VRaptor 4

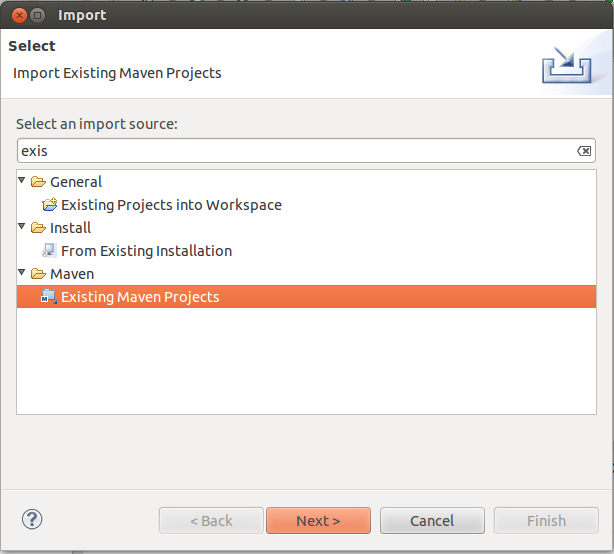
**Sobre o VRaptor 4**

O VRaptor 4 é a mais nova versão desse framework que te traz muita produtividade no desenvolvimento java web! Pensando em melhorar ainda mais essa área do desenvolvimento java, a Caelum criou e mantém o VRaptor. Nessa nova versão temos os [poderosos recursos do CDI](http://www.cdi-spec.org/) integrados ao nosso core, afinal essa especificação está cada vez mais ganhando espaço no mercado. Nesse curso, aprenderemos os principais recursos iniciais do VRaptor, como controle de fluxo de uma requisição web, validação de dados, interceptors, injeção de dependências, integração com plugins, entre outros recursos e boas práticas de desenvolvimento.

**Download e configuração do projeto**

Como ponto de partida, precisamos baixar a base do projeto que usaremos durante o curso. Esse projeto trabalhará no contexto de controle de estoque de produtos. Esse é um cenário simples, porém que nos permitirá usar muitos dos poderosos recursos dessa nova versão do framework! É só você [clicar aqui](https://github.com/alura-cursos/desenvolvimento-web-com-vraptor-4/archive/master.zip) para fazer o download do projeto.

Pronto! Agora que você já fez o download, basta extrair do zip e importar o projeto no eclipse. Pela facilidade de uso, vamos usar o maven para gerenciamento de nossas dependências. Se você não conhece o maven, você pode gostar do [nosso curso voltado para essa ferramenta](http://www.alura.com.br/cursos-online-java/maven). No Eclipse, basta você selecionar a opção File > Import e selecionar a opção Import (Existing Maven Project into workspace).



Repare que é um projeto Java Web normal. Temos por enquanto as classes Produto, que só possui um id, nome, valor e quantidade:

package br.com.caelum.vraptor.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity

public class Produto {

@GeneratedValue @Id

private Long id;

private String nome;

private Double valor;

private Integer quantidade;

// getters e setters

}

E também temos a classe ProdutoDao, que vai isolar nosso acesso ao banco de dados. Para nos aproximar de uma aplicação real do dia a dia, vamos usar o [JPA com hibernate](http://www.alura.com.br/cursos-online-java/jpa)que é muito usado pelo mercado para auxiliar na persistência dos dados. Não se preocupe, você não precisa ter nenhum conhecimento dessa ferramenta para prosseguir o curso. Repare que nossa classe Produto possui algumas anotações para indicar que é uma tabela do banco de dados e que seu id é auto gerado, e usaremos a classe EntityManager do JPA para fazer nosso CRUD (cadastro, leitura, alteração e remoção de produtos), ele nos dá uma forma mais orientada a objetos de fazer essas operações, assim trabalhamos em um ambiente mais familiar do que fazer as queries na mão e programar pensando de forma relacional.

public class ProdutoDao {

private final EntityManager em;

public ProdutoDao(EntityManager em) {

this.em = em;

}

public void adiciona(Produto produto) {

em.persist(produto);

}

public void remove(Produto produto) {

em.remove(produto);

}

public void busca(Produto produto) {

em.find(Produto.class, produto.getId());

}

@SuppressWarnings("unchecked")

public List<Produto> lista() {

return em.createQuery("select p from Produto p")

.getResultList();

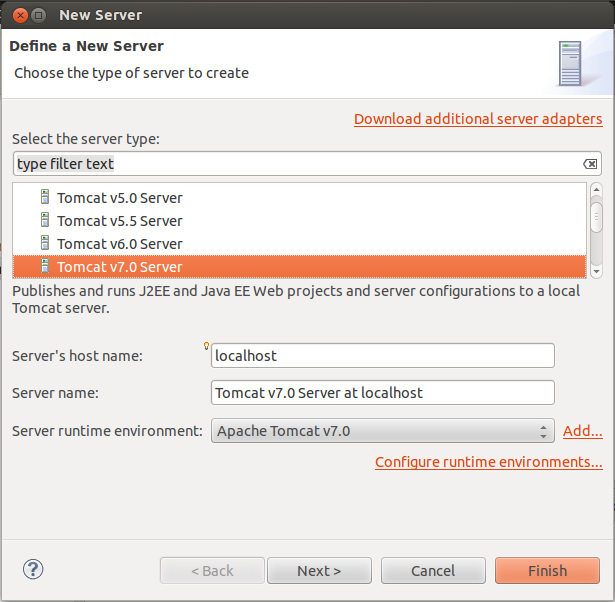
}

}

**Instalando o Tomcat 7**

Vamos usar o tomcat como container web para executar nossa app de controle de estoque. Se você estiver mais adaptado com outro container ou servidor de aplicação, você pode preferir usá-lo, porém é importante lembrar que o VRaptor 4 usa recursos do JavaEE 7, como o CDI 1.1, portanto seu servidor precisará suportar essa versão da especificação. Além do Tomcat e o Jetty (a partir da versão 8), alguns dos servidores já testados e suportados são o Glassfish 4 e WildFly 8.0.

Você pode baixar o Apache Tomcat 7 [por esse link](http://tomcat.apache.org/download-70.cgi). Depois de baixar, apenas extraia o zip para uma pasta que preferir em seu computador, e em seguida clique com o botão direito dentro da aba (view) Server. Deve aparecer uma caixa de dialogo com a opção New Server, vamos selecionar essa opção e em seguida escolhemos a opção Apache Tomcat 7. Agora precisamos informar o local em que a pasta do tomcat foi descompactada:



Pronto! Agora que finalizamos, já temos o Tomcat como opção de servidor da aba Server.

**Primeiro Controller do VRaptor 4**

A principio queremos que ao acessar a uri "/" sejamos direcionados para uma página html simples com uma mensagem dizendo que esse mapeamento deu certo. Para isso, vamos criar a classe ProdutoController no pacote br.com.caelum.vraptor.controller e adicionar a anotação @Controller. Ao adicionar essa anotação estamos ensinando ao VRaptor que a função dessa classe será receber as informações que chegam pela requisição http do usuário, ela quem vai determinar que ao acessar uma url alguma regra de negócio de nossa aplicação deverá ser executada.

@Controller

public class ProdutoController {

}

Vamos adicionar um método chamado inicio, que receberá a anotação @Path("/")indicando seu caminho.

@Controller

public class ProdutoController {

@Path("/")

public void inicio() {

}

}

Está quase tudo pronto! Agora só precisamos criar o arquivo inicio.jsp que possui um html simples de introdução ao projeto:

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Projeto de Produtos</title>

</head>

<body>

<h1>Primeira lógica com VRaptor 4!</h1>

</body>

</html>

Mas como ensinar ao VRaptor que ao acessar a uri "/" ele deve encaminhar para o arquivo inicio.jsp? A resposta é muito simples, basta adicionar esse arquivo na pasta WEB-INF/jsp/produto/. Por convenção o VRaptor espera que ao acessarmos o método inicio da classe ProdutoController, queremos utilizar a página chamada inicio(com o mesmo nome do método) que está dentro da pasta produto (que é o nome da classe, sem o sufixo controller).

Em outras palavras, o nome do método reflete no nome do arquivo jsp, e o nome do controller (sem o sufixo controller) reflete no nome da pasta em que esse arquivo está. Portanto neste caso, basta adicionar o arquivo no caminho /WEB-INF/jsp/produto/inicio.jsp.

Vamos testar para garantir que tudo está funcionando. Para executar esse nosso primeiro controller, adicionamos o projeto no tomcat clicando com o botão direito do mouse e selecionando a opção Add and Remove. Selecione o projeto vraptor-produtos e clique em Finish. Click no icone de start ou utilize o atalho CTRL+ALT+R para subir o servidor.

Pronto! Basta acessar o link <http://localhost:8080/vraptor-produtos/> para ver nossa primeira lógica!

**Criando uma listagem de produtos**

Agora que já conhecemos algumas convenções do VRaptor, vamos exercitar nosso conhecimento criando uma listagem simples de produtos. Para isso, vamos adicionar um método chamado lista na classe ProdutoController.

@Controller

public class ProdutoController {

@Path("/produto/lista")

public List<Produto> lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

return dao.lista();

}

}

Para simplificar o código, isolamos a regra de criação da classe EntityManager do JPA na classe JPAUtil:

public class JPAUtil {

public static EntityManager criaEntityManager() {

EntityManagerFactory factory = Persistence

.createEntityManagerFactory("default");

return factory.createEntityManager();

}

}

Complicado? Não se preocupe, o VRaptor ainda vai nos ajudar e muito em todo esse trabalho!

Repare que nosso método lista está retornando uma lista de produtos, mas como acessar essa lista em nossa JSP? Podemos fazer uso da Expression Language (EL) para recuperar o valor na JSP, algo como: ${nomeDaVariavelAqui}. Mas qual o nome da variável?

Por convenção, o nome da variável disponibilizada em nossa JSP será produtoList, ou seja, sempre o nome da classe com a primeira letra minúscula e o sufixo List. Já se o método retornasse um único Produto, o nome da variável disponibilizada na JSP seria apenas produto.

Sabendo disso, podemos agora criar a lista.jsp dentro da pasta WebContent/WEB-INF/jsp/produto.

Nesse arquivo vamos apenas criar uma tabela com o nome, valor e quantidade de produtos. Para preencher o corpo dessa tabela, vamos iterar por essa lista de produtos, em EL ${produtosList}.

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

<html>

<head>

<title>Lista de Produtos</title>

</head>

<body>

<table>

<thead>

<tr>

<th>Nome</th>

<th>Valor</th>

<th>Quantidade</th>

</tr>

</thead>

<tbody>

<c:forEach items="${produtoList}" var="produto">

<tr>

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

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

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

</tr>

</c:forEach>

</tbody>

</table>

</body>

</html>

Para estilizar essa página sem nos preocuparmos muito em ficar criando css, vamos usar o [twitter bootstrap](http://getbootstrap.com/). Basta adicionar seu import no inicio do seu html, e adicionar as classes dele de formatação da tabela. Para centralizar, também vamos criar uma divcom classe container que o bootstrap fará todo o trabalho.

<html>

<link rel="stylesheet" type="text/css" href="../bootstrap/css/bootstrap.css">

<body>

<div class="container">

<h1>Listagem de Produtos</h1>

<table class="table table-stripped table-hover table-bordered">

<!-- restante do código omitido -->

</table>

</body>

</html>

Comó já temos alguns produtos pré cadastrados, vamos reiniciar o tomcat e acessar o link <http://localhost:8080/vraptor-produtos/produto/lista> para ver o resultado!



Com o Eclipse aberto, selecione a opção Import -> Import (Existing Maven Project into workspace). Depois selecione o projeto, marque a opção Copy Projects into Workspace e confirme a importação no Eclipse.

Agora vamos configurar o Tomcat 7 no Eclipse. Após baixá-lo [aqui](http://tomcat.apache.org/download-70.cgi), descompacte-o em qualquer pasta.

No Eclipse, acesse a aba Servers. Selecione New -> Server e escolha a versão 7 do Tomcat. Em seguida, selecione onde você baixou o Tomcat, e clique em Next.

Agora basta vincular o novo projeto ao Tomcat. Clique no projeto "vraptor-produtos" e depois em Add. Finalize a operação.

Tudo configurado! Agora basta subirmos o Tomcat, clicando no servidor e escolhendo a opção START. Em seu console deverá aparecer a seguinte mensagem: "VRaptor {VERSAO} successfuly initialized". Tudo deu certo? Qual a versão que apareceu em seu console?

Crie a classe ProdutoController dentro do pacote br.com.caelum.vraptor.controller e anote-a com @Controller.

Agora, crie um método chamado sobre, e adicione o caminho (@Path) para que ele seja invocado quando o usuário acessar a URL /produto/sobre.

Para concluir, crie o arquivo sobre.jsp dentro da pasta WEB-INF/jsp/produto e adicione o texto: "Esse é o projeto VRaptor-Produtos, criado por SEU\_NOME".

Pronto! Agora reinicie o servidor e acesse a url localhost:8080/vraptor-produtos/produto/sobre.

Coloque aqui o código do seu controller e jsp:

O Controller deve ficar parecido com:

package br.com.caelum.vraptor.controller;

import br.com.caelum.vraptor.Controller;

import br.com.caelum.vraptor.Path;

@Controller

public class ProdutoController {

@Path("/produto/sobre")

public void sobre() {

}

}

E o jsp:

<html>

<head>

<meta charset=UTF-8">

<title>Projeto de Produtos</title>

</head>

<body>

<h1>Esse é o projeto VRaptor-Produtos,

criado por Rodrigo Turini!</h1>

</body>

</html>

Vamos agora criar a função de listagem de produtos da nossa aplicação, para isso adicione o método lista na classe ProdutoController. Adicione o @Path com o caminho: /produto/lista. Lembre-se que esse método precisa criar um ProdutoDao e para isso vamos precisar de um EntityManager do JPA. Use o JPAUtil.criaEntityManager() para criar esse objeto. Além disso, não se esqueça que esse método deve dar um return na lista de produtos, para que ela seja acessada pelo jsp.

Depois disso, basta criar o arquivo lista.jsp dentro da pasta WEB-INF/jsp/produto, e montar a tabela com os valores que o controller devolver. Para exibir os valores você precisará fazer um forEach, para isso você precisará fazer algo como:

<!-- não se esqueça desse import para usar o c:forEach! -->

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

<table>

<c:forEach var="produto" items="${produtoList}">

<tr>

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

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

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

</tr>

</c:forEach>

</table>

Quando concluir, reinicie o servidor e acesse a url localhost:8080/vraptor-produtos/produto/lista para ver o resultado! Compartilhe conosco o seu método do controller e a lista.jsp.

