Spring MVC I: Criando aplicações web
13%

AULA 03 Cadastro de preços do produto

ATIVIDADES
01 Vídeo 1 23min
02 Vídeo 2 07min
03 Cadastro de preços do produto
04 Adicionando os preços do livro
05 Utilização do ModelAndView
06 Configuração de persistência dos preços

03 Cadastro de preços do produto

PRÓXIMA ATIVIDADE

Até este ponto do curso, já fizemos o cadastro de produtos e tivemos a certeza de que tudo está funcionando corretamente. Nossos produtos já estão sendo cadastrados no banco de dados. Mas se verificarmos alguns livros na Casa do Código, veremos que há mais algumas informações que precisamos incluir sobre os produtos.

Vamos voltar ao formulário(`form.jsp`):

```
<form action="casadocodigo/produtos" method="POST">
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas"/>
    </div>
    <button type="submit">Cadastrar</button>
</form>
```

Uma das informações que queremos adicionar é o **Preço**. No caso da Casa do Código, os livros têm três opções de compra. O **Ebook**, **Impresso** e o **Combo** (ebook e impresso juntos). Faremos o mesmo em nosso sistema. Sendo assim, modifique o formulário (`form.jsp`) de produtos adicionando esses três novos campos:

```
<form action="/casadocodigo/produtos" method="post">
[...]
    <div>
        <label>E-book</label>
        <input type="text" name="ebook"/>
    </div>
    <div>
        <label>Impresso</label>
        <input type="text" name="impresso"/>
    </div>
    <div>
        <label>Combo</label>
        <input type="text" name="combo"/>
    </div>
[...]
</form>
```

Dessa forma, temos as três opções de preço. Agora precisamos modificar a classe `Produto` , para que possa guarda-los também.

Imagine colocar os três atributos de preço na classe `Produto` . Estaríamos duplicando informações. Agora imagine se tivermos que adicionar no futuro, outras variações de preço além dessas três... Teríamos que adicionar mais atributos em nossa classe. Pensando assim, parece uma boa ideia ter uma **lista de preços**.

O **preço** representa uma algo importante em nosso negócio. Criaremos então uma classe que representa o preço do produto e neste teremos uma lista com cada um dos preços (Ebook, Impresso e Combo). Sendo assim, modifique a classe `Produto` adicionando o atributo `precos` .

```
@Entity
public class Produto {
    [...]
    private List<Preco> precos;
    [...]
}
```

Observação: Estamos usando atributos privados, então lembre-se sempre de gerar os **getters and setters.** Use os atalhos do Eclipse!

Agora criaremos a classe `Preco` que terá dois atributos. O `valor` , que é o preço propriamente dito e o `tipo` . O tipo limita-se as três opções que temos atualmente: **Ebook, Impresso e Combo.**

Poderíamos usar `String` s para identificar os tipos de preço, mas teríamos que fazer várias verificações na `String` s. Para evitar isso, vamos criar um atributo `TipoPreco` e um `ENUM` que lista nossas três opções de preço. Então teremos a classe `Preco` :

```
public class Preco {
    private BigDecimal valor;
    private TipoPreco tipo;
    [...]
}
```

Depois, iremos criar também o `ENUM TipoPreco` :

```
public enum TipoPreco {
    EBOOK, IMPRESSO, COMBO;
}
```

Observação: A classe `Preco` e o `enum TipoPreco` devem ficar no pacote `br.com.casadocodigo.loja.models` .

Podemos fazer uma relação de produtos com seus preços em duas tabelas diferentes no banco de dados, usando o `id` do produto para estabelecer essa relação ***OneToMany***, ou seja, um produto para vários preços. Mas neste contexto, isso não faria muito sentido, porque teríamos um `id` para o preço e não precisamos disso, pois não vamos reutilizar o preço do produto.

Faremos essa relação de uma outra forma, marcaremos o atributo `List<Preco> precos` da classe `Produto` com a anotação `@ElementCollection` indicando que este atributo é uma coleção de elementos:

```
@Entity
public class Produto {
    [...]
    @ElementCollection
    private List<Preco> precos;
    [...]
}
```

E para que o **Spring** possa relacionar e portar os elementos de preço para dentro desta coleção, devemos marcar a classe `Preco` com uma a anotação `Embeddable` :

```
@Embeddable
public class Preco {
    [...]
}
```

Note que já estamos tomando todos os cuidados para que os preços de nossos sejam flexíveis. A classe `Produto` tem uma lista de `Preco` e o `enum TipoPreco` tem os tipos de preços dos nossos produtos. O nosso `form.jsp` está fixo com as três opções que deixamos disponíveis. Vamos deixá-la mais flexível também.

