

# Bootcamp Java Developer

Fase 3 - Java Architect  
Módulo 30



# Asociaciones

# Asociaciones

Permiten mapear objetos propios dentro de **Hibernate / JPA**. Se suelen usar especialmente para las colecciones. Para ello existen tres tipos de relaciones que generalmente podemos encontrar en un modelo relacional:

- **One to Many / Many to One**  
Uno a muchos / Muchos a uno.
- **One to One.**  
Uno a uno.
- **Many to Many.**  
Muchos a muchos.



## One to Many / Many to One

En esta clase de relaciones *Uno a muchos / Muchos a uno* existe una entidad que tiene como atributo una colección de entidades de otro tipo. A la entidad que posee la colección se la conoce como **extremo One**. A la entidad a la que pertenecen los elementos de la colección se la conoce como **extremo Many**. La entidad del *extremo Many* se relacionará como máximo con una entidad del *extremo One*.

El siguiente ejemplo permite comprender cómo funcionan estas relaciones.

### Ítem y Factura:

- Una *Factura* podrá tener *N ítems* (Extremo **One**).
- Cada *Ítem* pertenece como máximo a una *Factura*. (Extremo **Many**).




Las clases para este modelo serían:

### Factura

```
[code]
public class Factura {

    private Long codigo = null;
    private Set<Item> items = new HashSet<Item>();

}
[/code]
```



### Item

```
[code]
public class Item {

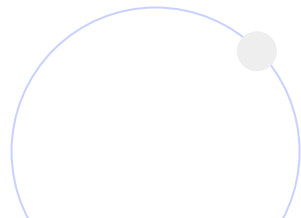
    private Long id = null;
    private Factura factura;
    private Long cantidad;
    private String nombreProducto;

}
[/code]
```



Para **definir cómo se mapearán las clases** hay **ciertos aspectos del negocio** que primeramente deberíamos tener definidos:

1. **En nuestro sistema agregaremos los ítems a la factura**, lo que es más común a la hora de trabajar con objetos complejos, en lugar de a cada ítem *setearle* la factura asociada.
2. **El ciclo de vida del ítem depende del de la factura**. Si no existe la factura no tienen sentido sus ítems. Si se elimina una factura, se eliminan también sus ítems.
3. **Acceso a la información**: siempre que traigo los datos de una factura traigo los datos de sus ítems, o accedo a los datos de los ítems en una segunda instancia. Existen las dos posibilidades, analizaremos las dos situaciones con las opciones que nos da Hibernate.
4. Existen otros aspectos que irán surgiendo a medida que avancemos, no obstante por ahora podemos comenzar a atacar el mapeo de los objetos.



Empezamos con **Item**:

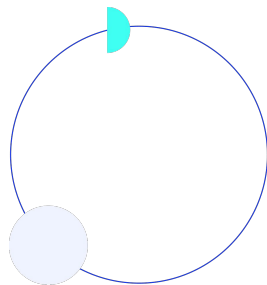
```
[code]
@Entity
@Table(name = "items")
public class Item {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "it_id")
    private Long id = null;

    @ManyToOne
    @JoinColumn(name = "fac_id")
    private Factura factura;

    @Column(name = "it_cantidad")
    private Long cantidad;

    @Column(name = "it_descripcion")
    private String nombreProducto;
}
[/code]
```



Este mapeo se lo conoce como ***mapeo unidireccional muchos a uno***.

Observaciones:

1. **@ManyToOne**: Este es el **extremo *Many*** de la relación. La tabla items tendrá una FK a la tabla facturas. Esta *annotation* puede contener diferentes configuraciones con respecto al tipo de *cascading* y el *fetching*, las cuales abordaremos más adelante.
2. **@JoinColumn(name = "fac\_id")**: Indica el nombre de la columna de la tabla a través de la cual se establece la relación.

Con este simple mapeo sería suficiente para establecer la relación entre ambas entidades, no obstante nos puede interesar recorrer la relación en forma bidireccional.





Para ello, mapearemos el otro extremo de la relación:

```
[code]
@Entity
@Table(name = "facturas")
public class Factura {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "fac_id")
    private Long codigo = null;

    @OneToMany
    @JoinColumn(name = "fac_id")
    private Set<Item> items = new HashSet<Item>();
}
[/code]
```

Observaciones:

1. **@OneToMany**: Explicita la relación desde el extremo "one". Esta *annotation* puede contener diferentes configuraciones con respecto al tipo de *cascading* y el *fetching*.

**De esta forma tenemos el mapeo básico de nuestras entidades**, no obstante todavía no tuvimos en cuenta los aspectos conversados al inicio del mapeo.



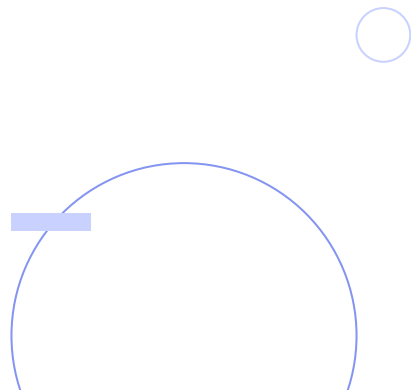
El punto uno se refería a la forma en que se iría armando el objeto compuesto, como los ítems se agregarían a la factura:

```
[code]
Factura f = new Factura();
Item i = new Item();
f.getItems().add(i);
[/code]
```

El objetivo sería lograr que **cuando se persista el objeto Factura también se persistan los ítems y queden relacionados mediante la FK.**

Por defecto Hibernate asume que los dos extremos de la relación se sincronizan con la base de datos. Con lo cual, tanto si agrego un ítem a una factura o viceversa, este cambio se persistirá en la base de datos.

