene lo bueno. Tenemos que indicar que este campo es de una relación muchos a uno (porque hay muchos de esta tabla que están relacionados con uno de la otra) Lo hacemos así:

```
	@ManyToOne(cascade= {CascadeType.PERSIST,CascadeType.MERGE,
CascadeType.DETACH, CascadeType.REFRESH})
	@JoinColumn(name="idcategoria")
	private Categorias categoria;
```

Indicamos que es muchos a uno con `@ManyToOne`. Con la propiedad cascade indicamos lo que queremos que pase cuando se modifique la clase madre, hemos puesto todos menos delete, porque no queremos que al borrar una categoría se borren los productos. En este enlace se explica el significado de cada uno de los tipos:

https://www.baeldung.com/jpa-cascade-types

La anotación `@JoinColumn` nos indica a que campo de la tabla hacemos referencia

También es conveniente poner en las categorías la lista de los productos asociados. Si en el caso anterior hemos puesto la anotación @ManyToOne en este caso ponemos @OneToMany:

```
@OneToMany(mappedBy="categoria",
			cascade= {CascadeType.PERSIST,CascadeType.MERGE,
CascadeType.DETACH, CascadeType.REFRESH})
	private List<Productos> productos;

	public List<Productos> getProductos() {
		return productos;
	}

	public void setProductos(List<Productos> productos) {
		this.productos = productos;
	}

	public void addProductos(Productos producto) {
		if (productos==null) {
			productos=new ArrayList<Productos>();
		}
		productos.add(producto);
		producto.setCategoria(this);
	}
```

Analicemos el código. En primer lugar tenemos la anotación `@OneToMany` para indicar la relación. La propiedad :

```
mappedBy="categoria",
```

hace referencia al campo de Productos que enlaza con la categoría. Recordemos que en productos teníamos:

```
private Categorias categoria;
```

Esta es la que enlaza. La propiedad de cascada es igual que en el caso anterior.

Un producto sólo tiene una categoría, pero una categoría tendrá varios productos, así que tendremos una lista de ellos. Finalmente implementaremos un ArrayList pero recordad lo que comenté sobre la tendencia de definir las variables por el interfaz y no por el tipo.

Después tenemos un getter y un setter clásico y por último un método que nos ayudará a añadir productos a una categoría. Comprobamos que la lista de productos no esté vacía, si lo está la creamos. Añadimos el producto a la lista y establecemos para ese producto como categoría la misma en la que estamos (this).

Con esto ya podemos manejar productos y categorías y asignar unos a otros y viceversa. Yo puedo crear un producto y asignarle una categoría o crear productos y añadirlos a la categoría.

Ejemplo de la primera manera:

```
	Productos tomate = new Productos("Tomate", "Rojo como el pecado", 10);
	Categorias cat = session.get(Categorias.class, 2);
	tomate.setCategoria(cat);
	session.save(tomate);
```

Ejemplo de la segunda:

```
Productos pimiento = new Productos("Pimiento", "Del Padrón", 5);
Productos coliflor = new Productos("Coliflor", "Blanca y radiante", 7);

cat.addProductos(coliflor);
cat.addProductos(pimiento);
session.save(coliflor);
session.save(pimiento);
```

Si vamos a la base de datos veremos que se han añadido los productos y asignados a sus categorías correspondientes.

En el ejemplo he recuperado una categoría existente pero podríamos crear una nueva… lo dejo como ejercicio.

Al estar relacionados cuando recuperamos un producto se carga también su categoría y viceversa. Lo podemos ver en los siguientes ejemplos:

Primero cambio el toString del producto:

```
@Override
	public String toString() {
		return "Productos [idproducto=" + idproducto + ", nombre=" +
nombre + ", descripcion=" + descripcion
				+ ", stock=" + stock +", categoría="+categoria+"]";
	}
```

Y luego cuando lo cargamos veremos la categoría:

```
		Productos prod = session.get(Productos.class, 1);
                System.out.println(prod);
```

```
Resultado:

Productos [idproducto=1, nombre=Manzana, descripcion=Manzana saludable,
stock=3, categoría=Categorias [idcategoria=1, nombre=Deliciosas frutas,
productos=Manzana|Pera|]]
```

