

Desenvolvimento Ágil para a Web 2.0 com VRaptor, Hibernate e AJAX

Curso FJ-28



Conheça mais da Caelum.



Cursos Online

www.caelum.com.br/online



Casa do Código

Livros para o programador
www.casadocodigo.com.br



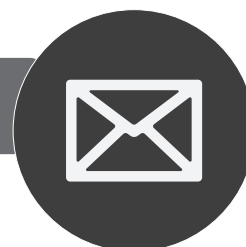
Blog Caelum

blog.caelum.com.br



Newsletter

www.caelum.com.br/newsletter



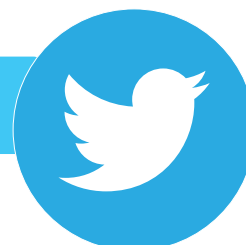
Facebook

www.facebook.com/caelumbr



Twitter

twitter.com/caelum



Sobre esta apostila

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no final do índice.

Baixe sempre a versão mais nova em: www.caelum.com.br/apostilas

Esse material é parte integrante do treinamento Desenv. Ágil para Web 2.0 com VRaptor, Hibernate e AJAX e distribuído gratuitamente exclusivamente pelo site da Caelum. Todos os direitos são reservados à Caelum. A distribuição, cópia, revenda e utilização para ministrar treinamentos são absolutamente vedadas. Para uso comercial deste material, por favor, consulte a Caelum previamente.

www.caelum.com.br

Sumário

1	O curso	1
1.1	Objetivo do curso	1
1.2	Sobre o curso: Hibernate, VRaptor, JSP e AJAX	1
1.3	Onde posso aprender mais do VRaptor?	2
2	O sistema	3
2.1	A necessidade do cliente	3
2.2	Partes do projeto	4
3	Novo projeto	5
3.1	Definindo as tecnologias	5
3.2	Montando o ambiente de desenvolvimento	6
3.3	Exercícios: montando o ambiente de desenvolvimento	7
3.4	Exercícios: configurando o deploy da aplicação	12
4	Persistindo os dados com o Hibernate	15
4.1	A camada de persistência do projeto	15
4.2	Sobre o Hibernate	16
4.3	Preparando o Hibernate	16
4.4	Configurando o Hibernate	17
4.5	Exercícios: Configurando o banco	18
5	Cadastrando Produtos	20
5.1	Modelando um produto	20
5.2	Anotando a classe Produto	21
5.3	Exercícios - Modelando o produto	23
5.4	Adicionando um produto no banco	24
5.5	Exercícios	25
5.6	Outras operações com produto	26
5.7	Exercícios - outras operações com produto	27
5.8	Discussão em sala - mais sobre o Hibernate	30
6	Refatorando	31
6.1	Analisando o código atual	31
6.2	Refactoring	31
6.3	Aprendendo a refatorar	32
6.4	Exercícios	33
6.5	Comentários são sempre necessários?	37
6.6	Refatorando para criar os DAOs	38
6.7	Exercícios	38

6.8	Discussão em sala	46
7	VRaptor	47
7.1	Sobre o VRaptor	47
7.2	Como instalar	47
7.3	Como configurar	47
7.4	Primeiro exemplo com o VRaptor	48
7.5	Exercícios	50
7.6	Redirecionando para uma view	50
7.7	Exercícios	51
7.8	Disponibilizando informações para a view	52
7.9	Disponibilizando coleções para a view	53
7.10	Exercícios	54
8	Criando o Controlador de Produtos	57
8.1	Listando produtos	57
8.2	Quais são minhas dependências?	59
8.3	Injeção de Dependências	60
8.4	Exercícios	61
8.5	Cadastrando um produto	65
8.6	Criando o formulário HTML	66
8.7	Exercícios	67
8.8	Redirecionar para listagem depois de adicionar	70
8.9	Exercícios	72
8.10	Atualizando e removendo produtos	74
8.11	Exercícios	77
8.12	Discussão em sala - VRaptor	80
9	Refatorando os DAOs	81
9.1	Injeção de dependências no DAO	81
9.2	Exercícios	83
9.3	Analisando o código	86
9.4	Escopos definidos pelo VRaptor	86
9.5	Fechando a sessão do Hibernate	89
9.6	Exercícios	90
10	Validando formulários	93
10.1	Validator	93
10.2	Exercícios	96
10.3	Para saber mais: Hibernate Validator	98
10.4	Exercícios Opcionais	100
11	REST	102