RESPONDA

### Opinião do instrutor

Método do Controller:

@Path("/produto/lista")

public List<Produto> lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao produtoDao = new ProdutoDao(em);

return produtoDao.lista();

}

arquivo lista.jsp:

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

<!-- inicio do html e head omitido -->

<body>

<h1>Listagem de produtos</h1>

<table>

<thead>

<tr>

<th>Nome</th>

<th>Valor</th>

<th>Quantidade</th>

</tr>

</thead>

<tbody>

<c:forEach var="produto" items="${produtoList}">

<tr>

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

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

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

</tr>

</c:forEach>

</tbody>

</table>

</body>

</html>

O twitter bootstrap já está baixado e adicionado em seu projeto, tudo que você precisa fazer é adicionar o import desse css e utilizar as suas classes para estilizar a tabela de listagem de produtos.

Algumas classes que você pode adicionar na tabela são:

<table class="table table-striped table-bordered table-hover">

<! -- continuação do código da tabela -->

Depois acesse novamente a listagem de produtos para ver o resultado. O que achou do bootstrap? Já tinha usado em algum projeto?

Qual o nome da variável colocada no request pelo VRaptor no caso a seguir:

public List<Produto> lista() {

// codigo do metodo omitido

}

E se o método retornar um único objeto, como:

public Produto exibe() {

// codigo do metodo omitido

}

No primeiro método, que retorna um List<Produto>, a variável no request se chamará produtoList. Para o segundo caso, o nome será produto. O VRaptor sempre pega o nome do tipo do objeto e coloca a primeira letra em minúscula. Produto vira produto, Cliente vira cliente, NotaFiscal vira notaFiscal. Como é uma lista concatena ainda no final o List, como foi o caso do produtoList.

# Código com muita responsabilidade

Quando um código faz muita coisa além de seu propósito, ou seja possui muita responsabilidade, ele se torna cada vez mais dificil de manter e testar. Se tornam partes problemáticas do nosso sistema.

Alguns exemplos muito conhecidos com a linguagem java são aqueles Servlets que além de realizar toda a lógica, ainda geravam o html que era devolvido como resposta ao usuário. Ou então os JSPs, que além de cuidar do HTML tinham pedaços de código java e até mesmo código SQL. Por exemplo, no lugar de usar a classe ProdutoControllercom o método lista que criamos, poderíamos ter feito tudo isso no arquivo lista.jsp, algo como:

<%@page import="br.com.caelum.vraptor.model.Produto"%>

<%@page import="java.util.List"%>

<%@page import="br.com.caelum.vraptor.util.JPAUtil"%>

<%@page import="javax.persistence.EntityManager"%>

<%@page import="br.com.caelum.vraptor.dao.ProdutoDao"%>

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

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Lista de produtos</title>

</head>

<body>

<table>

<thead>

<tr>

<th>Nome</th>

<th>Preço</th>

<th>Quantidade</th>

</tr>

</thead>

<tbody>

<%

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

List<Produto> produtoList = dao.lista();

for(Produto produto : produtoList) {

%>

<tr>

<td><%= produto.getNome() %></td>

<td><%= produto.getValor() %></td>

<td><%= produto.getQuantidade() %></td>

</tr>

<% } %>

</tbody>

</table>

</body>

</html>

Estranho? Isso porque é só um pedacinho de código. Se fosse alguma função mais complexa esse código estaria ainda mais carregado. Repare que nesse mesmo arquivo temos imports das classes java, código java, código html, e cada nova necessidade que aparecer em nossa listagem vai ficar nele também. Imagine se tivermos varios fors e ifs, como ficaria bagunçado o fechamento das indentações dessas lógicas sempre com o <% } %> misturado no html.

Para não sobrecarregar alguma das partes de nosso sistema, uma solução comum é separar suas responsabilidades.

# Como organizar meu código web?

Ao desenvolver essa primeira funcionalidade da nossa aplicação, a listagem de produtos, já foi possivel notar que nosso código tem uma certa distribuição. Vamos entender agora como ele foi e será organizado ao decorrer do curso, assim como as motivações para utilizar esse padrão de organização.

Quando um usuário executa alguma ação em nosso sistema, por exemplo abrir listagem de produtos, alguma regra de negócio de nosso código é executada e por fim o usuário tem como resposta as informações exibidas em sua tela.

Repare que para isso acontecer passamos por 3 fases, ou **camadas** como costumam ser chamadas. Uma delas é a nossa regra de negócio (listar todos os produtos, neste caso). Outra camada é a de resposta ao usuário, responsável por dispor as informações ao final da ação. Por fim, precisamos de alguém para intermediar essas duas camadas, saber qual a lógica deverá ser executada de acordo com a ação realizada pelo usuário.

# O padrão arquitetural MVC

Essas 3 camadas são a base de um conhecido padrão arquitetural, o MVC (Model-View-Controller). Ele propoe uma forma educada de lidarmos com a distribuição do nosso código, que é:

**Model** : Onde temos nossas regras de negócio, são classes que podem representar nossas entidades (como a classe Produto) ou que te ajudam a trabalhar com os dados da nossa aplicação (como a ProdutoDao).

**View** : Onde temos nossas interfaces de uso do sistema, em geral nossas páginas web. A lista.jsp é um exemplo de item da camada de View.

**Controller** : Onde temos os intermediários que conectam as outras duas outras camadas. Essa camada recebe informações da View e transforma essas informações em chamadas das nossas regras de negócio, da camada Model. Como é o caso do nosso ProdutoController.

# Outros frameworks que utilizam o padrão MVC

Existem diversos frameworks que assim como o VRaptor utilizam o padrão arquitetural Model-View-Controller. Alguns dos principais exemplos são o [Spring MVC](http://www.alura.com.br/cursos-online-java/spring-mvc), que também é escrito em java, e em outras linguagens temos o [Ruby on Rails](http://www.alura.com.br/cursos-online-ruby/ruby-on-rails) e [Asp.NET MVC](http://www.alura.com.br/cursos-online-net/aspnetmvc). Independente da linguagem ou framework que você utilizar, conhecer MVC é um passo fundamental para o desenvolvimento web.

Quais os motivos para não colocar código de lógica junto com o código HTML?

Misturar os dois códigos dificulta a compreensão tanto do designer quanto do desenvolvedor. Em casos extremos, a mistura de SQL dentro do HTML deixa o código ainda mais difícil de compreender, isto é, difícil de manter.

Isso dificulta também a reutilização de partes do código, que costumam ser copiadas e coladas entre arquivos distintos. Ao separar o Java do HTML, o código Java pode ser reutilizado mais facilmente.

No MVC, o VRaptor trabalha em qual camada?

Na camada de Controller, ele é quem controla as entradas e dispara as requisições internas para os controllers e suas views.

Quando uma requisição chega ao servidor, qual a ordem de execução comum?

Quando uma requisição chega, ela é prontamente atendida pelo VRaptor. O framework então decide qual action chamar. A action executa, e ao final, diz ao framework qual JSP exibir. O VRaptor por fim, pega a JSP, a processa, e envia para o usuário final, finalizando a requisição.

Para que nossa aplicação de controle de estoque de produtos funcione como esperamos precisamos adicionar mais alguns comportamentos, como por exemplo a adição de produtos, remoção, entre outras operações importantes pra este dominio.

Vamos adicionar esses comportamentos em nossa classe controladora de Produtos, a ProdutoController!

Por sinal, é interessante saber que é uma prática comum separar os comportamentos de nossas regras de negócio em controllers específicos para cada recurso que estamos manipulando. Por exemplo, em nosso ProdutoController eu adiciono todos os comportamentos relacionados ao meu modelo Produto. Mas e se eu for adicionar um formulário de cadastro de usuários, faria sentido adicionar esse comportamento na mesma classe ProdutoController? O código até funcionaria, mas para que fique melhor organizado a melhor opção seria criar uma classe UsuarioController que isola as operações com o modelo Usuario.

Lembrando que para que uma classe seja um Controller do VRaptor, tudo que você precisa fazer é adicionar a anotação @Controller. Por convenção, também adicionamos o sufixo Controller ao nome da classe.

# Criando um cadastro de produtos

Para criar a funcionalidade de cadastro de produtos, primeiro precisamos fornecer ao usuário uma página com o formulário que deverá ser preenchido com as informações necessárias para esse cadastro.

Então vamos começar criando a página formulario.jsp dentro da pasta /WEB-INF/jsp/produto/, com três inputs para o usuário adicionar o nome, valor e quantidade do produto que será criado:

<html>

<body>

<form action="">

Nome: <input type="text" />

Valor: <input type="text" />

Quantidade: <input type="text" />

<input type="submit" value="Adicionar" />

</form>

</body>

</html>

Para que o usuário possa acessar esse formulário, precisamos adicionar esse mapeamento na classe controladora de nossos produtos. Quero que ao acessar a URL /produto/formulario o VRaptor redirecione o pedido do usuário para esse formulario.jsp que acabamos de criar. Para isso vamos adicionar o método formulario() com a anotação @Path indicando o caminho.

@Path("/produto/formulario")

public void formulario(){

}

Apenas lembrando algumas convenções, é muito importante que esse método se chame formulario, pois assim o VRaptor irá procurar pelo arquivo formulario.jsp, e como já vimos, esse arquivo precisa estar dentro da pasta /WEB-INF/jsp/produto/.

Estamos quase lá, só precisamos agora criar uma ação para o nosso formulário, para que alguma lógica de nosso controller seja utilizada quando o usuário clicar em Adicionar. Vamos adicionar um novo método na classe ProdutoController, chamado adiciona:

@Path("/produto/adiciona")

public void adiciona() {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(??); // preciso de um produto aqui!!!

em.getTransaction().commit();

}

Repare que agora, além de criar um EntityManager, precisamos abrir uma transação e commitar ao final do processo. Operação como adição e remoção de produtos sempre precisam estar envolvidas em uma transação, para que só seja concluída caso o proceso todo tenha acontecido com sucesso. Logo veremos mais a respeito e melhorar esse processo de trabalhar com as transações do JPA.

Vamos adicionar essa nova ação no campo action do nosso formulário:

<html>

<body>

<form action="<c:url value='/produto/adiciona'/>">

Nome: <input type="text" />

Valor: <input type="text" />

Quantidade: <input type="text" />

<input type="submit" value="Adicionar" />

</form>

</body>

</html>

Repare que usamos o c:url na action de nosso formulário. Essa taglib tem como função buscar o caminho relativo e é uma prática comum utilizá-la em links e endereços.

# Recebendo parâmetros da Requisição

Mas como podemos receber essas informações do formulário em nosso método adiciona? Isso é muito simples com VRaptor! Poderíamos começar adicionando um **name** para cada campo do nosso formulário, e adicionar parâmetros com esses mesmos nomes em nosso método adiciona. Por exemplo, vamos adicionar os names como **nome**, **valor** e **quantidade**:

<html>

<body>

<form action="<c:url value='/produto/adiciona'/>">

Nome: <input type="text" name="nome" />

Valor: <input type="text" name="valor" />

Quantidade: <input type="text" name="quantidade" />

<input type="submit" value="Adicionar" />

</form>

</body>

</html>

Agora em nosso método adiciona, basta adicionar os parâmetros com os mesmos nomes que demos no atributo name dos campos:

@Path("/produto/adiciona")

public void adiciona(String nome, double valor, int quantidade) {

// restante do código omitido

}

Agora, tudo que precisamos fazer é transformar isso tudo em um Produto. Podemos criar esse objeto e setar os valores:

@Path("/produto/adiciona")

public void adiciona(String nome, double valor, int quantidade) {

Produto produto = new Produto();

produto.setNome(nome);

produto.setValor(valor);

produto.setQuantidade(quantidade);

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

}

Achou trabalhoso? Então podemos simplificar mais! Repare que o **nome**, **valor** e **quantidade** são representados pela classe Produto em nosso sistema, podemos apenas pedir ao VRaptor um Produto com as informações que passamos como parâmetro na requisição! Para isso basta adicionar o parâmetro Produto no método adiciona:

@Path("/produto/adiciona")

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

}

Agora só precisamos modificar o atributo **name** nos inputs do nosso formulário, para produto.nomeDoCampo:

<html>

<body>

<form action="<c:url value="/produto/adiciona"/>">

Nome: <input type="text" name="produto.nome"/>

Valor: <input type="text" name="produto.valor"/>

Quantidade: <input type="text" name="produto.quantidade"/>

<input type="submit" value="Adicionar" />

</form>

</body>

</html>

Essa é outra convenção do VRaptor. O nome do atributo que passamos no método adiciona é **produto**, portanto no name dos inputs HTML eu passo o valor produto.nomeDoCampo. Dessa forma, quando pedimos um atributo chamado produtocomo parâmetro ao VRaptor, ele procura na requisição os valores de texto que começam com produto.algumValor e depois tenta preencher esse valor usando o seu setter.

Por exemplo, quando eu uso o name produto.nome o VRaptor chama o método setNome(String nome) passando o texto que eu digitei no campo de texto do HTML como valor. Basta ir separando por pontos que o VRaptor sabe que deve setar esses valores, e caso não encontre o campo, apenas ignora e não preenche nada.

# Concluindo o cadastro de produtos

Ao invocar o método adiciona(Produto produto) do nosso controller, a sua lógica será executada e o VRaptor vai tentar redirecionar a requisição do usuário para a página adiciona.jsp! Portanto, para concluir nosso cadastro precisamos criar essa página:

<html>

<body>

Produto adicionado com sucesso!

</body>

</html>

Lembrando, para que tudo funcione como esperado esse arquivo precisa estar no diretório /WEB-INF/jsp/produto/.

Vamos testar essa nova funcionalidade! Após restartar o servidor, vamos abrir o formulário de produtos na url <http://localhost:8080/vraptor-produtos/produto/formulario> e adicionar um novo produto!

# Estilizando o formulário

Vamos adicionar as classes e estrutura do twiter bootstrap para deixar o nosso formulário com uma aparência melhor. Para isso, basta adicionar suas classes seletoras. No final, nosso jsp deve ficar assim:

<div class="container">

<h1>Adicionar Produto</h1>

<form action="<c:url value='/produto/adiciona'/>" >

Nome:

<input class="form-control" type="text" name="produto.nome" value="${produto.nome}"/>

Valor:

<input class="form-control" type="text" name="produto.valor" value="${produto.valor}"/>

Quantidade:

<input class="form-control" type="text" name="produto.quantidade" value="${produto.quantidade}"/>

<button type="submit" class="btn btn-primary">Adicionar</button>

</form>

</div>



Assim como no texto e vídeo, crie o formulário e ação de cadastro de produtos. Lembre-se de adicionar os métodos formulario (mapeado para /produto/formulario) e adiciona (mapeado para /produto/adiciona). O método formulario apenas redireciona para a view formulario.jsp, já o método adiciona precisa receber um Produto e criar um ProdutoDao para chamar seu método adiciona(...).