Lo mismo en categoría, cambio el toString:

```
@Override
	public String toString() {
		String prods="";
		for(Productos p:productos) {
			prods+=p.getNombre()+"|";
		}
		return "Categorias [idcategoria=" + idcategoria + ", nombre=" +
nombre+", productos="+prods + "]";
	}
```

```
Y al leer la categoría saldrán sus productos:

	Categorias cat = session.get(Categorias.class, 1);
	System.out.println(cat);
	System.out.println(cat.getProductos());
```

```
Resultado:

Categorias [idcategoria=1, nombre=Deliciosas frutas, productos=Manzana|Pera|]
[Productos [idproducto=1, nombre=Manzana, descripcion=Manzana saludable,
stock=3, categoría=Categorias [idcategoria=1, nombre=Deliciosas frutas,
productos=Manzana|Pera|]], Productos [idproducto=2, nombre=Pera,
descripcion=No ha salido de un olmo, stock=13, categoría=Categorias
[idcategoria=1, nombre=Deliciosas frutas, productos=Manzana|Pera|]]]
```

¿Vemos algo extraño en estos resultados? Si lo has visto bien y si no no te preocupes que ya te lo cuento. Cuando cargamos la categoría del producto también se cargan sus productos ¿Es este un bucle infinito? No, pero tenemos que tener claras las opciones de carga dentro de Hibernate.

## Eager vs Lazy

Hay dos tipos de carga: **Eager** que carga los datos relacionados aunque no se hayan pedido y **Lazy** que carga los datos relacionados sólo cuando se piden. Cada relación tiene sus tipos predeterminados:

@OneToOne: Eager

@OneToMany: Lazy

@ManyToOne:Eager

@ManyToMany: Lazy

Pero nosotros podemos sobreescribir este tipo en la propia anotación indicando el tipo de fetch que queremos. Por ejemplo en productos tenemos un ManyToOne con Eager que nos recuperará la categoría aunque no la necesitemos. No es una mala opción, porque seguramente cuando recuperamos un producto será buena idea recuperar su categoría. Pero si quisiéramos demorar la carga lo más posible, lo podemos indicar en el campo de la siguiente manera:

```
@ManyToOne(cascade= {CascadeType.PERSIST,CascadeType.MERGE,
CascadeType.DETACH, CascadeType.REFRESH},
                    fetch=FetchType.LAZY)
      @JoinColumn(name="idcategoria")
      private Categorias categoria;
```

Ahora sólo se cargarán los datos de la Categoría cuando solicitemos información acerca de ella.

En la práctica nosotros no notaremos nada, ya que cuando necesitemos la información Hibernate se encargará de solicitarla. Pero si escogemos siempre el tipo de fetch Lazy ahorraremos recursos cuando se realice la primera petición.

Sí podremos notar diferencia si cerramos la sesión (`session.close();`) y accedemos al objeto. Si el tipo de fetch es eager los datos se habrán cargado y podremos acceder a los objetos relacionados. En caso contrario no se podrán cargar.

## Many to Many

Vamos a ver cómo implementar una relación N-N en Hibernate. En mi caso tengo dos tablas, productos y proveedores que tienen este tipo de relación. En la base de datos tengo una tabla producto proveedor. Tengo que crear la clase Proveedores que será más o menos como las que hemos hecho hasta ahora:

```
package com.trifulcas.hibernate.entidades;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
```

```java
@Table(name = "proveedores")
public class Proveedores {

	@Id
	@Column(name = "idproveedor")
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int idproveedor;

	@Column(name = "nombre")
	private String nombre;

	@Column(name = "nif")
	private String nif;

	@Column(name = "poblacion")
	private String poblacion;

	public Proveedores() {

	}

	public Proveedores(String nombre, String nif, String poblacion) {
		super();
		this.nombre = nombre;
		this.nif = nif;
		this.poblacion = poblacion;
	}

	public int getIdproveedor() {
		return idproveedor;
	}

	public void setIdproveedor(int idproveedor) {
		this.idproveedor = idproveedor;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getNif() {
		return nif;
	}

	public void setNif(String nif) {
		this.nif = nif;
	}

	public String getPoblacion() {
		return poblacion;
	}

	public void setPoblacion(String poblacion) {
		this.poblacion = poblacion;
	}
```

```
	@Override
	public String toString() {
		return "Proveedores [idproveedor=" + idproveedor + ", nombre=" +
nombre + ", nif=" + nif + ", poblacion="
				+ poblacion + "]";
	}

}
```

