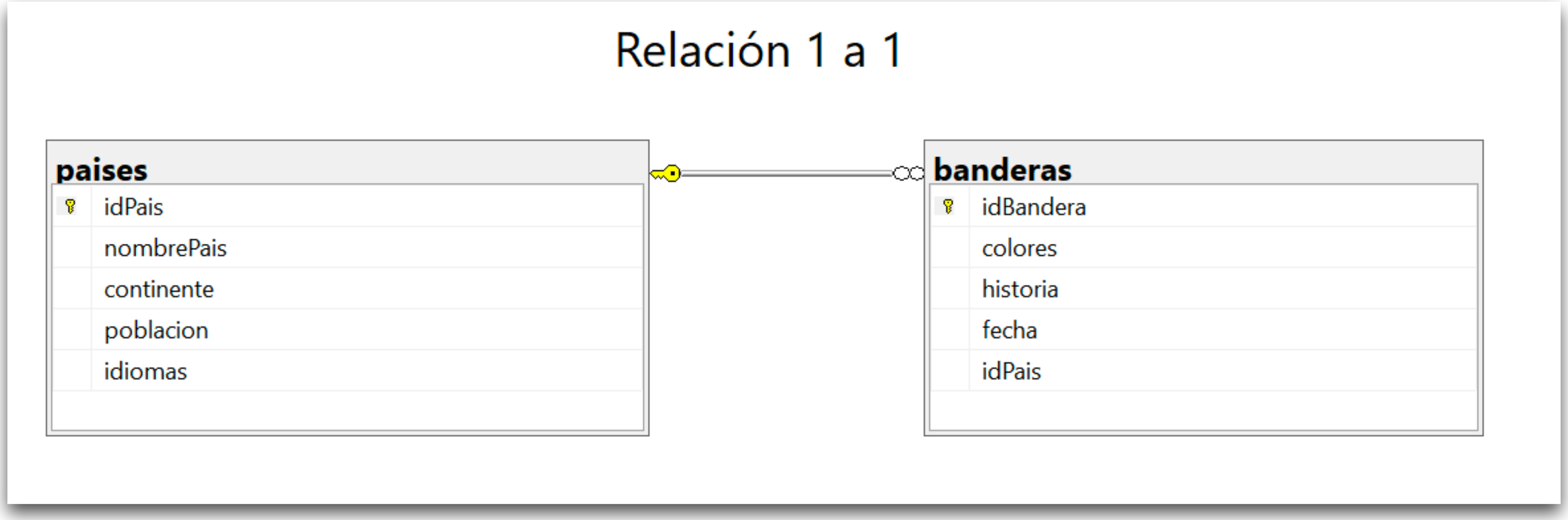
Trabajando con JPA

Relaciones entre entidades

Índice del capítulo

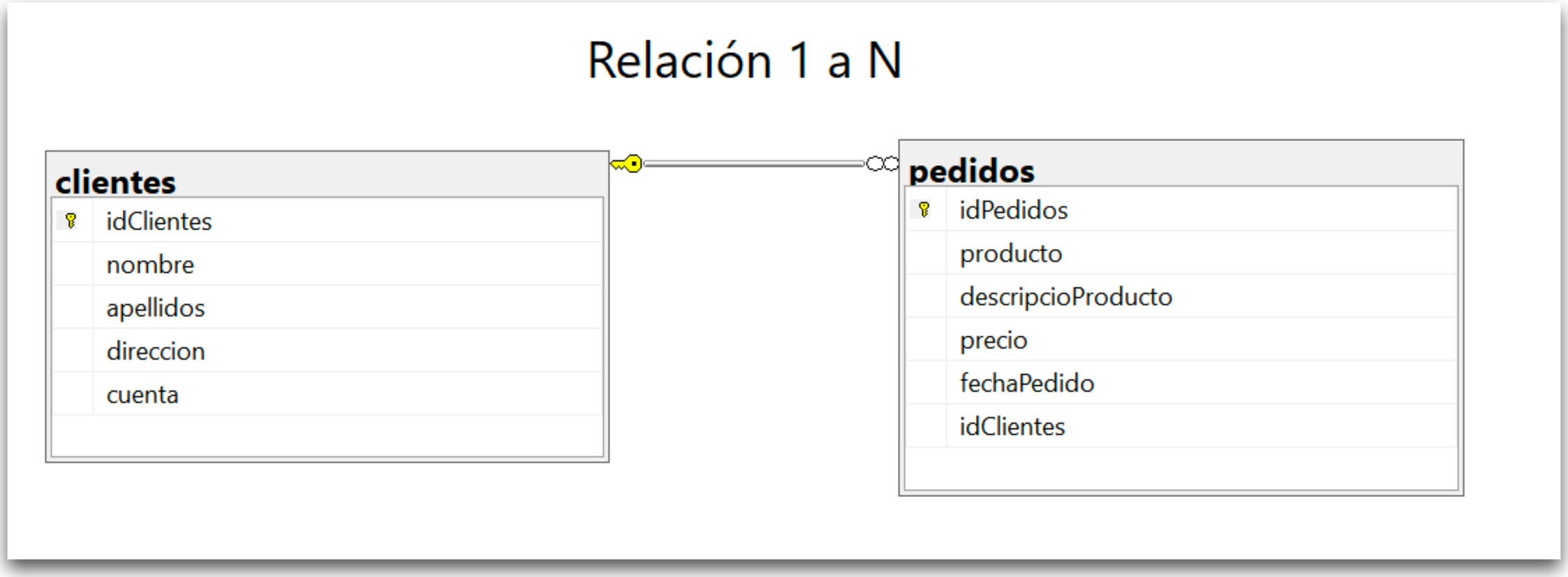
Introducción

- Existen **distintos tipos de relaciones** entre las entidades:
  - **Uno a Uno:** en esta relación, cada entidad está asociada directamente con una única entidad.



Introducción

- (cont.):
  - **Uno a Muchos:** cada entidad puede estar asociada con muchas. Normalmente esta asociación se fija con un objeto de tipo **List, Set, Map, SortedSet o SortedMap**.

