Mapeamentos e Relacionamentos

Temos algumas categorias para fazer mapeamentos e relacionamentos:

- @OneToOne: *Relacionamento de um para um **(1 : 1)***
- @OneToMany: *Relacionamento de um para muitos **(1 : *)***
- @ManyToOne: *Relacionamento de muitos para um **(* : 1)***
- @ManyToMany: *Relacionamentos de muitos para muitos **(* : *)***

Mapeamentos unidirecionais vs bidirecionais

Antes de tudo, é preciso notar-se que cada um dos relacionamentos **@OneToOne**, **@OneToMany**, **@ManyToOne** e **@ManyToMany** pode ser unidirecional ou bidirecional.

No relacionamento unidirecional entre duas entidades **A** e **B**, partindo-se da entidade **A**, eu chego facilmente a uma instância da entidade **B**, mas não consigo facilmente fazer o caminho contrário. Já no relacionamento bidirecional, eu também posso a partir da entidade **B**, facilmente navegar de volta para a entidade **A**.

Mapeamento @ManyToOne unidirecional

O **@ManyToOne** significa **muitos-para-1**. Neste nosso exemplo (vou supor que o campo categoria na tabela evento deveria se chamar **categoria_id**), teríamos isso:

```
@Entity
@Table(name = "evento")
public class Evento {

    // ... Outros campos ...

    @ManyToOne
    @JoinColumn(name = "categoria_id")
    private Categoria categoria;

    // ... Outros campos e métodos ...

    public Categoria getCategoria() {
        return categoria;
    }
}
```

O lado **Many** é o da classe que envolve isso tudo, no caso **Evento**. O lado **One** é o da entidade relacionada, no caso **Categoria**. Ou seja, muitos eventos para uma categoria. A mesma regra se aplica também ao **@ManyToOne**, **@OneToOne** e **@ManyToMany** (veremos mais sobre eles abaixo).

Isso acontece porque um evento tem apenas uma categoria, mas uma categoria pode ter muitos eventos. Com esse mapeamento, podemos fazer isso:

```
Evento e = ...;
Categoria c = e.getCategoria();
```

Mapeamento @OneToMany unidirecional

O **@OneToMany** é o oposto do que o **@ManyToOne**, ou seja é o **1-para-muitos**. Por exemplo, poderíamos fazer isso:

```
@Entity
@Table(name = "categoria")
public class Categoria {

    // ... Outros campos ...

    @OneToMany
    @JoinColumn(name = "categoria_id") // Esta coluna está na tabela "evento".
    private List<Evento> eventos;

    // ... Outros campos e métodos ...

    public List<Evento> getEventos() {
        return eventos;
    }
}
```

Com esse mapeamento, podemos fazer isso:

```
Categoria c = ...;
List<Evento> eventos = c.getEventos();
```

Mapeamentos @OneToMany e @ManyToOne bidirecionais

Se você tiver os dois casos acima ao mesmo tempo, onde a partir de **Evento** eu chego em **Categoria** e a partir de **Categoria** eu chego em **Evento**, o resultado vai ser que o

mapeamento vai dar errado. Por quê? Porque o **JPA** verá dois mapeamentos distintos, um deles de **Evento** para **Categoria** e outro mapeamento diferente de **Categoria** para **Evento**. Ocorre que esses dois mapeamentos são um só!
É aí que entra o campo **mappedBy**:

```
@Entity
@Table(name = "evento")
public class Evento {

    // ... Outros campos ...

    @ManyToOne
    @JoinColumn(name = "categoria_id")
    private Categoria categoria;

    // ... Outros campos e métodos ...

    public Categoria getCategoria() {
        return categoria;
    }
}

@Entity
@Table(name = "categoria")
public class Categoria {

    // ... Outros campos ...

    @OneToMany(mappedBy = "categoria")
    private List<Evento> eventos;

    // ... Outros campos e métodos ...

    public List<Evento> getEventos() {
        return eventos;
    }
}
```

Nesta relação bidirecional, o **mappedBy** diz que o outro lado da relação que é o dono dela e que o campo que a modela é o de nome categoria. Observe que esse é o nome do campo na classe **Evento** aqui no **Java**, e não o nome do campo no banco de dados!
Em geral, é recomendável que o lado da relação que termine com o **toOne** seja o dono da relação.
É importante em relacionamentos bidirecionais, sempre ligar os dois lados da relação antes de persistir no **EntityManager**:

```
Evento e = ...;
Categoria c = ...;
e.setCategoria(c);
c.eventos.add(e);
```

Mapeamento @OneToOne

Se você usar o **@OneToOne** você modela o caso **1-para-1**. Outro exemplo seria você pode fazer com que uma **avaliação** pertença a apenas uma **pessoa**, mas nesse tipo de relacionamento, você também está informando que uma **pessoa** só pode ter uma **avaliação**.

