05

Relacionamentos bidirecionais

Transcrição

[00:00] Olá, pessoal! Continuando aqui então, agora que já mapeamos o relacionamento do pedido com os itens pedidos, aquele relacionamento muitos para muitos, quando tem mais colunas do que as colunas de *join* entre as tabelas, acabamos ficando com um relacionamento bidirecional.

[00:15] Porque, se pararmos para pensar, a classe ItemPedido, ela já está mapeando o relacionamento com a classe Pedido. Ela já está fazendo aquele mapeamento de "itens_pedido" para "pedidos".

[00:27] Mas, na classe Pedido, também tem o mesmo mapeamento, o mapeamento do mesmo relacionamento, que, no caso aqui, com é o lado contrário, é *one to many*. Eu também estou mapeando o "itens_pedido", então é de "pedidos" para "itens_pedido", eu mapeei essas duas pontas, de "itens_pedido" para "pedidos" e de "pedidos" para "itens_pedido".

[00:48] Isso configura um relacionamento bidirecional, um relacionamento das duas pontas, os dois lados estão se mapeando. Porém, sempre que isso acontecer, nós temos que tomar um certo cuidado, porque, por padrão, se não indicarmos que esse é um relacionamento bidirecional, a JPA, ela não vai entender, ela vai supor que isso é um novo mapeamento, que é um novo relacionamento.

[01:11] Em vez de termos essas cinco tabelas, as do *slide*, ela vai gerar uma sexta tabela. Então ela não vai considerar que esse mapeamento `private List<ItemPedido> itens;` , do "Pedido", é o lado oposto desse mapeamento

`private Pedido pedido;`, da classe "ItemPedido". Ela vai supor que é outro relacionamento e ela vai criar uma sexta tabela, uma segunda tabela de *join*.

[01:29] Nós podemos até simular isso, podemos abrir aquela nossa classe de teste. Deixa eu abrir ela aqui, está no pacote "testes > CadastroDeProduto". Vou só rodar essa classe para ele gerar aqui no console os *logs* do *hibernate*. Vamos olhar nos *logs* de criação de tabelas.

[01:47] Perceba, ele *logou* aqui. Ele criou a tabela de categorias. Criou a tabela de clientes, ok, a nossa tabela de clientes. Criou a nossa tabela `itens_pedido`, que é a tabela de *join*, ok. Criou a tabela de pedidos e a tabela de produto, aqui embaixo.

[02:08] Contudo, antes, ele criou essa tabela `pedidos_itens_pedido`. Não existe essa sexta tabela aqui no *slide*.

[02:14] Aqui está o problema. Ele criou essa tabela aqui justamente por conta do relacionamento de “Pedido” para “ItemPedido”. Então ele já tinha visto o mapeamento de ItemPedido para Pedido, que é o *many to one*, mas, quando ele chegou aqui, no `@OneToMany`, que é o lado contrário, ele não sabe que isso é o lado contrário do relacionamento, para ele isso é um novo relacionamento.

[02:36] Por isso ele gerou uma nova tabela. Para isso não acontecer, no lado que tem *to many*, temos que colocar aqui, abrir parênteses e colocar um atributo, chamado `@OneToMany(mappedBy = " ")`, que é para indicar: olha, JPA, esse relacionamento, ele já está mapeado lá do outro lado. Que outro lado? Na classe ItemPedido, pelo atributo chamado `pedido`.

[02:59] Aqui nós passamos o nome do atributo que está no outro lado do relacionamento, `(mapped By = "pedido")`. É só fazer isso e pronto, resolveu o problema. Se rodarmos de novo aquela classe de teste - deixa eu rodar ela aqui. Vamos lá para cima, ver os *logs*. Criou a tabela de categorias, de clientes, `itens_pedido`, ok, pedido, produto.

[03:21] Acabou, para baixo é só *alter table* para ele gerar as chaves estrangeiras, as *foregin keys*. Agora ele não gerou aquela sexta tabela, a segunda tabela de *join*, agora ele entendeu que esse relacionamento *one to many* é o lado oposto do *many to one* que já está mapeado naquela entidade ItemPedido.

[03:41] Então cuidado quando for fazer um relacionamento bidirecional porque um dos lados é o lado inverso e você terá que colocar o `mappedBy` . Outra dica, agora de boas práticas. Eu, particularmente, sempre prefiro, quando tem um relacionamento por uma lista, já inicializar a lista aqui na declaração do atributo.

