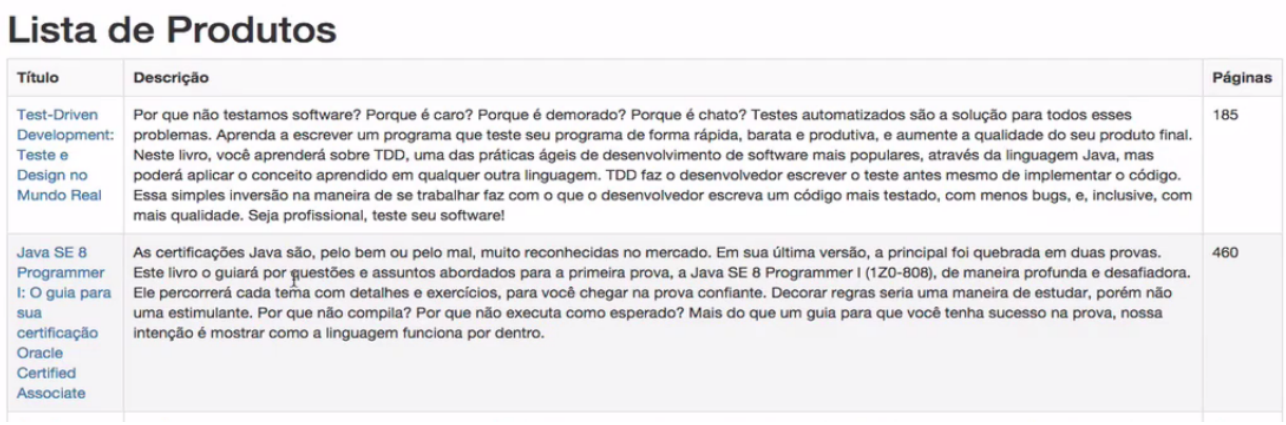
06

Open EntityManager In View

Transcrição

Uma funcionalidade interessante para os gestores da aplicação é que ao entrar na listagem dos produtos, seja possível verificar os preços de cada produto sem a necessidade de ir até a página de detalhes do produto. No painel administrativo, veja como nossa listagem se encontra no momento. Perceba que ela está sem as informações de preço.



Lista de Produtos

Título	Descrição	Páginas
Test-Driven Development: Teste e Design no Mundo Real	Por que não testamos software? Porque é caro? Porque é demorado? Porque é chato? Testes automatizados são a solução para todos esses problemas. Aprenda a escrever um programa que teste seu programa de forma rápida, barata e produtiva, e aumente a qualidade do seu produto final. Neste livro, você aprenderá sobre TDD, uma das práticas ágeis de desenvolvimento de software mais populares, através da linguagem Java, mas poderá aplicar o conceito aprendido em qualquer outra linguagem. TDD faz o desenvolvedor escrever o teste antes mesmo de implementar o código. Essa simples inversão na maneira de se trabalhar faz com o que o desenvolvedor escreva um código mais testado, com menos bugs, e, inclusive, com mais qualidade. Seja profissional, teste seu software!	185
Java SE 8 Programmer I: O guia para sua certificação Oracle Certified Associate	As certificações Java são, pelo bem ou pelo mal, muito reconhecidas no mercado. Em sua última versão, a principal foi quebrada em duas provas. Este livro o guiará por questões e assuntos abordados para a primeira prova, a Java SE 8 Programmer I (1Z0-808), de maneira profunda e desafiadora. Ele percorrerá cada tema com detalhes e exercícios, para você chegar na prova confiante. Decorar regras seria uma maneira de estudar, porém não uma estimulante. Por que não compila? Por que não executa como esperado? Mais do que um guia para que você tenha sucesso na prova, nossa intenção é mostrar como a linguagem funciona por dentro.	460

Para a exibição dos preços dos produtos, precisaremos acessar o atributo `precos` no objeto `produto` da `lista.jsp`. Na tabela da listagem de produtos será adicionada mais uma coluna.

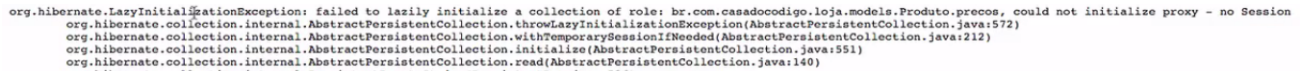
```
<table class="table table-bordered table-striped table-hover">
    <tr>
      <th>Título</th>
      <th>Descrição</th>

      <th>Preços</th>

      <th>Páginas</th>
    </tr>
    <c:forEach items="${produtos }" var="produto">
      <tr>
        <td>
          <a href="${s:mvcUrl('PC#detalhe').arg(0,produto.id).build() }">${produto.titulo }</a>
        </td>
        <td>${produto.descricao }</td>

        <td>${produto.precos }</td>

        <td>${produto.paginas }</td>
      </tr>
    </c:forEach>
</table>
```

Neste ponto, uma observação válida a ser feita é que o atributo `precos` no objeto `produto` trata-se de um vetor, um *array*. Ao tentar imprimir um objeto deste tipo de forma direta como estamos fazendo, o método `toString` de cada um dos objetos dentro do vetor será chamado.

