03

Mapeando uma entidade

Transcrição

Já fizemos as configurações e podemos continuar. Mas antes, vamos corrigir dois detalhes que acabei errando no vídeo anterior. Na propriedade da URL do JDBC, antes do `"h2:mem:loja"`, faltou o prefixo `jdbc:`. Portanto, o correto é `"jdbc:h2:mem:loja"`. E na propriedade do usuário do banco de dados não é `username"` e, sim, `user"`. É complicado querer decorar, o melhor é pegar de exemplos na internet ou de outro projeto.

Enfim, temos as informações do banco de dados configuradas para a JPA. A JPA já sabe se comunicar com o JDBC e passar essas configurações na hora em que precisar acessar o banco de dados. Agora configuraremos a parte de persistência. Nosso projeto é uma loja, como faremos para integrar? Como ensinaremos para a JPA a nossa tabela no banco de dados?

No nosso caso, teremos, inicialmente, a seguinte **Tabela de produtos**:

Produtos	
id	bigint
nome	varchar
descricao	varchar
preco	decimal

Vamos imaginar que criaremos essa tabela de produtos no banco de dados e que ela tem essas quatro colunas: uma chamada "id", que é a chave primária do banco de dados (autogerada pelo banco de dados); o "nome", que é um

"varchar" (um texto, uma descrição do produto); a "descricao" do produto, que também é um texto, um "varchar"; e o "preco", que é um "decimal".

Então, tendo essa tabela no banco de dados, como ensinaremos e a configuraremos para que seja representada de alguma maneira no código Java? Na JPA, isso será feito por uma classe Java, que na JPA é chamda de entidade. Nós mapeamos todas as tabelas no banco de dados por uma entidade, que nada mais é do que uma classe Java.

Vamos criar a classe que representará um produto no banco de dados. Já que é uma classe Java, ficará no "src/main/java". Apertaremos "Ctrl + N", em "Wizards", digitaremos "class" para filtrar e selecionaremos "Next". Na próxima tela, trocaremos o pacote de "loja" para "br.com.alura.loja", e o nome da classe será "Produto".

Essa será a nossa classe chamada `Produto.java`, que representará um produto no banco de dados. Vamos apenas adicionar ".modelo" no pacote, isto é, `br.com.alura.loja.modelo` e podemos prosseguir.

Na JPA, precisamos lembrar que a ideia não é que ela seja uma especificação para um ORM. Com a ORM nós fazemos o mapeamento objeto-relacional. Nós precisamos desse mapeamento, dessa ligação entre o lado da orientação objetos com o lado do mundo relacional do banco de dados. Isso é feito na classe `public class Produto {`, e é ela que está representando a tabela de produtos, portanto, é assim que a indicaremos para a JPA.

A partir da versão 2.0 da JPA, podemos fazer tudo via anotações. Então, em cima da classe, podemos colocar uma anotação da JPA que é o `@Entity`. Assim, é como se disséssemos: JPA, está vendo essa classe `Produto`? Ela é uma entidade, ou seja, existe uma tabela no banco de dados que está mapeando, e que é o espelho dessa classe. Então, é para isso que serve essa anotação `@Entity`.

Agora, apertamos "Ctrl + Shift + O" para importar. Ele sugeriu duas opções para importar e precisamos escolher com cuidado. Existe a opção `javax.persistence.Entity` e a `org.hibernate.annotations.Entity`. A primeira é a anotação da especificação da JPA. A segunda, é a do Hibernate. Então, importaremos a do `javax.persistence.Entity`, que é a da especificação.

Se não queremos ficar presos ao Hibernate - a uma implementação - e desejamos usar o máximo possível a especificação, porque se um dia quisermos trocá-la, não teremos que mexer em todas as classes. Portanto, precisamos ter cuidado com tudo que importarmos.

```
package br.com.alura.loja.modelo;

import javax.persistence.Entity;

@Entity
public class Produto {
```

COPIAR CÓDIGO

Com isso, o Hibernate JPA já sabe que a classe `Produto` está mapeando uma tabela no banco de dados. Só que, no banco de dados, o nome da tabela é "produtos" no plural. Como é possível dizer isso para a JPA, já que não queremos nomear a classe como "Produtos" no plural e com “p” minúsculo? Algo que contrariaria as convenções do Java.

Eventualmente, se o nome da tabela não for o mesmo da entidade, teremos que ensinar isso para a JPA, porque, por padrão, ela considera que o nome da tabela é o mesmo nome da entidade (no nosso caso, não é). Para fazer essa configuração, adicionaremos mais uma anotação em cima da classe que é o `@Table`. Apertaremos "Ctrl + Shift + O" para importar e, de novo, selecionaremos `javax.persistence.Table`.

Na anotação `@Table`, abriremos parênteses, selecionaremos o atributo `name:String - Table` com a qual passaremos o nome da tabela que é `name =`

"produtos" .

```
package br.com.alura.loja.modelo;

import javax.persistence.Entity;

@Entity
@Table(name = "produtos")
public class Produto {
```