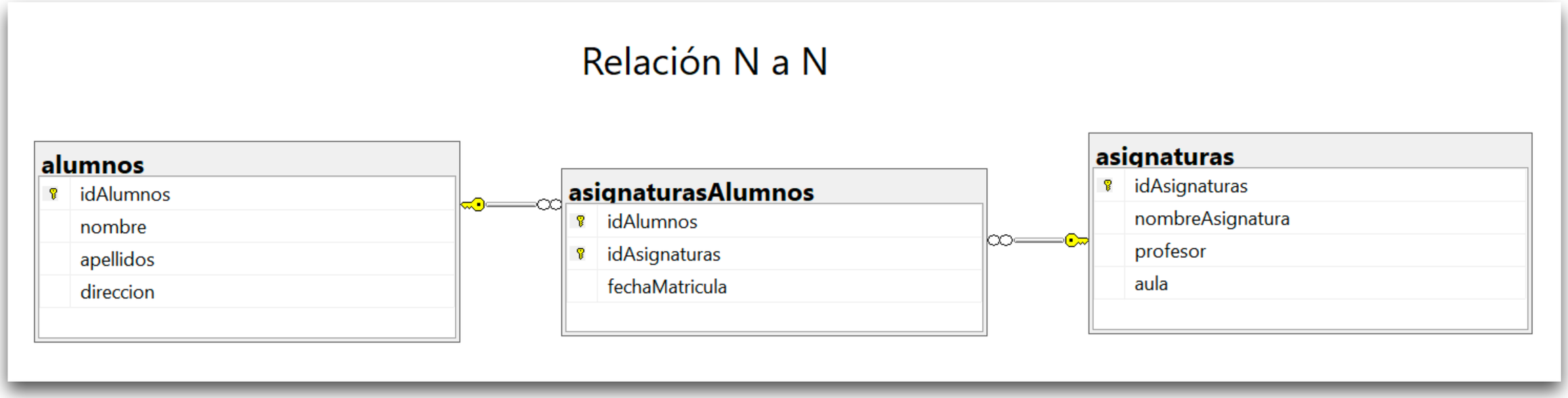


Introducción

- (cont.):
  - **Muchos a Uno:** es justo la relación opuesta a Uno a Muchos. Normalmente se concreta mediante el uso de un mapeo bidireccional.

Introducción

- (cont.):
  - **Muchos a Muchos**. varias entidades está relacionadas a su vez con varias entidades. Cada entidad tiene un objeto de tipo List o Set, que hace referencia a la otra entidad. Esto resulta en una tabla de unión en la que se definen las relaciones.
    - Un ejemplo de este caso de uso es **Productos** y **Categorías**: un **Producto** puede pertenecer a varias **Categorías**, y a su vez en cada **Categoría** puede haber varios **Productos**.



Introducción

- Atendiendo a la **direccionalidad**, la relación entre dos entidades puede ser:
  - **Unidireccional**: el mapeo se realiza **únicamente en una dirección**. Uno de los extremos de la relación no sabrá nada acerca del otro extremo
  - **Bidireccional**: ambos extremos de la relación saben de los dos. Esta es la opción recomendada por JPA, puesto que permite navegar por el gráfico de objetos en ambas direcciones.
- El **extremo propietario** de la relación será el que mantenga la **clave ajena (foreign key)** de la base de datos.
  - En una relación **Uno a Uno**, es el extremo donde se especifica la clave ajena.
  - En una relación **Uno a Muchos, o Muchos a Uno**, es el lado del "**Muchos**".

Introducción

- JPA nos permite definir estas **relaciones** mediante el uso de **anotaciones** sobre nuestras clases:
  - La anotación @OneToMany.
  - La anotación @ManyToOne.
  - La anotación @OneToOne.
  - La anotación @ManyToMany.

La anotaciones básicas

- Introducción.
- Relaciones unidireccionales one to many.
- Relaciones bidireccionales one to many/many to one.
- One to one.
- Many to many.

Introducción

- Las asociaciones **Uno a Uno (One to One)** se implementan mediante la **anotación @OneToOne**.
- Las asociaciones **Uno a Muchos (One to Many)** se implementan mediante la **anotación @OneToMany**.
- Las asociaciones **Muchos a Uno (Many to One)** se implementan mediante la **anotación @ManyToOne**.
- Las asociaciones **Muchos a Muchos (Many to Many)** se implementan mediante la **anotación @ManyToMany**.
- El **atributo mappedBy** se usa para definir el campo que posee la referencia en la relación.

Relaciones unidireccionales one to many

- Introducción.
- La anotación @OneToMany.
- Ejemplo.
- La anotación @JoinTable.
- La anotación @JoinColumn.

Introducción

- Las **relaciones uno a muchos** se caracterizan por incluir una entidad donde tenemos un **objeto principal** y **colección de objetos de otra entidad** relacionados directamente.
- Estas relaciones se definen mediante **colecciones**, pues tendremos una serie de objetos pertenecientes al objeto principal.
- Se desarrollan gracias a **la anotación @OneToMany**.

La anotación @OneToMany

- La anotación tiene los siguientes **atributos**:

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element	Description
`CascadeType[]`	`cascade`	(Optional) The operations that must be cascaded to the target of the association.
`FetchType`	`fetch`	(Optional) Whether the association should be lazily loaded or must be eagerly fetched.
`java.lang.String`	`mappedBy`	The field that owns the relationship.
`boolean`	`orphanRemoval`	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.
`java.lang.Class`	`targetEntity`	(Optional) The entity class that is the target of the association.

La anotación @OneToMany

- (cont.):
  - **cascade**: (Opcional) Las operaciones que se deben conectar en cascada al destino de la asociación.
  - **fetch**: (Opcional) Si la asociación debe cargarse de forma perezosa o debe buscarse con entusiasmo.
  - **mappedBy**: El campo que posee la relación.
  - **orphanRemoval**: (Opcional) booleano que indica si aplicar la operación de eliminación a las entidades que se han eliminado de la relación y aplicar en cascada la operación de eliminación a esas entidades.
  - **targetEntity**: (Opcional) La clase de entidad que es el destino de la asociación.

Ejemplo

- Un ejemplo clásico para entender este tipo de relaciones son las **facturas**, pues tendremos una Entidad cabecera donde tendremos los **datos principales de la factura**, como podría ser serie, cliente, total, fecha de expedición, etc.

- Por otra parte, la factura tendrá una **serie de líneas** que representa cada uno de los **productos vendidos**.

Ejemplo

- La clase Invoice:

```
@Entity
@Table(name="INVOICES")
public class Invoice {

   @Id
   @Column(name="ID")
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private long id;
   ...

   @OneToMany
   private List<InvoiceLine> lines;
```

Ejemplo

- Como la relación es **unidireccional**, InvoiceLine no hace referencia a Invoice de ninguna forma;

```
@Entity
@Table(name = "invoice_lines")
public class InvoiceLine {
   @Id
   @Column(name = "ID")
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(name = "PRODUCT", nullable = false)
   private String product;

   @Column(name = "PRICE", nullable = false)
   private double price;
   ...

}
```

Ejemplo

- Se se ejecuta el código y se analiza la estructura de la base de datos:

```
create table INVOICE (id bigint generated by default as identity, INVOICE_AMOUNT double, INVOICE_CIF varchar(255), DATE date, primary key (id))
create table INVOICE_INVOICE_LINE (Invoice_id bigint not null, lines_id bigint not null)
create table INVOICE_LINE (id bigint generated by default as identity, PRICE double not null, PRODUCT varchar(255) not null, primary key (id))
```

- En base de datos, el problema no se resuelve mediante una **clave ajena**, se hace mediante una tabla intermedia, lo que conocemos como **una JOIN TABLE**.

- Como podemos ver, JPA de forma automática decide el nombre de la tabla y sus columnas correspondientes. La **anotación @JoinTable** nos permite modificar ese comportamiento por defecto.

Ejemplo

- En el ejemplo anterior, si decidimos eliminar una factura, cualquier línea de factura asociada quedará como una **referencia pendiente u objeto huérfano**.
- Para resolver este problema, Hibernate proporciona **algunas propiedades** que se pueden utilizar para **propagar la eliminación** y **limpiar objetos huérfanos**.
- Ampliemos nuestra anotación @OneToMany en List<InvoiceLine> lines en el objeto Invoice. Esta anotación también se puede utilizar con varias opciones para personalizar el comportamiento y el mapeo de relaciones. La opción que puede resultar útil para la eliminación en cascada es **cascade**.
- Esencialmente, la cascada nos permite definir **qué operación** en la entidad principal debe conectarse en cascada a las **entidades secundarias** relacionadas.
- Se describe más adelante en **este capítulo**.

