

CAPÍTULO 17

Apêndice - VRaptor3 e produtividade na Web

"Aquele que castiga quando está irritado, não corrige, vinga-se"
— Michel de Montaigne

Neste capítulo, você aprenderá:

- O que é Inversão de Controle, Injeção de Dependências e *Convention over Configuration*;
- Como utilizar um framework MVC baseado em tais ideias;
- Como abstrair a camada HTTP da sua lógica de negócios;
- Como *não* utilizar arquivos XML para configuração da sua aplicação;
- A usar o framework MVC *VRaptor 3*.

17.1 - MOTIVAÇÃO: EVITANDO APIs COMPLICADAS

Vamos lembrar como fica um código utilizando um controlador MVC simples para acessar os parâmetros do request, enviados pelo cliente.

É fácil notar como as classes, interfaces e apetrechos daquele controlador infectam o nosso código e surge a necessidade de conhecer a API de servlets a fundo. O código a seguir mostra uma Action do Struts 1 que utiliza um DAO para incluir um contato no banco de dados.

```
public class AdicionaContato implements Action {  
  
    public String executa(HttpServletRequest req,  
        HttpServletResponse res) throws Exception {  
        Contato contato = new Contato();  
        contato.setNome(req.getParameter("nome"));  
        contato.setEndereco(req.getParameter("endereco"));  
    }  
}
```

```
contato.setEmail(req.getParameter("email"));

ContatoDao dao = new ContatoDao();
dao.adiciona(contato);

return "/ok.jsp";
}
}
```

Baseado no código acima, percebemos que estamos fortemente atrelados a `HttpServletRequest` e seu método `getParameter`. Fora isso, usamos diversas classes estranhas ao nosso projeto: `Action`, `HttpServletRequest` e `HttpServletResponse`. Se estivéssemos controlando melhor a conexão, seria necessário importar `Connection` também! Nenhuma dessas classes e interfaces citadas faz parte do nosso projeto! Não é o código que modela minha lógica!

É muito chato, e nada prático, repetir isso em toda a sua aplicação. Sendo assim, visando facilitar esse tipo de trabalho, vimos que o *Struts Action*, por exemplo, utiliza alguns recursos que facilitam o nosso trabalho:

```
public class AdicionaContato extends Action {

    public ActionForward execute(ActionMapping map, ActionForm form,
        HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        Contato contato = ((ContatoForm) form).getContato();

        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

        return map.findForward("ok");
    }
}
```

Mas, mesmo assim, imagine os limites de tal código:

- Você **não** pode receber mais de um action form;
- Sua classe **deve** estender `ActionForm`. Se você queria estender outra, azar;
- Você **deve** receber todos esses argumentos que não foi você quem criou;
- Você **deve** retornar esse tipo que não foi você quem criou;
- Você **deve** trabalhar com Strings ou tipos muito pobres. O sistema de conversão é complexo para iniciantes;
- O código fica muito alienado: ele é escrito de tal forma que o programador precisa agradar o framework e não o framework agradar o programador;

- Você acaba criando classes repetidas: deve criar dois beans repetidos ou parecidos, ou ainda escrever muito código xml para substituir um deles.

Dado esses problemas, surgiram diversos outros frameworks, inclusive diversos patterns novos, entre eles, Injeção de Dependências (*Dependency Injection*), Inversão de Controle (*Inversion of Control - IoC*) e o hábito de **preferir** ter convenções em vez de configuração (*Convention over Configuration - CoC*).

Imagine que possamos deixar de estender Action:

```
public class AdicionaContato {
    public ActionForward execute(ActionMapping map, ActionForm form,
        HttpServletRequest req, HttpServletResponse res)
        throws Exception {
        Contato contato = ((ContatoForm) form).getContato();

        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

        return map.findForward("ok");
    }
}
```

Nesse momento estamos livres para mudar nosso método execute. Desejamos que ele se chame adiciona, e não um nome genérico como execute, sem contar que não precisamos de todos aqueles quatro argumentos. Afinal, não utilizamos o request e response:

```
public class AdicionaContato {
    public ActionForward adiciona(ActionMapping map,
        ActionForm form) throws Exception {

        Contato contato = ((ContatoForm) form).getContato();

        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

        return map.findForward("ok");
    }
}
```

Aos poucos, o código vai ficando mais simples. Repare que na maioria das vezes que retornamos o ActionForward, o retorno é "ok". Será que sempre temos que fazer isso? Não seria mais fácil não retornar valor algum, já que sempre retornamos o mesmo valor? Portanto, vamos remover esse valor de retorno ao invés de usar esse estranho ActionMapping:

```
public class AdicionaContato {
    public void adiciona(ActionForm form) throws Exception {

        Contato contato = ((ContatoForm) form).getContato();
```

```
ContatoDao dao = new ContatoDao();
dao.adiciona(contato);

}
}
```

Por fim, em vez de criar uma classe que estende `ActionForm`, ou configurar toneladas de XML, desejamos receber um `Contato` como parâmetro do método.

Portanto nada mais natural que o parâmetro seja `Contato`, e não `ContatoForm`.

```
public class AdicionaContato {

    public void adiciona(Contato contato) throws Exception {

        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

    }
}
```

O resultado é um código bem mais legível, e um controlador menos intrusivo no seu código: nesse caso nem usamos a API de servlets!

17.2 - VANTAGENS DE UM CÓDIGO INDEPENDENTE DE REQUEST E RESPONSE

Você desconecta o seu programa da camada web, criando ações ou comandos que não trabalham com request e response.

Note que, enquanto utilizávamos a lógica daquela maneira, o mesmo servia de adaptador para a web. Não precisamos mais desse adaptador, nossa própria lógica que é uma classe Java comum, pode servir para a web ou a qualquer outro propósito.

