05

Mapeando relacionamentos

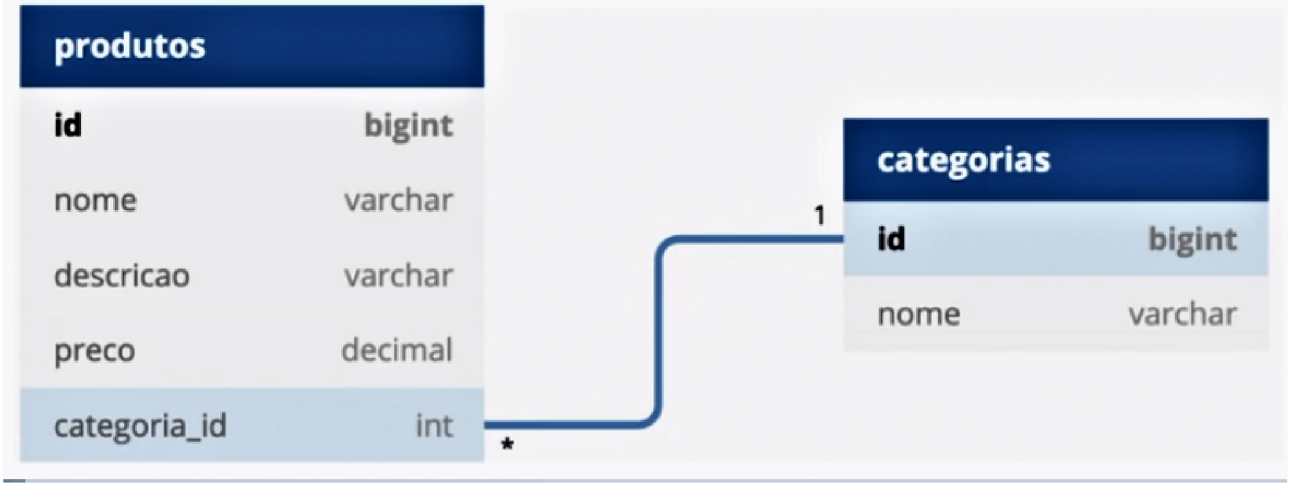
Transcrição

Na última aula, nós fizemos o mapeamento do *enum*. Nesta aula, nós precisaremos fazer uma alteração nele. Quando utilizamos *enum* no Java, passamos as constantes, que no nosso caso são, `CELULARES`, `INFORMATICA` e `LIVROS`, mas elas são fixas conforme o que está no código.

O problema é esse, para o sistema, não existe uma flexibilidade, portanto, se o usuário precisar cadastrar uma nova categoria, teremos que mexer no código fonte da aplicação, fazer um novo *build*, um novo *deploy* e subir outra vez a aplicação no ar.

Seria mais interessante se os próprios usuários pudessem cadastrar essas categorias para que fosse mais flexível. No sistema mesmo teria uma tela, um cadastro de categorias. Então, não queremos mais que fique fixo, mas sim deixar via sistema, com uma tabela para armazenar essas categorias. Ao fazer essa alteração, `Categoria` não será mais um *enum*, mas, sim, uma entidade, pois teremos uma tabela de Categoria.

A ideia é, justamente, a que está no seguinte diagrama.



Antes tínhamos apenas a tabela de "produtos". Agora, temos também a de "categorias" com duas colunas: o "id", que é a chave primária; e o "nome", referente ao nome da categoria. Existe um relacionamento entre o produto e a categoria. Todo produto pertence a uma categoria.

Então, na tabela de produtos, precisaremos adicionar a coluna "categoria_id", que, na verdade, é uma chave estrangeira, uma FK (Foreign Key) que aponta para o "id" na tabela de "categorias". Agora, precisaremos fazer um mapeamento desse relacionamento, que é algo que não tínhamos visto ainda. A JPA suporta normalmente mapear um relacionamento entre tabelas. Vamos ver como isso ficará.

O primeiro passo, na `Categoria.java`, é que `Categoria` não será mais um *enum*, portanto, trocaremos para `class`. Apagaremos as constantes, e teremos dois atributos, o `id` e o `nome`.

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

public class Categoria {

    private Long id;
        private String nome;

}
```

COPIAR CÓDIGO

A classe `Categoria` é uma entidade, então precisamos colocar as anotações da JPA, que podemos pegar da classe `Produto.java`. Então, pegaremos o `@Entity` e o `@Table(name = "produtos")`. Vamos copiar e colar, trocando o nome da tabela por `"categorias"`.

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

import javax.persistence.Entity;

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

    private Long id;
        private String nome;

}
```

COPIAR CÓDIGO

Copiaremos também o `@Id` e `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

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

import javax.persistence.Entity;

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

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

```
}
```

COPIAR CÓDIGO

Tudo certo! O `nome` é `String`, ele mapeará como *varchar*, não muda nada. Só precisamos agora dos *Getters* e *Setters*. Criaremos também um construtor, assim como fizemos na entidade de produto. Então, no atalho, selecionaremos "Source > Generate Constructor". Na próxima tela, desmarcaremos o "id" (porque ele é gerado pelo banco de dados) e deixaremos só o "nome". Agora basta apertar "Generate".