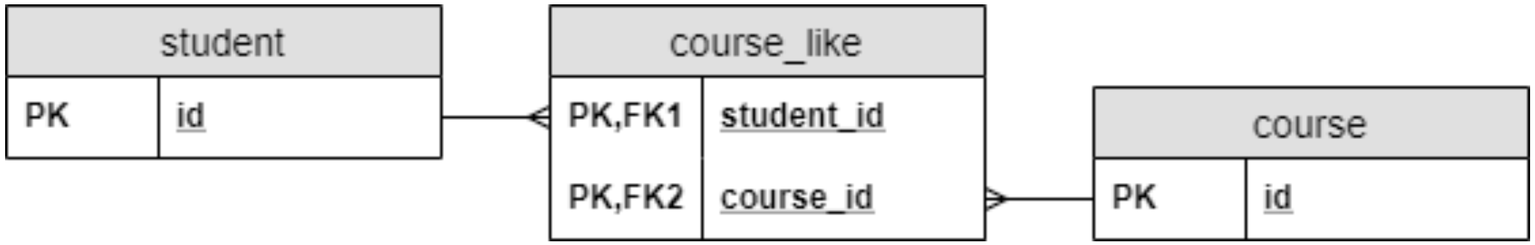
Ejemplo

- JPA también ofrece una opción para establecer el nivel cascada **para todas las operaciones**. Esto se llama **CascadeType.ALL**.
- Si solo queremos eliminar en cascada las entidades secundarias cuando se elimina la entidad principal, entonces podemos usar **CascadeType.Remove**.
- La segunda opción que usaremos junto con cascada es **orphanRemoval**. Nuestra factura quedaría de la siguiente forma:

```
@Entity
@Table(name="INVOICES")
public class Invoice {
   ...

   @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
   private List<InvoiceLine> lines;
```

La anotación @JoinTable

- Se usa para **personalizar una tabla** de asociación, por ejemplo:



- Esta anotación tiene una **serie de atributos útiles**.

La anotación @JoinTable

- Atributo **name**: Especifica el nombre de la tabla de unión que se creará en la base de datos.

```
@JoinTable(
    name = "student_course"
)
```

- Atributo **joinColumns**: Especifica las **columnas de clave ajena** de la entidad propietaria a la tabla de combinación. Estas columnas representan la relación entre la entidad propietaria y la tabla de unión. Utilice la **anotación @JoinColumn** dentro de joinColumns para definir las columnas de clave externa.

```
@JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name = "student_id")
)
```

La anotación @JoinTable

- Atributo **inverseJoinColumns**:
  - Especifica las columnas de clave ajena de la entidad inversa a la tabla de combinación.
  - Estas columnas representan la relación de la **entidad inversa** con la tabla de combinación. Utilice la anotación @JoinColumn dentro de inverseJoinColumns para definir las columnas de clave externa.

```
@JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
)
```

La anotación @JoinTable

- En el ejemplo anterior, si introducimos la siguiente modificación:

```
@JoinTable(name = "INTERMEDIO",
                        joinColumns = @JoinColumn(name = "factura_id"),
                        inverseJoinColumns = @JoinColumn(name = "linea_id"))
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
List<InvoiceLine> lines = new ArrayList<>();
```

- La tabla generada será:

```
create table INVOICE (id bigint generated by default as identity, INVOICE_AMOUNT double, INVOICE_CIF varchar(255), DATE date, primary key (id))
create table INTERMEDIO (factura_id bigint not null, linea_id bigint not null)
create table INVOICE_LINE (id bigint generated by default as identity, PRICE double not null, PRODUCT varchar(255) not null, primary key (id))
```

La anotación @JoinColumn

- ¿Podemos implementar esta **asociación unidireccional** de otra forma? ¿Por ejemplo, con una clave ajena? La respuesta es sí.

- Se puede conseguir trabajando con la **anotación @JoinColumn** que nos permite definir las características de esa clave ajena.

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "invoice_id")
List<InvoiceLine> lines = new ArrayList<>();
```

- No se crea una tabla de unión, se define una **clave ajena** en la tabla **INVOICE_LINE**:

```
create table INVOICE (id bigint generated by default as identity, INVOICE_AMOUNT double, INVOICE_CIF varchar(255), DATE date, primary key (id))
create table INVOICE_LINE (id bigint generated by default as identity, PRICE double not null, PRODUCT varchar(255) not null, invoice_id bigint, primary key (id))
```

La anotación @JoinColumn

- Sus atributos se describen a continuación:
  - **name**: Indica el nombre con el que se deberá de crear la columna dentro de la tabla.
  - **referencedColumnName**: Se utiliza para indicar sobre **qué columna se realizará el Join** de la otra tabla. Por lo general no se suele utilizar, pues JPA asume que la **columna es el ID de la Entidad** objetivo.
  - **unique**: Crea un constraints en la tabla para impedir valores duplicados (default false).
  - **nullable**: Crea un constraints en la tabla para impedir valores nulos (default true).
  - **updatable**: Le indica a JPA si el valor deberá actualizarse durante el proceso de actualización (default true).

La anotación @JoinColumn

- (cont.):
  - **columnDefinition**: Esta propiedad es utiliza para indicar la instrucción SQL que se deberá utilizar al crear la columna en la base de datos. Esta nos ayuda a definir exactamente como se creará la columna sin depender de la configuración de JPA.
  - **table**: Le indicamos sobre qué tabla deberá realizar el JOIN, normalmente **no es utilizada**, pues JPA asume la **tabla por medio de la entidad objetivo**.
  - **foreignKey**: Se utiliza para especificar o controlar la generación de una restricción de clave externa cuando la generación de tablas está vigente. Si no se especifica este elemento, se aplicará la estrategia de clave externa por defecto del proveedor de persistencia:

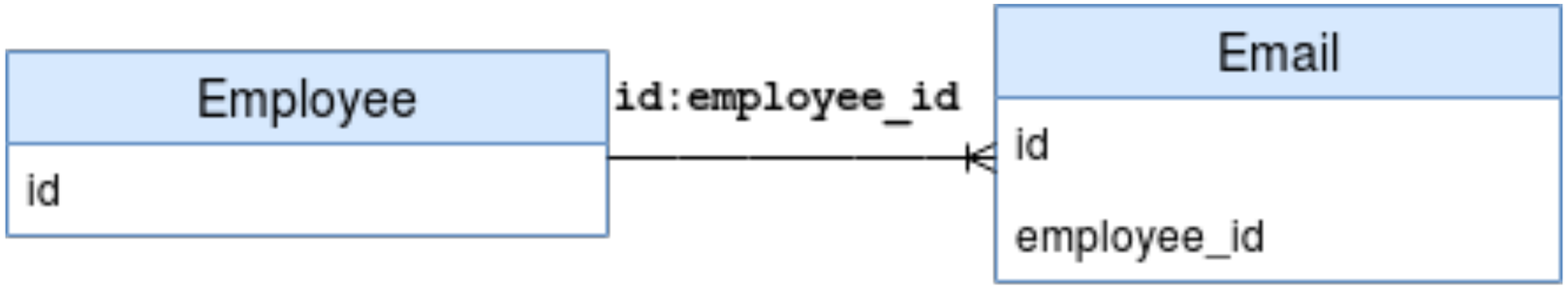
```
@ForeignKey(value=ConstraintMode.PROVIDER_DEFAULT)
```

Relaciones bidireccionales one to many/many to one

- Introducción.
- La anotación @ManyToOne.
- La anotación @JoinColumn.
- La anotación @OneToMany.
- El atributo mappedBy.

Introducción

- Supongamos que tenemos dos entidades: Employee e Email.
- Claramente, un **empleado puede tener varias direcciones de correo electrónico**. Sin embargo, una **dirección de correo** electrónico dada debe pertenecer **exactamente a un solo empleado**.
- Esto significa que comparten una asociación de uno a muchos:



- También en nuestro **modelo de base de datos**, tendremos una **clave ajena employee_id** en nuestra entidad Email que se refiere al **atributo id de un empleado**.

La anotación @ManyToOne

- En una **relación Uno-a-Muchos/Muchos-a-Uno**, el **lado propietario** generalmente se define en el **lado muchos** de la relación y suele ser el lado que posee **la clave ajena**.