Além disso, você recebe todos os objetos que precisa para trabalhar, não se preocupando em buscá-los. Isso é chamado de injeção de dependências. Por exemplo, se você precisa do usuário logado no sistema e, supondo que ele seja do tipo `Funcionario`, você pode criar um construtor que requer tal objeto:

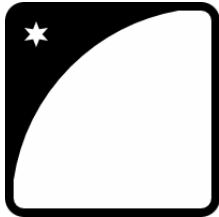
```
public class AdicionaContato {

    public AdicionaContato(Funcionario funcionario) {
        // o parametro é o funcionario logado no sistema
    }

    public void adiciona(Contato contato) throws Exception {
```

```
ContatoDao dao = new ContatoDao();  
dao.adiciona(contato);  
  
}  
}
```

Você pode também fazer o curso FJ-21 dessa apostila na Caelum



Querendo aprender ainda mais sobre Java na Web e Hibernate? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-21** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

[Consulte as vantagens do curso *Java para Desenvolvimento Web*.](#)

17.3 - VRAPTOR 3

Tudo o que faremos neste capítulo está baseado no framework opensource **VRaptor 3**. Sua documentação pode ser encontrada em:

<http://www.vraptor.com.br/>

Iniciativa brasileira, o VRaptor foi criado inicialmente para o desenvolvimento de projetos internos do Instituto de Matemática e Estatística da USP, pelos então alunos do curso de ciência da computação.

O site do G.U.J (www.guj.com.br), fundado em 2002 pelos mesmos criadores do VRaptor, teve sua versão nova escrita em VRaptor 2, e hoje em dia roda na versão 3. Este framework é utilizado em projetos open source e por várias empresas no Brasil e também pelo mundo.

Consulte o site para tutoriais, depoimentos, screencasts e até palestras filmadas a respeito do framework.

Para aprendermos mais do VRaptor, vamos utilizar seus recursos básicos em um pequeno projeto, e vale notar a simplicidade com qual o código vai ficar.

17.4 - A CLASSE DE MODELO

O projeto já vem com uma classe de modelo pronta, chamada **Produto**, que utilizaremos

em nossos exemplos, e é uma entidade do Hibernate:

```
@Entity
public class Produto {

    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;

    @Temporal(TemporalType.DATE)
    private Calendar dataInicioVenda;
    // getters e setters

}
```

A classe ProdutoDAO também já existe e utiliza o hibernate para acessar um banco de dados (mysql).

```
public class ProdutoDAO {

    private Session session;

    public ProdutoDao() {
        this.session = new HibernateUtil().getSession();
    }

    public void adiciona(Produto p) {
        Transaction tx = session.beginTransaction();
        session.save(p);
        tx.commit();
    }

    public void atualiza(Produto p) {
        Transaction tx = session.beginTransaction();
        session.update(p);
        tx.commit();
    }

    public void remove(Produto p) {
        Transaction tx = session.beginTransaction();
        session.delete(p);
        tx.commit();
    }

    @SuppressWarnings("unchecked")
    public List<Produto> lista() {
        return session.createCriteria(Produto.class).list();
    }
}
```

O foco desse capítulo não está em como configurar o Hibernate ou em boas práticas da

camada de persistência portanto o código do DAO pode não ser o ideal por utilizar uma transação para cada chamada de método, mas é o ideal para nosso exemplo.

17.5 - MINHA PRIMEIRA LÓGICA DE NEGÓCIOS

Vamos escrever uma classe que adiciona um Produto a um banco de dados através do DAO e do uso do VRaptor 3:

```
@Resource
public class ProdutoController {

    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }
}
```

Pronto! É assim que fica uma ação de adicionar produto utilizando esse controlador (e, futuramente, vamos melhorar mais ainda).

E se surgir a necessidade de criar um método *atualiza*? Poderíamos reutilizar a mesma classe para os dois métodos:

```
@Resource
public class ProdutoController {

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }

    // a ação atualiza
    public void atualiza(Produto produto) {
        new ProdutoDao().atualiza(produto);
    }
}
```

O próprio controlador se encarrega de preencher o produto para chamar nosso método. O que estamos fazendo através da anotação `@Resource` é dizer para o VRaptor disponibilizar essa classe para ser instanciada e exposta para a web. Dessa forma será disponibilizado uma URI para que seja possível invocar a lógica desejada, seja de adição de contato, seja de atualização.

Não existe nenhum XML do VRaptor que seja de preenchimento obrigatório. Só essa classe já é suficiente.

E o JSP com o formulário? Para fazer com que a lógica seja invocada, basta configurarmos corretamente os campos do formulário no nosso código html. Não há segredo

algum.

Note que nosso controller se chama `ProdutoController` e o nome do método que desejamos invocar se chama `adiciona`. Com isso, o VRaptor invocará este método através da URI `/produto/adiciona`. Repare que em nenhum momento você configurou esse endereço. Esse é um dos pontos no qual o VRaptor usa diversas convenções dele, ao invés de esperar que você faça alguma configuração obrigatória.

Tire suas dúvidas no novo G.U.J. Respostas



O G.U.J. é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do G.U.J. é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

17.6 - REDIRECIONANDO APÓS A INCLUSÃO

Precisamos criar uma página que mostre uma mensagem de sucesso, aproveitamos e confirmamos a inclusão mostrando os dados que foram incluídos:

```
<html>
  Seu produto foi adicionado com sucesso!<br/>
</html>
```

Mas qual o nome desse arquivo? Uma vez que o nome do controller é `produto`, o nome da lógica é `adiciona`, o nome de seu arquivo de saída deve ser: `WebContent/WEB-INF/jsp/produto/adiciona.jsp`.

Você pode alterar o redirecionamento padrão da sua lógica, enviando o usuário para um outro jsp, para isso basta receber no construtor do seu controller um objeto do tipo `Result`. Esse objeto será passado para o seu controller através de Injeção de Dependências.