[03:58] Aqui já colocar `private List<ItemPedido> itens = new ArrayList<>();` para inicializar essa lista como uma lista vazia, porque senão teremos que sempre ficar fazendo aquele *if, if* lista foi instanciada? Se a lista é nula, dá *new* na lista; *if* a lista é nula, dá *new* na lista. Então teríamos que fazer isso o tempo inteiro. Para evitar essa checagem, já inicializamos aqui a coleção.

[04:22] Ele sempre vai começar uma coleção vazia, então eu não preciso verificar se está nula, nunca estará nula, sempre estará instanciada e com uma coleção vazia. Uma outra dica de boa prática: como é um relacionamento muitos para muitos, quando formos instanciar esses objetos, temos que lembrar de *setar*, de, no "Pedido", adicionar nessa lista o "ItemPedido", mas no "ItemPedido", que está sendo adicionado, também temos que *setar* o "Pedido".

[04:50] Lembre: é bidirecional, os dois lados têm que se conhecer. Para evitar espalhar esse tipo de código, é comum, é uma boa prática, aqui, por exemplo, no nosso caso, na classe "Pedido", criarmos um método e esse método é que vai adicionar um item nessa lista. Nesse método utilitário, ele já faz esse vínculo dos dois lados do relacionamento.

[05:12] Então podemos criar um método aqui, em "Pedido", por exemplo, `public void adicionarItem()` . Então, na classe de "Pedido", eu tenho um método para adicionar um item, e ele recebe como parâmetro um `(ItemPedi`

`item)` . Aqui dentro, o que eu faço? `item.setPedido(this);` . No item eu vou *setar* o pedido como sendo o *this*, o próprio pedido, a própria classe atual.

[05:39] Eu também vou pegar aqui `this.itens` , que é a lista, e vou adicionar esse `this.itens.add(item);` . Dessa maneira eu estou vinculando os dois lados, o item conhece o pedido e o pedido conhece o item, eu adicionei na lista de itens esse novo item. Inclusive, lembra que a nossa classe de "ItemPedido", ela não tem em `Pedido pedido` , ela não tem um preço unitário?

[06:09] Temos que lembrar de sempre passar esse preço unitário. Podemos ter um método aqui. Na verdade, um construtor. O construtor, eu gerei o construtor padrão e o que recebe os atributos. Nesse construtor que recebe os atributos, eu já não recebo o pedido?

[06:25] E eu não tenho que guardar qual é o preço unitário dele? Eu posso fazer isso direto no construtor para não esquecer, `this.precoUnitario = produto` , passado como um parâmetro, `= produto.getPreco();` . Com isso, na hora que eu der um *new* no "ItemPedido", eu não passo um produto? Eu já passo qual é o preço unitário dele já para não esquecer.

[06:49] Só alguns detalhes, algumas dicas, para você não esquecer, e principalmente relacionado com relacionamento bidirecional. Quando você tem esse relacionamento bidirecional, você tem que colocar o `mappedBy` , geralmente do lado do `@OneToMany` , é do lado do `@OneToMany` que você coloca o `mappedBy` .

[07:08] Essa *string* `"pedido"` que você passa aqui é o nome do atributo do outro lado. É o nome do atributo, aqui é `"pedido"` , então nessa *string* tem que ser o mesmo nome do atributo. A dica também de inicializar as coleções, não só para *one to many*, *many to many* também, sempre que tiver lista, já inicializa a coleção.

[07:25] E você criar um método utilitário nessa classe, para receber o item, q é o elemento desta coleção. E *setar* os dois lados do relacionamento, para não

ter perigo de você esquecer de *setar*. Geralmente o pessoal esquece de *setar* esse lado aqui, geralmente o código fica assim.

[07:41] Itens, adiciona esse pedido. Só que o item fica sem o pedido, ele não tem o outro lado vinculado. Então não se esqueça de fazer esse lado do relacionamento. Por hoje, esse era o objetivo do nosso vídeo, entender o esquema do mapeamento bidirecional, que temos que mapear os dois lados do relacionamento, e umas dicas de boas práticas quando lidarmos com coleções e em relação ao mapeamento bidirecional.

[08:08] No próximo vídeo vamos fazer um exemplo, eu vou criar um produto, adicionar um item e ver como é que funciona isso na prática, naquela nossa classe *main*, vamos testar esse relacionamento e ver se ele vai inserir tudo certo. Vejo vocês lá, um abraço.