Storage: Implementación

IT BOARDING

BOOTCAMP

Índice

IT BOARDING

BOOTCAMP

STORAGE IMPLEMENTACIÓN

// Modelado de Relaciones

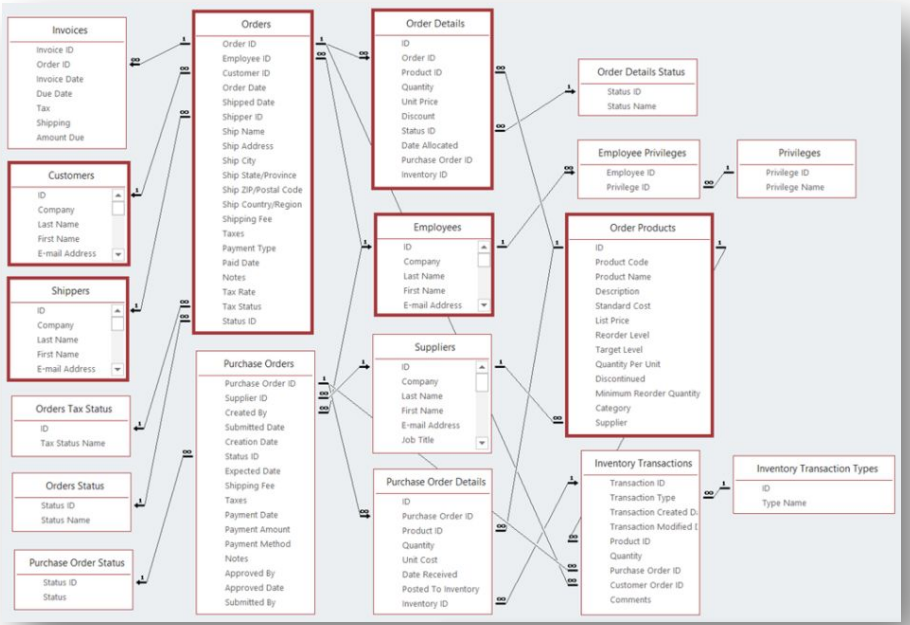
IT BOARDING

BOOTCAMP

// Relaciones

Una relación es una **conexión** entre dos tipos de entidades.
Veremos **cuatro tipos de relaciones** y cómo modelarlas en Hibernate.

- **OneToOne**
- **OneToMany**
- **ManyToOne**
- **ManyToMany**



IT BOARDING

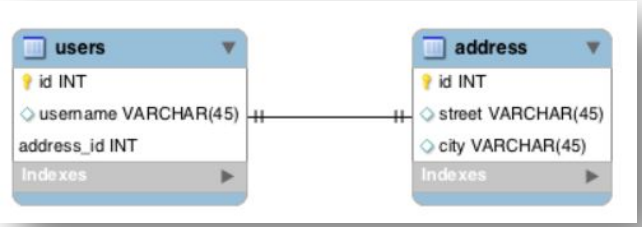
BOOTCAMP

Uno a Uno

@OneToOne: En el caso de que una clase tenga un atributo del tipo de otra clase, y solo uno, podemos realizar el mapeo utilizando esta anotación.

Podemos utilizarla junto con la anotación **@JoinColumn** para configurar el nombre de la columna en la tabla padre (users) que mapea a la clave primaria de la segunda tabla (address).

La anotación **@JoinColumn**, solo se utiliza del lado del propietario (users) por lo que debe escribirse en la clase donde es inyectada (address), mientras que **@OneToOne** debe escribirse en las dos.

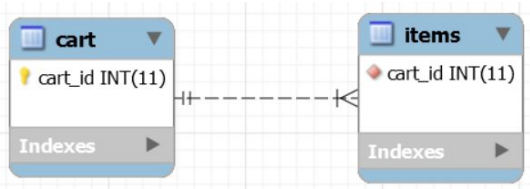



```java
@Entity
@Table(name = "users")
public class User {

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

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

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

```java
@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
}
```

Uno a muchos

@OneToMany: Es necesaria cuando una fila en la tabla debe mapearse a múltiples filas en otra tabla.



En este caso tenemos una tabla Cart que tiene muchos items. La propiedad «mappedBy» indica a Hibernate qué atributo de la clase padre utilizaremos para mapear un atributo de la clase hijo (en este caso Cart utilizará el atributo cart de la clase Items para generar el mapeo).

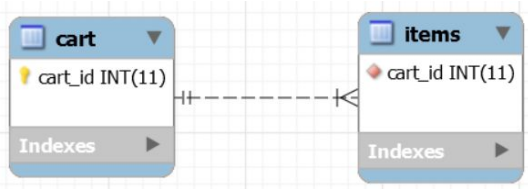
También podría añadirse una referencia a Cart dentro de Items, usando **@ManyToOne**, logrando así una relación bidireccional, es decir que podríamos acceder a Items desde Carts, y también a Carts desde Items.

```java
@Entity
@Table(name="CART")
public class Cart {

    //...