Dessa forma, um exemplo de redirecionar para outro jsp após a execução seria:

```
@Resource
public class ProdutoController {
    private Result result;

    public ProdutoController(Result result) {
        this.result = result;
    }
}
```



```
}  
  
// a ação adiciona  
public void adiciona(Produto produto) {  
    new ProdutoDao().adiciona(produto);  
    result.forwardTo("/WEB-INF/jsp/outroLugar.jsp");  
}  
}
```

E a criação do arquivo `WEB-INF/jsp/outroLugar.jsp`. Dessa forma, você pode condicionar o retorno, de acordo com o que acontecer dentro do método que realizará toda a sua lógica, controlando para onde o usuário será redirecionado através de if's.

Além de termos a possibilidade de redirecionar para outro JSP, também podemos redirecionar para uma outra lógica. Novamente podemos fazer isso através do objeto `Result`. Basta dizermos que vamos enviar o usuário para outra lógica e indicarmos de qual `Controller` será essa lógica e também qual método deverá ser chamado.

```
public void remove(Produto produto) {  
    dao.remove(produto);  
    result.redirectTo(ProdutoController.class).lista();  
}
```

Isso fará com que logo após a lógica de remoção seja executada, o usuário execute a lógica de listagem que desenvolveremos a seguir.

17.7 - CRIANDO O FORMULÁRIO

Poderíamos criar nossos formulários como JSPs diretos dentro de `WebContent`, e acessarmos os mesmos diretamente no navegador, mas isso não é uma prática aconselhada, pois, futuramente podemos querer executar alguma lógica antes desse formulário e para isso ele teria que ser uma lógica do `VRaptor` e provavelmente teríamos que mudar sua URL, quebrando links dos usuários que já os possuíam gravados, por exemplo.

Vamos tomar uma abordagem diferente, pensando desde o começo que futuramente precisaremos de uma lógica executando antes do formulário, vamos criar uma lógica vazia, que nada mais fará do que repassar a execução ao formulário através da convenção do `VRaptor`.

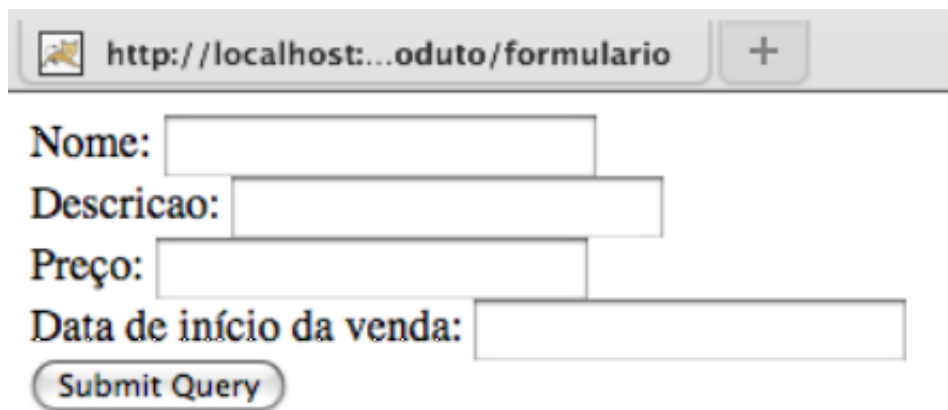
```
@Resource  
public class ProdutoController {  
    public void formulario() {  
    }  
}
```

Podemos acessar o nosso formulário pelo navegador através da URL:

/produto/formulario.

Já os parâmetros devem ser enviados com o nome do parâmetro que desejamos preencher, no nosso caso, produto, pois, é o nome do parâmetro do método adiciona:

```
<html>
  <form action="produto/adiciona">
    Nome: <input name="produto.nome"/><br/>
    Descricao: <input name="produto.descricao"/><br/>
    Preço: <input name="produto.preco"/><br/>
    Data de início de venda: <input name="produto.dataInicioVenda"/><br/>
    <input type="submit"/>
  </form>
</html>
```



17.8 - A LISTA DE PRODUTOS

O próximo passo será criar a listagem de todos os produtos do sistema. Nossa lógica de negócios, mais uma vez, possui somente aquilo que precisa:

```
@Resource
public class ProdutoController {

    public void lista() {
        new ProdutoDao().lista();
    }
}
```

Mas dessa vez precisamos disponibilizar esta lista de produtos para a nossa camada de visualização. Pensando em código java, qual é a forma mais simples que temos de retornar um valor (objeto) para alguém? Com um simples return, logo, nosso método lista poderia retornar a própria lista que foi devolvida pelo DAO, da seguinte forma:

```
public List<Produto> lista() {
    return new ProdutoDao().lista();
}
```

Dessa forma, o VRaptor disponibilizará na view um objeto chamado `produtoList`, que você possui acesso via *Expression Language*.

Repare que nossa classe pode ser testada facilmente, basta instanciar o bean `ProdutoController`, chamar o método `lista` e verificar se ele retorna ou não a lista que esperávamos. Todas essas convenções evitando o uso de configurações ajudam bastante no momento de criar testes de unidade para o seu sistema.

Por fim, vamos criar o arquivo `lista.jsp` no diretório `WebContent/WEB-INF/jsp/produto/` utilizando a taglib core da JSTL:

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

<h1>Produtos</h1>
<table>
<c:forEach var="produto" items="${produtoList}">
  <tr>
    <td>${produto.nome}</td>
    <td>${produto.preco}</td>
    <td>${produto.descricao}</td>
    <td><fmt:formatDate pattern="dd/MM/yyyy"
      value="${produto.dataInicioVenda.time}" />
    </td>
  </tr>
</c:forEach>
</table>
```

E chame o endereço: <http://localhost:8080/controle-produtos/produto/lista>, o resultado deve ser algo parecido com a imagem abaixo:



Nova editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de



experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

17.9 - EXERCÍCIOS

Vamos criar um novo projeto do Eclipse, usando um projeto já começado com o VRaptor.

1. Crie um novo **Dynamic Web Project** chamado **controle-produtos**.
2. Clique da direita no nome do projeto **controle-produtos** e vá em **Import > Archive File**. Importe o arquivo **Desktop/caelum/21/controle-produtos.zip**.
3. Gere a tabela Produto. Procure a classe GeraTabela e execute o método main.
4. Associe o projeto com o Tomcat e acesse no seu navegador a URL:
<http://localhost:8080/controle-produtos>
5. Vamos criar a parte de listagem dos produtos.
 - a. Crie a classe ProdutoController no pacote `br.com.caelum.produtos.controller`, com o método para fazer a listagem (**Não se esqueça de anotar a classe com @Resource**):