11.1	O que é REST	102
11.2	Características e vantagens	102
11.3	O triângulo do REST	103
11.4	Mudando a URI da sua lógica: @Path	104
11.5	Mudando o verbo HTTP dos seus métodos	105
11.6	Refatorando o ProdutosController	106
11.7	Exercícios	110
12	AJAX e efeitos visuais	112
12.1	O que é AJAX?	112
12.2	Um pouco de JQuery	112
12.3	Validando formulários com o JQuery	113
12.4	Criando a busca de produtos	115
12.5	Melhorando a busca: Autocomplete	118
12.6	Exercícios	121
12.7	Para saber mais: Representation	125
13	Criando o Carrinho de Compras	127
13.1	O modelo do Carrinho	127
13.2	Controlando o carrinho de compras	128
13.3	Visualizando os itens do carrinho	132
13.4	Removendo itens do carrinho	135
13.5	Exercícios	136
14	Autenticação	142
14.1	Criando Usuários	142
14.2	Efetuando o login	146
14.3	Restringindo funcionalidades para usuários logados	149
14.4	Interceptor	150
15	Apêndice - Download e Upload	154
15.1	Exercícios	154
16	Apêndice - Integrando VRaptor e Spring	158
16.1	Como fazer a integração?	158
16.2	Integrando o Transaction Manager do Spring	158
16.3	Exercícios: Transaction Manager	159
17	Apêndice: Mudando a View Padrão: Velocity	163
17.1	Exercícios: Configurando o Velocity	163
17.2	Exercícios: Mudando o Template Engine de uma única lógica	164
17.3	Exercícios: Mudando o resultado de todas as lógicas para Velocity	165
	Índice Remissivo	166

Versão: 17.o.6

CAPÍTULO 1

O curso

“Dizem que os homens nunca se contentam e, quando se lhes dá alguma coisa, pedem sempre um pouco mais. Dizem ainda que essa é uma das melhores qualidades da espécie e que foi ela que tornou o homem superior aos animais, que se contentam com o que têm.”
– A pérola, John Steibeck.

1.1 OBJETIVO DO CURSO

Uma das grandes vantagens de utilizar a plataforma Java é a quantidade de opções: são centenas de frameworks opensource de qualidade que podemos escolher para desenvolver um projeto. Mas qual deles escolher?

Depois da escolha dos frameworks ainda temos um outro problema: fazer com que eles trabalhem juntos de maneira coesa. Para isso é necessário conhecer as boas práticas e tratar muito bem do ciclo de vida dos objetos “caros”, como threads, conexões, seções e arquivos. Um erro no gerenciamento desses objetos pode ser fatal para a performance e escalabilidade do seu sistema.

Nesse curso, além de estudarmos frameworks para o desenvolvimento web, é necessário ficar atento para a importância de como vamos fazê-los trabalhar junto. Essa costura pode ser aplicada em diversos outros casos, mesmo com outros frameworks, e é baseada essencialmente no bom isolamento de classes, usando inversão de controle e injeção de dependências, assunto que será tratado recorrentemente durante o curso.

1.2 SOBRE O CURSO: HIBERNATE, VRAPTOR, JSP E AJAX

Um dos frameworks que é encontrado na grande maioria dos projetos atualmente é o Hibernate, que será utilizado nesse curso para persistência. Veremos mais do que o simples gerenciamento de estado e de queries, pois será praticado como tratar de maneira correta a `Session` e a `Transaction` envolvida.

O VRaptor, iniciativa brasileira para Controller MVC, é utilizado para facilitar o desenvolvimento web, evitando o contato com as classes pouco amigáveis do `javax.servlet`, deixando o código legível e desacoplado, ideal para testes.

Java Server Pages é utilizado como camada View, mostrando as principais tags e problemas enfrentados. Juntamente com o VRaptor, o framework javascript JQuery será utilizado para nossas requisições AJAX consumindo dados no formato JSON.

1.3 ONDE POSSO APRENDER MAIS DO VRAPTOR?

Você pode encontrar uma vasta documentação em português no site do VRaptor: <http://www.vraptor.com.br/>

O fórum do GUJ também é uma excelente fonte de informação, e abriga o fórum oficial do VRaptor: <http://www.guj.com.br/>

CAPÍTULO 2

O sistema

Veremos como será o sistema que vamos desenvolver e as tecnologias que utilizaremos.

2.1 A NECESSIDADE DO CLIENTE

Nossa empresa foi contratada para desenvolver um sistema de compras online. O cliente quer que o sistema seja acessado através da web, e deixou claro que devemos nos preocupar muito com a facilidade de navegação e a interação dos usuários com o sistema.