Como estamos usando o JPA, além de criar o EntityManager precisaremos iniciar uma transação antes de chamar o método do dao, e commitar essa transação logo no final, algo como:

public void adiciona(Produto produto){

// cria o EM e também o DAO

em.getTransaction().begin();

// chama o método adiciona do dao

em.getTransaction().commit();

}

Não esqueça de no final criar o aquivo adiciona.jsp para informar ao usuário que o produto foi adicionado com sucesso!

Depois de concluir, compartilhe o código do seu controller e jsps criadas.

Primeiro precisamos do método formulario no Controller:

@Path("/produto/formulario")

public void formulario() {

}

Repare que ele não faz nada, só devolve o formulario.jsp. Vamos criá-lo:

<html>

<body>

<h1>Adicionar Produto</h1>

<form action="<c:url value='/produto/adiciona'/>">

Nome:

<input type="text" name="produto.nome" value="${produto.nome}"/>

Valor:

<input type="text" name="produto.valor" value="${produto.valor}"/>

Quantidade:

<input type="text" name="produto.quantidade" value="${produto.quantidade}"/>

<button type="submit">Adicionar</button>

</form>

</body>

</html>

Agora vamos criar o método adiciona no Controller:

@Path("/produto/adiciona")

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao produtoDao = new ProdutoDao(em);

em.getTransaction().begin();

produtoDao.adiciona(produto);

em.getTransaction().commit();

}

E por fim, a página adiciona.jsp, na pasta WEB-INF/jsp/produto/:

<html>

<head>

<meta charset=UTF-8">

<title>Projeto de Produtos</title>

</head>

<body>

Produto adicionado com sucesso!

</body>

</html>

Adicione o bootstrap e nosso css base no formulário de adição de produtos. Para isso, você vai precisar colocar os imports:

<link rel="stylesheet" type="text/css" href="../bootstrap/css/bootstrap.css">

<link rel="stylesheet" type="text/css" href="../base.css">

Lembre-se de adicionar a class="form-control" nos inputs de texto, e a class="btn btn-primary" no button do seu form. Ao final, cole aqui seu código do formulario.jsp.

RESPONDA

### Opinião do instrutor

O seu código deverá ficar parecido com:

<html>

<link rel="stylesheet" type="text/css" href="../bootstrap/css/bootstrap.css">

<link rel="stylesheet" type="text/css" href="../base.css">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Projeto de Produtos</title>

</head>

<body>

<div class="container">

<h1>Adicionar Produto</h1>

<form action="<c:url value='/produto/adiciona'/>" method="post">

Nome:

<input class="form-control" type="text" name="produto.nome" value="${produto.nome}"/>

Valor:

<input class="form-control" type="text" name="produto.valor" value="${produto.valor}"/>

Quantidade:

<input class="form-control" type="text" name="produto.quantidade" value="${produto.quantidade}"/>

<button type="submit" class="btn btn-primary">Adicionar</button>

</form>

</div>

</body>

</html>

Um outro passo importante é adicionarmos a função de remover os produtos! Crie o método remove no seu ProdutoController, ele deve ser mapeado para a url /produto/remove e receber um Produto como parâmetro. Não se esqueça que para remover do banco de dados, vamos precisar do ProdutoDao, e assim como no método adiciona, precisaremos abrir uma transação!

@Path("/produto/remove")

public void remove(Produto produto){

// cria o EM e DAO

em.getTransaction().begin();

// remove o produto do banco

em.getTransaction().commit();

}

Depois disso, coloque um link na tabela de listagem de produtos, do lista.jsp. Para passar o id do produto pela request você pode passar esse id como argumento na URL, da seguinte forma:

<td>

<a href="<c:url value='/produto/remove?produto.id=${produto.id}'/>">Remover</a>

</td>

Crie também a página remove.jsp informando ao usuário que esse produto foi removido com sucesso!

Ao final, compartilhe conosco o método remove do seu controller e o conteúdo do seu arquivo remove.jsp.

Método remove do ProdutoController:

@Path("/produto/remove")

public void remove(Produto produto){

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

em.getTransaction().begin();

dao.remove(produto);

em.getTransaction().commit();

}

nova coluna com link na lista.jsp:

<td>

<a href="<c:url value='/produto/remove?produto.id=${produto.id}'/>">Remover</a>

</td>

arquivo remove.jsp:

<html>

<head>

<meta charset=UTF-8">

<title>Projeto de Produtos</title>

</head>

<body>

Produto removido com sucesso!

</body>

</html>

Crie alguns links para facilitar a navegação de nosso projeto. Por exemplo, no arquivo lista.jsp poderiamos ter um link para adicionar produto:

<a href="<c:url value='/produto/formulario'/>">

Adicionar mais produtos!

</a>

Adicione também um link para acessar a listagem de produtos pelas páginas adiciona.jsp e remove.jsp. Ao final, compartilhe conosco todos os links criados.

### do instrutor

na lista.jsp:

<a href="<c:url value='/produto/formulario'/>">

Adicionar mais produtos!

</a>

e nos arquivos adiciona.jsp e remove.jsp:

<a href="<c:url value='/produto/lista'/>">

Voltar para a listagem

</a>

Repare que até agora criamos as seguintes rotas em nosso sistema:

ProdutoController.lista() => /produto/lista

ProdutoController.formulario() => /produto/formulario

ProdutoController.adiciona() => /produto/adiciona

ProdutoController.remove() => /produto/remove

Acabamos sempre mapeando o método para: nomeDoController/nomeDoMétodo!

Claro, assim como nas pastas da jsp, o nome do Controller sempre fica sem o sufixo Controller, e com a primeira letra minuscula: ProdutoController vira produto!

O que o VRaptor poderia fazer pra melhorar esse processo?

Criar uma convenção de nomes para as rotas! Sempre que possível, o VRaptor aplica convenções para seus processos, a fim de te deixar ainda mais produtivo! Essa convenção de rotas já existe, vamos conhecê-la melhor no próximo capítulo.

# Utilizando os Verbos do HTTP

Até agora mapeamos todos os nosso métodos do ProdutoController utilizando a anotação @Path. Dessa forma ele atende as requisições feitas independente do verbo HTTP utilizado no form. Repare em como ficou a nossa URL após cadastrar um novo produto utilizando o método padrão, que é o GET:

<http://localhost:8080/vraptor-produtos/produto/adiciona?produto.nome=Camiseta+Hollister&produto.valor=250%2C00&produto.quantidade=10>.

Quando utilizamos o método GET, todas as informações são enviadas pela URL. Nem sempre queremos deixar tudo visível dessa forma. Além disso, existe um limite para o tamanho da URL e ainda corremos o risco de perder algumas informações se o conteúdo que formos adicionar for muito grande!

Podemos modificar nosso formulário para utilizar o método POST. Dessa forma, os dados não ficarão mais visiveis na URL e tudo é passado dentro do corpo do protocolo HTTP. A URL ficará: <http://localhost:8080/vraptor-produtos/produto/adiciona>, sem os dados visíveis!

É muito importante lembrar que isso não é uma medida de segurança, afinal os dados ainda podem ser consultados pelo navegador do usuário. O ganho dessa abordagem é que o corpo do POST é muito maior que a URL do GET, e evitamos deixar os dados sempre tão expostos.

Mas afinal, quando usar cada verbo? É muito comum utilizarmos o método **GET** para listagens ou visualização de algum de nossos recursos, sempre em operações que não modificam os valores do sistema.

O **POST** para adição de informações a um recuso existente ou adição de um novo recurso, como estamos fazendo agora com o Produto.

Além disso, existem outros verbos como o **PUT** e **DELETE**, que são utilizados para atualização e remoção, respectivamente.

Para utilizar o método POST na adição de produtos, tudo que precisamos fazer é adicionar o atributo method="post" em nosso form do formulario.jsp:

<form action="<c:url value='/produto/adiciona'/>" method="post">

Nome: <input type="text" name="produto.nome"/>

Valor: <input type="text" name="produto.valor"/>

Quantidade: <input type="text" name="produto.quantidade"/>

<input type="submit" value="Adicionar" />

</form>

Agora em nosso método adiciona do ProdutoController, vamos adicionar a anotação @Post:

@Path("/produto/adiciona") @Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

}

Da mesma forma, podemos utilizar as anotações dos outros verbos HTTP como GET (@Get), PUT (@Put) ou DELETE (@Delete)

Vamos reiniciar o servidor e testar um novo cadastro de produtos!

Agora que vimos que tudo está funcionando, é possivel simplificar ainda mais o nosso código! Podemos passar o caminho diretamente pra anotação do verbo que escolhermos, assim não precisamos utilizar a anotação @Path:

@Post("/produto/adiciona")

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

}

# Convenção de nomes das nossas URLs

Até agora nomeamos as nossas urls da seguinte forma:

ProdutoController.lista() => /produto/lista

ProdutoController.formulario() => /produto/formulario

ProdutoController.adiciona() => /produto/adiciona

Repare que a url sempre começa com o nome do Controller (sem o sufixo Controller), e logo em seguida usamos o mesmo nome do método. Essa é uma outra convenção do VRaptor, para simplificar ainda mais nosso código podemos anotar nossos métodos apenas com as anotações de verbo, como o @Get ou @Post, e por padrão o VRaptor vai nomear as url dessa mesma forma, considerando o nome do Controller e método. Vamos alterar nosso código:

@Controller

public class ProdutoController {

@Get

public List<Produto> lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

return dao.lista();

}

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

}

}

Em um sistema de login, queremos que um método seja acessado apenas pelo verbo POST. Como fazer isso? Quais partes do sistema precisaríamos mudar? E se algum método for usar o GET?

Basta mudar a anotação do método de @Path para @Post, ou qualquer outro verbo do HTTP que eu queira utilizar. Além disso, no caso do POST em nosso jsp será preciso adicionar o atributo method="POST" no form. Para o caso do GET essa segunda alteração não é necessária, afinal ele é o método padrão de envio do form.

Mude o método adiciona para que utilize o verbo POST. Lembre-se que será necessário adicionar o atributo method="post" no formulario de adição de produtos! Após concluir, compartilhe conosco o seu método adiciona e o inicio do form.

RESPONDA

### Opinião do instrutor

O método adiciona deve ficar parecido com:

@Post("/produto/adiciona")

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao produtoDao = new ProdutoDao(em);

em.getTransaction().begin();

produtoDao.adiciona(produto);

em.getTransaction().commit();

}

Inicio do form da view formulario.jsp:

<form action="<c:url value='/produto/adiciona'/>" method="post">

Modifique os métodos lista e formulario para que passem a atender requisições do tipo GET, e também o método remove para atender requisições do tipo DELETE. Ao final, cole aqui o seu ProdutoController com as alterações.

RESPONDA

### Opinião do instrutor

A classe ProdutoController deve ficar assim:

@Controller

public class ProdutoController {

@Get("/produto/lista")

public List<Produto> lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao produtoDao = new ProdutoDao(em);

return produtoDao.lista();

}

@Get("/produto/formulario")

public void formulario() {

}

@Post("/produto/adiciona")

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao produtoDao = new ProdutoDao(em);

em.getTransaction().begin();

produtoDao.adiciona(produto);

em.getTransaction().commit();

}

@Delete("/produto/remove")

public void remove(Produto produto){

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

em.getTransaction().begin();

dao.remove(produto);

em.getTransaction().commit();

}

}

Para utilizar ainda mais as convenções do VRaptor, remova o caminho das rotas dos métodos do seu ProdutoController! Por exemplo, no lugar de @Get("/produto/lista")deixe apenas @Get. Reinicie o servidor e teste a sua aplicação. O que aconteceu?

RESPONDA

### Opinião do instrutor

Tudo continua funcionando, afinal a convenção do VRaptor é nomear as rotas com nomeDoController/nomeDoMetodo, assim como já estavamos fazendo.

Nosso processo de desenvolvimento fica ainda mais simples e produtivo quando conhecemos e utilizamos as convenções do VRaptor!

# Fluxo padrão de uma Request no VRaptor

Como vimos, quando o usuário envia uma requisição (request) para uma URL de nosso sistema, cabe ao VRaptor identificar qual a lógica que deverá ser executada. Ou seja, quando eu acesso a URL "/produto/lista", por exemplo, ele precisa identificar que esse caminho está associado ao método lista() da classe ProdutoController. Em seguida, ele segue a convenção de nomes da view para identificar que deve renderizar a página jsp que possui o mesmo nome do método, e que está na pasta com o mesmo nome do controller (sem esse sufixo). Neste caso /WEB-INF/jsp/produto/lista.jsp.

Excelente, já vimos que esse é o fluxo padrão de uma request no VRaptor! Mas como podemos modificar esse fluxo? Por exemplo, após adicionar um produto, dessa vez eu não quero que ele siga sua convenção e envie para a página adiciona.jsp (que possui o mesmo nome do método). Pelo contrário, gostaria de ser **encaminhado** para a lógica de listagem. Dessa forma já posso ver o meu novo produto adicionado!

# Modificando o fluxo da requisição com o Result

Para conseguir redefinir esse fluxo padrão precisamos utilizar um objeto do VRaptor, o Result!

Mas como criamos esse complexo objeto? Por onde começar!? Nós não estamos interessados em **criar** esse objeto, mas sim em **usar**! Nesse caso, podemos pedir para o VRaptor nos fornecer esse objeto já pronto! Para fazer isso, basta adicionar um construtor em nosso ProdutoController e adicionar o Result como atributo da classe.

Além disso, é muito importante lembrar que nesta nova versão do VRaptor precisamos anotar esse nosso construtor com @Inject.

private final Result result;

@Inject

public ProdutoController(Result result) {

this.result = result;

}

Agora que temos o objeto result pronto para uso, podemos usá-lo para encaminhar a requisição do usuário para outra lógica de nosso sistema! Fazer isso é muito simples, basta adicionar a linha result.forwardTo(NomeDaClasse).nomeDoMetodo() no final do método que possui a lógica que queremos encaminhar, neste caso o adiciona():

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

result.forwardTo(this).lista();

}

Repare que utilizamos o **this** para representar o Controller em que está o método, afinal é um método da própria classe ProdutoController. Se quisermos direcionar para um controller diferente, basta mudar o this pela classe do Controller.

Agora, ao final da lógica de adicionar um produto seremos redirecionados para a view lista.jsp, e não mais adiciona.jsp.

Essa é apenas uma possibilidade de uso do Result, podemos fazer muito mais com esse recurso!

# Enviar informações para a View com o Result

Agora quando adicionamos um novo produto em nosso sistema, estamos sendo direcionados para nossa listagem, porém sem nenhum feedback de que tudo deu certo! É sempre uma boa prática mostrar uma mensagem ou dar uma indicação visual de que o processo (neste caso de adição de Produto) foi concluido com sucesso.