```
@Resource
public class ProdutoController {

    public List<Produto> lista() {
        return new ProdutoDao().lista();
    }

}
```

- b. Crie o arquivo `lista.jsp` no diretório `WebContent/WEB-INF/jsp/produto` (crie pasta e jsp)

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

<h1>Produtos</h1>
<table>
<c:forEach var="produto" items="${produtoList}">
    <tr>
        <td>${produto.nome}</td>
        <td>${produto.preco}</td>
        <td>${produto.descricao}</td>
        <td>
```

```

        <fmt:formatDate pattern="dd/MM/yyyy"
        value="${produto.dataInicioVenda.time}" />
    </td>
</tr>
</c:forEach>
</table>

```

c. Teste a sua lógica: <http://localhost:8080/controle-produtos/produto/lista>.

6. Vamos criar a funcionalidade de adicionar produtos.

a. Na classe ProdutoController crie o método adiciona:

```

@Resource
public class ProdutoController {

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }
}

```

b. Após o produto ser gravado, devemos retornar para a listagem dos produtos. Para isso vamos adicionar o redirecionamento no método adiciona. Para fazermos esse redirecionamento vamos precisar receber um objeto do tipo Result no construtor do nosso ProdutoController:

```

@Resource
public class ProdutoController {
    private Result result;

    public ProdutoController(Result result) {
        this.result = result;
    }

    //método para fazer a listagem

    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
        result.redirectTo(ProdutoController.class).lista();
    }
}

```

c. Vamos criar a lógica para o formulário

```

@Resource
public class ProdutoController {

    public void formulario() {
    }

}

```

d. Crie o arquivo `formulario.jsp` no diretório `WebContent/WEB-INF/jsp/produto`

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
<%@ taglib
  uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>
<html>
  <head>
    <script type="text/javascript"
      src="<c:url value="/js/jquery.js"/>">
    </script>
    <script type="text/javascript"
      src="<c:url value="/js/jquery-ui.js"/>">
    </script>
    <link type="text/css"
      href="<c:url value="/css/jquery.css"/>" rel="stylesheet" />
  </head>
  <body>
    <form action="<c:url value="/produto/adiciona"/>">
      Nome: <input name="produto.nome"/><br/>
      Descricao: <input name="produto.descricao"/><br/>
      Preço: <input name="produto.preco"/><br/>
      Data de início da venda:
      <caelum:campoData id="dataInicioVenda"
        name="produto.dataInicioVenda"/>
      <br />
      <input type="submit"/>
    </form>
  </body>
</html>
```

e. Se achar conveniente, crie um arquivo para o cabeçalho faça-o importar as bibliotecas em *Javascript* e o *CSS* e inclua-o no `formulario.jsp`

f. Você importou alguma classe do pacote `javax.servlet`? É importante percebermos que não estamos atrelados com nenhuma API estranha. Apenas escrevemos código Java normal.

g. Teste o seu formulário: <http://localhost:8080/controle-produtos/produto/formulario>. Adicione alguns produtos diferentes.

7. Vamos fazer a exclusão de produtos.

a. Vamos primeiramente criar uma nova coluna na nossa tabela no `lista.jsp`, adicionando para cada produto um link para fazermos a exclusão:

```
<c:forEach var="produto" items="${produtoList}">
  <tr>
    <td>${produto.nome}</td>
    <td>${produto.preco}</td>
    <td>${produto.descricao}</td>
    <td>
      <fmt:formatDate pattern="dd/MM/yyyy"

```

```
value="${produto.dataInicioVenda.time}"/>
</td>
<td>
  <a href="<c:url value="/produto/remove"/>?produto.id=${produto.id}">
    Remover
  </a>
</td>
</tr>
</c:forEach>
```

b. Vamos criar a nossa nova lógica para exclusão no `ProdutoController` que redirecionará para a lógica de lista após a remoção do produto:

```
public void remove(Produto produto) {
    new ProdutoDao().remove(produto);
    result.redirectTo(ProdutoController.class).lista();
}
```

c. Acesse a lista de produtos em <http://localhost:8080/controle-produtos/produto/lista> e remova alguns produtos.

8. (Opcional) Faça a funcionalidade de alteração do produto da forma que achar conveniente

17.10 - APROFUNDANDO EM INJEÇÃO DE DEPENDÊNCIAS E INVERSÃO DE CONTROLE

O VRaptor utiliza bastante Injeção de Dependências e também permite que nós utilizemos em nossos `Controllers`. O grande ganho que temos com isso é que nosso código fica muito menos acoplado com outras bibliotecas, aumentando a testabilidade da nossa aplicação através de testes de unidade, assunto coberto no curso **FJ-22**.

Além disso, outra vantagem que sentimos ao utilizar Injeção de Dependências é a legibilidade de nosso código. Nós sabemos o que ele recebe, mas lendo o nosso código, não precisamos saber de qual lugar essa dependência está vindo, apenas como ela será utilizada. O fato de não sabermos mais como a nossa dependência será criada é o que chamamos de Inversão de Controle.

O código que desenvolvemos no `ProdutoController` possui um exemplo muito bom de código acoplado. Repare que todas as nossas lógicas (`adiciona`, `remove`, `lista` etc) sabem como o `ProdutoDao` tem que ser criado, ou seja, que precisamos instanciá-lo sem passar nenhum argumento pro seu construtor. Se um dia mudássemos o construtor para receber algum parâmetro, teríamos que mudar todos os métodos de nossa lógica e isso não é algo que queremos nos preocupar no dia a dia.

17.11 - INJEÇÃO DE DEPENDÊNCIAS COM O VRAPTOR

Podemos desacoplar os nossos métodos de `ProdutoController` fazendo com que o `Controller` apenas receba em seu construtor uma instância de `ProdutoDao`. E quando o `VRaptor` precisar criar o `ProdutoController` a cada requisição, ele também criará uma instância do `ProdutoDao` para passar ao `Controller`.

