

Spring MVC 1: Criando aplicações web

03 Listando os produtos

17%

AULA 04

Listando os produtos

ATIVIDADES

Vídeo 1

14min

Vídeo 2

09min

Listando os produtos

Utilização de métodos de requisição d...

Criando método que lista os produtos

Passando a lista de produtos para a view



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
</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 `páginas`). 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 usará 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)
}
```

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 retorando 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 simplesmente 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 atribudo `method` com o valor `POST` .

Para resolvermos o problema das todas duplicadas só precisaremos adicionar um novo parametro 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.

Mudando Encoding do Spring.

Note que nossa lista de produtos aparece com alguns caracteres estranhos. Isso acontece por que o **servidor** não conhece o **encoding** da requisição, então ele troca os caracteres especiais e com acentos por outros caracteres.

Lista de Produtos

Título	Descrição	Páginas
TDD com JAVA	teste suas aplica	220

Há várias formas de resolver este problema, mas vamos usar uma das mais simples. Criando Filtros! Dessa forma, ao receber a requisição o **Spring** filtra a requisição com o **encoding** que vamos configurar. Em nossa classe `ServletSpringMVC` dentro do pacote `br.com.casadocodigo.loja.conf` , vamos criar mais um método de configuração do **Spring**.

Exite um método chamado `getServletFilters` usado pelo **Spring** que espera receber um array de filtros. Então vamos criar um `CharacterEncodingFilter` , definir o `encoding` deste filtro usando o valor `"UTF-8"` , adicionar este filtro ao `array` de filtros e o retornar esse `array` para o **Spring**. Use os imports `import javax.servlet.Filter` e `org.springframework.web.filter.CharacterEncodingFilter`.

```
public class ServletSpringMVC extends AbstractAnnotationConfigDispatcher
    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter encodingFilter = new CharacterEncodingF
        encodingFilter.setEncoding("UTF-8");
        return new Filter[] {encodingFilter};
    }
}
```

Agora quando cadastrarmos novos produtos, os caracteres estarão normais.

Lista de Produtos

Título	Descrição	Páginas
TDD no Mundo Real	já Aprenda a usar teste unitário no mundo real	220

Melhorando rotas no controller

Vamos fazer agora um pequeno ajuste em nosso `ProdutosController` para deixar o mapeamento das rotas mais simples. Note que em todos os métodos usamos a anotação `@RequestMapping` passando sempre `/produtos` .

Para que não precisemos ficar passando `/produtos` em todos os métodos do controller, vamos pôr essa anotação em nossa classe. Assim podemos remover o `/produtos` de todos os métodos e o **Spring** se encarrega de carregar os mapeamentos baseados no mapeamento da classe. Sendo assim nossa classe `ProdutosController` deve ficar parecida com o código abaixo:

```
@Controller
@RequestMapping("produtos")
public class ProdutosController {
    [...]

    @RequestMapping("/")
    public ModelAndView form(){
        [...]
    }

    @RequestMapping(method=RequestMethod.POST)
    public String gravar(Produto produto){
        [...]
    }

    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView listar(){
        [...]
    }
}
```

Dessa forma, se acessamos `/produtos` via `GET` , o método `listar` será chamado. Se o acesso for via `POST` o método `gravar` será chamado. E o `/produtos/form` continua chamando o método `form` . Bem mais simples, certo?

Teste novamente as páginas de listagem e de cadastro de produtos. Tudo deve estar funcionando normalmente. Cadastre novos produtos e verifique que os caracteres estranhos também não aparecem mais.