É muito comum que o método `toString` seja sobrescrito em casos como este. Caso não, um código estranho de identificação do objeto será exibido na página. Por isso, definiremos o método `toString` na classe `Preco` que irá compor um texto com o tipo do preço e seu valor.

```
public String toString() {
    return this.tipo.name() + " - " + this.valor;
}
```

Agora que tudo está de acordo com o que queremos fazer (*exibir os preços na listagem dos produtos*) podemos testar a visualização da listagem de produtos. Mas ao tentarmos abrir a página, temos um erro:



O erro acontece por que ao tentar carregar os preços dos produtos, a conexão com o banco de dados já foi encerrada e com isto os preços não podem ser buscados. Os preços só estão sendo buscados em um segundo momento, quando estes realmente são necessários. Este é o padrão **Lazy Initialization** usado pelo *Hibernate* para carregamento de coleções.

Para resolvermos o problema, precisamos apenas alterar a consulta no banco de dados que carrega os produtos, para que também carregue junto com estes, seus preços. Isso é possível através do `join fetch`. No método `lista` da classe `ProdutoDAO` teremos:

```
public List<Produto> listar(){
    return manager.createQuery("select distinct(p) from Produto p join fetch p.precos").getResu
}
```

O uso do `distinct` é feito para que os produtos sejam distintos, pois ao realizar o `join` com a tabela de preços, o resultado dos produtos pode ser multiplicado para cada tipo de preço relacionado a um produto. Isso já resolve, mas vamos pensar um pouco mais no problema que estamos enfrentando.

O problema do *Lazy Initialization* acontece por que ao tentar carregar os preços dos produtos, a sessão com o banco de dados já tem sido encerrada. Ter que lembrar sempre de realizar o `join` não parece ser uma boa alternativa, é passivo de erro, podemos esquecer a qualquer momento deste detalhe. O que poderíamos fazer pra resolver isso de outra forma? Vejamos primeiro o processo.

O *Spring* carregará o `ProdutoDAO` e após isso iniciar uma sessão com o banco de dados. Carregará os produtos e depois encerrará a sessão. O passo seguinte é passar os dados para a `lista.jsp` e ai é que o problema acontece. Os preços não podem ser carregados depois da sesão finalizada. Este é o comportamento padrão.

Uma solução é manter a sessão com o banco de dados até que a visualização da página seja carregada. Assim o carregamento dos preços serão feitos sem nenhum problema. Um recurso que perpassa todo este processo são os *Filtros*. Há um filtro pronto especifico para a solução deste problema chamado `OpenEntityManagerInViewFilter`.

Adicionaremos este filtro na cadeia de filtros carregados no método `getServletFilters` da classe `ServletSpringMVC`.

```
@Override
protected Filter[] getServletFilters() {
    CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();
```

```
        encodingFilter.setEncoding("UTF-8");
        return new Filter[] {encodingFilter, new OpenEntityManagerInViewFilter()};
    }
```

Com este filtro configurado, podemos remover o `join` adicionado anteriormente e ver que a listagem continua funcionado. Mas cuidado, o uso do *Lazy Inicialization* que faz esse carregamento tardio dos recursos pode nos trazer problemas. Vejamos.

Ao carregar a listagem dos produtos, verifique o resultado da consulta executata pelo *Hibernate* no terminal. Conte quantas consultas foram executadas para uma listagem de apenas 3 produtos:

```
Tomcat v7.0 Server at localhost [Apache Tomcat] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Sep 2, 2015, 4:50:36 PM)
Hibernate: select distinct produto0_.id as id1_0_, produto0_.dataLancamento as dataLanc2_0_, produto0_.
Hibernate: select precos0_.Produto_id as Produto_1_0_0_, precos0_.tipo as tipo2_1_0_, precos0_.valor as
Hibernate: select precos0_.Produto_id as Produto_1_0_0_, precos0_.tipo as tipo2_1_0_, precos0_.valor as
Hibernate: select precos0_.Produto_id as Produto_1_0_0_, precos0_.tipo as tipo2_1_0_, precos0_.valor as
```

Perceba que uma consulta carrega os produtos e após isso, para cada produto é realizada uma nova consulta para o carregamento de seus preços. Este problema é muito comum nas aplicações web (também conhecido como **N+1** onde para cada registro, uma nova consulta precisa ser feita).

Este resultado é alcançado quando não usamos a estratégia de *join* nas tabelas do banco de dados. Mas e quando usamos o *join*? O que acontece? Vejamos:

```
Hibernate: select distinct produto0_.id as id1_0_, produto0_.dataLancamento as dataLanc2_0_, produto0_.
```

Apenas uma consulta é feita! O *join* já faz o carregamento dos preços ao buscas os produtos e por isso, não será mais necessário buscá-los quando a listagem for ser exibida.

Sempre que possível, use o *join* nestes casos. Esta estratégia evita a sobrecarga do acesso ao banco de dados. Em nossa aplicação, deixaremos as duas configurações do jeito que estão, com o *join* e o *OpenEntityManagerInViewFilter*. O primeiro para que o número de consultas seja o menor possível e o segundo para que o erro de carregamento de dados no banco não seja apresentado caso esqueçamos de realizar o *join* em outra parte da aplicação.