```
public class ProdutoController {  
    private Result result;  
    private ProdutoDao produtoDao;  
  
    public ProdutoController(Result result, ProdutoDao produtoDao) {  
        this.result = result;  
        this.produtoDao = produtoDao;  
    }  
  
    //métodos para adicionar, excluir e listar produtos  
}
```

No entanto, se executarmos qualquer lógica nossa aplicação vai parar de funcionar. Isso porque precisamos dizer para o `VRaptor` que `ProdutoDao` é uma classe que ele vai gerenciar.

Podemos fazer isso através da anotação `@Component` no nosso `ProdutoDao`.

Aqui, em um caso mais usual, `ProdutoDao` seria uma interface, e nosso componente seria uma classe concreta como, por exemplo, `HibernateProdutoDao`. O `VRaptor` vai saber descobrir que deve injetar um `HibernateProdutoDao` para quem precisa um `ProdutoDao`. Dessa forma desacoplamos ainda mais nosso código do controller, podendo passar mocks para ele quando formos testá-lo, como veremos no FJ-22 e FJ-28.

```
@Component  
public class ProdutoDao {  
  
    //construtor e métodos do Dao  
}
```

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

17.12 - ESCOPOS DOS COMPONENTES

Por padrão, o VRaptor criará um `ProdutoDao` por requisição. Mas nem sempre é isso que queremos. Portanto, podemos mudar esse comportamento através de anotações que determinam em qual escopo o nosso componente ficará:

- `@RequestScoped` - o componente será o mesmo durante a requisição
- `@SessionScoped` - o componente será o mesmo durante a sessão do usuário
- `@ApplicationScoped` - o componente será o mesmo para toda a aplicação
- `@PrototypeScoped` - o componente será instanciado sempre que requisitado

Para definirmos um escopo, o nosso `ContatoDao` poderia ficar da seguinte forma:

```
@Component
@RequestScoped
public class ProdutoDao {

    //construtor e métodos do Dao
}
```

Como os componentes são gerenciados pelo VRaptor?

Uma das boas ideias do VRaptor é o de não reinventar a roda e reutilizar funcionalidades e frameworks já existentes quando possível.

Uma das partes na qual o VRaptor se utiliza de frameworks externos é na parte de Injeção de Dependências, na qual ele utiliza o framework `Spring` que possui um container de Injeção de Dependências.

Aprendemos `Spring` a fundo no curso FJ-27 que cobre em detalhes o framework.

17.13 - EXERCÍCIOS: USANDO INJEÇÃO DE DEPENDÊNCIAS PARA O DAO

1. Vamos diminuir o acoplamento do `ProdutoController` com o `ProdutoDao` recebendo-o no construtor.

a. Precisamos fazer com que o `ProdutoDao` seja um componente gerenciado pelo VRaptor. Para isso, vamos anotá-lo com `@Component` e `@RequestScoped`.

```
@Component
@RequestScoped
public class ProdutoDao {

    //construtor e métodos do Dao
}
```

b. Basta indicarmos que queremos receber `ProdutoDao` no construtor do `ProdutoController` e utilizarmos o DAO recebido para fazermos nossas operações com o banco de dados. Não se esqueça de alterar seus métodos.

```
@Resource
public class ProdutoController {
    private Result result;
    private ProdutoDao produtoDao;

    public ProdutoController(Result result, ProdutoDao produtoDao) {
        this.result = result;
        this.produtoDao = produtoDao;
    }

    public List<Produto> lista() {
        return produtoDao.lista();
    }

    public void adiciona(Produto produto) {
        produtoDao.adiciona(produto);
        //redirecionamento
    }

    public void remove(Produto produto) {
        produtoDao.remove(produto);
        //redirecionamento
    }
}
```

c. Reinicie sua aplicação e verifique que tudo continua funcionando normalmente.

17.14 - ADICIONANDO SEGURANÇA EM NOSSA APLICAÇÃO

Nossa aplicação de controle de produtos permite que qualquer pessoa modifique os dados de produtos. Isso não é algo bom, pois, pessoas sem as devidas permissões poderão acessar essa funcionalidade e guardar dados inconsistentes no banco.

Precisamos fazer com que os usuários façam login em nossa aplicação e caso ele esteja logado, permitiremos acesso às funcionalidades.

Para construir a funcionalidade de autenticação, precisamos antes ter o modelo de `Usuario` e o seu respectivo DAO com o método para buscar o `Usuario` através do login e senha.

```
@Entity
public class Usuario {
    @Id @GeneratedValue
    private Long id;

    private String nome;

    private String login;

    private String senha;

    //getters e setters
}

@Component
@RequestScoped
public class UsuarioDao {
    private Session session;

    public UsuarioDao() {
        this.session = new HibernateUtil().getSession();
    }

    public Usuario buscaUsuarioPorLoginESenha(Usuario usuario) {
        Query query = this.session.
            createQuery("from Usuario where " +
                "login = :pLogin and senha = :pSenha");
        query.setParameter("pLogin", usuario.getLogin());
        query.setParameter("pSenha", usuario.getSenha());
        return (Usuario) query.uniqueResult();
    }
}
```

Como já possuímos o modelo de usuário, precisamos guardar o usuário na sessão.

Já aprendemos que podemos criar um componente (`@Component`) que fica guardado na sessão do usuário. Portanto, vamos utilizar essa facilidade do VRaptor.