O Result também possui um método para pendurar informações na requisição, assim podemos recuperá-las em nossas views. Esse método é o include(String chave, Object valor). Repare que ele recebe dois parâmetros: o primeiro é a chave, ou nome da variável que será disponibilizada na view. O segundo parâmetro é o objeto que será retornado quando esse nome de variável for usado.

Vamos usar esse método para retornar uma mensagem informando que tudo deu certo no cadastro do produto. Para isso, basta adicionar o include() no método adiciona.

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

result.include("mensagem", "Produto adicionado com sucesso!");

result.forwardTo(this).lista();

}

Agora no arquivo lista.jsp podemos recuperar essa mensagem com uso da Expression Language (EL).

<c:if test="${not empty mensagem}">

<div class="alert alert-success">${mensagem}</div>

</c:if>

Repare que usamos um c:if pra apenas mostrar a div com a mensagem se ela não for vazia.

Pronto! Agora, ao adicionar um produto, além de sermos encaminhados para a lógica de listagem, temos uma mensagem indicando que tudo correu bem.

Com o Result podemos desponibilizar qualquer objeto na view, inclusive a nossa lista de produtos! Atualmente fazemos isso com o retorno do método:

@Get

public List<Produto> lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

return dao.lista();

}

Podemos fazer o mesmo utilizando o método include():

@Get

public void lista() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

result.include("produtoList", dao.lista());

}

Eu não preciso chamar essa variável de produtoList, claro.

# Forward x Redirect

Repare que no final do nosso método adiciona demos um forward para o método lista. Essa é uma situação bem comum, e um tanto perigosa. Quando terminamos de adicionar um produto, temos a lista.jsp carregada na tela, porém a URL continua com /produto/adiciona! Da mesma forma, todos os parâmetros que enviamos na requisição de adicionar um produto, continuam na requisição. Para testar, após adicionar um produto, pressione o F5 para recarregar a página. O que aconteceu?

Isso mesmo, o produto foi duplicado em nosso banco de dados! Encontramos um bugem nosso código.

A solução desse problema é bem simples, no lugar de fazer um forward podemos usar fazer um redirect! A classe Result também possui o método redirectTo, que é usado de forma parecida:

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

A grande diferença é que o forward acontece do lado do servidor, a lógica é encaminhada e no final a view lista.jsp é renderizada e devolvida ao usuário. Diferente do redirect, que acontece do lado do cliente. Nosso sistema devolve uma resposta ao navegador do usuário, que faz uma nova requisição para a outra lógica (sem manter os dados da primeira requisição).

Por sinal, é uma boa prática sempre usar o redirect após fazer uma requisição do tipo POST, como em nosso exemplo.

# Outros formatos de resposta

O Result possui outras formas de devolver um objeto para a view. Sua API simples permite serializarmos qualquer objeto ou lista de objetos de nosso controller em diferentes formatos, como JSON e XML, para que sejam disponibilizados como em um WebService.

Para esse trabalho utilizamos seu método use(), que pode receber a classe Results.xml() como parâmetro. Ficamos com o seguinte código: result.use(Results.xml()).

Agora precisamos indicar qual objeto (ou lista de objetos, que é nosso caso) deve ser transformada em um XML. Pra isso encadeamos de forma fluente o método from(), algo como:

result.use(Results.xml()).from(dao.lista());

Para finalizar chamamos o método serialize, que vai aplicar essa serialização. Então, para criar um método que devolve um XML de todos os nosso produtos cadastrados, fazemos algo como:

@Get

public void listaEmXml() {

EntityManager em = JPAUtil.criaEntityManager();

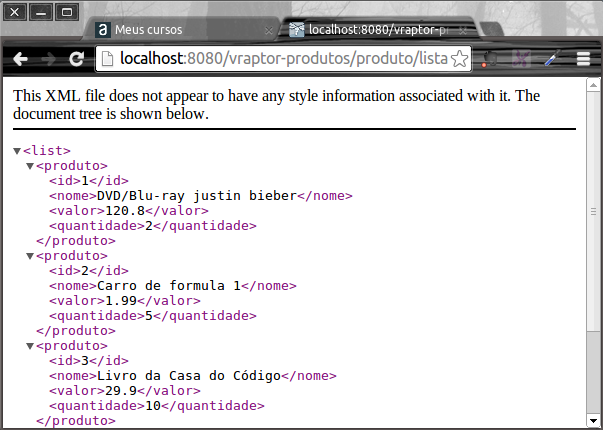
ProdutoDao dao = new ProdutoDao(em);

result.use(Results.xml()).from(dao.lista()).serialize();

}

Para testar, reinicie o tomcat e acesse a url <http://localhost:8080/vraptor-produtos/produto/listaEmXml>.

Ele deve te devolver um XML parecido com esse, com os produtos que você cadastrou.



Opcionalmente também podemos adicionar ou excluir informações nesse XML que será gerado. Fazemos isso com os métodos include e exclude, por exemplo:

@Get

public void listaEmXml() {

EntityManager em = JPAUtil.criaEntityManager();

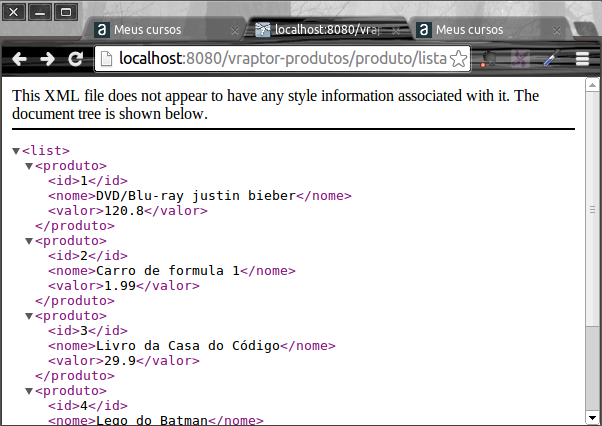
ProdutoDao dao = new ProdutoDao(em);

result.use(Results.xml()).from(dao.lista())

.exclude("quantidade").serialize();

}

Nessa caso, o XML devolvido não possui mais o campo quantidade:



Para fazer esse método nos devolver um JSON, só precisamos mudar o tipo de Resultsque passamos para o método use. Bastante simples, não acha? Vamos implementar esse novo método!

@Get

public void listaEmJson() {

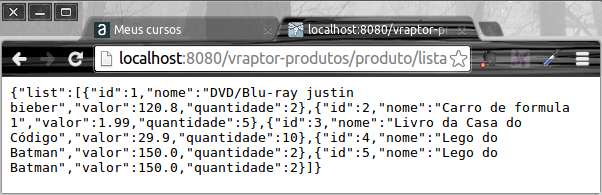
EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

result.use(Results.json()).from(dao.lista()).serialize();

}

Para testar, reinicie o tomcat e acesse a url <http://localhost:8080/vraptor-produtos/produto/listaEmJson>.



Como vimos, seguindo o fluxo padrão de uma request no VRaptor, quando o usuário acessa a URL /produto/adiciona, o método invocado será o adiciona da classe produtos. E qual será a view (jsp) renderizada? Dentro de qual pasta?

RESPONDA

### Opinião do instrutor

A não ser que o fluxo seja redirecionado pra algum outro lugar, a convenção é que a view será a adiciona.jsp dentro da pasta WEB-INF/jsp/produto.

Qual interface do VRaptor podemos utilizar para modificar o fluxo padrão de uma request? Essa é a unica responsabilidade dessa classe?

RESPONDA

### Opinião do instrutor

A interface é a Result. E além disso, ela possui outras responsabilidades, como incluir objetos na view e serialização de objetos.

Modifique o método adiciona para que redirecione para a lista de produtos ao final de sua execução. Lembre-se que pra isso você vai precisar da classe Result injetada, então será necessário pedir esse objeto no construtor:

private final Result result;

@Inject

public ProdutoController(Result result) {

this.result = result;

}

@Deprecated

ProdutoController() {

this(null); // para uso do CDI

}

Opcionalmente, faça o mesmo para o método remove.

Após concluir, cole aqui o seu método adiciona:

RESPONDA

### Opinião do instrutor

Basta adicionar o result.forwardTo(this).lista() no final do método:

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

result.forwardTo(this).lista();

}

Inclua uma mensagem de sucesso na lista.jsp informando que o produto foi adicionado após executar o método adiciona. Utilize o result.include(chave, valor)para mandar essa informação para a view, e a Expression Language ${chave} para recuperar a mensagem no seu jsp.

Depois que concluir, responda aqui com seu código do método adiciona e o html que você adicionou no lista.jsp.

RESPONDA

### Opinião do instrutor

No método adiciona, só precisamos adicionar a mensagem:

result.include("mensagem", "Produto adicionado com sucesso!");

Seu código completo ficou dessa forma:

@Post

public void adiciona(Produto produto) {

EntityManager em = JPAUtil.criaEntityManager();

em.getTransaction().begin();

ProdutoDao dao = new ProdutoDao(em);

dao.adiciona(produto);

em.getTransaction().commit();

result.include("mensagem", "Produto adicionado com sucesso!");

result.forwardTo(this).lista();

}

Na lista.jsp adicionamos o seguinte conteúdo:

<c:if test="${not empty mensagem}">

<div class="alert alert-success">${mensagem}</div>

</c:if>

Note que já utilizamos as classes do bootstrap pra estilizar a mensagem.

Modifique o método lista do ProdutoController, para que no lugar de retornar uma lista de produtos, apenas a inclua na view com o Result.include(...). Qual das duas formas você prefere?

RESPONDA

### Opinião do instrutor

Não existe uma resposta certa para essa pergunta, afinal parte da sua preferência. Mas a vantagem de utilizar o Result.include(...) é que além de poder personalizar o nome do objeto que será enviado para a view, você também pode enviar diversos objetos em um único método, diferente do return que é único.

Crie o método listaXML que retorna a lista de produtos em formato xml. Lembre-se que pra isso você pode usar o result.use(Results.xml()).from(...).

Também adicione um link na lista.jsp para que facilmente possamos acessar essa lista em formato xml. Após concluir, compartilhe o resultado aqui:

RESPONDA

### Opinião do instrutor

Código do método listaEmXml:

@Get

public void listaEmXml() {

EntityManager em = JPAUtil.criaEntityManager();

ProdutoDao dao = new ProdutoDao(em);

result.use(Results.xml()).from(dao.lista()).serialize();

}

Na lista.jsp, adicionamos o link para esse método:

<a href="<c:url value='/produto/listaEmXml'/>">

Acesse essa lista em XML!

</a>

# Injeção de Dependências com CDI

Esse conceito de pedir um objeto que precisamos no lugar de criá-lo, que utilizamos com a classe Result no capitulo anterior, é um dos principios da Injeção de Dependências! A idéia é que no lugar de você sair criando todos os objetos que a classe precisa (como estamos fazendo com o nosso DAO e EntityManager do JPA), esses objetos devem ser fornecidos a você.

Repare que isso diminui muito o acoplamento de nossas classes! A minha classe ProdutoController não precisa conhecer o processo de criação e dependencias da classe Result para poder usá-la. Mas repare que ainda estamos com um nível de acoplamento muito alto nessa classe, já que ela está com toda a responsabilidade de criar o ProdutoDao, tendo que criar um EntityManager para que isso funcione. Nosso código está muito amarrado, e pelo ponto de vista da orientação a objetos isso é um problema.

Não se preocupe, já vamos aplicar esse mesmo conceito de Injeção de Dependências (ou DI) na construção desses outros objetos. Mas primeiramente, vamos entender um pouco mais como isso é possivel, e conhecer algumas de suas regras.

Para injetar essa dependência utilizamos a anotação @Inject. Essa anotação vem do CDI, que é a especificação do java que diz respeito a injeção de dependências. O VRaptor 4 foi implementado tendo o CDI como container interno! Se você quiser conhecer um pouco mais de CDI, você pode se interessar por [esse post do blog da caelum](http://blog.caelum.com.br/use-cdi-no-seu-proximo-projeto-java/).

Para que o CDI consiga gerenciar as nossas classes, só precisamos adicionar o arquivo beans.xml dentro do diretório /META-INF/ de nosso projeto. Para facilitar, já deixamos esse arquivo adicionado para você.

Uma regra importante para utilizarmos essa injeção pelo construtor é que você precisa adicionar um construtor default em sua classe! Nosso controller vai ficar assim:

@Controller

public class ProdutoController {

private final Result result;

@Inject

public ProdutoController(Result result) {

this.result = result;

}

@Deprecated

ProdutoController() {

this(null); // apenas para uso do CDI!

}

// outros métodos da classe

}

Para evitar que alguém chame esse construtor por engano em testes e até mesmo para deixar explicito que é apenas para uso do CDI, é uma prática comum adicionar um @Deprecated nesse construtor vazio, e deixar sua visibilizade default.

Uma outra forma de fazer a injeção de dependências é diretamente pelo atributo da classe:

@Controller

public class ProdutoController {

@Inject private Result result;

// outros métodos da classe

}

É uma forma mais enxuta, porém consideramos uma boa prática a injeção pelo construtor. Isso facilita a testabilidade da classe. É claro, você ainda pode testar suas classes que utilizam injeção direta pelo atributo utilizando reflection, criando um setterpara os atributos ou qualquer outro tipo de estratégia para inserir um valor de teste naquele atributo, porém passar a informação pelo construtor é uma saida bem mais simples e elegante.

# Diminuindo acoplamento de nossas classes com DI

Vamos aplicar o mesmo conceito de DI para diminuir o acoplamento de nosso ProdutoController.

Primeiro, vamos apagar toda a logica de criação do ProdutoDao e EntityManager, esse código não pertence mais ao nosso Controller! Agora, vamos pedir para que o VRaptor (com CDI) nos entregue esse nosso DAO pronto! Basta adicionar o parâmetro ProdutoDao em nosso construtor, assim como fizemos com o Result. No final, nosso controller deve ficar assim:

@Controller

public class ProdutoController {

private final Result result;

private final ProdutoDao dao;

@Inject

public ProdutoController(Result result, ProdutoDao dao) {

this.result = result;

this.dao = dao;

}

@Deprecated

ProdutoController() {

this(null, null); // apenas para uso do CDI!

}

@Get("/")

public void index() {

System.out.println("primeiro projeto com VRaptor 4!");

}

@Get

public List<Produto> lista() {

return dao.lista();

}

@Get

public void formulario() {

}

@Post

public void adiciona(Produto produto) {

dao.adiciona(produto);

result.forwardTo(this).lista();

}

// outros métodos que eu tenha implementado

}

Muito mais simples, não acha? Vamos modificar agora o ProdutoDao, para que tudo funcione.