As seguintes funcionalidades foram solicitadas pelo cliente:

- Cadastrar, atualizar, listar e remover produtos
- Buscar produtos
- Adicionar, remover e listar produtos do carrinho de compras
- Cadastrar, atualizar, listar e remover usuários
- Sistema de login
- Efetuar a compra, devendo solicitar do cliente a forma de pagamento

Em um projeto real, esses requisitos seriam garimpados com o passar do tempo junto ao cliente, para desenvolver o sistema de forma incremental e iterativa, utilizando métodos ágeis como Scrum (curso PM-83) e práticas XP (curso PM-87).

2.2 PARTES DO PROJETO

Para facilitar o desenvolvimento, vamos separar em tarefas o que devemos fazer, que serão vistas a partir do próximo capítulo.

- 1) Decisão de tecnologias
- 2) Montagem do ambiente de desenvolvimento
- 3) CRUD produto
- 4) Carrinho de compras
- 5) CRUD usuário
- 6) login
- 7) Compra de produto

CAPÍTULO 3

Novo projeto

“Não há uma verdade fundamental, apenas há erros fundamentais”

– Bachelard , Gaston

Neste capítulo, vamos definir as tecnologias do projeto e vamos criar e configurar nosso projeto inicial.

3.1 DEFININDO AS TECNOLOGIAS

Nosso projeto será todo em Java e utilizará o VRaptor 3.x como framework Web. Para a persistência, usaremos o Hibernate com anotações rodando sobre o MySQL, que pode ser mudado caso desejado.

Na parte de interação com os usuários, teremos bastante Ajax. Para facilitar o uso do Ajax, vamos utilizar uma biblioteca chamada JQuery, junto com alguns de seus plugins.

Como servidor, vamos usar o Apache Tomcat pela sua popularidade, mas podemos facilmente rodar a aplicação no Jetty ou Glassfish.

SERVIDORES E JAVA EE

Para uma discussão mais detalhada sobre servidores de aplicação Java EE e servlet containers, consulte os capítulos iniciais da apostila do curso **FJ-21**.

INTERNET EXPLORER 6

Os CSSs, Javascripts e HTMLs usados no curso foram desenvolvidos para o Firefox 3.5+ sem se preocupar com compatibilidade com outros browsers, então algumas funcionalidades podem não funcionar em outros browsers, principalmente no Internet Explorer 6.

Ao desenvolver um sistema você deveria se preocupar com a compatibilidade entre os browsers, e você pode saber mais sobre isso no curso **WD-43 | Desenvolvimento Web com HTML, CSS e JavaScript**.

Resumindo as tecnologias:

Java

- Hibernate Core - JBoss
- Hibernate Annotations - JBoss
- VRaptor - Caelum
- JSTL - Sun
- Tomcat - Apache
- MySQL - Sun

Ajax

- JQuery - JQuery

3.2 MONTANDO O AMBIENTE DE DESENVOLVIMENTO

Para o ambiente de desenvolvimento, vamos utilizar o Eclipse com o plugin WTP. A versão que utilizaremos no curso será a mais nova, a 3.6, que possui o codinome Helios.

Para utilizar o Eclipse, basta fazer o download, descompactar o zip baixado e depois executá-lo. Isso é uma grande vantagem em relação a programas que exigem a instalação. O Eclipse é a IDE líder de mercado. Formada por um consórcio liderado pela IBM, possui seu código livre. Para fazer o download, acesse o site do Eclipse e baixe a distribuição "*Eclipse IDE for Java EE Developers*".

<http://www.eclipse.org/downloads>

Vamos também configurar o Tomcat para rodar nossa aplicação.

3.3 EXERCÍCIOS: MONTANDO O AMBIENTE DE DESENVOLVIMENTO

- 1) Na Caelum, execute o Eclipse pelo ícone que está em seu Desktop. Em casa, faça a instalação como citado anteriormente.
- 2) Descompacte o arquivo `apache-tomcat-X.X.X.zip` em seu Desktop. O arquivo está na pasta **Caelum/28** em seu Desktop.

Em casa, você pode fazer o download do Tomcat 6.x em: <http://tomcat.apache.org/>

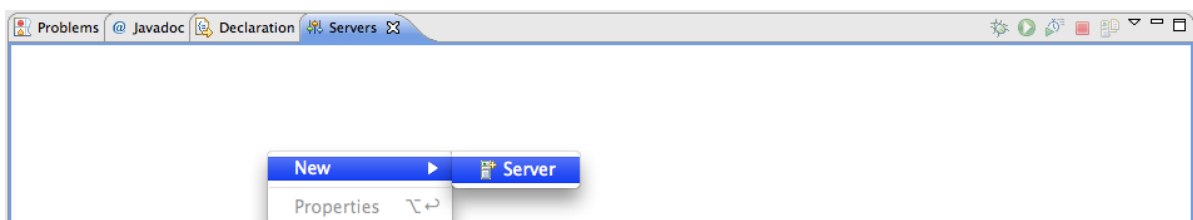
- 3) No Eclipse, abra a view **Servers**. Para tal, você pode ir em *Window* e depois *Show View* e escolher a opção *Servers*.