Este comportamiento puede ser modificado mediante el concepto de **mappedBy** para lograr una solución más genérica aplicable a la mayoría de los casos.



## MappedBy

Con esta configuración podemos lograr que **las modificaciones que se realicen en la colección del extremo One no se sincronicen con la BD.**

```
[code]
@OneToMany(mappedBy="factura")
[/code]
```

Esto indica que el atributo persistente es **“factura”** en la clase **Item**. Así logramos que:

```
[code]
factura.getItems().add(item);
[/code]
```

**Sea ignorado** por la base de datos a la hora de persistir.

```
[code]
Item.setFactura(factura);
[/code]
```

**Sea persistido** en la base de datos, llegado el momento de persistir los cambios.



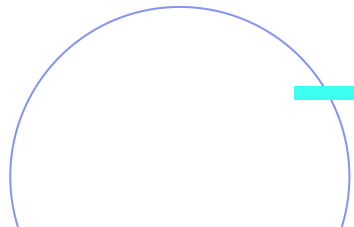
## Cascading

Este tema se encuentra relacionado con el segundo aspecto a tener en cuenta que planteamos en el inicio:

```
[code]
Factura fac = new Factura();
Item it = new Item();
fac.getItems().add(it);
DB.save(fac);
[/code]
```

El ciclo de vida de ambos objetos está relacionado, tanto si agrego como si remuevo un ítem en el momento de persistir la información en la DB todos los cambios quedan reflejados.

Para ello utilizamos el parámetro **Cascade**, este parámetro es **unidireccional**, puede declararse **en ambos extremos**.



```
[code]
@OneToMany(cascade = CascadeType.ALL, mappedBy = "factura")
private Set<Item> items = new HashSet<Item>();
[/code]
```

**CascadeType.ALL** indica que se propaguen todas las operaciones posibles desde la entidad fuente al destino. Un último aspecto a tener en cuenta con respecto al cascadeo es la existencia de objetos huérfanos, para lograr que al eliminar la relación entre un ítem y una factura el registro del ítem se elimine de la base de datos es necesario configurar la opción llamada **delete-orphan**:

```
[code]
@OneToMany(cascade = CascadeType.ALL, mappedBy="factura", orphanRemoval=true)
[/code]
```

# Fetching

El tercer punto que abordamos es la forma en que buscaremos la información a la base de datos, la diferencia radica en que si vamos a buscar una factura a la base de datos traemos siempre la información de los ítems, o esta información la traemos en una segunda instancia.



## Existen dos opciones:

- **LAZY:** al traer un ítem, no nos traerá aún la factura asociada. Al traer el ítem se realizará un SELECT y otro cuando se desee obtener la información de la factura.
- **EAGER:** realiza un Join entre la tabla ITEM y FACTURA, para poder cargar ambos objetos en el momento de obtener el ítem desde la Base de datos.

```
[code]  
@OneToMany(fetch = FetchType.EAGER/LAZY).  
[/code]
```

Por defecto, el fetching es **LAZY** para colecciones y **EAGER** en casos *One-To-Many*.



## One-To-One

Es una **relación simple uno a uno** donde una tabla tiene una FK a otra.

Veamos el ejemplo de la derecha:



```
[code]
@Entity
@Table(name = "CLIENTE")
public class Cliente {
    ...
        @OneToOne
        @JoinColumn(name = "DIR_ID")
        private Direccion direccion;
    ...
}

@Entity
@Table(name = "DIRECCION")
public class Direccion {
    ...
        @OneToOne(mappedBy="direccion")
        private Cliente cliente;
    ...
}
[/code]
```



## Many-To-Many

En un modelo relacional las relaciones ***muchos a muchos*** se encuentran representadas con una **tabla cruzada entre entidades**. En un modelo de objetos **no existen entidades para representar estas tablas cruzadas** ya que no es necesario para poder resolver la problemática.

A la hora de mapear esta relación **podemos optar por utilizar dos relaciones *Many-To-One***, de la forma ya vista, en caso de que tenga sentido para nuestro sistema mapear la tabla intermedia, ya sea porque posee información adicional o por algún otro motivo.

Si optamos por ignorar la tabla intermedia **podremos mapear una relación *Many-To-Many***.

Imaginemos un *Producto*, asociado al ítem en una relación *One-To-Many*, y a una clase *Categoría*, en una relación *Many-To-Many* con la clase *Producto*.



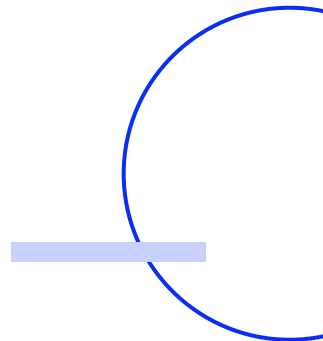
Para obtener los ítems asociados a la categoría a partir de la misma, en **forma unidireccional**:

```
[code]
@Entity
@Table(name = "CATEGORIA")
public class Categoria {

    ...

    @ManyToMany
    @JoinTable(name = "CATEGORIA_ITEM",
        joinColumns = {@JoinColumn(name = "CAT_ID")},
        inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")})
    private Set<Item> items = new HashSet<Item>();

    ...
}
[/code]
```



**Bidireccional:**

```
[code]
@ManyToMany(mappedBy = "items")
private Set<Categoria> categorias = new HashSet<Categoria>();
[/code]
```



**¡Sigamos  
trabajando!**