03

Mapeamento de herança

Transcrição

[00:00] Olá, pessoal! Continuando então, vamos estudar outro conceito importante, que eventualmente é comum nas aplicações, o conceito de herança. Às vezes você tem que modelar as classes baseado em herança e eventualmente ser entidade da JPA. A JPA, ela tem suporte para mapeamento de herança no banco de dados. Vamos simular e ver como é que funciona isso.

[00:21] Vamos imaginar a nossa classe, a nossa entidade "Produto", imagine que agora eu quero ter subprodutos. Agora eu tenho um produto, mas eu quero também ter, além das informações que já tem no produto, *id*, nome, descrição, preço, data de cadastro e categoria, eu quero ter mais informações que são específicas de cada tipo de produto.

[00:41] Por exemplo, se for um livro, vai ter o nome do autor, o número de páginas, a editora. Se for um produto de informática, qual é a marca, o modelo do aparelho, enfim. Só que no Java, eu não quero deixar todas essas possibilidades dentro da mesma classe "Produto", então eu vou criar subclasses, utilizar herança. Vamos fazer isso aqui.

[01:03] Eu vou criar uma nova classe, no pacote de "modelo", chamada "Livro". Aqui eu quero representar um livro. E um livro, ele é um produto, então eu vou usar herança: `public class Livro extends Produto` . Tudo o que tem no "Produto", o "Livro" tem, só que a diferença é que o "Livro", ele tem mais dois atributos.

[01:22] Tem o autor, que é uma *sting*, `private String autor;`, e tem um `private Integer numeroDePaginas;`. Tem esses dois atributos. Vou gerar aqui os *getters* e *setters*, fazer aquele mesmo esquema, "Source > Generate getters and setters...", selecionar tudo, ok. Vou criar o construtor *default* primeiro, embora a ordem não interesse. Vou gerar aqui o construtor usando todos os atributos e está aqui a nossa entidade.

[02:00] Da mesma maneira, eu quero ter, além da classe "Livro", vou copiar e colar a classe "Livro", eu quero ter a classe "Informatica". "Informatica" nada mais é do que um produto também, e ele tem `private String marca;` e `private Integer modelo;`, essa é a única diferença, tem marca, marca, marca, marca, modelo, modelo, modelo. Eu vou gerar de novo aqui os *getters* e *setters*.

[02:31] "Source > Generate getters and setters". Ok. Dar uma formatada aqui no código, pronto. Então herança, conforme existe na orientação a objetos. Só que e a JPA? Como é que isso vai funcionar para a JPA? Como é que eu faço o mapeamento de herança? Vai depender de como você quer que seja, como você quer que fique a modelagem do banco de dados.

[02:57] A JPA, ela tem suporte para algumas estratégias. Vamos estudar as duas principais. Por exemplo, uma alternativa, seria você usar essa estratégia aqui, que é chamada de *single table*.

[03:08] Embora no Java você tenha subclasses, você tem a classe "Produto" só com os atributos comuns e as subclasses "Livro" e "Informatica" herdando de "Produto" e cada uma com os seus atributos em específico, você pode, no banco de dados, ter uma única tabela, um tabelão gigantesco com todos os atributos misturados.

[03:29] Então tem ID, nome, descrição, preço, data, autor, número de páginas, marca e modelo. Percebe? Dá para ficar dessa maneira. Essa aqui, a vantagem é que você tem mais performance, já que é uma única tabela, não são tabelas quebradas, não vai ter *join*, então é mais performático. A desvantagem é que

fica tudo misturado em uma mesma tabela. Mas a JPA suporta esse modelo, se for o que você desejar.

[03:52] Para fazer isso, na classe "Produto", em cima da entidade `Produto`, que é a classe base, a classe mãe, você coloca uma anotação, que é o `@Inheritance()`. Para dizer: olha, JPA, essa classe aqui, eu vou usar ela como herança. Nos parênteses, tem o atributo `(strategy = )`, que você passa `(strategy = InheritanceType.SINGLE_TABLE)`.

[04:15] A estratégia é um tabelão gigantesco. Ele já assume que "Livro" e "Informatica" são duas entidades. Você tem que colocar o `@Entity` em "Livro", `@Entity` em "Informatica". Pronto. O `@Id` não precisa, já está herdando da classe "Produto", esses atributos já serão herdados. Pronto. A anotação `@Inheritance` é só na classe pai, na classe base.

[04:45] Vamos rodar uma daquelas classes de teste. Cadê? A "TesteCriteria", vou rodar esta aqui. "Run As". Vamos ver como é que ele vai gerar as tabelas para mim. Vamos subir tudo, cadê as tabelas? Categoria, cliente, item pedido, pedido, produto.

[05:04] Ele criou a tabela "produtos". Está lá ID, data, descrição, nome, preço, marca, modelo, autor, número de páginas. Ele criou certo. Porém ele criou mais um atributo, chamado "DTYPE". Por quê? Como é um tabelão, quando fizermos um *select*, fizermos uma consulta com a JPQL ou com o *criteria*, o *hibernate*, ele precisa saber: eu trouxe um registro dessa única tabela, esse registro, ele é uma instância de livro ou de informática?