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

    public Categoria(String nome) {
        this.nome = nome;
    }
```

COPIAR CÓDIGO

Quando alguém der `new` na classe `Categoria`, já tem que passar o nome. Vamos gerar o método `getNome()` e colocar também um `setNome`. Em teoria, nós não precisaríamos do *Setter*, porque, ele já vem na hora de dar `new`, mas, vamos deixar aqui, caso precise. Já temos a nossa entidade de `Categoria` mapeada.

```
    public Categoria(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
```

```
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
```

COPIAR CÓDIGO

É simples mapear uma tabela do banco de dados, basta criar a classe e colocar as anotações da JPA, declarar os atributos e fazer os mapeamentos usando as anotações da JPA conforme a necessidade. Porém, na classe `Produto.java`, `Categoria` agora não é mais um *enum*, então, apagaremos o `@Enumerated(EnumType.STRING)`.

Agora, teremos um problema, porque a JPA detectará que, na entidade `Produto`, existe um atributo `Categoria`, e que seu tipo não é mais primitivo do Java, não é mais um *enum* e nem uma classe do Java. Ela detectará que esse tipo é uma outra classe do nosso projeto e que essa classe é uma entidade (`@Entity`). Então, a JPA automaticamente saberá que isso é um relacionamento entre `Produto` e `Categoria`, isto é, um relacionamento de duas entidades.

Desta maneira, temos a obrigação de dizer à JPA qual é a cardinalidade desse relacionamento. Se um produto tem uma única categoria ou várias categorias, é um para um, um para muitos, muitos para um, ou seja, qual é a cardinalidade. Se observarmos o desenho do diagrama, veremos que, de produtos para categorias, temos: muitos para um. Isto é, um produto tem uma única categoria, mas uma categoria pode estar vinculada a vários produtos.

Então, de produtos para categorias: asterisco, 1. Que quer dizer: muitos para um. Na JPA, para informarmos que a cardinalidade desse relacionamento é "muitos para um", temos uma anotação, `@ManyToOne`. Ou seja, muitos produtos estão vinculados com uma `Categoria`. Uma categoria pode ter vários produtos, mas o produto tem uma única categoria.

@ManyToOne

```
    private Categoria categoria;

    public Produto(String nome, String descricao, BigDecimal
preco, Categoria categoria) {
        this.nome = nome;
        this.descricao = descricao;
        this.preco = preco;
        this.categoria = categoria;
    }
```

COPIAR CÓDIGO

Existem algumas anotações da JPA para relacionamento, o `@ManyToOne`, `@OneToMany` (que é o contrário), `@OneToOne` e `@MayToMany`. A escolha dependerá da cardinalidade, do tipo de relacionamento entre as tabelas. Agora, no `CadastroDeProduto.java` está dando um erro em `Categoria.CELULARES`, porque `Categoria` não é mais um *enum*. Precisamos de uma categoria cadastrada no banco de dados para associá-la com esse produto.

Sendo assim, vamos colocar uma categoria `celulares`, mas dará um erro de compilação. Vamos selecionar o comando "Ctrl + 1"e criar uma variável local. Nós precisamos da `Categoria celulares`, então, `new Categoria()`, e passaremos o nome `"CELULARES"`.

```
public static void main(String[] args) {
    Categoria celulares = new Categoria("CELULARES");
```

COPIAR CÓDIGO

Mas, nós precisamos salvar essa categoria no banco de dados antes de salvar o produto, ou teremos problemas. Vamos rodar e ver o problema selecionando "Run As > 1 Java Application". No Console, veremos que deu uma *exception* "Transient Property Value Exception". Significa que, existe um valor na propriedade que é "transient", não está salvo, não está persistido.

Na próxima aula, nós discutiremos sobre estados da entidade para saber o que é "transient" e como funciona essa transição de estados de uma entidade na JPA. Mas, a questão é: nós temos uma categoria e acabamos de instanciá-la, mas ela não está salva no banco de dados. Na hora em que criamos um produto, nos vinculamos a essa categoria que não está persistida no banco de dados.

E, na hora que chamamos o `dao.cadastrar(celular);`, a JPA detectou. Ela foi fazer um *insert* do produto, mas viu que esse produto estava relacionado com uma categoria, mas essa categoria ainda não tinha sido persistida no banco de dados. Assim, ela considera que isso é um erro e lança uma *exception*.

O correto é salvar a categoria e, depois, salvar o produto. Sendo assim, a categoria já estará gerenciada pela JPA e estará na tabela, terá um id, e será possível fazer o relacionamento. Então, precisamos cadastar no banco de dados a categoria. Nós havíamos criado uma classe `ProdutoDAO`, podemos criar, também, uma categoria DAO seguindo o mesmo modelo.

Portanto, vamos selecionar com o comando "Ctrl + C" o `ProdutoDao.java` e colar e renomear para `CategoriaDao` em "Enter a new name for 'ProdutoDao'". Agora basta apertar "Ok". Vamos abrir o `CategoriaDao.java`, ele recebe o `EntityManager` da mesma maneira. Enfim, é a mesma estrutura, as classes DAO serão parecidas. Mas, em `CategoriaDao.java`, vez de ser um `Produto`, será uma `Categoria`. Depois, apertaremos "Ctrl + Shift + O" para importar.