COPIAR CÓDIGO

Com isso, ensinamos a JPA que, embora o nome da entidade seja `Produto` , o nome da tabela é `produtos` . Agora ela já sabe que, ao fazer a ligação, ela precisará fazer também a conversão. Dentro da classe, nós temos os atributos, que nada mais são do que o espelho das colunas no banco de dados. A nossa tabela tem quatro colunas (id, nome, descricao e preco) e nós as transformaremos em atributos. Vamos adicioná-los:

```
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Uma curiosidade é que o nome dos atributos é exatamente igual ao nome das colunas no banco de dados. Logo, isso é algo que não precisaremos ensinar para a JPA, ela já assume que o nome da coluna é o mesmo do atributo dentro da entidade. Se fosse diferente, isto é, se o nome da coluna "descricao" fosse "desc", por exemplo, como ensinaríamos para a JPA caso não quiséssemos chamar o atributo de desc e, sim, de "descricao"?

Neste caso, nós colocaríamos, em cima do atributo, uma anotação chamada `@Column` (e apertaríamos "Ctrl + Shift + O" para importar). Da mesma maneir

existe uma atributo chamado `name`, seguido dele, passaríamos o nome da coluna no banco de dados `"desc"`. Ou seja, `"Column(name = "desc")`. É como se disséssemos para a JPA: o nome do atributo é `descricao`, mas o nome da coluna, `@Column`, é `desc`.

Desta maneira, é possível ensinar a JPA quando o nome da coluna for diferente do nome do atributo. No nosso caso, vamos apagar essa anotação, porque o nome da coluna é exatamente igual ao nome do atributo.

Este processo que fizemos tem o nome de **mapeamento**, isto é, fizemos o mapeamento de uma entidade, ensinamos ao Java e JPA que a classe `Produto` representa uma tabela, que o nome da tabela é diferente, no banco de dados, do nome da classe. Ensinamos também quais são os atributos que serão mapeados como colunas.

Só temos mais um detalhe importante para a JPA. No banco de dados, a coluna "id" é a chave primária. Nós precisamos informar qual é a "*primary key*", a chave primária da tabela no mundo relacional. Também precisamos informar para a JPA que, dos quatro atributos, o primeiro, que se chama `id`, é a chave primária, já que ele não associa automaticamente.

Em cima do atributo `id`, colocaremos uma notação chamada `@Id` e apertamos "Ctrl + Shift + O" para importar. No nosso caso, ele importou diretamente do `javax.persistence.Id`. Como, geralmente, quem cuida do `id`, da chave primária é o banco de dados e não a aplicação, também precisamos ensinar para a JPA que quem gerará o identificador não é a aplicação e, sim, o banco de dados.

Quando formos salvar um produto, o `id` estará nulo. Não tem problema, porque é o banco de dados que vai gerar o próximo `id`. Podemos configurar isso com outra notação, que colocamos em cima do atributo `id`, que é o `@GeneratedValue`, isto é, para dizer como o valor da chave primária é gerado.

```
@Id
@GeneratedValue()
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Existe um parâmetro que precisamos passar que é a estratégia, `strategy`, isto é, qual é a estratégia de geração da chave primária. Isso dependerá do banco de dados, alguns usam SEQUENCE, outros não. Então, nas estratégias, temos três opções: IDENTITY; SEQUENCE; e TABLE. Geralmente, utilizamos a IDENTITY, quando não tem SEQUENCE no banco de dados, ou SEQUENCE, quando tem. No nosso caso, será IDENTITY, já que não temos SEQUENCE no H2.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Feito isso, falta apenas gerar os *Getters e Setters*. Abriremos um atalho com o botão direito e selecionaremos "Source > Generate Getters and Setters". Na próxima tela, marcaremos todos os atributos com "Sellect All" e apertaremos "Generate". Retornando ao `Produto.java`, selecionaremos o comando "Ctrl + Shift + F" para formatar e está pronto o mapeamento da entidade.

Então, é assim que mapeamos quais classes vão representar tabelas no banco de dados. Depois conheceremos outras anotações. Quando tivermos relacionamentos de tabelas, aprenderemos a mapear também esses relacionamentos. Enfim, veremos tudo isso com calma durante o curso.

Um último detalhe para fecharmos esse vídeo. Pela JPA, em relação a toda entidade, além de precisarmos ir até a classe e adicionar anotações da JPA para fazer o mapeamento, também deveríamos adicionar a classe no `persistence.xml`. Fora das `properties` e dentro do `persistence-unit`, exite outra tag chamada `class`.

Pela JPA, deveríamos passar todas as classes/entidades do nosso projeto, ou seja, passaríamos o caminho completo da classe, `br.com.alura.loja.modelo.Produto`. Pela JPA, para cada entidade que mapearmos, além de mapear na classe, temos que adicioná-la no `persistence.xml` com a tag `class`.

```
<class>br.com.alura.loja.modelo.Produto</class>
```

COPIAR CÓDIGO

Porém, se tivermos utilizando o Hibernate, não precisamos adicionar a tag `class`, porque ele consegue encontrar automaticamente as classes/entidades do nosso projeto. Essa é uma particularidade do Hibernate, pode ser que as outras implementações não façam isso e, portanto, teremos que, manualmente, adicionar o `class`.

Como estamos utilizando o Hibernate, e esse processo é meio trabalhoso: ao criar uma nova tabela no banco, temos que criar a classe, fazer o mapeamento e adicionar no `persistence.xml`. Para não esquecermos de nada, não vamos adicionar no nosso código, pois o Hibernate encontrará automaticamente.

Outro detalhe importante, se adicionarmos uma entidade no `class`, temos que adicionar todas. Se esquecermos alguma, o Hibernate só olhará para as que estiverem declaradas. Ou mapeamos todas, ou nenhuma.

Esse era o objetivo do vídeo de hoje, mostrar como fazemos o mapeamento de uma entidade. Agora, já temos o `persistence.xml` com as configurações do

banco, já temos uma entidade mapeada e, no próximo vídeo, veremos como fazer para cadastrar um produto no banco de dados (dentro da JPA e das classes Java).