[05:35] Como só tem uma única tabela, como o *hibernate* vai saber? Ele precisa que tenha mais uma coluna na classe. Por padrão, ele chama essa coluna de "DTYPE", e ela é um "varchar". O que ele coloca nessa coluna? Se salvarmos um registro, o que ele insere? Ele insere, na coluna "DTYPE" o nome da classe.

[05:57] Então ele vai inserir o registro, se for um objeto do tipo informática, e insere na coluna a palavra "Informatica". Se for um objeto do tipo livro, ele

insere a palavra "Livro". Tem como você personalizar o nome dessa coluna e o tipo, para invés de ser a *string* com o nome da classe, ser um número, ser só uma letra, enfim, dá para personalizar isso também.

[06:17] É assim que funciona a estratégia *single table*. Você tem a classe base anotada com um `@Inheritance`, `SINGLE_TABLE` a estratégia, e as classes filhas apenas `@Entity` e o `extends` normalmente. A outra estratégia seria essa estratégia aqui, que é a segunda mais comum.

[06:38] Você não quer uma tabela gigantesca, você quer uma classe para cada tabela, para cada subclasse, e a classe para a tabela pai, a tabela mãe, a tabela base. Na tabela "produtos", eu tenho só os atributos comuns, e na tabela "informatica" e "livros", eu tenho os atributos específicos, porém tem que ter o ID, que além de ser chave primária, é uma chave estrangeira.

[07:01] Então, quando eu inserir um livro, ele vai dar um *insert* na tabela "livros" e vai dar também um *insert* na tabela "produtos". Em "produtos" vai ter, sei lá, ID 33, na tabela "livros" vai estar o mesmo ID, 33. A chave primária é a chave estrangeira da tabela "produtos". Assim, a vantagem é que fica mais organizado, não fica aquele troço gigantesco, bagunçado.

[07:21] A desvantagem é que se eu fizer um *select*, uma *query* na tabela "livros", na entidade de livro, ele vai ter que fazer um *join* com a tabela "produtos" para carregar os dados que estão na tabela, os dados comuns. Então tem *join* na consulta, dependendo, pode ser um problema de performance. Para trocar, é simples. Na classe "Produto", você troca a estratégia: `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`.

[07:43] Pronto, acabou. Percebe como é fácil trocar a estratégia? É só você trocar no `@Inheritance`. E nessa estratégia você não precisa mexer em nada, as classes continuam iguais, continuam herdando da classe "Produto" e os atributos específicos em cada classe e os atributos comuns na classe base. Vamos rodar aquela nossa classe "TesteCriteria" e ver o que ele vai gerar.

[08:08] Seguindo, "Cannot use identity" produto. Classe "Produto" deu um problema aqui. Deixa eu dar uma conferida. É `TABLE_PER_CLASS` mesmo? Na verdade, não é `TABLE_PER_CLASS`, eu escrevi errado. A estratégia que eu queria mostrar aqui é a `JOINED`. Essa estratégia aqui é a *joined*.

[08:27] Vamos rodar novamente o "TesteCriteria". Cadê os *create table*? Categoria, cliente, olha.

[08:41] Ele criou uma tabela "Informatica", que é para a classe "Informatica", criou as colunas marca, modelo e o ID.

[08:53] Criou a tabela "Livro". Autor, número de páginas e o ID. E a tabela de "produtos". Nela tem só os atributos em comum, data de cadastro, ID, descrição.

[09:03] Então é a segunda estratégia comum, a estratégia *joined*, que é a estratégia que ele faz o *join*, você tem a tabela comum e as tabelas filhas, cada uma com os seus respectivos atributos e a tabela comum com os atributos, comuns para todos.

[09:15] Aqui não é *table per class*, é *joined*, confundi. Nesse caso, não tem aquela coluna de descriminação. Aqui o nome da tabela ficou "Livro" com L maiúsculo porque eu não coloquei o `@Table` nas subclasses. Está só o `@Entity`, por padrão o nome da classe é o mesmo nome que ele vai gerar para a tabela. Se eu quisesse trocar, teria que colocar o `@Table`, conforme já conhecemos.

[09:39] Então essas são as duas principais estratégias, *single table* e *joined*. Essa *table per class*, tem outras estratégias também, mas não são comuns. As duas principais, que vocês sempre vão encontrar, são essas duas. Vou deixar como um desafio para você dar uma lida no *table per class*, dar uma pesquisada em qual é a diferença dele para o *joined* e como é que faz para configurar na JPA, se tem que mudar alguma coisa.

[10:04] Provavelmente você já sabe que sim, porque eu cometi o erro e deu uma *exception*, então dê uma pesquisada, fica o desafio para você entender o *table per class*. Mas as duas principais *single table* e *joined*. Espero que vocês tenham gostado e no próximo vídeo nós vemos mais um recurso da JPA. Vejo vocês lá.