Como precisamos de uma transação antes de fazer todas as operações que precisam de transação em nosso DAO, por enquanto vamos abrir e commitar a transação em cada um dos métodos que for necessário (por enquanto o adiciona e remove). Essa não é a forma ideal de fazer isso, claro, mas logo vamos modificar isso!

public class ProdutoDao {

private final EntityManager em;

@Inject

public ProdutoDao(EntityManager em) {

this.em = em;

}

@Deprecated

ProdutoDao() {

this(null); // para uso do CDI

}

public void adiciona(Produto produto) {

em.getTransaction().begin();

em.persist(produto);

em.getTransaction().commit();

}

public void remove(Produto produto) {

em.getTransaction().begin();

em.remove(produto);

em.getTransaction().commit();

}

public void busca(Produto produto) {

em.find(Produto.class, produto.getId());

}

@SuppressWarnings("unchecked")

public List<Produto> lista() {

return em.createQuery("select p from Produto p").getResultList();

}

}

Mas repare que para criar um ProdutoDao, precisamos passar o EntityManager como parâmetro injetado e adicionar um construtor default. Só que agora, quem vai criar o ProdutoDao é o CDI! Como ele vai saber criar o EntityManager do JPA antes de criar o ProdutoDao? Nós sabemos que para criar esse objeto basta chamar a classe JPAUtil e usar seu método criaEntityManager(). O CDI não.

# Produção customizada de classes

Precisamos ensinar o CDI a criar um EntityManager! Esse processo é muito simples, basta eu criar uma classe comum, que será nossa Produtora, vamos chamá-la de EntityManagerProducer e criá-la no pacote br.com.caelum.vraptor.producers.

Agora, tudo que precisamos fazer é criar um método que retorna o EntityManagercriado. Para o CDI saber que precisa chamar esse método para conseguir criar um EntityManager, basta adicionar a anotação @Produces:

package br.com.caelum.vraptor.producers;

public class EntityManagerProducer {

@Produces @RequestScoped

public EntityManager criaEntityManager(){

return JPAUtil.criaEntityManager();

}

}

**Atenção: Não crie a classe produtora dentro do pacote br.com.caelum.vraptor.util, esse pacote é reservado e não é gerenciado pelo CDI.**

# Todas as nossas classes são managed beans

Repare que não precisamos usar nenhuma anotação especial para conseguir injetar nossa classe! Como isso foi possível? Como usamos o CDI para fazer a injeção de dependências, por padrão ele vai gerenciar todas as classes de nosso projeto! Ou seja, todas as nossas classes são passíveis de injeção de dependências (ou Managed Beans).

É claro, se eu não quiser essa opção, posso modificar meu arquivo beans.xml para que ele não gerencie nenhuma, ou gerencie apenas as classes com anotações do CDI.

# Tempo de vida dos componentes gerenciados

Mas quantas vezes o CDI vai criar essas dependências? Quanto tempo essas classes criadas vão existir em memória? Você quem decide ao definir um escopo para sua classe! Por exemplo, se quisermos criar um ProdutoDao a cada requisição, basta adicionar a anotação @RequestScoped na classe.

@RequestScoped

public class ProdutoDao {

// código omitido

}

Assim, se duas pessoas pedirem injetado o ProdutoDao em uma mesma requisição, a mesma instancia será passada. Outros escopos comuns são @SessionScoped e @ApplicationScoped.

Se você não definir nenhum escopo, por padrão sua classe vai ter o escopo @Dependent. Isso significa que a sua classe vai ser instanciada a mesma quantidade de vezes que a classe que a pediu injetada for.

O CDI é a especificação do JavaEE que diz respeito a injeção de dependências. Ao utilizar o VRaptor 4, já estamos utilizando CDI, afinal estamos integrados e utilizamos este recurso para nossas tarefas internas. E importante saber que o CDI é apenas a especificação, o documento, a implementação que utilizamos é o Weld 2. Você pode ler mais sobre CDI [nesse post](http://blog.caelum.com.br/use-cdi-no-seu-proximo-projeto-java/) ou em [seu site oficial](http://www.cdi-spec.org/).

Assim como no video, vamos abrir e fechar as transações nos métodos do nosso ProdutoDao. Lembre-se que só precisamos de transação nos métodos que modificam o produto.

Quando concluir, cole aqui a sua classe ProdutoDao.

RESPONDA

### Opinião do instrutor

public class ProdutoDao {

private final EntityManager em;

public ProdutoDao(EntityManager em) {

this.em = em;

}

public void adiciona(Produto produto) {

em.getTransaction().begin();

em.persist(produto);

em.getTransaction().commit();

}

public void remove(Produto produto) {

em.getTransaction().begin();

em.remove(busca(produto));

em.getTransaction().commit();

}

public Produto busca(Produto produto) {

return em.find(Produto.class, produto.getId());

}

@SuppressWarnings("unchecked")

public List<Produto> lista() {

return em.createQuery("select p from Produto p").getResultList();

}

}

O próximo passo é usar DI (injeção de dependências) em nosso ProdutoDao. Queremos pedir o EntityManager injetado. Mas como o CDI vai saber criar esse objeto? Crie um produtor para ele! Crie a nova classe dentro do pacote br.com.caelum.vraptor.producers:

Lembre-se que o método produtor precisa ter a anotação @Produces e retornar o que ele está produzindo, neste caso um EntityManager.

**Atenção: Não crie a classe produtora dentro do pacote br.com.caelum.vraptor.util, esse pacote é reservado e não é gerenciado pelo CDI.**

Compartilhe a sua classe produtora aqui:

RESPONDA

### Opinião do instrutor

Chamamos a classe de EntityManagerProducer:

package br.com.caelum.vraptor.producers;

public class EntityManagerProducer {

@Produces

public EntityManager producer() {

return JPAUtil.criaEntityManager();

}

}

**Importante**: A classe produtora **não** deve estar dentro de um pacote com o nome br.com.caelum.vraptor.util. Esse pacote é reservado e não será gerenciado pelo CDI.

Agora que temos um produtor para ele, podemos pedir o EntityManager injetado na classe ProdutoDao. Adicione o @Inject no construtor dessa classe, e não se esqueça que também é preciso adicionar o construtor default.

Além disso, como é o CDI quem vai criar o dao agora, também podemos pedi-lo injetado na classe ProdutoController! Adicione essa dependência no construtor de seu controller.

Compartilhe depois que concluir o código das suas classes ProdutoDao e ProdutoController.

RESPONDA

### Opinião do instrutor

A classe ProdutoDao ficou assim:

public class ProdutoDao {

private final EntityManager em;

@Inject

public ProdutoDao(EntityManager em) {

this.em = em;

}

@Deprecated

ProdutoDao() {

this(null); // CDI eyes only

}

public void adiciona(Produto produto) {

em.getTransaction().begin();

em.persist(produto);

em.getTransaction().commit();

}

// outros métodos

E a classe ProdutoController:

@Controller

public class ProdutoController {

private final Result result;

private final ProdutoDao produtoDao;

@Inject

public ProdutoController(Result result, ProdutoDao produtoDao) {

this.result = result;

this.produtoDao = produtoDao;

}

@Deprecated

ProdutoController() {

this(null, null);

}

// demais métodos

Ao marcar uma classe como @RequestScoped, quando o VRaptor/CDI cria esse componente?

RESPONDA

### Opinião do instrutor

Um componente anotado como @RequestScoped é instanciado a cada nova requisição. Ou seja, se ocorrerem 100 requisições, o VRaptor instanciará esse componente 100 vezes.

Devemos usar @SessionScoped quando deve existir apenas uma instância única daquele objeto por usuário. Um exemplo disso é a classe que guarda o identificador do usuário logado; esse objeto deve ser único durante todo o tempo em que um usuário estiver no sistema.

Vamos ensinar ao CDI que ele deve criar um novo EntityManager por request de nosso sistema! Para fazer isso, acesse sua classe produtora de EntityManager e adicione a anotação de escopo em cima do método anotado com @Produces. Poste seu método produtor quando concluir:

RESPONDA

### Opinião do instrutor

Agora sabemos que um EntityManager diferente é criado por request.

public class EntityManagerProducer {

@Produces @RequestScoped

public EntityManager producer() {

return JPAUtil.criaEntityManager();

}

}

Que classes podemos chamar de Managed Beans em nosso projeto? O que isso significa?

RESPONDA

### Opinião do instrutor

Como estamos usando o CDI para fazer a injeção de dependências, por padrão ele vai gerenciar todas as classes de nosso projeto! Ou seja, todas as nossas classes são passíveis de injeção de dependências, e podem ser chamadas de Managed Beans.

É claro, se não quisermo essa opção, podemos modificar o arquivo beans.xml para que ele não gerencie nenhuma, ou gerencie apenas as classes com anotações do CDI.

# Aplicando regras de validação em nosso formulário

Por enquanto nosso formulário não está validando nenhuma entrada de dados, por exemplo, tente adicionar um produto sem nome, quantidade ou com algum valor negativo. Todas essas operações funcionam, mas não deveriam. Faz parte de nosso trabalho validar esses valores e enviar uma mensagem ao usuário caso alguma de nossas regras não tenha sido atendida.

Mas como fazer a validação em nosso Controller? Isso é muito simples, poderiamos fazer um simples if, usar o Result para incluir uma mensagem na view e redirecionar o fluxo de volta para o formulário! Algo como:

@Post

public void adiciona(Produto produto) {

if (produto.getQuantidade() < 0) {

result.include("mensagem", "Não pode cadastrar

um produto com quantidade negativa!");

result.forwardTo(this).formulario();

return;

}

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

Mas repare que isso é muito trabalhoso! Se eu for validar mais campos, preciso repetir esse if varias vezes. Cada vez mais aumentamos nosso código e sua complexidade. O VRaptor facilita muito esse processo de validação com o uso de um objeto especializado, o Validator.

# Validando com VRaptor Validator

Como criar um Validator? Lembre-se do conceito de injeção de dependências, nós não queremos **criar**, e sim **usar** esse objeto. Vamos pedir para que o VRaptor nos forneça! Basta adicionar esse parâmetro no construtor:

@Controller

public class ProdutoController {

private final Result result;

private final ProdutoDao dao;

private final Validator validator;

@Inject

public ProdutoController(Result result, ProdutoDao dao, Validator validator) {

this.result = result;

this.dao = dao;

this.validator = validator;

}

@Deprecated

ProdutoController() {

this(null, null, null); // apenas para uso do CDI!

}

// restante do código omitido

}

Para aplicar a mesma validação que fizemos à pouco, mas agora com o Validator, podemos utilizar seu método check passando uma condição booleana e um objeto que representa uma mensagem de validação, o SimpleMessage. Caso seu resultado seja false, essa mensagem é adicionada numa lista de erros de validação para que posteriormente seja apresentada ao usuário na view. Algo como:

validator.check(produto.getQuantidade() > 0,

new SimpleMessage("erro", "Não pode cadastrar um produto com quantidade negativa!"));

Repare que invertemos a condição para > 0, afinal é isso que queremos garantir.

A classe SimpleMessage recebe dois argumentos, o primeiro é uma categoria e o segundo a mensagem em si. Essa categoria ajuda quando queremos agrupar mensagens para apresentar na view. Veremos mais sobre como usá-la a seguir.

# Redirecionando o fluxo em caso de erros de validação

Nosso próximo passo é configurar para qual página queremos redirecionar o cliente caso exista algum erro de validação. Para fazer isso, podemos usar o método onErrorUsePageOf():

validator.onErrorUsePageOf(this).formulario();

Nosso método adiciona agora pode validar os casos em que o usuário tentar cadastrar um produto com quantidade negativa:

@Post

public void adiciona(Produto produto) {

validator.check(produto.getQuantidade() > 0,

new SimpleMessage("erro", "Não pode cadastrar um produto com quantidade negativa!"));

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

Podemos agora criar novas validações adicionando novas chamadas ao método check.

# Isolando mensagens e internacionalização

Mas repare que a mensagem de validação foi inserida diretamente em nosso código, dessa forma nossos Controllers ficam cheios de mensagens e fica dificil internacionalizar seus valores. Essa não é uma boa prática.

Além da SimpleMessage existe uma outra opção de mensagem, a I18nMessage! No lugar de receber diretamente a mensagem, essa classe recebe uma **chave** para a verdadeira mensagem, que estará configurada em um arquivo de configuração de nosso projeto. Vamos mudar nosso código para utilizar essa implementação.

@Post

public void adiciona(Produto produto) {

validator.check(produto.getQuantidade() > 0,

new I18nMessage("erro", "produto.quantidade.negativa"));

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

Agora basta adicionar o arquivo de configuração dessas mensagens em nosso src, o arquivo deve se chamar messages.properties.

Dentro do arquivo, nós colocamos a **chave** e o **valor** (mensagem) correspondente ao erro:

produto.quantidade.negativa = Não pode cadastrar um produto com quantidade negativa!

Após aplicar essa configuração, vamos reiniciar o tomcat e tentar adicionar um produto com quantidade negativa. Repare, ao validar ele redirecionou para o formulario (antes de adicionar o produto no banco de dados) e nos mostrou a mensagem de validação que escrevemos no arquivo **messages.properties**.

# Exibindo os erros em nossa View

Repare que quando existe um erro de validação, nós estamos perdendo as informações que digitamos anteriomente. Se for um formulário com vários campos, com certeza vai incomodar bastante o usuário. Para manter esses dados em caso de erro de validação, basta adicionar o atributo value nos inputs com o atributo do produto na Expression Language:

Nome: <input type="text" name="produto.nome" value="${produto.nome}"/>

Valor: <input type="text" name="produto.valor" value="${produto.valor}"/>

Quantidade: <input type="text" name="produto.quantidade" value="${produto.quantidade}"/>

Apenas com essa mudança já podemos testar que ao preencher um produto com quantidade negativa e submeter o formulario, o VRaptor vai redirecionar para o formulário com os campos populados.

O último passo é mostrar para o usuário a mensagem de validação. O Validator do VRaptor automaticamente inclui na view uma variável chamada errors. Podemos recuperar essa variável com o uso de EL, e iterar sobre seus valores:

<c:forEach var="error" items="${errors}">

${error.category} - ${error.message}<br/>

</c:forEach>

Vamos adicionar esse trecho de código em nosso formulario.jsp, logo depois da tag form:

<!-- inicio do arquivo omitido -->

</form>

<c:if test="${not empty errors}">

<div class="alert alert-danger">

<c:forEach var="error" items="${errors}">

${error.category} - ${error.message}<br />

</c:forEach>

</div>

</c:if>

</div>

</body>

</html>

Pronto, agora basta restartar o servidor e tentar cadastrar um produto com quantidade negativa novamente. Repare que agora a mensagem de validação é apresentada para o usuário.