```
@Component
@SessionScoped
public class UsuarioLogado {
    private Usuario usuarioLogado;

    public void efetuaLogin(Usuario usuario) {
        this.usuarioLogado = usuario;
    }

    //getter pro usuarioLogado
}
```

Basta construirmos o Controller que fará o login do nosso usuário. Vamos criar uma classe chamada LoginController que receberá via construtor um UsuarioLogado e o UsuarioDao. Após a verificação de que o usuário informado está cadastrado o guardaremos dentro do nosso componente UsuarioLogado. Se o usuário existir, a requisição será redirecionada para a listagem dos produtos.

```
@Controller
public class LoginController {
    private UsuarioDao usuarioDao;
    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginController(UsuarioDao usuarioDao,
        UsuarioLogado usuarioLogado, Result result){
        this.usuarioDao = usuarioDao;
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void autentica(Usuario usuario) {
        Usuario autenticado = usuarioDao
            .buscaUsuarioPorLoginESenha(usuario);
        if(autenticado != null) {
            usuarioLogado.efetuaLogin(autenticado);
            result.redirectTo(ProdutoController.class).lista();
        }
    }
}
```

Basta criarmos o formulário para fazermos o login. Vamos utilizar a mesma estratégia do formulário para cadastro de produtos, criando um método no nosso Controller que redirecionará para o formulário.

```
@Controller
public class LoginController {
    //atributos, construtor e métodos para efetuar o login

    public void formulario() {
    }
}
```

E o JSP contendo o formulário, que estará em WEB-INF/jsp/login e se chamará formulario.jsp:

```
<html>
<body>
    <h2>Login no Controle de Produtos</h2>
    <form action="login/autentica">
        Login: <input type="text" name="usuario.login" /><br />
        Senha: <input type="password" name="usuario.senha" />
        <input type="submit" value="Autenticar" />
    </form>
</body>
```

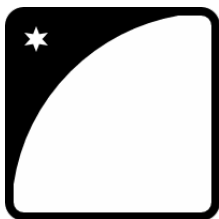
</html>

Ainda precisamos fazer com que se o login seja inválido, o usuário volte para a tela de login:

```
public void autentica(Usuario usuario) {
    Usuario autenticado = dao.buscaUsuarioPorLoginESenha(usuario);
    if (autenticado != null) {
        usuarioLogado.efetuaLogin(autenticado);
        result.redirectTo(ProdutoController.class).lista();
        return;
    }
    result.redirectTo(LoginController.class).formulario();
}
```

Pronto, nossa funcionalidade de Login está pronta!

Você não está nessa página a toa



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso *Java para Desenvolvimento Web*.](http://www.caelum.com.br/apostila-java-web/apendice-vraptor3-e-produtividade-na-web/)

17.15 - INTERCEPTANDO REQUISIÇÕES

Mas como garantir que o usuário está mesmo logado na nossa aplicação no momento em que ele tenta, por exemplo, acessar o formulário de gravação de produtos?

O que precisamos fazer é verificar, antes de qualquer lógica ser executada, se o usuário está guardado na sessão. Podemos fazer isso através de **Interceptors**, que funcionam de forma parecida com os **Filters** que aprendemos anteriormente. A vantagem é que os **Interceptors** nos fornecem facilidades a mais que estão ligadas ao VRaptor, algo que a API de **Filter** não nos provê.

Para criarmos um **Interceptor** basta criarmos uma classe que implementa a interface `br.com.caelum.vraptor.Interceptor` e anotá-la com `@Intercepts`.

Ao implementarmos a interface, devemos escrever dois métodos: `intercept` e `accepts`.

- `intercept`: Possui o código que fará toda a lógica que desejamos executar antes e depois da

requisição ser processada.

- **accepts:** Método que indica através de um retorno `boolean` quem deverá ser interceptado e quem não deverá ser interceptado.

O `Interceptor` como qualquer componente, pode receber em seu construtor outros componentes e no nosso caso ele precisará do `UsuarioLogado` para saber se existe alguém logado ou não.

Dessa forma, o nosso `Interceptor` terá o seguinte código:

```
@Intercepts
public class LoginInterceptor implements Interceptor {

    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginInterceptor(UsuarioLogado usuarioLogado, Result result) {
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void intercept(InterceptorStack stack,
        ResourceMethod method, Object instance)
        throws InterceptionException {
        if(usuarioLogado.getUsuario() != null) {
            stack.next(method, instance);
        } else {
            result.redirectTo(LoginController.class).formulario();
        }
    }

    public boolean accepts(ResourceMethod method) {
        ResourceClass resource = method.getResource();

        return !resource.getType().isAssignableFrom(LoginController.class);
    }
}
```

Pronto, temos nossa funcionalidade de autenticação e também a parte de autorização finalizadas.

17.16 - EXERCÍCIOS: CONSTRUINDO A AUTENTICAÇÃO E A AUTORIZAÇÃO

1. Vamos permitir que nossos usuários possam efetuar Login na aplicação.

- a. Crie o componente `UsuarioLogado` no pacote `br.com.caelum.produtos.component` com o seguinte código:

```

@Component
@SessionScoped
public class UsuarioLogado {
    private Usuario usuarioLogado;

    public void efetuaLogin(Usuario usuario) {
        this.usuarioLogado = usuario;
    }

    public Usuario getUsuario() {
        return this.usuarioLogado;
    }
}

```

b. Faça com que `UsuarioDao` seja um componente anotando-o com `@Component` e indique que ele deverá estar no escopo de requisição (`@RequestScoped`).