```
@Entity
public class Email {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    private Employee employee;

    // ...
}
```

La anotación @ManyToOne

- La anotación incluye los siguientes atributos:

Optional Elements

Modifier and Type	Optional Element	Description
`CascadeType[ ]`	`cascade`	(Optional) The operations that must be cascaded to the target of the association.
`FetchType`	`fetch`	(Optional) Whether the association should be lazily loaded or must be eagerly fetched.
`boolean`	`optional`	(Optional) Whether the association is optional.
`java.lang.Class`	`targetEntity`	(Optional) The entity class that is the target of the association.

La anotación @JoinColumn

- La **anotación @JoinColumn** define una columna que **unirá dos entidades**.
- Define la **columna de clave ajena** de una entidad y su **campo de clave principal asociado**.
- Esta anotación nos permite crear relaciones entre entidades, guardando y accediendo a datos rápidamente.
- Generalmente se usa con las **anotaciones @ManyToOne o @OneToOne** para definir una asociación.

La anotación @JoinColumn

- Como podemos ver, define ese mapeo físico real en el lado **propietario**:

```
@Entity
public class Email {
    ...
   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = “employee_id”, referencedColumnName = "id")
   private Employee employee;
}
```

- Simplemente significa que **nuestra entidad Email** tendrá una **columna clave ajena** llamada **employee_id** que se refiere al identificador único de **nuestra entidad Employee**.

La anotación @OneToMany

- Una vez que hemos definido el lado propietario de la relación, JPA ya tiene toda la información que necesita para mapear esa relación en nuestra base de datos.
- Para que **esta asociación sea bidireccional**, todo lo que tendremos que hacer es definir el **lado de referencia**.
- El **lado inverso o de referencia** simplemente se mapea en el lado propietario con la anotación **@OneToMany**:

```
@Entity
public class Employee {
   ...
   @OneToMany(...)
   private List<Email> emails;
}
```

El atributo mappedBy

- Debemos usar el **atributo mappedBy** de la anotación **@OneToMany** para hacerlo.

```
@Entity
public class Employee {
   ...

   @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
   private List<Email> emails;
}
```

- Este atributo define el **nombre de la propiedad en el otro extremo**, en este caso **employee**.

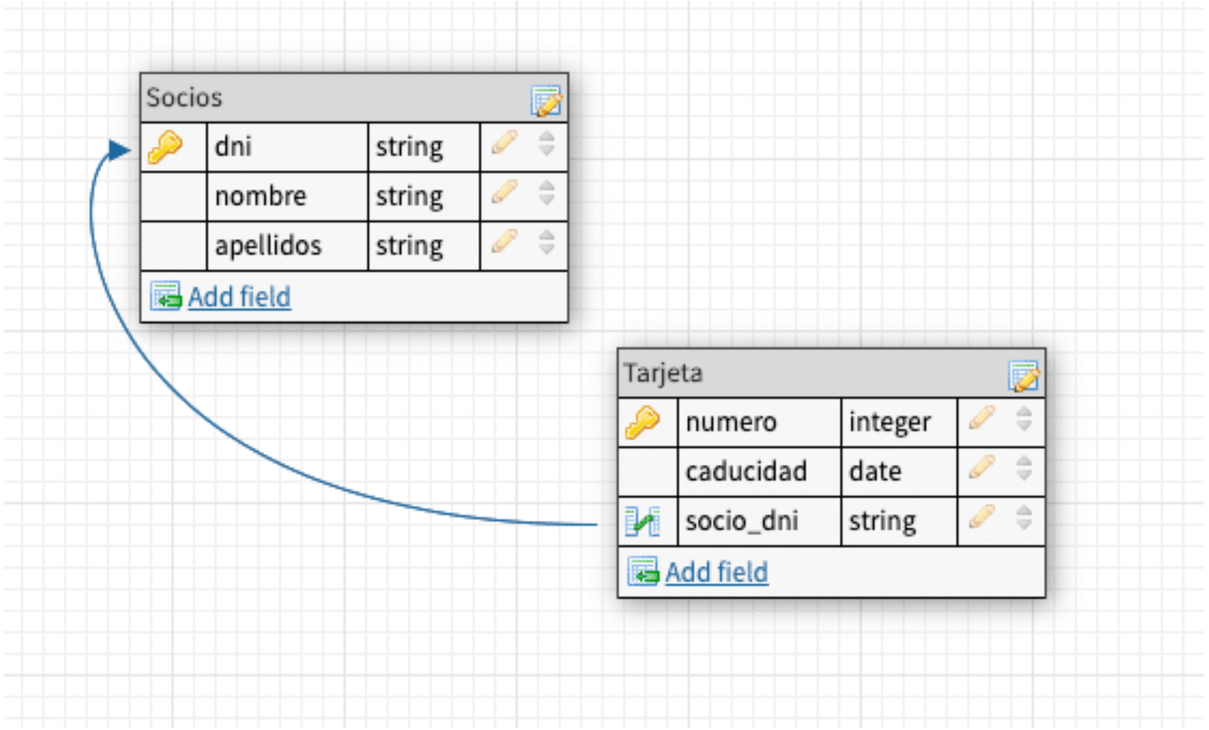
One to one

- Introducción.
- La anotación @OneToOne.
- Usando una clave ajena.
- Usando una clave principal compartida.
- Usando una Join Table.

Introducción

- En muchas ocasiones nos encontramos trabajando con JPA y construyendo relaciones @OneToMany @ManyToOne ya que son las más habituales pero **nos olvidamos de las relaciones @OneToOne** que aunque no son las más habituales existen y tienen casuísticas bastante comunes.

- Imaginémonos un **gimnasio** en el cual cada **socio** tiene una **tarjeta** que le permite el acceso. Estamos ante una **relación 1 a 1** ya que **cada socio tiene asignada su tarjeta** y cada **tarjeta pertenece a un socio**.



Introducción

- La **anotación @OneToOne** será responsable de definir este tipo de relación.
- Ejemplo:

```
@Entity
@Table(name = "address")
public class Address {


   ...
   @OneToOne(mappedBy = "address")
   private User user;
}
```

- Posteriormente profundizaremos en sus atributos.

Introducción

- Vamos a analizar diferentes formas de crear **asignaciones uno a uno** en JPA:
  - Usando una clave ajena.
  - Usando una clave principal compartida.
  - Usando una Join Table.

La anotación @OneToOne

- La anotación incluye los siguientes atributos:

Optional Elements

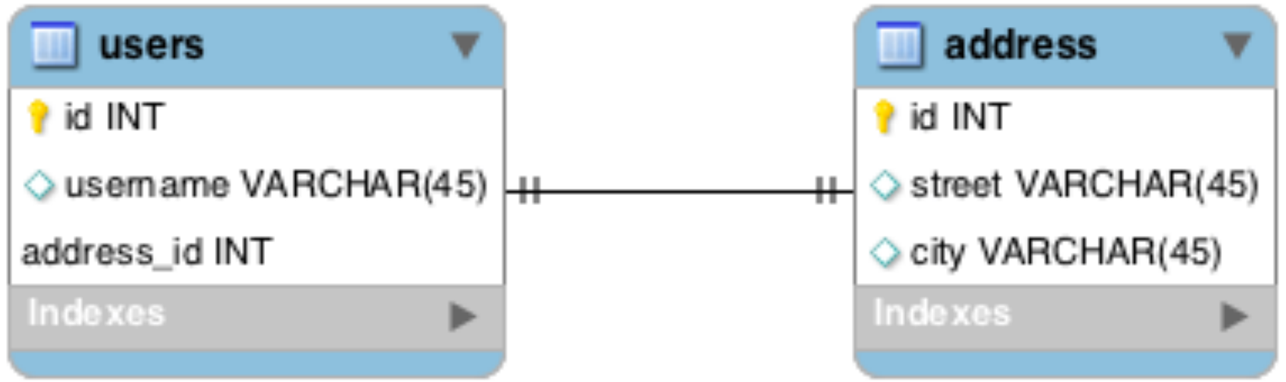
Modifier and Type	Optional Element	Description
`CascadeType[]`	`cascade`	(Optional) The operations that must be cascaded to the target of the association.
`FetchType`	`fetch`	(Optional) Whether the association should be lazily loaded or must be eagerly fetched.
`java.lang.String`	`mappedBy`	(Optional) The field that owns the relationship.
`boolean`	`optional`	(Optional) Whether the association is optional.
`boolean`	`orphanRemoval`	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.
`java.lang.Class`	`targetEntity`	(Optional) The entity class that is the target of the association.

Usando una clave ajena

- Supongamos que estamos construyendo un **sistema de administración de usuarios** y nuestro jefe nos pide que almacenemos una **dirección postal para cada usuario**. Un usuario tendrá una dirección de correo y una dirección de correo tendrá solo un usuario vinculado.

- Este es un ejemplo de **una relación uno a uno**, en este caso entre el usuario y la dirección.

- Echemos un vistazo al siguiente diagrama que representa un mapeo uno a uno basado en una clave ajena:



Usando una clave ajena

- La entidad User:

```
@Entity
@Table(name = "users")
public class User {

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

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "address_id", referencedColumnName = "id")
   private Address address;
   ...
}
```

Usando una clave ajena

- La entidad User:
  - Tenga en cuenta que colocamos la **anotación @OneToOne** en el campo de la entidad relacionada, **Address**.
  - Además, debemos colocar la **anotación @JoinColumn** para configurar el **nombre de la columna** en la **tabla users** que se asigna a la clave principal en **la tabla address**. Si no proporcionamos un nombre, JPA seguirá algunas reglas para seleccionar uno por defecto.

Usando una clave ajena

- La entidad Address:
  - Finalmente, tenga en cuenta en la siguiente entidad que **no usaremos la anotación @JoinColumn** allí.
  - Esto se debe a que **solo lo necesitamos en el lado propietario** de la relación de clave ajena.
  - En pocas palabras, **el propietario de la columna de clave ajena obtiene la anotación @JoinColumn**.

Usando una clave ajena

- La entidad Address:

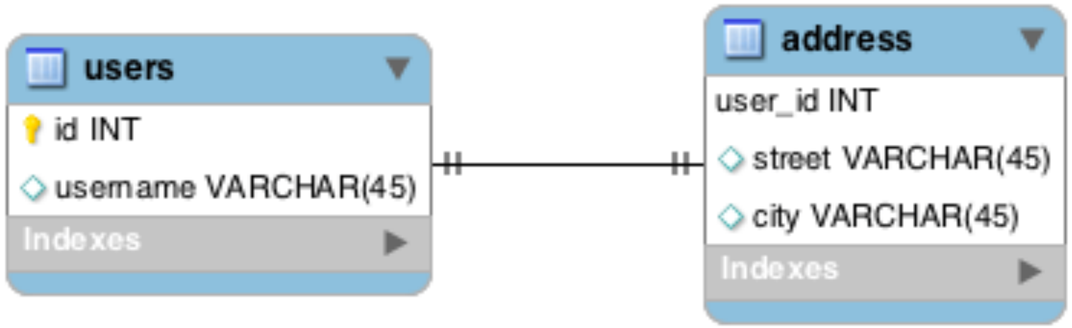
```
@Entity
@Table(name = "address")
public class Address {

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

   @OneToOne(mappedBy = "address")
   private User user;

   //... getters and setters
}
```

Usando una clave ajena

- La entidad Address:
  - También necesitamos colocar la **anotación @OneToOne** aquí también.
  - Eso es porque esta es **una relación bidireccional**.
  - El lado de la dirección de la relación se llama **el lado no propietario**.

Usando una clave principal compartida

- En esta estrategia, en lugar de crear una **nueva columna address_id**, marcaremos la columna de la clave principal (**user_id**) de la tabla de **direcciones** como la clave ajena de la tabla de usuarios:



- Hemos optimizado el espacio de almacenamiento utilizando el hecho de que estas entidades tienen una relación uno a uno entre ellas.

Usando una clave principal compartida

- La entidad User:

```
@Entity
@Table(name = "users")
public class User {

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

    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Address address;

    //... getters and setters
}
```

Usando una clave principal compartida

- Nota:
  - La **anotación @PrimaryKeyJoinColumn** especifica que la **clave principal de una entidad** es una clave ajena para otra entidad.
  - Asigna una **relación entre dos entidades** al convertir sus columnas de **clave principal** en **claves ajenas** en la tabla de entidades.

Usando una clave principal compartida

- La entidad Address:

```
@Entity
@Table(name = "address")
public class Address {

    @Id
    @Column(name = "user_id")
    private Long id;

    @OneToOne
    @MapsId
    @JoinColumn(name = "user_id")
    private User user;