# Utilizando as anotações do Bean Validation

O VRaptor possui integração com o Bean Validation, dessa forma conseguimos utilizar suas anotações para validar os campos dos nossos modelos. Vamos fazer a mesma validação da quantidade, agora anotando esse atributo da classe Produto com @Min(value=0)

@Entity

public class Produto {

@GeneratedValue @Id

private Long id;

private String nome;

private Double valor;

@Min(value=0)

private Integer quantidade;

// continuação do código

}

Para que a validação aconteça, precisamos adicionar a anotação @Valid no parametro Produto do nosso método adiciona:

@Post

public void adiciona(@Valid Produto produto) {

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

Vamos testar! Basta restartar o servidor e tentar cadastrar um produto com valor negativo.

Excelente, ele está validando! Mas note que a mensagem foi:

adiciona.produto.quantidade - must be greater than or equal to 0

Podemos modificar essa mensagem adicionando o atributo message nas anotações de validação, neste caso na @Min:

@Min(value=0, message="Minha mensagem")

private Integer quantidade;

Mas para evitar inserir varias mensagens de validação em nosso código, vamos isolar no arquivo de internacionalização. Basta adicionar a chave(key) da sua mensagem entre {}:

@Min(value=0, message="{produto.quantidade.negativa}")

private Integer quantidade;

E agora adicionar a chave e valor no arquivo .properties. Uma diferença importante aqui, é que o Bean Validation exige que você chame o arquivo de ValidationMessages.properties, e não de messages.properties como estavamos fazendo. Vamos acessar o arquivo ValidationMessages.properties que já está dentro da pasta src/main/resources, e adicionar o conteúdo:

produto.quantidade.negativa = Não pode cadastrar um produto com quantidade negativa!

Pronto! Vamos testar mais uma vez agora, tudo o que precisamos fazer é restartar o servidor e tentar adicionar um novo produto com quantidade negativa.

Vamos adicionar mais algumas regras de validação! Outros casos bem comuns que costumamos validar no dia a dia é por exemplo o tamanho do texto do nome do produto. E também que o campo nome do produto não pode ser vazio. Podemos adicionar as anotações @Size e @NotEmpty para garantir isso:

@Entity

public class Produto {

@GeneratedValue @Id

private Long id;

@NotEmpty @Size(min=0)

private String nome;

@Min(value=0)

private Double valor;

@Min(value=0)

private Integer quantidade;

// continuação do código

Faça a validação do produto que recebemos no método adiciona antes de salvá-lo.

Receba o Validator no construtor e verifique que a quantidade é maior que zero utilizando o método check. Caso contrário, mostre a página do formularionovamente.

Use a classe SimpleMessage para criar a mensagem de validação.

RESPONDA

### Opinião do instrutor

O método adiciona deve ficar assim:

@Post

public void adiciona(Produto produto) {

validator.check(produto.getQuantidade() > 0,

new SimpleMessage("erro", "Não pode cadastrar um produto com quantidade negativa!"));

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

E no formulario.jsp:

<c:if test="${not empty errors}">

<div class="alert alert-danger">

<c:forEach var="error" items="${errors}">

${error.category} - ${error.message}<br />

</c:forEach>

</div>

</c:if>

Repare que já adicionamos as classes do bootstrap pra estilizar nossas mensagens de erro.

Repare que utilizando a classe SimpleMessage temos que adicionar o texto da mensagem em nosso código. Assim como no video, extraia essa mensagem pro arquivo /src/main/resources/messages.properties já presente em nosso projeto. Adicione nesse arquivo:

produto.quantidade.negativa = Não pode cadastrar um produto com quantidade negativa!

E agora substitua a classe SimpleMessage pela I18nMessage em seu código:

validator.check(produto.getQuantidade() > 0,

new I18nMessage("erro", "produto.quantidade.negativa"));

Reinicie o servidor e teste a validação no navegador. Deverá aparecer a mensagem em português.

Compartilhe ao final o código do seu método adiciona.

RESPONDA

### Opinião do instrutor

Código do método adiciona:

@Post

public void adiciona(Produto produto) {

validator.check(produto.getQuantidade() > 0,

new I18nMessage("erro", "produto.quantidade.negativa"));

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

Assim como no video, utilize o Bean Validation para fazer as validações de nosso Produto. Remova ou comente a linha de código que faz a validação (validator.check(...)) em nosso método adiciona, e agora adicione a anotação @Valid no parâmetro Produto.

lembre-se de anotar o atributo quantidade da classe Produto com o @Min(value=0).

Depois reinicie o servidor para garantir que está tudo funcionando.

Poste aqui o seu método adiciona.

RESPONDA

### Opinião do instrutor

Nosso método adiciona agora ficou assim:

@Post

public void adiciona(@Valid Produto produto) {

validator.onErrorUsePageOf(this).formulario();

dao.adiciona(produto);

result.include("mensagem", "Produto adicionado com sucesso!");

result.redirectTo(this).lista();

}

As mensagens de validação padrão do Bean Validation são em inglês! Como podemos exibir as mensagens em português na nossa view? Modifique o que for necessário para que isso funcione. Lembre-se, no caso do Bean Validation as mensagens devem ficar no arquivo ValidationMessages.properties.

RESPONDA

### Opinião do instrutor

No modelo Produto, adicionamos a mensagem:

@Min(value=0, message="{produto.quantidade.negativa}")

private Integer quantidade;

Depois disso, só foi preciso adicionar o arquivo ValidationMessages.properties com o seguinte conteúdo:

produto.quantidade.negativa = Não pode cadastrar um produto com quantidade negativa!

Adicione mais regras de validação em sua classe Produto. Um exemplo de validação é: O nome do produto nunca pode ser vazio ou nullo, você pode anotar esse atributo da classe com @NotEmpty e @NotNull

Quais outras validações poderiamos aplicar para esse modelo?

RESPONDA

### Opinião do instrutor

Existem diversas outras anotações de validação, como por exemplo a @Size(min=3)para garantir que o nome tem pelo menos 3 digitos.

Você acha que a validação feita no client-side, ou seja, no browser do usuário, através de Javascript eliminaria a necessidade da validação no server-side? Que problemas isso traria para a aplicação?

RESPONDA

### Opinião do instrutor

A validação server-side é importantíssima. Ela é que vai garantir que nenhum dado inválido seja inserido na sua base de dados. A validação client-side é amigável para o usuário, mas só ela não resolve, afinal o cliente pode desabilitar o javascript, por exemplo.

Nossa aplicação atualmente permite a **todos** adicionar, remover e listar produtos. Mas em uma aplicação real isso não deveria ser possível! É importante para nossa regra de negócio que apenas os usuários autorizados e que estejam logados em nosso sistema possam fazer qualquer consulta ou modificação em nosso estoque. Para que isso seja possível vamos começar esse trabalho fornecendo um meio de o usuário se logar em nossa aplicação.

**Criando um formulário de Login**

Vamos criar a classe LoginController no pacote br.com.caelum.vraptor.controllerque será a responsável por disponibilizar uma página de login e também terá a lógica de autenticação da nossa aplicação. A princípio, vamos apenas adicionar o método formulario:

@Controller

public class LoginController {

@Get

public void formulario() {

}

Precisamos agora criar o formulario.jsp para qual esse método será mapeado. Lembrando que esse formulario deve estar dentro da pasta WEB-INF/jsp/login/:

<html>

<link rel="stylesheet" type="text/css" href="../bootstrap/css/bootstrap.css">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Formulário de Login</title>

</head>

<body>

<div class="container">

<form class="form-signin" action="<c:url value='/login/autentica'/>" method="post">

<h2 class="form-signin-heading">Faça login para acessar o VRaptor-Produtos</h2>

<input type="text" class="form-control" name="usuario.nome" placeholder="Nome"/>

<input type="password" class="form-control" name="usuario.senha" placeholder="Senha"/>

<button class="btn btn-lg btn-primary btn-block" type="submit">Login</button>

</form>

</div>

</body>

</html>

Note que para facilitar o trabalho já estilizamos esse formulário utilizando o *twitter bootstrap*.

**Fazendo a autenticação de usuários**

Agora precisamos criar o método de autorização de acesso dos usuários. Esse método deve validar o nome e a senha enviado pelo formulário, se tudo der certo vamos encaminhar a requisição para a listagem de produtos, caso contrário validar os dados e reenchaminhar para a tela de login. Já deixamos criadas as classes Usuario e UsuarioDao em nosso projeto. Só precisamos agora criar o método autentica que executará nossa regra de autenticação. Ele deve receber um Usuario como parâmetro, e validar se esse usuário é cadastrado chamando o método existe da classe UsuarioDao:

@Post

public void autentica(Usuario usuario) {

if(!dao.existe(usuario)){

// se não existe, vamos redirecionar para página

}

// ok, podemos encaminhar para a listagem de produtos

}

Vamos utilizar a classe Validator do VRaptor para nos auxiliar a incluir a mensagem de validação e redirecionar para o formulário. E também precisaremos do Result para redirecionar o usuário para a listagem caso não exista nenhum erro de validação. Pedindo esses objetos injetados, assim como fizemos nos outros capítulos, o nosso código da classe LoginController deve ficar assim:

@Controller

public class LoginController {

private final UsuarioDao dao;

private final Result result;

private final Validator validator;

@Inject

public LoginController(UsuarioDao dao, Result result,

Validator validator) {

this.dao = dao;

this.result = result;

this.validator = validator;

}

@Deprecated

LoginController() {

this(null, null, null, null); // para uso do CDI

}

@Get

public void formulario() {

}

@Post

public void autentica(Usuario usuario) {

if(!dao.existe(usuario)){

validator.add(new I18nMessage("login", "login.invalido"));

validator.onErrorUsePageOf(this).formulario();

}

result.redirectTo(ProdutoController.class).lista();

}

}

Como estamos usando o I18nMessage, precisamos adicionar a chave login.invalidono arquivo messages.properties:

// outras mensagens omitidas

login.invalido = Nome ou senha inválido

E para exibir essa mensagem, vamos adicionar o EL errors após a tag form do formulario.jsp:

<!-- inicio do jsp omitido -->

</form>

<c:if test="${not empty errors}">

<div class="alert alert-danger">

<c:forEach var="error" items="${errors}">

${error.category} - ${error.message}<br />

</c:forEach>

</div>

</c:if>

</div>

Agora basta reiniciar o servidor e acessar a url do nosso formulário: <http://localhost:8080/vraptor-produtos/login/formulario>. Tente logar com um usuário que não existe para ver que a validação está funcionando, e logo em seguida com um usuário existente (pode ser o **vraptor**, que utilizamos no vídeo) para que você seja redirecionado para a lista de produtos.

**Adicionando o usuário logado na sessão**

Da forma que está, o usuário está sendo autenticado e logo em seguida perdemos essa informação. Como podemos fazer para que a aplicação se lembre que esse usuário já fez o login em nosso sistema? Precisamos guardar essa informação em algum objeto que viva por mais tempo que uma *request*, ou seja, um objeto com escopo maior.

Vamos criar a classe UsuarioLogado, e determinar o escopo dela como @SessionScoped. Assim, cada sessão de usuário terá uma classe dessa mantendo suas informações em memória.

@SessionScoped

@Named

public class UsuarioLogado {

private Usuario usuario;

public Usuario getUsuario() {

return usuario;

}

public void setUsuario(Usuario usuario) {

this.usuario = usuario;

}

}

Note que além da anotação de escopo, utilizamos a anotação @Named. Essa é uma anotação do CDI utilizada para disponibilizar objetos na jsp. Dessa forma, não precisamos fazer o result.include(...) dessa classe para que ela fique acessível em qualquer página do nosso sistema!

O último passo é settar o usuário na classe UsuarioLogado quando ele for autenticado:

@Post

public void autentica(Usuario usuario) {

if(!dao.existe(usuario)){

validator.add(new I18nMessage("login", "login.invalido"));

validator.onErrorUsePageOf(this).formulario();

}

usuarioLogado.setUsuario(usuario);

result.redirectTo(ProdutoController.class).lista();

}

Para garantir que o usuário está sendo mantido na sessão do navegador, vamos modificar a lista.jsp para exibir o nome desse usuário quando ele estiver logado. Basta modificar o texto da tag h1 da listagem, para:

<h1>Listagem de Produtos do ${usuarioLogado.usuario.nome}</h1>

Excelente, vamos tentar reiniciar o servidor agora. Note que recebemos a seguinte exception:

org.jboss.weld.exceptions.DeploymentException: WELD-000072:

Bean declaring a passivating scope must be passivation capable.

Bean: Managed Bean [class br.com.caelum.vraptor.model.UsuarioLogado]

Isso aconteceu pois o CDI determina que sempre que você anota uma classe com @SessionScoped, essa classe precisa ser serializável, ou seja, implementar Serializable. Vamos modificar nossa classe UsuarioLogado para atender essa condição:

@SessionScoped

@Named

public class UsuarioLogado implements Serializable {

// restante do código omitido

}

Agora sim podemos restartar o tomcat e acessar a página da lista de produtos:

* uma vez antes de logar no sistema (o nome do usuário não estará disponível)
* e mais uma vez logo após se autenticar (agora sim, o nome deve estar na lista)

Note que os dados do usuário logado estão sendo mantidos na sessão e disponibilizados na jsp. Mesmo quando fazemos varios *refresh*s na página!



Assim como no vídeo, comece criando um formulário de login, o formulario.jsp, e logo em seguida a classe LoginController com o método formulario mapeando esse jsp.

Após concluir, compartilher o código do seu controller e jsp conosco.

RESPONDA

### Opinião do instrutor

O LoginController com o método formulario:

@Controller

public class LoginController {

@Get

public void formulario() {

}

e o formulario.jsp:

<html>

<link rel="stylesheet" type="text/css" href="../bootstrap/css/bootstrap.css">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Formulário de Login</title>

</head>

<body>

<div class="container">

<form class="form-signin" action="<c:url value='/login/autentica'/>" method="post">

<h2 class="form-signin-heading">Faça login para acessar o VRaptor-Produtos</h2>

<input type="text" class="form-control" name="usuario.nome" placeholder="Nome"/>

<input type="password" class="form-control" name="usuario.senha" placeholder="Senha"/>

<button class="btn btn-lg btn-primary btn-block" type="submit">Login</button>

</form>

</div>

</body>

</html>

Repare que já utilizamos o bootstrap para estilizar nosso formulário.

Crie a classe UsuarioLogado para manter as informações do usuário na sessão. Lembre-se que essa classe precisa ter a anotação de escopo @SessionScoped e também o @Namedpara que o objeto dessa classe fique disponível na jsp.

Não se esqueça que como é uma classe com escopo de sessão, você precisa dizer que ela implementa Serializable.

Quando terminar compartilhe o código da sua classe UsuarioLogado.