```

@Component
@RequestScoped
public class UsuarioDao {

    //metodos e construtor
}

```

c. Crie a classe `LoginController` dentro do pacote `br.com.caelum.produtos.controller`:

```

@Resource
public class LoginController {
    private UsuarioDao usuarioDao;
    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginController(UsuarioDao usuarioDao,
        UsuarioLogado usuarioLogado, Result result) {
        this.usuarioDao = usuarioDao;
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void autentica(Usuario usuario) {
        Usuario autenticado = usuarioDao
            .buscaUsuarioPorLoginESenha(usuario);
        if(autenticado != null) {
            usuarioLogado.efetuaLogin(autenticado);
            result.redirectTo(ProdutoController.class)
                .lista();
            return;
        }
        result.redirectTo(LoginController.class).formulario();
    }

    public void formulario() {
    }
}

```

d. Vamos criar a tela para permitir com que os usuários se loguem na aplicação, crie um

arquivo chamado `formulario.jsp` dentro de `WEB-INF/jsp/login` com o conteúdo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <body>
    <h2>Login no Controle de Produtos</h2>
    <form action="<c:url value="/login/autentica"/>">
      Login: <input type="text" name="usuario.login" /><br />
      Senha: <input type="password" name="usuario.senha" />
      <input type="submit" value="Autenticar" />
    </form>
  </body>
</html>
```

e. Vamos criar o interceptador para não deixar o usuário acessar as funcionalidades do nosso sistema sem ter logado antes. Crie a classe `LoginInterceptor` no pacote

`br.com.caelum.produtos.interceptor:`

```
@Intercepts
public class LoginInterceptor implements Interceptor {

    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginInterceptor(UsuarioLogado usuarioLogado,
        Result result) {
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void intercept(InterceptorStack stack,
        ResourceMethod method, Object instance)
        throws InterceptionException {

        if(usuarioLogado.getUsuario() != null) {
            stack.next(method, instance);
        } else {
            result.redirectTo(LoginController.class).formulario();
        }
    }

    public boolean accepts(ResourceMethod method) {
        ResourceClass resource = method.getResource();

        return !resource.getType().isAssignableFrom(LoginController.class);
    }
}
```

f. Tente acessar <http://localhost:8080/controle-produtos/produto/lista> e como você não efetuou o login, você é redirecionado para a devida tela de login.



g. Efetue o seu login. Verifique o seu banco de dados para conseguir um login válido. Caso não exista ninguém cadastrado, insira um usuário no banco com o comando abaixo e tente efetuar o login:

```
insert into Usuario (nome, login, senha)
values ('Administrador', 'admin', 'admin123');
```

17.17 - MELHORANDO A USABILIDADE DA NOSSA APLICAÇÃO

Sempre que clicamos no link *Remover* na listagem dos produtos uma requisição é enviada para o endereço `/produto/remove`, a exclusão é feita no banco de dados e em seguida **toda** a listagem é recriada novamente.

A requisição ser enviada e a exclusão ser feita no banco de dados são passos obrigatórios nesse processo, mas será que precisamos recriar toda a listagem outra vez?

Podemos destacar alguns pontos negativos nessa abordagem, por exemplo:

- Tráfego na rede: Recebemos como resposta todo o HTML para regerar a tela inteira. Não seria mais leve apenas removermos a linha da tabela que acabamos de excluir ao invés de recriarmos toda a tabela?
- Demora na resposta: Se a requisição demorar para gerar uma resposta, toda a navegação ficará comprometida. Dependendo do ponto aonde estiver a lentidão poderá até ficar uma tela em branco que não dirá nada ao usuário. Será que não é mais interessante para o usuário permanecer na mesma tela em que ele estava e colocar na tela uma mensagem indicando que está realizando a operação? Como a tela continuará aberta sempre, ele pode continuar sua navegação e uso normalmente.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no



mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil. Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](http://www.casodocodigo.com.br)

17.18 - PARA SABER MAIS: REQUISIÇÕES: SÍNCRONO X ASSÍNCRONO

Nós podemos enviar requisições em nossas aplicações web que não "travem" a navegação do usuário e mantenha a aplicação disponível para continuar sendo utilizada. As requisições tradicionais, com as quais estamos acostumados é o que chamamos de *Requisições Síncronas*, na qual o navegador ao enviar a requisição interrompe a navegação e reexibe toda a resposta devolvida pelo servidor.

Mas podemos utilizar um outro estilo, que são as *Requisições Assíncronas*. Nelas, quando a requisição é enviada, o navegador continua no estado em que estava antes, permitindo a navegação. Quando a resposta é dada pelo servidor é possível pegá-la e apenas editar algum pedaço da página que já estava exibida antes para mostrar um conteúdo novo (sem recarregar toda a página).

17.19 - PARA SABER MAIS: AJAX

Podemos fazer requisições assíncronas através de uma técnica conhecida como AJAX (*Assynchronous Javascript and XML*). Essa técnica nada mais é do que utilizar a linguagem *Javascript* para que em determinados eventos da sua página, por exemplo, o clique de um botão, seja possível enviar as requisições de forma assíncrona para um determinado lugar, recuperar essa resposta e editar nossa página para alterarmos dinamicamente o seu conteúdo.

Um dos grandes problemas de AJAX quando a técnica surgiu é que era muito complicado utilizá-la, principalmente porque haviam incompatibilidades entre os diversos navegadores existentes. Muitos tratamentos tinham que ser feitos, para que as funcionalidades fossem compatíveis com os navegadores. E dessa forma, tínhamos códigos muito grandes e de difícil legibilidade.

Hoje em dia temos muito mais facilidade para trabalhar com AJAX, através de bibliotecas

de *Javascript* que encapsulam toda a complexidade nos fornecem uma API agradável para trabalhar, como por exemplo, jQuery, YUI (Yahoo User Interface), ExtJS e assim por diante.

17.20 - ADICIONANDO AJAX NA NOSSA APLICAÇÃO

Aqui no curso utilizaremos o jQuery, mas tudo o que fizermos também poderá ser feito com as outras bibliotecas. Para detalhes de como fazer, consulte suas documentações.

Vamos colocar AJAX na remoção dos produtos. Ao clicarmos no link *Remover*, vamos disparar um evento que estará associado com uma função em Javascript, que utilizará o jQuery para enviar a requisição, pegar a resposta que será uma mensagem indicando que o produto foi removido e colocar essa mensagem logo acima da listagem dos produtos.