Uma outra maneira de fazer isso, e acessar qualquer tela ou menu, é através da pragmática tecla de atalho **Ctrl+3**. Depois digite *servers*.

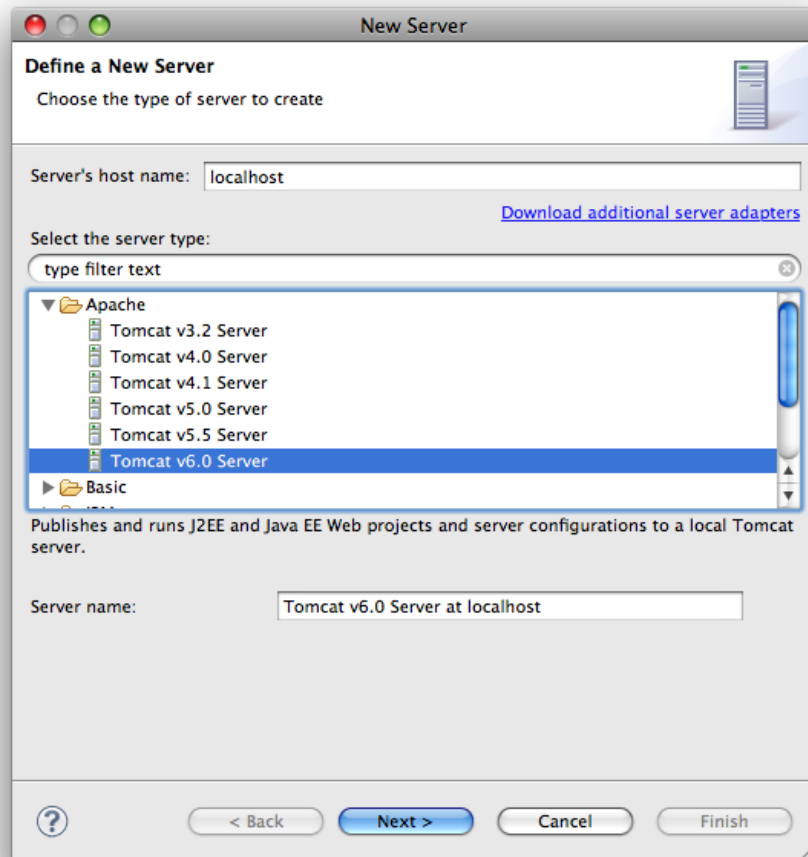


- 4) Configure o Tomcat seguindo os passos abaixo:

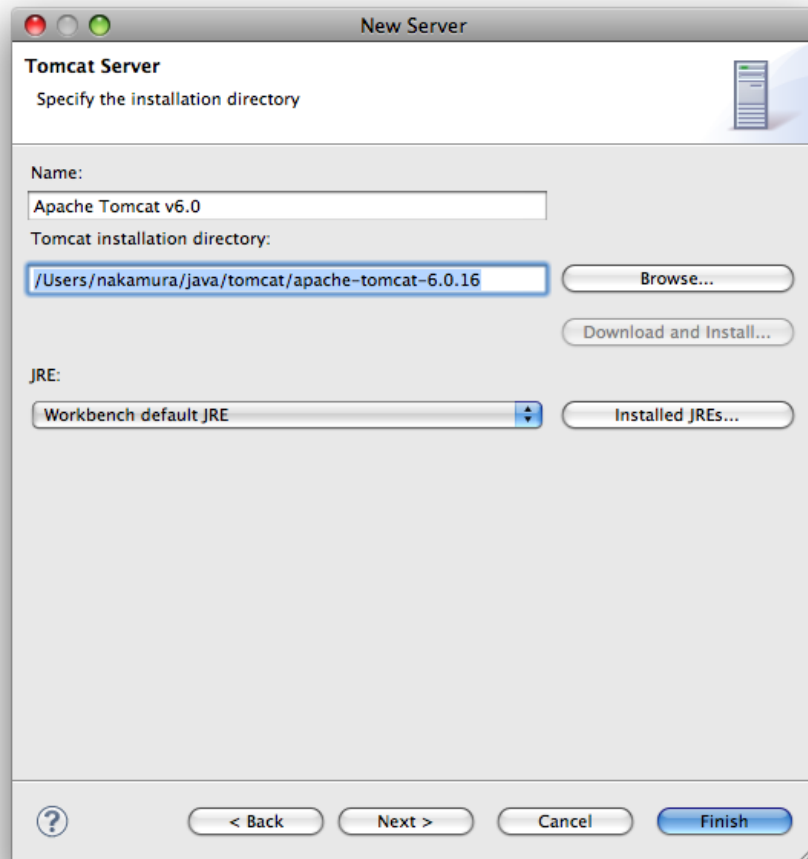
- Botão direito na view *Servers*
- Escolha a opção *New*
- Escolha a opção *Server*