```
    public CategoriaDao(EntityManager em) {
        this.em = em;
        }

        public void cadastrar(Categoria categoria) {
            this.em.persist(categoria);
        }

}
```

COPIAR CÓDIGO

Está pronta a nossa classe DAO. Agora, em `CadastroDeProduto.java`, nós criamos a `Categoria celulares`, o `Produto celular`, o `EntityManager`, o `ProdutoDao`, vamos apenas renomear a variável `dao` para `produtoDao`, porque agora precisaremos ter duas classes DAO - dois objetos DAO.

```
ProdutoDao produtoDao = new ProdutoDao(em);
```

COPIAR CÓDIGO

Nós precisaremos instanciar também o `CategoriaDao`. Vamos copiar a linha anterior e, ao invés de `ProdutoDao`, será `CategoriaDao`. O nome da variável será `categoriaDao`, e apertaremos "Ctrl + Shit + O" para importá-la. Algo interessante é que podemos compartilhar o mesmo `EntityManager` entre as várias classes DAO.

```
CategoriaDao categoriaDao = new CategoriaDao(em);
```

COPIAR CÓDIGO

Então, nós iniciamos a transação. Logo abaixo, fizemos o *commit*. Antes de salvar o `produtoDao` no banco de dados, chamaremos `categoriaDao.cadastrar()`, passando a categoria `celulares`.

```
EntityManager em = JPAUtil.getEntityManager();
        ProdutoDao produtoDao = new ProdutoDao(em);
        CategoriaDao categoriaDao = new CategoriaDao(em);

        em.getTransaction().begin();

        categoriaDao.cadastrar(celulares);
        produtoDao.cadastrar(celular);
```

```
        em.getTransaction().commit();
        em.close();
    }

}
```

COPIAR CÓDIGO

Agora, sim, salvamos a categoria, `CategoriaDao.cadastrar(celulares)`, no banco de dados, e depois salvamos o produto, `produto.Dao.cadastrar(celular)`, no banco de dados, já que ele está vinculado com essa categoria. Por isso, é provável que não aconteça mais a *exception* do "transient property value", pois o produto está associado com uma categoria que, sim, está persistida.

Vamos rodar ("Run As > 1 Java Application") e verificar se ele faz o *insert* corretamente no banco de dados. Ele fez o *insert* da categoria, "Hibernate: insert into categorias", e o *insert* do produto, "Hibernate: insert into produtos". Também criou a tabela de categorias, "Hibernate: create table categorias", e a tabela de produtos, "Hibernate: create table produtos".

Ele também fez um "alter table" para adicionar a chave estrangeira na tabela produtos, que agora precisa da coluna do id da categoria. Portanto, ele alterou a tabela e criou a "constraint" *foreign key*.

É assim que fazemos mapeamento de relacionamento na JPA. Sempre que temos uma entidade, onde o atributo é uma outra entidade, a JPA automaticamente identifica que é um relacionamento, e então, precisamos indicar qual é a cardinalidade. Ela não tem um valor padrão para a cardinalidade, nós que precisamos configurar, e isso dependerá de como escolheremos modelar o banco de dados.

Precisamos saber como está modelado no banco de dados para escolher a anotação de relacionamento. O último detalhe é que, na hora de persistir,

temos que ter cuidado, porque, se estamos persistindo com uma entidade e vinculando com outra entidade, essa outra precisa estar persistida antes, ou receberemos uma *exception* "transiente property value excepetion".

O objetivo da aula de hoje foi discutir um pouco sobre relacionamentos, aprendemos como mapeá-lo e também como funciona a parte de persistência. Espero que tenham gostado. Na próxima aula discutiremos sobre os estados das entidades, como funciona o "transient" para a JPA. Vejo vocês lá!! Abraços!!