    //... getters and setters
}
```

Usando una clave principal compartida

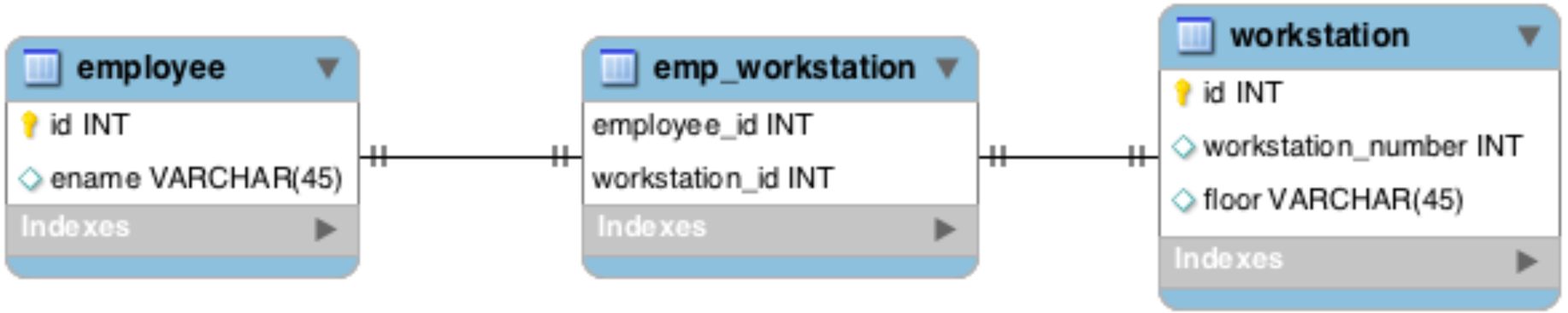
- El **atributo mappedBy** ahora se mueve a la **clase User** ya que la clave ajena ahora está presente en la **tabla address**.
- También agregamos la **anotación @PrimaryKeyJoinColumn**, que indica que la clave principal de la entidad **User** se usa como valor de clave externa para la entidad Address asociada.
- Todavía tenemos que definir un campo @Id en **la clase Address**. Pero tenga en cuenta que esto hace referencia a la columna **user_id** y ya **no usa la anotación @GeneratedValue**.
- Además, en el campo que hace referencia al Usuario, agregamos **la anotación @MapsId**, que indica que los valores de la clave principal **se copiarán de la entidad User**.

Usando una Join Table

- Las **asignaciones uno a uno** pueden ser de dos tipos: **opcionales y obligatorias**. Hasta ahora, solo hemos visto **relaciones obligatorias**.

- Ahora imaginemos que nuestros empleados se asocian a una estación de trabajo. Es uno a uno, pero a veces **un empleado puede no tener una estación de trabajo** y viceversa.

- Las estrategias que hemos comentado hasta ahora **nos obligan a poner valores nulos** en la columna para manejar relaciones opcionales. Por lo general, pensamos en relaciones de muchos a muchos cuando consideramos una Join Table, pero en este caso puede ayudarnos a eliminar estos valores nulos:



Usando una Join Table

- La entidad Employee:

```
@Entity
@Table(name = "employee")
public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = "id")
   private Long id;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinTable(name = "emp_workstation",
    joinColumns = { @JoinColumn(name = "employee_id", referencedColumnName = "id") },
    inverseJoinColumns = { @JoinColumn(name = "workstation_id", referencedColumnName = "id") })
   private WorkStation workStation;
}
```

Usando una Join Table

- La entidad WorkStation:

```
@Entity
@Table(name = "workstation")
public class WorkStation {


   ...
  @OneToOne(mappedBy = "workStation")
   private Employee employee;
}
```

- @JoinTable le indica a JPA que emplee la **estrategia de unir tablas** mientras mantiene la relación.
- Además, **Employee** es el propietario de esta relación, ya que elegimos usar la anotación de la tabla de unión en ella.

Many to many

- Introducción.
- Usando join table.

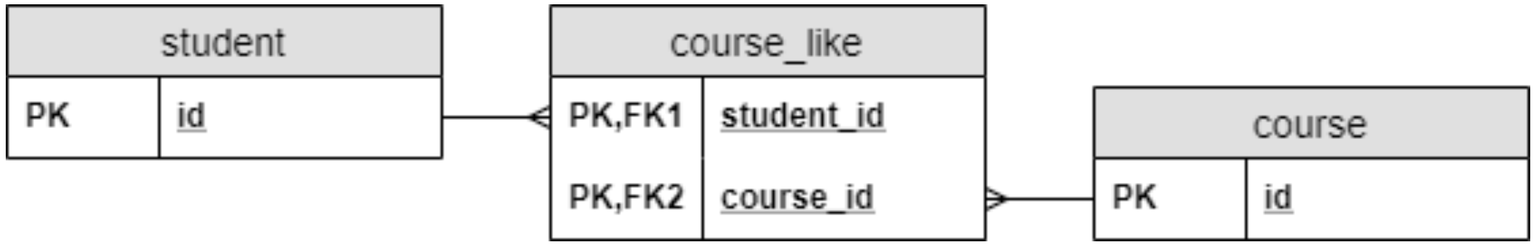
Introducción

- Una **relación** es una **conexión entre dos tipos de entidades**. En el caso de una relación de **muchos a muchos**, ambos lados pueden relacionarse con **múltiples instancias** del otro lado.
- Tenga en cuenta que es posible que los tipos de entidad estén en una relación consigo mismos.
- Piense en el ejemplo de modelar árboles genealógicos: cada nodo es una persona, por lo que si hablamos de la relación padre-hijo, ambos participantes serán una persona.
- Sin embargo, no hay tanta diferencia si hablamos de una relación entre tipos de entidad únicos o múltiples. Dado que es **más fácil pensar en las relaciones entre dos tipos de entidades diferentes**, lo usaremos para ilustrar nuestros casos.

Introducción

- Tomemos el ejemplo de los **estudiantes** marcando los cursos que **les gustan**. A un estudiante le pueden gustar muchos cursos, y a muchos estudiantes les puede gustar el mismo curso:

- Como sabemos, en los RDBMS podemos crear relaciones con claves ajena. Dado que ambos lados deberían poder hacer referencia al otro, necesitamos **crear una tabla separada** para contener las claves externas:



Introducción

- A esta tabla se la llama **join table**.
- En una tabla de unión, la **combinación de las claves foráneas** será su **clave principal compuesta**.

Usando join table

- La entidad Student:

```
@Entity
class Student {

   @Id
   Long id;

   @ManyToMany
   Set<Course> likedCourses;

   // additional properties
   // standard constructors, getters, and setters
}
```

Usando join table

- La entidad Course:

```
@Entity
class Course {

   @Id
   Long id;

   @ManyToMany
   Set<Student> likes;

   // additional properties
   // standard constructors, getters, and setters
}
```

Usando join table

- Además, tenemos que configurar cómo modelar la relación en el RDBMS. El **lado propietario** es donde configuramos la relación. Usaremos la **clase Student**.
- Podemos hacer esto con la **anotación @JoinTable** en la clase Student. Proporcionamos el **nombre de la tabla de unión** (course_like), así como las **claves foráneas con las anotaciones @JoinColumn**.
- El **atributo joinColumn** se conectará al lado propietario de la relación y el **inverseJoinColumn** al otro lado:

```
@ManyToMany
@JoinTable(
  name = "course_like",
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id"))
Set<Course> likedCourses;
```

Usando join table

- Importante:
  - Tenga en cuenta que **no es necesario** usar **@JoinTable** o incluso **@JoinColumn**. JPA generará los **nombres de las tablas y columnas** automáticamente.
  - Sin embargo, la **estrategia que usa JPA** no siempre coincidirá con las **convenciones** de nomenclatura que usamos.
  - Por lo tanto, necesitamos la posibilidad de configurar nombres de tablas y columnas de forma manual.

Usando join table

- En el otro lado de la relación (target side) solo tenemos que proporcionar el **nombre del campo**, que mapea la relación, por esto establecemos el **atributo mappedBy de la anotación @ManyToMany** en la clase Course:

```
@ManyToMany(mappedBy = "likedCourses")
Set<Student> likes;
```

- Tenga en cuenta que dado que **una relación de muchos a muchos** no tiene **un lado propietario en la base de datos**, podríamos configurar la **tabla de combinación** en la clase Course y hacer referencia a ella desde la clase Student.

Recuperación o búsqueda de datos

- Introducción.
- Trabajando con tipos fetch.
- Trabajando con lazy.

Introducción

- Existen distintos **tipos de captura o búsqueda (Fetch Types)** en función de cómo se efectúa la búsqueda de los datos en la relación:
  - **Lazy**: el dato no se solicita hasta que se referencia. Es decir, si se tiene una lista, JPA va a esperar a que se haga una consulta sobre la lista para obtener los datos de la base de datos.
  - **Eager**: los datos se consultan por adelantado.
- El **enumerado FetchType** nos permite manejar este concepto **programáticamente**.

Trabajando con tipos fetch

- Introducción.
- El enumerado FetchType.
- Comportamiento por defecto.
- FetchType.EAGER.
- FetchType.LAZY.

Introducción

- **FetchType** define cuándo se obtienen las **entidades relacionadas de la base de datos** y es uno de los **elementos cruciales** para un nivel de persistencia rápido.
- En general, desea **recuperar** las entidades que utiliza de la forma **más eficiente posible**. Pero eso no es tan fácil. ¿Obtiene todas las relaciones con una consulta u obtiene solo la entidad raíz e inicializa las relaciones tan pronto como las necesite?
- Vamos a explicar **ambos enfoques** con más detalle a continuación.

El enumerado FetchType

Package javax.persistence

Enum FetchType

java.lang.Object
    java.lang.Enum<FetchType>
        javax.persistence.FetchType

All Implemented Interfaces:

`java.io.Serializable, java.lang.Comparable<FetchType>`

```
public enum FetchType
extends java.lang.Enum<FetchType>
```

Enum Constant Summary

Enum Constants

Enum Constant	Description
EAGER	Defines that data must be eagerly fetched.
LAZY	Defines that data can be lazily fetched.

Comportamiento por defecto

- Cuando comenzó con JPA, lo más probable es que no supiera acerca de FetchTypes o que le dijeron que siempre usara FetchType.LAZY. En general, es una buena recomendación. Pero, ¿qué significa exactamente? ¿Y cuál es el valor por defecto si no define FetchType?

- El valor por defecto depende de la cardinalidad de la relación. Todas las relaciones **a uno** usan **FetchType.EAGER** y todas las relaciones **a muchos FetchType.LAZY**, es decir:

  - Para relación Uno a Muchos: **Lazy**.

  - Para relación Muchos a Uno: **Eager**.

  - Para relación Muchos a Muchos: **Lazy**.

  - Para relación Uno a Uno: **Eager**.

Comportamiento por defecto

- Este comportamiento por defecto puede cambiarse gracias al **atributo fetch** que incluyen las anotaciones que define relaciones.
- Ejemplo:

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

 @OneToMany(mappedBy = "order", fetch = FetchType.EAGER)
 private Set<OrderItem> items = new HashSet<OrderItem>();

 ...

}
```

FetchType.EAGER

- **FetchType.EAGER** le dice a JPA que obtenga **todos los elementos de una relación al seleccionar la entidad raíz**.

- Como se explicó anteriormente, este es el valor por defecto para las relaciones uno a uno y puede verlo en los siguientes fragmentos de código. Vamos a utilizar el FetchType por defecto (EAGER) para la relación de **muchos a uno** entre la entidad **OrderItem** y **Product**.

```
@Entity
public class OrderItem implements Serializable
{

  @ManyToOne
  private Product product;

  ...
}
```

FetchType.EAGER

- Cuando obtengo una entidad de tipo **OrderItem** de la base de datos, JPA también obtendrá la **entidad Product** relacionada.

```
OrderItem orderItem = em.find(OrderItem.class, 1L);
log.info("Fetched OrderItem: "+orderItem);
Assert.assertNotNull(orderItem.getProduct());
```

```
05:01:24,504 DEBUG SQL:92 - select orderitem0_.id as id1_0_0_, orderitem0_.order_id as order_id4_0_0_,
orderitem0_.product_id as product_5_0_0_, orderitem0_.quantity as quantity2_0_0_, orderitem0_.version as version3_0_0_,
order1_.id as id1_2_1_, order1_.orderNumber as orderNum2_2_1_, order1_.version as version3_2_1_, product2_.id as
id1_1_2_, product2_.name as name2_1_2_, product2_.price as price3_1_2_, product2_.version as version4_1_2_ from
OrderItem orderitem0_ left outer join purchaseOrder order1_ on orderitem0_.order_id=order1_.id left outer join Product
product2_ on orderitem0_.product_id=product2_.id where orderitem0_.id=?
05:01:24,557  INFO FetchTypes:77 - Fetched OrderItem: OrderItem , quantity: 100
```

FetchType.EAGER

- Esto parece ser **muy útil al principio**. Hacer un join de las entidades requeridas y obtenerlas todas en una sola consulta es muy eficiente.
- Pero tenga en cuenta que JPA **SIEMPRE** buscará la **entidad Producto** para su pedido, incluso si **no la usa en su código**.
- Si la entidad relacionada **no es demasiado grande**, esto **no es un problema** para las relaciones de uno a uno. Pero lo más probable es que ralentice su aplicación si la usa para **una relación de muchos** que no necesita para su caso de uso.
- JPA tiene que buscar **decenas o incluso cientos de entidades adicionales**, lo que genera una **sobrecarga significativa**.

FetchType.LAZY

- **FetchType.LAZY** le dice a JPA que **solo obtenga las entidades relacionadas** de la base de datos **cuando usa la relación**.
- Esta es una **buena idea,** en general, porque no hay razón para seleccionar entidades que no necesita para su caso de uso.
- Por ejemplo, la relación de **uno a muchos** entre las entidades **Order y OrderItem** utiliza el FetchType por defecto para las relaciones de varios, lo cual es perezoso.

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

  @OneToMany(mappedBy = "order")
  private Set<OrderItem> items = new HashSet<OrderItem>();
  ...
}
```

FetchType.LAZY

- FetchType utilizado no influye en el código. Puede llamar al **método getOrderItems()** como cualquier otro método getter.