    @OneToMany(mappedBy="cart")
    private Set<Items> items;

    // getters and setters
}
```

```java
@Entity
@Table(name="ITEMS")
public class Items {

    //...
    @ManyToOne
    @JoinColumn(name="cart_id", nullable=false)
    private Cart cart;

    public Items() {}

    // getters and setters
}
```

Muchos a Uno

@ManyToOne: Es necesaria cuando muchas instancias de una entidad son mapeadas a una instancia de otra entidad. Por ejemplo muchos Items en una Cart.

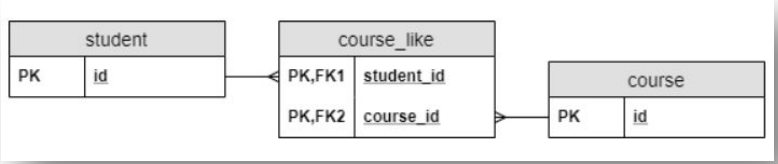



```java
@Entity
@Table(name="ITEMS")
public class Items {

    //...
    @ManyToOne
    @JoinColumn(name="cart_id", nullable=false)
    private Cart cart;

    public Items() {}

    // getters and setters
}
```

Esto lo logramos usando el «mappedBy» para el atributo items en la clase Cart, y agregando la anotación **@ManyToOne** sobre el atributo cart, convirtiéndose así Items en la tabla padre.

Muchos a Muchos

@ManyToMany: Se trata del caso en que muchas entidades se relacionan con muchas entidades de otro tipo (aunque en ocasiones puede ser del mismo).



Por ejemplo un estudiante puede tener muchos cursos, y un curso puede tener muchos estudiantes.

Para modelar esta relación es necesario crear una tabla separada que contenga las claves secundarias de ambas tablas cuya clave primaria compuesta va a ser la combinación de ambas claves primarias.

Para mapear esta relación anotamos los atributos de ambas clases (que son Sets) con **@ManyToMany** y configuramos la clase propietaria indicando la unión con **@JoinTable.**

```java
@Entity
class Student {

    @Id
    Long id;

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

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

```java
@Entity
class Course {

    @Id
    Long id;

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

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

joinColumn indica el atributo de unión de la clase Student, y **inverseJoinColumns** el de la clase Course.

Finalmente, en la clase Course, agregamos **«mappedBy»** para indicar qué campo de la clase Student va a mapear ese atributo.

// @JoinColumn

IT BOARDING

BOOTCAMP

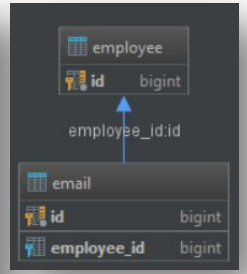
@JoinColumn

En una relación **@OneToOne** puede utilizarse para indicar que una columna en la entidad padre se referencia a una clave primaria en la entidad hijo. Por ejemplo, Office se relaciona con Address a través de una clave foránea, que es addressId.

```
@Entity
public class Office {
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "addressId")
    private Address address;
}
```

En una relación **@OneToMany** podemos usar el atributo “mappedBy” para indicar que la columna referencia a otra entidad. Por ejemplo, Email que es la entidad padre, se une a Employee a través de la id.

```
@Entity
public class Employee {

    @Id
    private Long id;

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

@Entity
public class Email {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

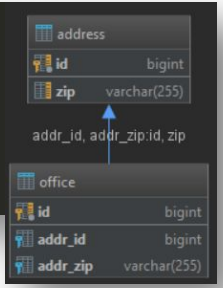


@JoinColumns

Puede utilizarse en los casos en que nuestra intención sea crear joins de múltiples columnas.

En la entidad Office crearemos dos claves foráneas que apunten a las columnas ID y ZIP en la entidad Address.

```
@Entity
public class Office {
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumns({
        @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
        @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
    })
    private Address address;
}
```



// Cascading

IT BOARDING

BOOTCAMP

Cascading

La mayoría de las veces las relaciones entre entidades dependen de la existencia de otra entidad.

Sin una la otra no podría existir, y si modificamos una deberíamos modificar ambas.

Por lo tanto, si realizamos una acción sobre una entidad, la misma acción debe aplicarse a la entidad asociada.

```java
@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;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String street;
    private int houseNumber;
    private String city;
    private int zipCode;
    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

Las más utilizadas son:

CascadeType.ALL: Propaga todas las operaciones de una entidad padre al hijo.

CascadeType.PERSIST: Propaga la operación de persistencia del padre al hijo. Cuando se guarda una Person, se guarda también la Address.

CascadeType.REMOVE/DELETE: Propaga las operaciones de borrado.

// Fetch Type

IT BOARDING

BOOTCAMP

Fetch Type

Es requerido especificar un Fetch Type al usar cualquiera de las asociaciones (relaciones) ya que define cuando se recupera la información de la BD y cargarla en memoria.

Cuando Hibernate accede a una entidad que se relaciona con otra mediante asociaciones, ¿cuándo recupera la información?

Este comportamiento puede configurarse utilizando el FetchType:

- **LAZY**: La colección no se recupera de la BD hasta que se hace una operación sobre ella. Por defecto es true. Si lo seteamos en false, cuando se recupere la información de A se traerá toda la información de los objetos relacionados de B. Como ventaja es más performante y reduce requerimientos de memoria.
- **EAGER**: Trae las entidades hijas junto con el padre.

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String street;
    private int houseNumber;
    private String city;
    private int zipCode;
    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

Si realizo un list() de las Address, Hibernate recupera los objetos Address, y sólo cuando de manera programática se accede a los objetos Address trae los datos asociados de Person.

Gracias.

IT BOARDING

BOOTCAMP