02

Preparando Controller, DAO e JSP

Transcrição

Nos últimos capítulos fizemos o cadastro completo dos produtos da nossa aplicação e já integramos esse cadastro com o banco de dados. Fizemos o formulário (`form.jsp`) ter os campos necessários para o cadastro de um novo produto. Fizemos também os campos dos tipos de preços serem criados dinamicamente através de um loop (`forEach`).

Criamos na classe `Produto` um atributo que guarda uma lista de preços. Também criamos a classe `Preco` que guarda o `valor` e o `tipo` do preço, sendo que, para o `tipo` do preço, usamos um `enum` chamado `TipoPreco` para guardar as opções de preços que temos em nossa aplicação, sendo elas: **Ebook**, **Impresso** e **Combo**.

Agora faremos a listagem desses produtos.

Para aproveitarmos um pouco um código que já temos, vamos copiar todo o código que está no `form.jsp` . Criar um novo arquivo `JSP` chamado `lista.jsp` no mesmo diretório onde está o `form.jsp` e colar o código do `form.jsp` no `lista.jsp` .

Como esta será uma página que apenas lista nossos produtos, não precisamos do formulário, sendo assim, apague o código referente ao formulário de cadastro (`<form>`). O código restante deve algo assim:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Livros de Java, Android, iPhone, Ruby, PHP e muito mais - Casa do Código</title>
    </head>
    <body>

    </body>
</html>
```

Nesta página então, criaremos uma tabela onde serão listados os produtos usando os seguintes dados: Título, descrição e quantidade de páginas. Sendo assim, no corpo da página (dentro da tag `<body>`) crie a estrutura básica da tabela, algo parecido com o código:

```
[...]
<body>
    <table>
        <tr>
            <td>Título</td>
            <td>Descrição</td>
            <td>Páginas</td>
        </tr>
    </table>
```

```
</body>
[...]
```

As tags `table`, `tr`, `td` representam a tabela (`table`), as linhas (`tr`) e as colunas (`td`). Esta será nossa tabela, esta primeira `tr` será a cabeçalho. As próximas linhas, já devem ser preenchidas com os dados dos produtos.

No capítulo anterior, conhecemos o `forEach`. O `forEach` foi usado para percorrer uma lista e criar os campos do formulário. Faremos algo parecido aqui. Vamos percorrer uma **lista de produtos**. Usaremos a mesma estrutura do `forEach` aqui.

Como cada uma das próximas linhas da tabela será um produto. E em cada linha, vamos imprimir os dados do produto (`titulo`, `descricao` e `paginas`). Então podemos facilmente escrever algo como:

```
[...]
<table>
    <tr>
        <td>Título</td>
        <td>Descrição</td>
        <td>Páginas</td>
    </tr>

    <c:forEach items="${produtos}" var="produto">
        <tr>
            <td>${produto.titulo}</td>
            <td>${produto.descricao}</td>
            <td>${produto.paginas}</td>
        </tr>
    </c:forEach>
</table>
[...]
```

Vamos também adicionar um título nessa nossa página, pra não ficar só a tabela sem nenhuma descrição. Usaremos a tag `h1` antes da tabela com o título: **Lista de Produtos**. `<h1>Lista de Produtos</h1>`.

Já temos quase tudo pronto, precisamos somente fazer com que nosso `ProdutoDAO` acesse o banco de dados e crie essa lista de produtos que queremos exibir. E por último fazer o mapeamento no `ProdutosController` retornando a lista de produtos para a `view` `lista.jsp`.

Crie na classe `ProdutoDAO` o método `listar` que usuará o `EntityManager` para criar uma consulta no banco de dados e retornar uma lista de produtos. O código é parecido com:

```
public List<Produto> listar(){
    return manager.createQuery("select p from Produto p", Produto.class).getResultList();
}
```

O `getResultList` irá criar uma lista com os resultados da consulta ao banco de dados.

Nosso segundo passo é fazer com que nosso `ProdutosController` use o método `listar` do `ProdutoDAO` e retornar essa lista de produtos para a `view`.

```
public ModelAndView listar(){
    List<Produto> produtos = produtoDao.listar();
    ModelAndView modelAndView = new ModelAndView("/produtos/lista");
    modelAndView.addObject("produtos", produtos);
    return modelAndView;
}
```

Nenhuma surpresa até aqui não é? Estamos usando recursos que já aprendemos anteriormente. Acessando o banco de dados, criando um `ModelAndView` para anexar objetos que serão usados em nossa `view` e retornando a lista.

O `ModelAndView` é uma classe do **Spring** que faz um relacionamento de um modelo (`model`) com uma visualização (`view`) . Este além de poder disponibilizar um outro objeto qualquer para a `view` pode fazer outras operações, como redirecionamento de páginas, entre outras. Veremos mais sobre `ModelAndView` posteriormente.

Vamos fazer com que a lista de produtos fique na url `/produtos` e faz sentido, certo? Quando acessamos `/produtos` queremos ver uma lista de produtos. Agora adicionaremos esse mapeamento de rota. O metodo `listar` deve ficar assim:

```
@RequestMapping("/produtos")
public ModelAndView listar(){
    List<Produto> produtos = produtoDao.listar();
    ModelAndView modelAndView = new ModelAndView("/produtos/lista");
    modelAndView.addObject("produtos", produtos);
    return modelAndView;
}
```

Com tudo isso pronto, podemos iniciar o servidor e tentar acessar a lista de produtos em `localhost:8080/casadocodigo/produtos/` . Mas algo parece não funcionar bem, temos um erro no console. Veja a mensagem de erro:

```
Ambiguous mapping found. Cannot map 'ProdutosController'.
```

Isso acontece porque temos duas rotas em nosso `controller` apontando para a mesma url. Quando o acesso for feito, o **Spring** não vai saber qual método chamar do **controller**. Vamos resolver isso!

Podemos diferenciar as rotas simplemente mudando a url que o método mapeia. Mas vamos diferenciar as rotas de uma outra forma. Vamos diferenciar pelos métodos usados pelo **protocolo HTTP**.

Quando acessamos uma página, digitando uma url ou clicando em links, estamos fazendo um `GET` . Quando estamos clicando em nosso botão de cadastrar produtos por exemplo, geralmente estamos fazendo um `POST` . Se você verificar o formulário (`form.jsp`) verá o atributo `method` com o valor `POST` .

Para resolvermos o problema das todas duplicadas só precisaremos adicionar um novo parâmetro no `@RequestMapping` usando o `enum RequestMethod` do **Spring**, definindo assim qual método `HTTP` vai ser usado para chamar aquele método do `controller` .

Quando fizermos um `GET` para `/produtos` o **Spring** deve chamar o método `listar` do nosso `ProdutosController` . Quando fizermos um `POST` para `/produtos` ele deve chamar o `gravar` , enviando um produto para ser gravado no banco de dados.

Modificaremos então as anotações de `@RequestMapping` do método `gravar` e `listar` do nosso `ProdutosController`, fazendo essa diferenciação. Veja o código como fica:

```
@RequestMapping(value="/produtos", method=RequestMethod.POST)
    public String gravar(Produto produto){
        [...]
    }


@RequestMapping(value="/produtos", method=RequestMethod.GET)
public ModelAndView listar(){
    [...]
}
```

Tente iniciar o servidor novamente e acessar a página de produtos, tudo deve funcionar normalmente.