Veja o código do `form` atual:

```
<form action="/casadocodigo/produtos" method="post">
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas" />
    </div>
    <div>
        <label>Ebook</label>
        <input type="text" name="ebook"/>
    </div>
    <div>
        <label>Impresso</label>
        <input type="text" name="impresso"/>
    </div>
    <div>
        <label>Combo</label>
        <input type="text" name="combo"/>
    </div>
    <button type="submit">Cadastrar</button>
</form>
```

Já que teremos uma lista de preços, podemos fazer um `forEach` nessa lista e exibir todos os preços de forma dinâmica. O arquivo `form.jsp` ficará assim:

```
<form action="/casadocodigo/produtos" method="post">
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas" />
    </div>


    <c:forEach items="${tipos}" var="tipoPreco" varStatus="status">
        <div>
            <label>${tipoPreco}</label>
            <input type="text" name="precos[${status.index}].valor" />
            <input type="hidden" name="precos[${status.index}].tipo" va
        </div>
    </c:forEach>


    <button type="submit">Cadastrar</button>
</form>
```

Perceba que estamos fazendo um `forEach` com `JSTL` em uma coleção ou lista que se chama `tipos` e acessando cada tipo de preço através da variável `tipoPreco` . Estamos também usando o `index` da `varStatus="status"` que serve como uma espécie de contador.

Mas antes de utilizarmos a lista de tipoPreco, temos que fazer uma pequena alteração no ProdutosController. Teremos que mudar o método form para retornar um objeto do tipo ModelAndView.

```
@RequestMapping("/produtos/form")
    public ModelAndView form(){

        ModelAndView modelAndView = new ModelAndView("produtos/form");
        modelAndView.addObject("tipos", TipoPreco.values());

        return modelAndView;
    }
```

Reparem que no construtor de ModelAndView passamos o endereço da View para que o Spring entenda qual o arquivo que ele deverá retornar ao navegador.

Observação: Para que as tags da `JSTL` funcionarem, lembre-se de fazer o importe da `taglib` logo após a **diretiva** de página `JSP` no início do arquivo. Observe:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEnc

<!-- Import da taglib -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
[...]
```

Observação: Caso não conheça a `JSTL` , recomendamos que faça o curso de Java e JSTL: Tags para facilitar o desenvolvimento JSP disponível aqui no Alura.

Nosso laço(`forEach`), manipula o html do formulário para criar os campos de tipos de preços, e o resultado deste laço é algo como:

```
<div>
    <label>EBOOK</label>
    <input type="text" name="precos[0].valor" />
    <input type="hidden" name="precos[0].tipo" value="EBOOK" />
</div>

<div>
    <label>IMPRESSO</label>
    <input type="text" name="precos[1].valor" />
    <input type="hidden" name="precos[1].tipo" value="IMPRESSO" />
</div>

<div>
    <label>COMBO</label>
    <input type="text" name="precos[2].valor" />
    <input type="hidden" name="precos[2].tipo" value="COMBO" />
</div>
```

Perceba que no `name` dos `input` s do formulário, estamos usando: `precos[numero].valor` e `precos[numero].tipo` . Quando enviarmos nosso formulário, o **Spring** ao fazer o `bind` dos atributos do `Produto` , irá detectar que estamos passando valores para o atributo `precos` em determinadas posições da lista. Ele também detectará que os valores são do tipo `tipoPreco` e irá preencher corretamente a lista.

Note que quando fazemos `precos[numero].tipo` , estamos acessando na lista de preços, o atributo `tipo` de um objeto do tipo `Preco` . E lembre-se que o `tipo` é recuperado da `enum TipoPreco` . O `input` do tipo `hidden` serve para que passemos o tipo do preço: **Ebook, Impresso e Combo.** Estes campos são ocultos ao usuário. Não queremos que ele modifique os valores destes campos e insira informações inválidas sobre os tipos de preços em nosso sistema.

Teste cadastrar um novo produto agora. Tudo deve funcionar corretamente. Observe também que no console do Eclipse deve aparecer algo como:

```
Produto [titulo=TDD com JAVA, descricao=TDD com JUnit, paginas=220]
Hibernate: insert into Produto (descricao, paginas, titulo) values (?, ?, ?)
Hibernate: insert into Produto_precos (Produto_id, tipo, valor) values (?, ?, ?)
Hibernate: insert into Produto_precos (Produto_id, tipo, valor) values (?, ?, ?)
Hibernate: insert into Produto_precos (Produto_id, tipo, valor) values (?, ?, ?)
```

Esta saída é do `Hibernate` mostrando o `SQL` gerado. Ela também indica que temos uma nova tabela em nosso banco de dados chamada `Produto_precos` . Mostra também que nesta tabela temos um campo `Produto_id` , que referencia a qual produto aquele preço se refere. Se acessamos nosso banco e fizermos um `SELECT * from Produto_precos;` teremos algo como a seguinte saída:

saída do select no banco de dados

Tipo: 0, 1 e 2? Como assim? Não deveria ser **Ebook, Impresso e Combo**? Para responder essa pergunta, devemos lembrar que por padrão o `enum` associa um texto a um número, iniciando de zero. Então, faz sentido sim. Essa associação depende da ordem dos elementos, sendo assim: **0 = Ebook, 1 = Impresso e 2 = Combo.**

Assim terminamos de cadastrar nossos produtos e seus respectivos preços. Experimente cadastrar mais alguns itens, porque no próximo capítulo iremos trabalhar com a listagem de produtos.

TIRAR DÚVIDA PRÓXIMA ATIVIDADE