RESPONDA

### Opinião do instrutor

@SessionScoped

@Named

public class UsuarioLogado implements Serializable{

private Usuario usuario;

public Usuario getUsuario() {

return usuario;

}

public void setUsuario(Usuario usuario) {

this.usuario = usuario;

}

}

Crie o método autentica na classe LoginController. Você pode utilizar o método existe da classe UsuarioDao para validar as informações de usuário, e também o Validator do VRaptor para adicionar uma mensagem caso o login digitado seja inválido.

Para testar, tente se autenticar com algum login inválido.

RESPONDA

### Opinião do instrutor

@Post

public void autentica(Usuario usuario) {

if(!dao.existe(usuario)){

validator.add(new I18nMessage("login", "login.invalido"));

validator.onErrorUsePageOf(this).formulario();

}

usuarioLogado.setUsuario(usuario);

result.redirectTo(ProdutoController.class).lista();

}

Na listagem de produtos (lista.jsp), adicione a EL ${usuarioLogado.usuario.nome}para exibir o nome do usuario que está logado em seu sistema.

O que acontece se nenhum usuário estiver logado e você acessar essa lista? O que deveria acontecer?

RESPONDA

### Opinião do instrutor

Vamos conseguir acessar a lista sem nenhum problema, apenas o nome do usuário que não será exibido, afinal não existe um usuário na sessão.

O ideal é que ninguém consiga acessar nenhuma informação interna de nosso sistema sem estar autorizado.

# Autorizando o acesso as lógicas com Interceptors

Estamos a um passo de completar essa funcionalidade de login. Repare que da forma que está, mesmo quando o usuário não está logado ele consegue acessar todas as funcionalidades do nosso sistema. Isso não pode continuar assim, apenas quem tem autorização de acesso deveria conseguir acessar as informações e fazer modificações em nosso sistema.

Mas como adicionar esse comportamento em todas as lógicas? Eu até poderia colocar um if em todos os meus métodos do controller, perguntando para a classe UsuarioLogado se ela possui um usuário diferente de null, algo como:

@Get

public void lista() {

if (usuarioLogado.getUsuario() != null) {

result.include("produtoList", dao.lista());

return;

}

result.redirectTo(LoginController.class).formulario();

}

Mas imagine o trabalho que isso daria! Além de que essa mudança vai poluir bastante o nosso código, e sempre precisaríamos lembrar de adicionar essa condição ao adicionarmos um novo método.

Essa é uma necessidade muito comum no desenvolvimento web, conseguir executar alguma lógica **durante**, **antes** ou **depois** do código de nossos controllers serem executados. Chamamos uma classe com essa habilidade de Interceptor.

Para criar um **interceptor** do VRaptor 4, tudo que precisamos fazer é criar uma classe comum e adicionar a anotação @Intercepts nessa classe. Vamos criar um interceptor chamado AutorizadorInterceptor dentro do pacote br.com.caelum.vraptor.interceptor. Esse interceptor vai fazer essa condição antes de executar o código do nosso Controller, e permitir a execução da lógica do nosso Controller apenas caso o usuário esteja logado.

@Intercepts

public class AutorizadorInterceptor {

}

Precisamos adicionar um método com o código que será executado **antes** e **depois** da lógica da nossa aplicação ser executada. Vamos chamá-lo de intercepta. Existem algumas regras para esse método funcionar, ou seja, interceptar uma requisição, e a primeira delas é que ele precisa estar anotado com @AroundCall.

Outra regra é que esse método precisa receber como parâmetro a classe SimpleInterceptorStack, cujo método next() vai indicar o ponto em que o código será executado:

@Intercepts

public class AutorizadorInterceptor {

@AroundCall

public void intercepta(SimpleInterceptorStack stack){

// antes de executar meu código

stack.next(); // executa o código

// depois de executar meu código

}

}

Excelente, agora só precisamos adicionar a condição para permitir ou não a execução desse método. Assim como nos Controllers podemos utilizar **Injeção de Dependências**em nossos Interceptors do VRaptor. Vamos então pedir o UsuarioLogado injetado, e fazer a condição redirecionando para o formulario sem executar nossas lógicas caso o usuário seja nulo.

@Intercepts

public class AutorizadorInterceptor {

private final Result result;

private final UsuarioLogado usuarioLogado;

@Inject

public AutorizadorInterceptor(Result result,

UsuarioLogado usuarioLogado) {

this.result = result;

this.usuarioLogado = usuarioLogado;

}

@Deprecated

AutorizadorInterceptor() {

this(null, null); // para uso do CDI

}

@AroundCall

public void intercept(SimpleInterceptorStack stack) {

if (usuarioLogado.getUsuario() == null) {

result.redirectTo(LoginController.class).formulario();

return;

}

stack.next();

}

}

Agora que tudo parece estar pronto, vamos tentar acessar a página de login da nossa aplicação: <http://localhost:8080/vraptor-produtos/login/formulario>. Repare que após tentar carregar a página por um tempo, o navegador nos respondeu com a seguinte mensagem: "This webpage has a redirect loop".

Isso aconteceu devido a condição que adicionamos em nosso interceptor! Repare:

if (usuarioLogado.getUsuario() == null) {

result.redirectTo(LoginController.class).formulario();

return;

}

Para acessar qualquer lógica (inclusive a do método formulario que é apenas redirecionar para sua jsp) a condição usuarioLogado.getUsuario() == null vai ser sempre **true**. Afinal, sem abrir o formulário e enviar os dados para o método autenticao objeto usuarioLogado nunca vai ter um usuário válido.

# Determinando quais métodos serão interceptados

Precisamos ensinar ao interceptor do VRaptor que esses métodos não devem ser interceptados! Para fazer isso, basta adicionar um método anotado com @Accepts em nosso interceptor. Esse método deve retornar um **boolean** que é a condição que a classe precisa atender para ser interceptada:

@Accepts

public boolean accepts() {

return // alguma condição aqui

}

Um caso de uso muito comum do accepts é quando marcamos um método de nosso projeto com alguma anotação nossa, assim dentro do accepts só precisamos validar se o método que está sendo interceptado possui essa anotação, assim definir se vamos ou não interceptar aquele método. Por exemplo, vamos criar a anotação @Public dentro do pacote br.com.caelum.vraptor.annotation:

@Documented

@Retention(value=RetentionPolicy.RUNTIME)

@Target(value=ElementType.METHOD)

public @interface Public {

}

Agora, no método formulario e autentica da classe LoginController adicionamos essa nova annotation:

@Public

@Get

public void formulario() {

}

@Public

@Post

public void autentica(Usuario usuario) {

// código omitido

}

E em nosso método accepts, do AutorizadorInterceptor dizemos que só deve aceitar (interceptar) se o método possuir essa anotação que criamos, a @Public. Mas como saber qual o método que estamos interceptando? Isso é bem simples, o VRaptor possui uma objeto em cada requisição que guarda essa informação, o ControllerMethod. Vamos pedir essa classe injetada em nosso Interceptor:

@Intercepts

public class AutorizadorInterceptor {

private final Result result;

private final UsuarioLogado usuarioLogado;

private final ControllerMethod controllerMethod;

@Inject

public AutorizadorInterceptor(Result result,

UsuarioLogado usuarioLogado, ControllerMethod controllerMethod) {

this.result = result;

this.usuarioLogado = usuarioLogado;

this.controllerMethod = controllerMethod;

}

// restante do código omitido

}

Agora só precisamos adicionar a condição do método accepts. Ele só deve interceptar se o método não contiver a anotação @Public, portanto:

@Accepts

public boolean accepts() {

return !controllerMethod.containsAnnotation(Public.class);

}

Para testar, vamos restartar o servidor e acessar a listagem de produtos: <http://localhost:8080/vraptor-produtos/produto/lista>. Repare que agora, como não estamos logados, fomos redirecionados para a página de login com sucesso!

Vimos que para o caso do controle de acesso, do nosso AutorizadorInterceptor, foi muito conveniente utilizar o método anotado com @AroundCall, pois assim recebemos o SimpleInterceptorStack como parâmetro e assim é possível escolher se vamos ou não chamar o método next que permite a continuação do nosso código e assim a execução da lógica dos nossos controllers.

# Interceptando antes ou depois da execução

Mas note que existem algumas situações em que não queremos interceptar durante a execução de nosso lógica, e sim apenas antes, ou até mesmo apenas depois que tudo aconteceu. E nem sempre queremos condicionar se vamos ou não executar a lógica. Isso pode ser feito com o @AroundCall, claro. Eu poderia simplesmente inserir o código que quero executar antes da chamada do método next, ou apenas depois.

Mas para simplificar ainda mais esse trabalho, o VRaptor conta com mais dois tipos de métodos nesse novo modelo de interceptor, o @AfterCall e o @BeforeCall. Vejamos um exemplo:

@Intercepts

public class AlgumInterceptor {

@BeforeCall

public void before() {

// código a ser executado antes da lógica

}

@AfterCall

public void after() {

// código a ser executado depois da lógica

}

}

Repare que para estes casos nós não precisamos injetar o SimpleInterceptorStack. A chamada do método next é implicita!

Vamos agora criar um Interceptor que nos mostra um **log** simples, com o nome do método que está sendo executado.

Bem, criar o interceptor nós já vimos como fazer, assim como um método @BeforeCall. Vamos então começar criando a classe LogInterceptor no pacote br.com.caelum.vraptor.interceptor e adicionar esse método:

@Intercepts

public class LogInterceptor {

@BeforeCall

public void before() {

// como saber qual método está sendo executado?

}

}

Assim como fizemos com o AutorizadorInterceptor, podemos pedir injetado o ControllerMethod para obter o nome do método que será executado. Vamos adicionar o contrutor e atributo no LogInterceptor:

private final ControllerMethod controllerMethod;

@Inject

public LogInterceptor(ControllerMethod controllerMethod) {

this.controllerMethod = controllerMethod;

}

Agora só precisamos printar o nome do método que está sendo executado! Algo como:

@Intercepts

public class LogInterceptor {

private final ControllerMethod controllerMethod;

@Inject

public LogInterceptor(ControllerMethod controllerMethod) {

this.controllerMethod = controllerMethod;

}

public LogInterceptor() {

this(null); // para uso do CDI

}

@BeforeCall

public void before() {

String nomeDoMetodo = controllerMethod.getMethod().getName();

System.out.println("Executando o método: " + nomeDoMetodo);

}

}

Fazendo o mesmo teste que fizemos anteriormente, restartando o tomcat e acessando nossa listagem de produtos, repare que no console temos a seguinte saída:

Executando o método: lista

# Interceptando só os métodos anotados

Não queremos mostrar o log com o nome de todos os métodos executados, pois isso vai deixar nosso console cheio de mensagens a todo momento. Em quero controlar quais métodos serão logados. Já vimos como fazer isso, podemos criar uma anotação para marcar esses métodos!

Vamos criar a anotação @Log dentro do pacote br.com.caelum.vraptor.annotation:

@Documented

@Retention(value=RetentionPolicy.RUNTIME)

@Target(value=ElementType.METHOD)

public @interface Log {

}

Agora, no método formulario do nosso ProdutoController adicionamos essa nova annotation:

@Log

@Get

public void formulario() {

}

E em nosso método accepts, do LogInterceptor dizemos que só deve aceitar (interceptar) se o método possuir essa anotação que criamos, a @Log:

@Accepts

public boolean accepts() {

return method.containsAnnotation(Log.class);

}

Para testar, basta restartar o servidor e acessar a página de lista de produtos: <http://localhost:8080/vraptor-produtos/produto/lista>. Repare que nenhuma informação foi logada, afinal o método lista não possui a anotação @Log.

Vamos acessar agora o nosso formulário: <http://localhost:8080/vraptor-produtos/produto/formulario>. Veja que a saída do console foi:

Executando o método: formulario

# Trabalhando com os accepts customizados

Para os casos mais comuns (como o que fizemos agora), o VRaptor nos provê alguns accepts customizados! Por exemplo, para ter o mesmo efeito que nosso método accepts do LogInterceptor, poderíamos ter anotado essa classe com @AcceptsWithAnnotations, dessa forma:

@AcceptsWithAnnotations(Log.class)

@Intercepts

public class LogInterceptor {

// continuação do código omitida

Vamos fazer um teste, basta **apagar** o nosso método accepts e manter apenas esse custom accepts.

O outro accept customizado é o @AcceptsWithPackage, que como o proprio nome indica, só intercepta as classes de um determinado pacote.

# Controlando a transação do JPA com interceptor

Agora que conhecemos o essencial de Interceptors, podemos usá-lo para melhorar um ponto de nosso código. Repare que nos métodos adiciona e remove da classe ProdutoDao foi preciso iniciar uma **transação** antes de executar a operação, e logo em seguida commitar essa transação, para que tudo seja realmente aplicado.

É essencial trabalhar com transações quando estamos trabalhando com banco de dados, afinal precisamos garantir a integridade de nossa base de dados, só modificar um valor caso tudo realmente de certo em nosso código.

É bastante trabalhoso fazer esse controlle de transação manualmente, sempre que vamos aplicar uma operação que modifique algum dado no banco precisamos **lembrar**de abrir e commitar a transação, além de que o código de todos os nossos DAOs ficam bastante acoplados com a forma de controlar transações do framework que estivermos usando.

Podemos automatizar esse processo criando um Interceptor que inicie uma transação quando uma requisição acontecer, e termine(commit) essa transação antes que a resposta seja enviada ao cliente! Vamos chamá-lo de ControleDeTransacaoInterceptor:

@Intercepts

public class ControleDeTransacaoInterceptor {

private final EntityManager em;

@Inject

public ControleDeTransacaoInterceptor(EntityManager em) {

this.em = em;

}

@Deprecated

ControleDeTransacaoInterceptor() {

this(null); // para uso do CDI

}

@AroundCall

public void intercept(SimpleInterceptorStack stack) {

em.getTransaction().begin();

stack.next();

em.getTransaction().commit();

}

}

Agora podemos **apagar as linhas** de transação do nosso ProdutoDao, desamarrando ele dessa responsabilidade.

Bem simples, não acha? Essa é uma prática muito comum do dia a dia de quem trabalha com JPA.

Um ponto importante é que mesmo que não use estamos abrindo transação em todas as requests! Poderíamos evitar isso criando uma anotação pra marcar quais métodos queremos que possuam uma transação, assim como fizemos com o @Log. Mas o custo de abrir uma transação e não utilizar não é grande, por isso acaba virando uma prática comum abrir essa transação sempre e não ter a obrigação de anotar todos os métodos que precisar dela, além de que evitamos o risco de esquecer dessa anotação em um método que precisaria da transação.

# Ordenando a execução dos interceptors, after e before

Repare que em nenhum momento configuramos a ordem que queremos executar nossos interceptor. Em alguns casos isso é extremamente importante! Por exemplo, o interceptor AutorizadorInterceptor deveria ser o primeiro a ser executado, afinal se o usuário não for autorizado não precisamos abrir uma transação ou logar as informações do método que ele está tentando acessar.