```
Order newOrder = em.find(Order.class, 1L);
log.info("Fetched Order: "+newOrder);
Assert.assertEquals(2, newOrder.getItems().size());
```

- JPA maneja la inicialización perezosa de forma transparente y **obtiene las entidades OrderItem** tan pronto como se llama al método getter.

FetchType.LAZY

- El log generado es:

```
05:03:01,504 DEBUG SQL:92 - select order0_.id as id1_2_0_, order0_.orderNumber as orderNum2_2_0_, order0_.version as version3_2_0_ from purchaseOrder order0_ where order0_.id=?
05:03:01,545  INFO FetchTypes:45 - Fetched Order: Order orderNumber: order1
05:03:01,549 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?
```

- Manejar las relaciones perezosas de esta manera es correcto si trabaja **con una sola entidad Order** o con una **pequeña lista de entidades**. Pero se convierte en un **problema de rendimiento** cuando lo hace en una gran lista de entidades.

FetchType.LAZY

- Como puede ver en los siguientes logs, JPA tiene que ejecutar una consulta SQL adicional para que cada **entidad Order obtenga sus OrderItems**.

```
05:03:40,936 DEBUG ConcurrentStatisticsImpl:411 - HHH000117: HQL: SELECT o FROM Order o, time: 41ms, rows: 3
05:03:40,939  INFO FetchTypes:60 - Fetched all Orders
05:03:40,942 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, ...
05:03:40,957 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, ...
05:03:40,959 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, ...
```

- Este comportamiento se denomina **problema n+1 select** y es el **problema de rendimiento más común**.

FetchType.LAZY

- Este código realiza **una consulta adicional para inicializar la relación**. Eso no parece un problema, pero calculemos la cantidad de consultas ejecutadas en un escenario más real. Digamos que tiene una entidad con **5 asociaciones que necesita inicializar**. Entonces obtendrá 1 + 5 = **6 consultas**.

- OK, esas son **5 consultas adicionales**. Eso todavía no parece un gran problema. Pero nuestra aplicación será utilizada por más de un usuario en paralelo.

- Digamos que su sistema tiene que servir a **100 usuarios paralelos**. Entonces obtendrá 100 + 5*100 = **600 consultas**. Eso se llama el **problema de select n+1**, y debería ser obvio que este no es un buen enfoque. Tarde o temprano, **la cantidad de consultas adicionales realizadas ralentizará su aplicación**.

- Por lo tanto, debe intentar evitar este enfoque y **echar un vistazo a otras opciones**.

FetchType.LAZY

- Hay dos formas de evitar estos problemas:
  - Puede usar **FetchType.EAGER** si sabe que **todos sus casos de uso que obtienen una entidad** de pedido también necesitan procesar las entidades de artículo de pedido relacionadas. **Eso casi nunca será el caso**.
  - Si hay algunos casos de uso que solo funcionan en entidades de pedido (que es lo más probable), debe **usar FetchType.LAZY en el mapeo de su entidad** y usar soluciones que inicialicen la relación cuando las necesite. **Estas estrategia será presentadas posteriormente**.

Operaciones en cascada

- Introducción.
- Tipos definidos por JPA.
- Tipos definidos por Hibernate.
- El atributo cascade.

Introducción

- Las **relaciones entre entidades** a menudo dependen de la existencia de una de ellas, por ejemplo, la relación **Persona-Dirección**. Sin la Persona, la entidad Dirección no tiene ningún significado propio. Cuando eliminamos la entidad Persona, nuestra entidad Dirección también debería eliminarse.

- Las operaciones en cascada son la manera de manejar este concepto. Cuando realizamos alguna acción sobre una entidad, la misma acción se aplicará a la entidad asociada.

- El **tipo de control en cascada** hace referencia a cómo los cambios de estado se propagan de los **objetos padres** a los **objetos hijos**.

- Existen tipo definidos por JPA y otros específicos definido por Hibernate. Vamos a describirlos.

Tipos definidos por JPA

- Todas las operaciones en cascada específicas de JPA se definen gracias al enumerado **javax.persistence.CascadeType** que contiene las siguientes entradas:
  - ALL.
  - PERSIST.
  - MERGE.
  - REMOVE.
  - REFRESH.
  - DETACH.

Tipos definidos por JPA

- **CascadeType.ALL:**
  - Propaga todas las operaciones, incluidas las específicas de Hibernate, de una entidad principal a una secundaria.
- **CascadeType.PERSIST:**
  - Las operaciones de guardado en las entidades padre se propagarán a las entidades relacionadas.
  - La operación persist hace que una instancia transitoria sea persistente. CascadeType.PERSIST propaga la operación persistente de una entidad principal a una secundaria. Cuando guardamos la entidad de persona, la entidad de dirección también se guardará.
- **CascadeType.MERGE**:
  - La operación merge copia el estado del objeto dado en el objeto persistente con el mismo identificador. CascadeType.MERGE propaga la operación merge de una entidad principal a una entidad secundaria.

Tipos definidos por JPA

- **CascadeType.REMOVE**:
  - Como sugiere el nombre, la operación remove elimina la fila correspondiente a la entidad de la base de datos y también del contexto persistente.
  - CascadeType.REMOVE propaga la operación de eliminación de la entidad principal a la secundaria. Similar a **CascadeType.REMOVE** de JPA, tenemos **CascadeType.DELETE**, que es específico de Hibernate. No hay diferencia entre los dos.
- **CascadeType.DETACH**:
  - La operación detach elimina la entidad del contexto persistente. Cuando usamos CascadeType.DETACH, la entidad secundaria también se eliminará del contexto persistente.

Tipos definidos por JPA

- **CascadeType.REFRESH**:
  - Las operaciones refresh vuelven a leer el valor de una instancia determinada de la base de datos. En algunos casos, podemos cambiar una instancia después de persistir en la base de datos, pero luego debemos deshacer esos cambios.
  - En ese tipo de escenario, esto puede ser útil. Cuando usamos esta operación con CascadeType.REFRESH, la entidad secundaria también se vuelve a cargar desde la base de datos cada vez que se actualiza la entidad principal.
- Por defecto, no se aplica ninguna operación de cascada.

Tipos definidos por Hibernate

- Hibernate admite **tres tipos adicionales** junto con los especificados por JPA. Estos tipos específicos de Hibernate están disponibles gracias al enumerado **org.hibernate.annotations.CascadeType**:
  - REPLICATE.
  - SAVE_UPDATE.
  - LOCK.

Tipos definidos por Hibernate

- **CascadeType.REPLICATE**:
  - La operación de replicación **se usa cuando tenemos más de una fuente de datos** y queremos que los datos estén sincronizados. Con CascadeType.REPLICATE, una operación de sincronización también se propaga a las entidades secundarias cuando se realiza en la entidad principal.
- **CascadeType.SAVE_UPDATE**:
  - CascadeType.SAVE_UPDATE propaga la misma operación a la entidad secundaria asociada. Es útil cuando usamos operaciones específicas de Hibernate como **save, update y saveOrUpdate**.
- **CascadeType.LOCK**:
  - De manera poco intuitiva, CascadeType.LOCK vuelve a adjuntar la entidad y su entidad secundaria asociada con el contexto persistente nuevamente.

El atributo cascade

- El **atributo cascade** se usa en la anotación que define la relación para indicar el tipo de cascada a aplicar, por ejemplo:

```
@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int id;
   private String name;
   @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
   private List<Address> addresses;
}
```