- 5) Escolha Tomcat v6.0 Server, dentro da pasta Apache

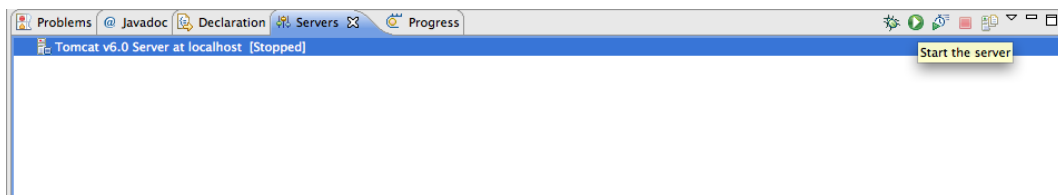


- 6) Clique no botão **Browse**, e indique o local onde foi feito o unzip do Tomcat (a pasta do Tomcat no seu Desktop):

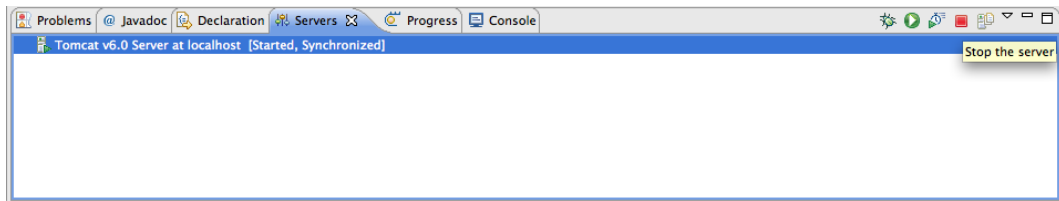


Clique em Finish.

- 7) Teste o servidor que acabamos de configurar. Na view Servers, selecione o Tomcat e clique no botão de iniciar:



- 8) Verifique se o log está limpo e se nenhuma mensagem de erro foi impressa. É muito importante, durante todo o curso, observar esse log para verificar se o servidor subiu sem erros e se tudo está rodando normalmente.
- 9) Agora que o Tomcat está configurado, vamos pará-lo um pouco para criar nosso projeto. Na view Servers, selecione o Tomcat e clique no botão de parar.



- 10) **Caso você esteja na Caelum siga essas instruções:** Vá na pasta Caelum/28 e abra o arquivo goodbuy.zip. Extraia o conteúdo desse arquivo para o Desktop.

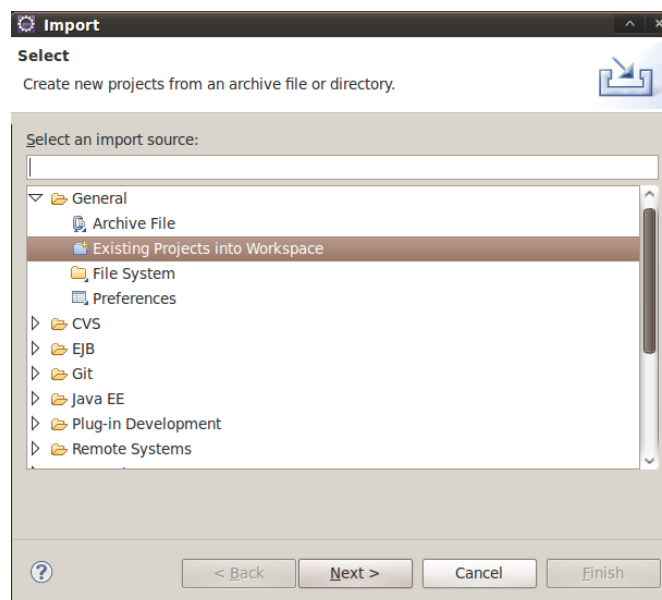
Caso você não esteja na Caelum e precise criar o projeto de casa:

Basta você baixar do site do VRaptor o *vraptor-blank-project*, descompacta-lo e importá-lo no Eclipse. Depois disso, adicione, dentro da pasta WebContent o conteúdo do zip que contém cabeçalho, rodapé, css e javascript básicos que usaremos durante o projeto. Você pode baixar em:

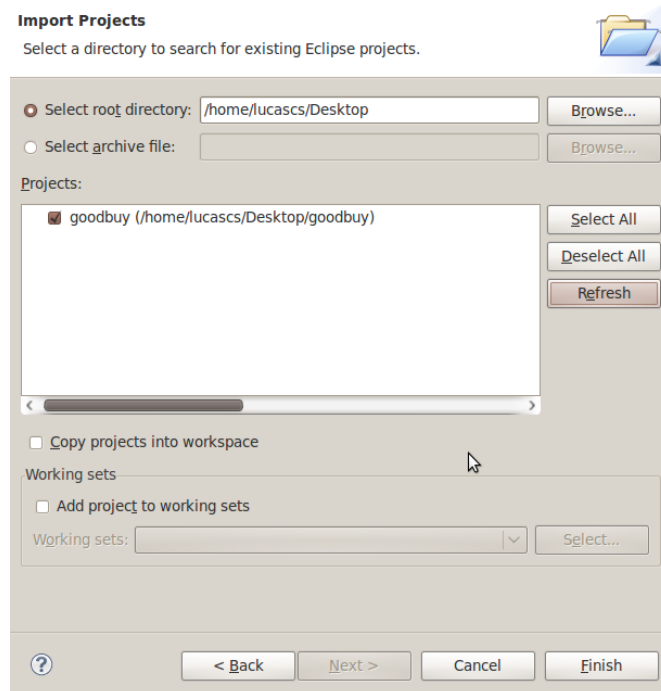
<http://www.caelum.com.br/download/caelum-java-web-vraptor-hibernate-ajax-fj28-auxiliar.zip>

- 11) Importe o projeto no eclipse seguindo os passos:

- 1) Vá em File
- 2) Escolha a opção Import
- 3) Escolha a opção General >> Import existing projects into workspace
- 4) Clique em Next

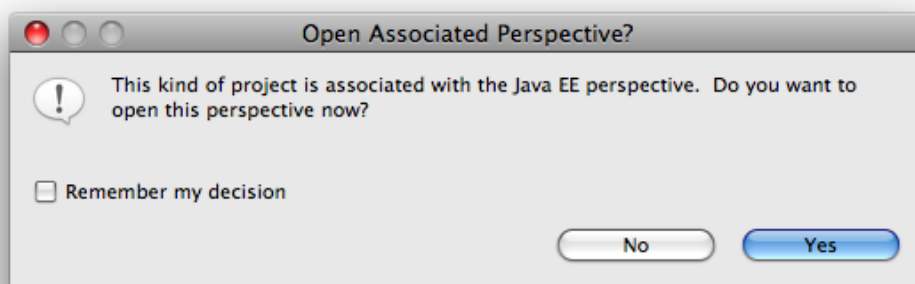


- 12) No Select root directory, selecione o Desktop e clique em Finish:



Este projeto já possui as configurações básicas do VRaptor e do Hibernate, para que não percamos muito tempo configurando. Mas essas configurações usadas serão explicadas com mais detalhe durante o curso.

- 13) O Eclipse perguntará se você quer trocar de perspectiva. Não será necessário, então vamos escolher a opção No.



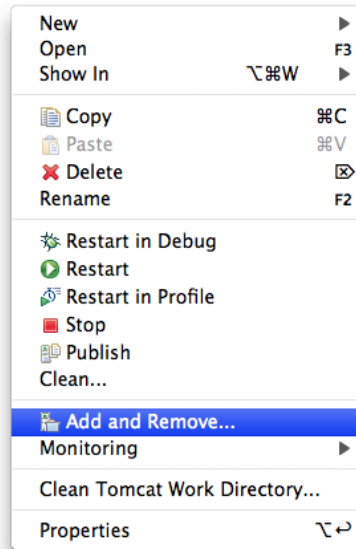
ESTRUTURA DE PASTAS DE UM PROJETO WEB

Para entender a estrutura de pastas dos projetos Web em Java, consulte o capítulo de servlets da apostila do curso FJ-21.