Ahora tenemos que indicar, tanto en productos como en proveedores, que tienen una relación N-N con la otra tabla. Ya que estamos en proveedores pongo el código que tenemos que poner en esta clase:

```
@ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE,
CascadeType.DETACH, CascadeType.REFRESH },
fetch = FetchType.LAZY)
	@JoinTable(name = "producto_proveedor",
	joinColumns = @JoinColumn(name="idproveedor"),
	inverseJoinColumns = @JoinColumn(name="idproducto"))
	private List<Productos> productos;

	public List<Productos> getProductos() {
		return productos;
	}

	public void setProductos(List<Productos> productos) {
		this.productos = productos;
	}

	public void addProducto(Productos producto) {
		if (productos == null) {
			productos = new ArrayList<Productos>();
		}
		productos.add(producto);

	}
```

Vamos a analizarlo línea por línea. La primera indica que tenemos una relación N-N el comportamiento en caso de cascada y el tipo de fetch:

```
@ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE,
CascadeType.DETACH, CascadeType.REFRESH },
fetch = FetchType.LAZY)
```

Nada extraño aquí. La siguiente línea es donde está la chicha. Indicamos cual es la tabla intermedia en la base de datos, cual es el campo que utilizamos para hacer el join, y cual es el campo que se va a utilizar para hacer el join con la otra entidad:

```
@JoinTable(name = "producto_proveedor",//Tabla de la BD
	joinColumns = @JoinColumn(name="idproveedor"), //Columna join
	inverseJoinColumns = @JoinColumn(name="idproducto"))//Columna join
otra tabla
```

En esta línea están todas las instrucciones para que Hibernate sepa como se implementa la relación N-N.

En Productos vamos a hacer exactamente lo mismo:

```
@ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE,
CascadeType.DETACH,
                    CascadeType.REFRESH }, fetch = FetchType.LAZY)
      @JoinTable(name = "producto_proveedor",
      joinColumns = @JoinColumn(name="idproducto"),
      inverseJoinColumns = @JoinColumn(name="idproveedor"))
      private List<Proveedores> proveedores;

      public List<Proveedores> getProvedores() {
            return proveedores;
      }

      public void setProvedores(List<Proveedores> proveedores) {
            this.proveedores = proveedores;
      }

      public void addProducto(Proveedores proveedor) {
            if (proveedores == null) {
                  proveedores = new ArrayList<Proveedores>();
            }
            proveedores.add(proveedor);

      }
```

Que es lo mismo pero desde el otro punto de vista. Una vez implementada la relación podemos crear Proveedores y Productos y añadirlos bien desde uno bien desde el otro y cuando guardemos los objetos se guardará también las relaciones. Veamos un ejemplo:

```
Proveedores paco= new Proveedores("Paco", "11111", "Turruncún");

session.save(paco);

Productos pepino = new Productos("Pepino", "Francés", 5);
Productos endivia = new Productos("Endivia", "Nada envidiosa", 7);

paco.addProducto(pepino);
paco.addProducto(endivia);
session.save(pepino);
session.save(endivia);
// commit de la transacción
session.getTransaction().commit();
```

En mi base de datos la tabla que relaciona productos y proveedores tiene un campo extra. En esta implementación no tenemos en cuenta ese campo, la anotación @ManyToMany sólo nos sirve para tablas intermedias sin campos añadidos. ¿Cómo podemos implementar ese caso? Creando en entidades una clase con los campos necesarios y creando dos relaciones @OneToMany.

## Código de ejemplo

Todo el código del ejercicio, incluyendo el sql de la base de datos, puede encontrarse aquí:

https://github.com/juanpablofuentes/Java/tree/master/hibernate-relaciones