Você faria isso assim:

```
@Entity
@Table(name = "avaliacao")
public class Avaliacao {

    // ... Outros campos ...

    @OneToOne
    @JoinColumn(name = "pessoa_id")
    private Pessoa pessoa;

    // ... Outros campos e métodos ...

    public Pessoa getPessoa() {
        return pessoa;
    }
}
```

Com isso, você pode fazer isso:

```
Avaliacao a = ...;
Pessoa avaliado = a.getPessoa();
```

Para fazer o contrário, é necessário que o relacionamento seja bidirecional:

```
@Entity
@Table(name = "pessoa")
public class Pessoa {

    // ... Outros campos ...

    @OneToOne(mappedBy = "pessoa")
    private Avaliacao avaliacao;

    // ... Outros campos e métodos ...

    public Avaliacao getAvaliacao() {
        return avaliacao;
    }
}
```

E então, tendo o relacionamento bidirecional:

```
Pessoa p = ...;
Avaliacao a = p.getAvaliacao();
```

Novamente, no caso de relacionamentos **bidirecionais**, sempre deve-se ligar os dois lados da relação antes de persistir no **EntityManager**:

```
Pessoa p = ...;
Avaliacao a = ...;
a.setPessoa(p);
p.setAvaliacao(a);
```

Mapeamento @ManyToMany

Vamos pensar em outro exemplo onde podemos fazer um relacionamento de muitos-para-muitos.

> Um tipo de pizza têm vários tipos de ingredientes.
> Um tipo de ingrediente pode fazer parte de vários tipos de pizza.

E vamos supor que tenhamos a tabela **pizza**, a tabela **ingrediente** e uma tabela intermediária **pizza_ingrediente**, onde cada linha contém a chave das outras duas tabelas.

```
@Entity
@Table(name = "pizza")
public class Pizza {

    // ... Outros campos ...

    @ManyToMany
    @JoinTable(
        name = "pizza_ingrediente",
        joinColumns = @JoinColumn(name = "pizza_id"),
        inverseJoinColumns = @JoinColumn(name = "ingrediente_id"),
    )
    private List<Ingrediente> ingredientes;

    // ... Outros campos e métodos ...

    public List<Ingrediente> getIngredientes() {
        return ingredientes;
    }
}
```

A anotação **@JoinTable** é responsável por fazer o mapeamento da tabela intermediária. O **joinColumns** representa o lado da entidade que é dona do relacionamento (**Pizza**) e o **inverseJoinColumns** o lado da entidade relacionada **(Ingrediente)**. Com isso tudo, é possível então fazer-se isso:

```
Pizza p = ...;
List<Ingrediente> ingredientes = p.getIngredientes();
```

Para fazer o relacionamento bidirecional, novamente temos o **mappedBy**:

```
@Entity
@Table(name = "ingrediente")
public class Ingrediente {

    // ... Outros campos ...

    @ManyToMany(mappedBy = "ingredientes")
    private List<Pizza> pizzas;

    // ... Outros campos e métodos ...

    public List<Pizza> getPizzas() {
        return pizzas;
    }
}
```

E então podemos fazer isso também:

```
Ingrediente i = ...;
List<Pizza> pizzas = i.getPizzas();
```

novamente, temos que lembrar de relacionar os dois lados:

```
Ingrediente mussarela = ...;
Ingrediente tomate = ...;
Ingrediente presunto = ...;
Ingrediente ovo = ...;

Pizza napolitana = ...;
Pizza portuguesa = ...;

napolitana.ingredientes.add(mussarela);
napolitana.ingredientes.add(tomate);
napolitana.ingredientes.add(presunto);

portuguesa.ingredientes.add(mussarela);
portuguesa.ingredientes.add(ovo);
portuguesa.ingredientes.add(presunto);

mussarela.pizzas.add(napolitana);
mussarela.pizzas.add(portuguesa);

presunto.pizzas.add(napolitana);
presunto.pizzas.add(portuguesa);

tomate.pizzas.add(napolitana);

ovo.pizzas.add(portuguesa);
```

Finalmente lembre-se disso:

> Se o relacionamento termina com `ToMany`, então você tem uma lista de entidades relacionadas. Se termina com `ToOne`, só há uma única entidade relacionada.