01

Iterando sobre coleções

Transcrição

O mundo java para web vive de JSPs. Não há muito como fugir delas, afinal, é a forma que temos para representar a camada visual das nossas aplicações.

O problema é que JSPs tendem a ser complicadas, afinal, ela contém código HTML, CSS, Javascript (que tendem a ser bem extensos), além de código Java que lida com lógica de apresentação (mostrar ou não mostrar uma tabela, uma informação, por exemplo).

Muitas JSPs fazem uso do que chamamos de **Scriptlets**. Scriptlets são código Java, espalhados dentro da JSP. Veja, por exemplo, a JSP abaixo.

Ela faz parte do projeto que será utilizado nesse curso. Você pode fazer o download dele em (http://s3.amazonaws.com/caelum-online-public/jstl/curso-jstl.zip)http://s3.amazonaws.com/caelum-online-public/jstl/curso-jstl.zip (http://s3.amazonaws.com/caelum-online-public/jstl/curso-jstl.zip). Ele é um projeto como outro qualquer; basta importá-lo no Eclipse.

Nesse momento, Não se preocupe em entender toda ela; apenas repare na quantidade de código Java que existe. Repare no "for", no "if", e etc:

```
<h1>Produtos</h1>
<div id="mensagem"></div>
<table width="100%">
    <tr>
        <td width="20%">Nome</td>
        <td>Preco</td>
        <td>Descricao</td>
        <td>Data de Inicio da Venda</td>
        <td>Usado?</td>
        <td width="20%">Remover?</td>
    </tr>

    <%
        List<Produto> produtoList = (List<Produto>) request.getAttribute("produtoList");
        for(Produto p : produtoList) {
    %>

        <tr id="produto<%= p.getId() %>">
            <td><%= p.getNome().toUpperCase() %></td>
            <td><%= p.getPreco() %></td>
            <td><%= p.getDescricao() %></td>
            <td><%= p.getDataInicioVenda().getTime() %></td>
            <% if(p.isUsado()) { %>
            <td>Sim</td>
            <% } else { %>
            <td>Não</td>
            <% } %>
            <td><a href="#" onclick="return removeProduto(<%= p.getId() %>)">Remover</a></td>
        </tr>
    <%
```

```
            }
        %>
    </table>
    <a href="/produtos/produto/formulario">Adicionar um produto</a>
```

O problema dessa JSP é que ela é difícil de manter. Veja que temos lugares onde abrimos chave (o `if`, por exemplo). Depois, para saber onde essa chave é fechada, é complicado.

Uma maneira de resolver isso seria se usássemos XML também para fazer a lógica Java. Afinal, já que HTML já é um XML, ficaria mais fácil de ler. Vamos começar, por exemplo, trocando aquele loop que temos por algo mais amigável. Veja o que temos nesse loop: uma lista chamada `produtoList`, que foi mandada para a JSP pela aplicação; o loop então passeia pela lista e cria a variável `p`, que contém cada um dos produtos. O loop, por sua vez, gera um monte de `tr`, preenchendo a tabela.

```
        <%
            List<Produto> produtoList = (List<Produto>) request.getAttribute("produtoList");
            for(Produto p : produtoList) {
        %>

                ... html aqui ....

                <% } %>
```

Veja o código abaixo, que faz a mesma coisa, mas dessa vez usando tags XML. Repare que a tag `forEach` recebe os mesmos parâmetros do loop anterior: a lista de produtos (no parâmetro items), e o nome da variável a ser criada para cada produto (no parâmetro var):

```
<c:forEach var="p" items="${produtoList}">
  ... html aqui ...
</c:forEach>
```

Existem várias tags como esse "forEach". Essa coleção de tags é chamada de **JSTL**. A ideia da JSTL é justamente facilitar a vida do programador na hora de escrever JSPs. A JSTL é hoje um padrão entre os desenvolvedores Java, e é muito utilizada. Existem diferentes "pacotes" na JSTL. Nesse momento, estamos estudando o pacote "core". Precisamos importá-lo na JSP:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Veja que definimos uma URI, que é o nome da biblioteca e um prefixo, que chamamos de "c". É por isso, aliás, que fizemos `c:forEach`. "c" indica que forEach está dentro do pacote importado. Um detalhe importante é que, apesar de parecer uma URL, a aplicação não está buscando isso na internet; é apenas o nome dela. A biblioteca da JSTL está dentro de um jar (o jstl.jar), que está importado no projeto.

Outro detalhe importante a reparar é o `${produtoList}`. Veja que usamos a notação com o caractere cifrão. Chamamos isso de EL (Expression Language). É assim que lidamos com as variáveis. Ali estamos dizendo que os ítens vem dessa variável.

Podemos fazer uso da EL para simplificar ainda mais coisa. Veja só que estamos "imprimindo" os valores do produto, usando:

```
<% = p.getNome() %>
```

Podemos usar a EL, e deixar isso ainda mais simples. Se fizermos `${p.nome}`, a EL irá, automaticamente, exibir o nome daquele objeto. Veja que não precisamos colocar o prefixo "get"; a EL já espera que o método siga essa convenção. Veja então no código abaixo que substituímos pelo uso da EL:

```
        <tr>
            <td width="20%">Nome</td>
            <td>Preco</td>
            <td>Descricao</td>
            <td>Data de Inicio da Venda</td>
        </tr>

<c:forEach var="p" items="${produtoList}">
            <tr id="produto${p.id}">
                <td>${p.nome}</td>
                <td>${p.preco}</td>
                <td>${p.descricao}</td>
                <td>${p.dataInicioVenda.time}</td>
            </tr>
</c:forEach>
```

Veja só como fica tudo mais simples. Simplificamos o código acima, tirando as duas últimas colunas, pois mais pra frente escreveremos aquele "if" também com JSTL. Misturar os dois não é tão fácil, e vamos evitar isso.

Veja só como nosso código final ficou muito mais fácil de ser entendido. É fácil de ver onde a tag "forEach" começa e termina; imprimir os valores do produto também ficou muito mais enxuto com o uso da EL.