Podemos definir essa ordem utilizando os atributos **after** e **before** da anotação @Intercepts. Por exemplo, para dizer que o ControleDeTransacaoInterceptor deve ser executados depois do AutorizadorInterceptor, podemos adicionar o atributo after=AutorizadorInterceptor.class:

@AcceptsWithAnnotations(Log.class)

@Intercepts(after=AutorizadorInterceptor.class)

public class ControleDeTransacaoInterceptor {

Também podemos receber mais de uma classe nos atributos after e before, por exemplo, agora o ControleDeTransacaoInterceptor deve ser executado depois do AutorizadorInterceptor e LogInterceptor:

@AcceptsWithAnnotations(Log.class)

@Intercepts(after={

AutorizadorInterceptor.class,

LogInterceptor.class

})

public class ControleDeTransacaoInterceptor {

Para testar, vamos adicionar a anotação @Log no método lista do ProdutoController. Agora basta reiniciar o servidor e tentar acessar esse método antes de fazer o login na aplicação. Repare que o log do método lista não apareceu no console. Vamos fazer o login e acessar novamente o método lista. Agora sim o log está disponível no console.

Crie a classe AutorizadorInterceptor com a anotação @Intercepts. Adicione o método intercept que deve receber um SimpleInterceptorStack como parâmetro e estar anotado com @AroundCall. Lembre-se que a chamada ao método stack.next() quem determina se a lógica do controller será ou não executada. Faça uma condição para que só chame o método next caso o usuário esteja presente na sessão (usuarioLogado.getUsuario() != null), assim como fizemos no vídeo. Caso contrario deverá redirecionar para o formulario de login.

Quando terminar, compartilhe o código do interceptor. Lembre-se que isso ainda não vai funcionar, pois precisamos liberar acesso aos métodos públicos da classe LoginController.

RESPONDA

### Opinião do instrutor

A lógica do seu interceptor deverá ficar parecida com:

@Intercepts

public class AutorizadorInterceptor {

private final Result result;

private final UsuarioLogado usuarioLogado;

@Inject

public AutorizadorInterceptor(Result result,

UsuarioLogado usuarioLogado) {

this.result = result;

this.usuarioLogado = usuarioLogado;

}

@Deprecated

AutorizadorInterceptor() {

this(null, null); // para uso do CDI

}

@AroundCall

public void intercept(SimpleInterceptorStack stack) {

if (usuarioLogado.getUsuario() == null) {

result.redirectTo(LoginController.class).formulario();

return;

}

stack.next();

}

}

Para marcar os métodos que não queremos interceptar, vamos criar a anotação @Public no pacote br.com.caelum.vraptor:

@Documented

@Retention(value=RetentionPolicy.RUNTIME)

@Target(value=ElementType.METHOD)

public @interface Public {

}

Agora adicione essa anotação nos métodos formulario e autentica da classe LoginController.

Para determinar que apenas os métodos que não possuem essa anotação devem ser interceptados, adicione o método accepts em seu interceptor:

@Accepts

public boolean accepts() {

return !method.containsAnnotation(Public.class);

}

Reinicie o servidor e tente acessar a listagem de produtos. O que aconteceu?

RESPONDA

### Opinião do instrutor

Você deverá ser redirecionado para o formulário de login, que como agora é publico não será interceptado. Seu interceptor deve ficar assim:

@Intercepts

public class AutorizadorInterceptor {

private final Result result;

private final UsuarioLogado usuarioLogado;

private final ControllerMethod controllerMethod;

@Inject

public AutorizadorInterceptor(Result result,

UsuarioLogado usuarioLogado, ControllerMethod controllerMethod) {

this.result = result;

this.usuarioLogado = usuarioLogado;

this.controllerMethod = controllerMethod;

}

@Accepts

public boolean accepts() {

return !method.containsAnnotation(Public.class);

}

@AroundCall

public void intercept(SimpleInterceptorStack stack) {

if (usuarioLogado.getUsuario() == null) {

result.redirectTo(LoginController.class).formulario();

return;

}

stack.next();

}

}

# Trabalhando com plugins do VRaptor

Vimos que criar um interceptor como o ControleDeTransacaoInterceptor é uma prática muito comum quando estamos usando o JPA, por exemplo. Além disso, tivemos que criar um **produtor do CDI** (@Produces) para a classe EntityManager, para tirar proveito da injeção de dependências. Essas são apenas algumas situações que os desenvolvedores que pretendem usar JPA com VRaptor precisam se preocupar para que tudo funcione.

A fim de otimizar esse processo e aumentar sua produtividade, o VRaptor possui diversos complementos criados pela comunidade ou pela própria caelum para facilitar ainda mais seu trabalho de desenvolvimento.

Um de seus complementos, ou **plugins** como são habitualmente chamados, é o **vraptor-jpa**. Ele te ajuda com todo esse trabalho de controle de transações e injeção de classes do JPA. Vamos usá-lo em nosso projeto!

# Instalação e uso do vraptor-jpa

Como estamos usando um projeto maven, para que ele nos forneça o vraptor-jpabasta adicionar essa dependência no arquivo pom.xml de nosso projeto. Se você preferir não usar o maven, basta [baixar o jar](http://search.maven.org/remotecontent?filepath=br/com/caelum/vraptor/vraptor-jpa/4.0.1/vraptor-jpa-4.0.1.jar) e adicionar ao seu classpath, como qualquer outra biblioteca de seu projeto java.

Vamos abrir o arquivo pom.xml, que se encontra na raiz do nosso projeto. Agora só precisamos adicionar as linhas abaixo dentro da tag dependencies:

<dependencies>

<!-- outras dependências já declaradas -->

<dependency>

<groupId>br.com.caelum.vraptor</groupId>

<artifactId>vraptor-jpa</artifactId>

<version>4.0.1</version>

</dependency>

</dependencies>

Para o maven baixar essa dependência, basta clicar com o botão direito do mouse no projeto vraptor-produtos, selecionar a opção Maven e depois Update Project. Ou você pode utilizar o atalho ALT + F5.

Como o plugin já faz todo o controle de transações e de produção de objetos, vamos apagar o nosso ControleDeTransacaoInterceptor e também o produtor de EntityManager, o EntityManagerProducer.

Pronto! Não precisamos de nenhuma configuração, basta restartar o servidor e testar para garantir que tudo está funcionando! Adicionar complementos ao VRaptor é um processo bem simples, não acha?

# Enviando e-mail com VRaptor-simplemail

Para deixar nossa app de controle de estoque de produtos ainda mais completa, vamos criar uma nova funcionalidade que envolva o envio de e-mails notificando que precisamos de um estoque maior pra determinado produto. Mas como enviar um e-mail em uma aplicação Java Web? Podemos utilizar outro complemento do VRaptor, o plugin **vraptor-simplemail**!

Esse é outro complemento muito utilizado no dia a dia do desenvolvedor web com VRaptor, pois ele te provê uma forma muito simples e prática de configurar e fazer envio de e-mails.

Para adicionar esse plugin em nosso projeto, vamos abrir novamente o arquivo pom.xml e adicionar sua dependência:

<dependencies>

<!-- outras dependências já declaradas -->

<dependency>

<groupId>br.com.caelum.vraptor</groupId>

<artifactId>vraptor-simplemail</artifactId>

<version>4.0.0</version>

</dependency>

</dependencies>

Vamos pedir para o maven baixar essa nova dependência, assim como fizemos com o outro plugin: um clique com o botão direito do mouse no projeto vraptor-produtos, selecionar a opção Maven e depois Update Project.

Pronto! Já podemos começar a utilizar o **vraptor-simplemail**!

Vamos começar adicionando um link na lista.jsp, que o usuário usará para pedir mais itens de um determinado produto em estoque:

<c:forEach itens="${produtoList}" var="produto">

<tr>

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

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

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

<td>

<a href="<c:url value='/produto/enviaPedidoDeNovosItens?

produto.nome=${produto.nome}'/>">Pedir mais itens!</a>

</td>

</tr>

</c:forEach>

Agora, em nosso Controller precisamos criar um método que vai receber essa requisição. Vamos chamá-lo de enviaPedidoDeNovosItens:

@Get

public void enviaPedidoDeNovosItens(Produto produto) {

// agora só precisamos enviar um e-mail notificando!

}

O vraptor-simplemail trabalha internamente com a biblioteca [Commons Email, da Apache](http://commons.apache.org/email/). Para criar um e-mail, precimos instanciar a classe SimpleEmail dessa biblioteca, e adicionar suas informações, algo como:

@Get

public void enviaPedidoDeNovosItens(Produto produto) throws EmailException {

Email email = new SimpleEmail();

email.setSubject("[vraptor-produtos] Precisamos de mais estoque");

email.addTo("rodrigo.turini@caelum.com.br");

email.setMsg("Precisamos de mais itens do produto" + produto.getNome());

}

Excelente, agora precisamos efetuar o envio desse e-mail que criamos. Para isso precisamos pedir injetada a classe Mailer em nosso ProdutoController, pois essa é a classe do vraptor-simplemail responsável pelo envio de e-mails.

@Controller

public class ProdutoController {

// outros atributos da classe

private final Mailer mailer;

@Inject

public ProdutoController(Result result, ProdutoDao dao,

Validator validator, Mailer mailer) {

this.result = result;

this.dao = dao;

this.validator = validator;

this.mailer = mailer;

}

E em nosso método enviaPedidoDeNovosItens só precisamos invocar seu método sendpassando esse nosso e-mail como argumento, e em seguida podemos redirecionar o usuário para a listagem de produtos:

@Get

public void enviaPedidoDeNovosItens(Produto produto) throws EmailException {

Email email = new SimpleEmail();

email.setSubject("[vraptor-produtos] Precisamos de mais estoque");

email.addTo("rodrigo.turini@caelum.com.br");

email.setMsg("Precisamos de mais itens do produto" + produto.getNome());

mailer.send(email);

result.redirectTo(this).lista();

}

Mas se tentarmos enviar um e-mail agora, qual será a conta de e-mail utilizado para o envio? De qual servidor? Com qual senha? Ainda precisamos informar essas configurações! Só que note que queremos enviar e-mails reais apenas em produção, nunca em testes ou desenvolvimento. Ou talvez até podemos precisar enviar em testes ou desenvolvimento, só que nunca na mesma conta de e-mail que usaremos em produção. Para facilitar esse trabalho, o vraptor-simplemail é integrado com a função de **Environment** (diferentes ambientes) do VRaptor 4.

Em nosso web.xml podemos definir qual ambiente estamos utilizando:

<context-param>

<param-name>br.com.caelum.vraptor.environment</param-name>

<param-value>production</param-value>

</context-param>

E de acordo com essa informação, ele vai procurar o arquivo chamado: **nomeDoAmbiente.properties**. Neste caso, ele vai procurar o arquivo **production.properties**. Vamos adicionar nesse arquivo as configurações básicas para um envio de e-mail:

vraptor.simplemail.main.server=smtp.gmail.com

vraptor.simplemail.main.port=587

vraptor.simplemail.main.tls=true

vraptor.simplemail.main.from=curso-alura-vraptor4@gmail.com

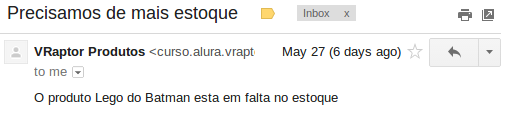
vraptor.simplemail.main.from.name=VRaptor Produtos

vraptor.simplemail.main.username=curso-alura-vraptor4@gmail.com

vraptor.simplemail.main.password=xxxxxx

Pronto, agora basta testar! Vamos abrir a listagem de produtos e clicar em Pedir mais itens! em algum dos campos.

Veja que rapidamente recebemos o e-mail com as informações!



Existem diversos outros plugins para tornar seu desenvolvimento com VRaptor ainda mais produtivo e sua aplicação ainda mais completa! Você pode conhecer alguns outros em nossa documentação. (link)

Adicione a seguinte dependência do vraptor-jpa em seu pom.xml:

<dependency>

<groupId>br.com.caelum.vraptor</groupId>

<artifactId>vraptor-jpa</artifactId>

<version>4.0.0.Final</version>

</dependency>

Depois de pedir para o maven atualizar as dependências (ALT + F5), apague a classe EntityManagerProducer e também a classe ControleDeTransacaoInterceptor (caso exista). Não se esqueça também de remover as linhas que fazem begin e commit na transação em sua classe ProdutoDao.

Pronto, você já pode reiniciar o tomcat e testar as operações do vraptor-produtos.

O que achou desse plugin?

RESPONDA

### Opinião do instrutor

O plugin vraptor-jpa é muito prático e produtivo! Tudo que você precisa fazer é adicioná-lo como dependência e tirar proveito das implementações prontas.

Adicione em seu pom.xml a dependência vraptor-simplemail:

<dependency>

<groupId>br.com.caelum.vraptor</groupId>

<artifactId>vraptor-simplemail</artifactId>

<version>4.0.0-RC2</version>

</dependency>

Atualize as dependências do maven (ALT + F5) e você estará pronto para usá-la!

Adicione em seu arquivo lista.jsp mais uma coluna com um link para envio de e-mail:

<td>

<a href="<c:url value='/produto/enviaPedidoDeNovosItens?

produto.nome=${produto.nome}'/>">Pedir mais itens!</a>

</td>

E também o método enviaPedidoDeNovosItens em seu controller:

@Get

public void enviaPedidoDeNovosItens(Produto produto) {

// agora só precisamos enviar um e-mail notificando!

}

Não se esqueça que para enviar o email você precisa pedir por injeção de dependências a classe Mailer.

Outro ponto é que você vai precisar adicionar as configurações em seu arquivo production.properties, por exemplo:

vraptor.simplemail.main.server=smtp.gmail.com

vraptor.simplemail.main.port=587

vraptor.simplemail.main.tls=true

vraptor.simplemail.main.from=curso-alura-vraptor4@gmail.com

vraptor.simplemail.main.from.name=VRaptor Produtos

vraptor.simplemail.main.username=curso-alura-vraptor4@gmail.com

vraptor.simplemail.main.password=xxxxxx

Reinicie o servidor e teste essa nova funcionalidade. Tudo funcionou? Como ficou seu código?

RESPONDA

### Opinião do instrutor

O método enviaPedidoDeNovosItens deve ficar parecido com:

@Get

public void enviaPedidoDeNovosItens(Produto produto) throws EmailException {

Email email = new SimpleEmail();

email.setSubject("[vraptor-produtos] Precisamos de mais estoque");

email.addTo("rodrigo.turini@caelum.com.br");

email.setMsg("Precisamos de mais itens do produto" + produto.getNome());

mailer.send(email);

result.redirectTo(this).lista();

}