Primeiramente, precisamos importar o jQuery na listagem de produtos, o arquivo WEB-INF/jsp/produto/lista.jsp. Podemos fazer isso através adicionando a tag <head> importando um arquivo Javascript, como a seguir:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <head>
    <script type="text/javascript"
      src="/controle-produtos/js/jquery.js">
    </script>
  </head>
  <body>
    <!-- continuação da pagina -->
```

Podemos alterar o link para disparar um evento ao ser clicado, e não mais chamar uma URL. O nosso link não nos enviará para lugar nenhum, será mais um artifício visual para exibi-lo como tal, portanto, o seu atributo href ficará com o valor #. Adicionaremos também ao link um evento do tipo onclick que fará uma chamada à função removeProduto que precisará do id do produto, para saber quem será removido:

```
<td>
  <a href="#" onclick="return removeProduto(${produto.id})">
    Remover
  </a>
</td>
```

Precisamos implementar a nossa função removeProduto(). Para isso, dentro do body da nossa página vamos colocar a função que receberá o id para enviar a requisição para a lógica de exclusão. A resposta gerada por essa lógica, nós vamos colocar em uma div cujo id se chamará mensagem:

```
<script type="text/javascript">
function removeProduto(id) {
    $('#mensagem')
        .load('/controle-produtos/produto/remove?produto.id=' + id);
}
</script>
```

No jQuery a # serve para especificar qual elemento você deseja trabalhar através do id desse elemento. Vamos criar a div antes da tabela da listagem dos produtos:

```
<html>
<head>
    <script type="text/javascript"
        src="/controle-produtos/js/jquery.js">
    </script>
</head>
<body>
    <h1>Produtos</h1>
    <div id="mensagem"></div>
    <!-- tabela para mostrar a lista dos produtos -->
```

Nossa remoção ainda não funcionará do jeito que queremos, pois, a resposta gerada está sendo a própria página da listagem e não uma mensagem de confirmação. Isso tudo acontece devido ao redirecionamento que colocamos na lógica para a remoção. Vamos retirar o redirecionamento e criarmos um .jsp para mostrar a mensagem de confirmação da exclusão.

O método remove do ProdutoController deverá possuir apenas o código da remoção:

```
public void remove(Produto produto) {
    produtoDao.remove(produto);
}
```

E vamos criar um novo .jsp que será chamado após a execução da remoção. Criaremos ele no diretório WEB-INF/jsp/produto com o nome remove.jsp e o seguinte conteúdo:

Produto removido com sucesso

Por fim, o último passo que precisamos fazer é remover da tabela o produto que foi excluído. Mais uma vez utilizaremos o jQuery para nos auxiliar. Vamos precisar identificar qual linha vamos remover da tabela. Para isso, todas as linhas (tr) terão uma identificação única que será composto pelo id de cada produto precedida pela palavra "produto", por exemplo, produto1, produto2 e assim por diante.

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

```

<fmt:formatDate pattern="dd/MM/yyyy"
  value="${produto.dataInicioVenda.time}" />
</td>
<td><a href="#" onclick="return removeProduto(${produto.id})">
  Remover
</a></td>
</tr>
</c:forEach>

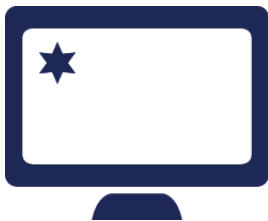
```

Em nossa função *Javascript* para remover o produto, podemos também remover a linha da tabela:

```
$('#produto' + id).remove();
```

Vale lembrar que a forma que apresentamos é apenas uma das formas de fazer, existem muitas outras formas diferentes de se atingir o mesmo comportamento.

Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](#)

17.21 - EXERCÍCIOS OPCIONAIS: ADICIONANDO AJAX NA NOSSA APLICAÇÃO

1. a. Na lista.jsp no diretório WEB-INF/jsp/produto, faça a importação do jQuery:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
  <script type="text/javascript"
    src="<c:url value="/js/jquery.js"/>">
  </script>
</head>
<body>
  <!-- continuação da pagina -->

```

- b. Altere os links para chamar uma função chamada removeProduto:

```

<td>
  <a href="#"
    onclick="return removeProduto(${produto.id})">
    Remover

```

```
</a>
</td>
```

c. Vamos criar a nossa função que executará o AJAX e removerá o item da lista. Adicione as seguintes linhas dentro do body da sua página `lista.jsp`:

```
<!-- inicio da pagina e import do javascript -->
<body>
  <script type="text/javascript">
    function removeProduto(id) {
      $('#mensagem')
        .load('<c:url value="/produto/remove"/>' +
          '?produto.id=' + id);
      $('#produto' + id).remove();
    }
  </script>

  <!-- continuação da pagina -->
</body>
```

d. Adicione uma div na sua página com o id mensagem, aonde será colocada a resposta devolvida pelo servidor:

```
<h1>Produtos</h1>
<div id="mensagem"></div>
<!-- tabela para mostrar a lista dos produtos -->
```

e. Dê um id para os `<tr>`, dessa forma você poderá apagá-los:

```
<tr id="produto${produto.id}">
```

f. Remova o redirecionamento do método `remove` do `ProdutoController`, de forma que ele fique como a seguir:

```
public void remove(Produto produto) {
    produtoDao.remove(produto);
}
```

g. Por fim, crie no diretório `WEB-INF/jsp/produto` o arquivo `remove.jsp` com o seguinte conteúdo:

Produto removido com sucesso

h. Acesse novamente a listagem dos produtos e faça a remoção deles. Perceba que a página não é recarregada quando você clica no *link*.

CAPÍTULO ANTERIOR:

[Apêndice - Integração do Spring com JPA](#)

PRÓXIMO CAPÍTULO:

[Apêndice - Java EE 6](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter