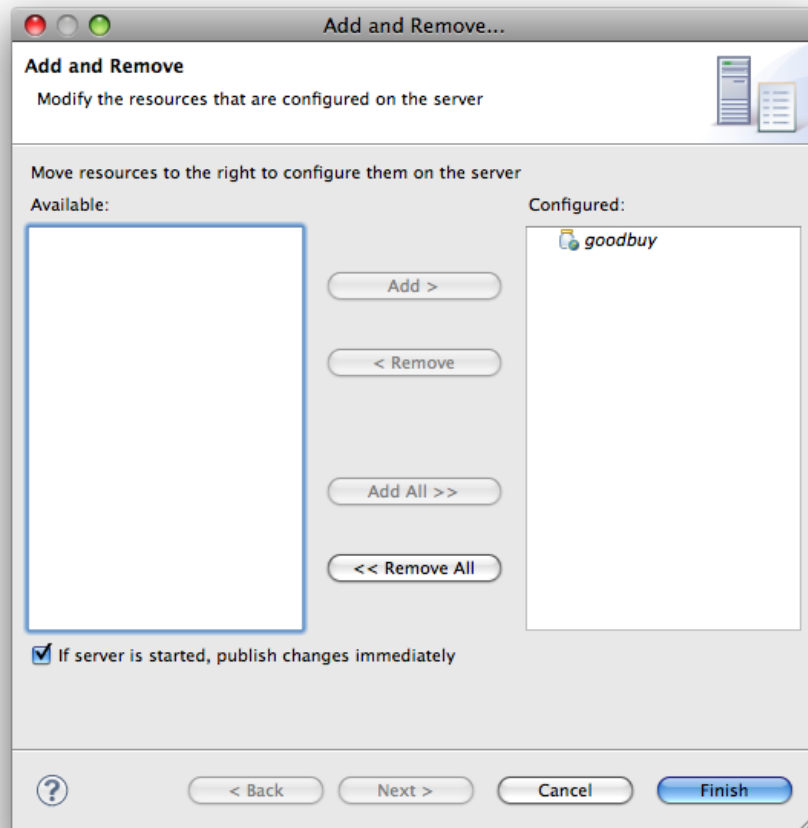
3.4 EXERCÍCIOS: CONFIGURANDO O DEPLOY DA APLICAÇÃO

1) Na aba de Servers, selecione o Tomcat que configuramos no exercício anterior e siga os passos abaixo:

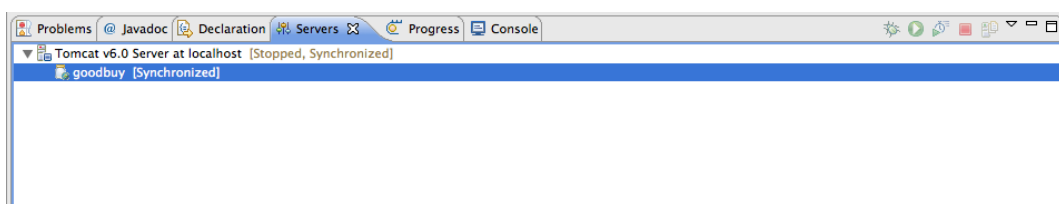
- Clique com o botão direito
- Selecione Add and Remove...



2) Passe o nosso projeto goodbuy para a direita, usando o botão Add All, e depois clique em Finish.

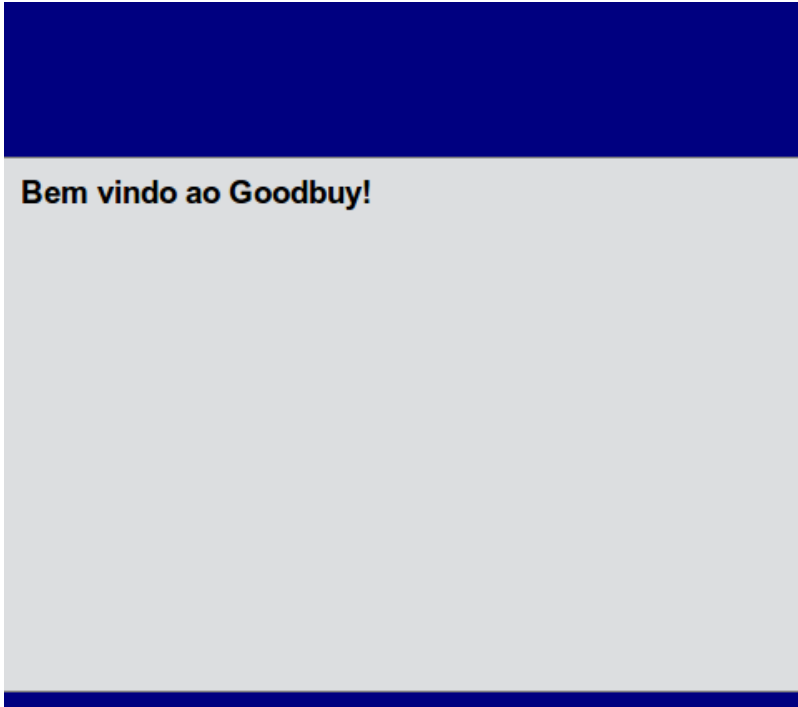


3) O projeto ficará associado ao Tomcat, conforme a figura abaixo.



Pronto! Nosso projeto está preparado para ser executado no servidor.

4) Inicie o servidor clicando com o botão direito e selecionando Start. Abra o firefox e digite na barra de endereços: <http://localhost:8080/goodbuy> .



Bem vindo ao Goodbuy!

Persistindo os dados com o Hibernate

4.1 A CAMADA DE PERSISTÊNCIA DO PROJETO

Nossa primeira tarefa será cadastrar, atualizar, listar e remover produtos do sistema. Para realizá-la, precisamos gravar as informações dos produtos no banco de dados. Para esse curso, utilizaremos o banco de dados MySQL, um banco de dados gratuito, de código aberto, simples e muito utilizado por diversas empresas.

Para maiores informações e para fazer o download, acesse a URL:

<http://www.mysql.com/>

Depois de instalado e configurado, o MySQL pode ser acessado com o usuário chamado **root** e a senha vazia (sem senha). Para testar, basta abrir o terminal e digitar o comando:

```
mysql -u root
```

Esse comando tenta fazer o login no mysql, informando o usuário **root** (-u root) e omitindo a senha, já que esse usuário não tem senha.

Para nos comunicarmos com o banco de dados, seja para uma consulta ou para algum tipo de alteração de dado, podemos usar a API que o Java SE disponibiliza, o JDBC. Um dos problemas do JDBC, quando usado diretamente, conforme mostramos no curso FJ-21, é que ele é muito trabalhoso além de que precisamos gerenciar muitos recursos importantes. Todas as informações têm de ser passadas uma a uma.

Para evitar ter que ficar fazendo todas as chamadas ao JDBC e ganharmos tempo e produtividade, vamos utilizar um framework para a persistência que já traz muitas funcionalidades já implementadas. O framework que utilizaremos será o Hibernate e ele será o responsável por fazer as chamadas à API do JDBC.

Vale lembrar que, tudo que é visto aqui no curso, é aplicável a outros frameworks de persistência, como seria o caso de usar o iBatis ou então o EclipseLink através de JPA. É muito importante perceber como vamos integrar todas as nossas ferramentas de maneira coesa e desacoplada.

4.2 SOBRE O HIBERNATE

O Hibernate é um framework **ORM - Object Relational Mapping**. É uma ferramenta que nos ajuda a persistir objetos Java em um banco de dados relacional. O trabalho do desenvolvedor é definir como os objetos são mapeados nas tabelas do banco e o Hibernate faz todo o acesso ao banco, gerando inclusive os comandos SQL necessários.

O Hibernate é um projeto opensource do grupo JBoss com muitos anos de história e liderança no mercado Java. Recentemente, boa parte das idéias do Hibernate e outros frameworks ORM foram padronizadas em uma especificação oficial do Java, a **JPA - Java Persistence API**. A JPA é uma especificação do JCP e possui várias implementações (o Hibernate, o Oracle Toplink, EclipseLink, OpenJPA etc).

MAIS SOBRE HIBERNATE

Neste curso, veremos muitos tópicos sobre Hibernate e como integrá-lo em nosso projeto com VRaptor3. Para tópicos avançados e mais detalhes, consulte o treinamento FJ-25:

<http://www.caelum.com.br/curso/fj25>

4.3 PREPARANDO O HIBERNATE

Para preparar o Hibernate, será necessário baixar dois ZIPs do site do Hibernate. Cada ZIP representa um projeto diferente.

O primeiro será o **Hibernate Core**, que se chama `hibernate-distribution-XXX.zip`. O segundo ZIP será do projeto **Hibernate Annotations**, já que queremos configurar nosso projeto com as anotações da JPA. Esse ZIP chama-se `hibernate-annotations-XXX.zip`. Faça os downloads diretamente em:

<http://www.hibernate.org>

Depois de fazer o download desses dois zips, basta descompactá-los e utilizar os JAR's que estão dentro de cada projeto. No exercício abaixo veremos quais JARs vamos precisar. A partir do Hibernate 3.5, esses jars todos vem numa única distribuição, a *core*.

CONFIGURANDO O MECANISMO DE LOGGING

O Hibernate utiliza o SL4J - Simple Logging Facade for Java - para o mecanismo de logging. Ele serve apenas como uma fachada para vários frameworks de logging, por exemplo o Log4J e o `java.util.logging`.

No nosso projeto utilizaremos o Log4J, então precisamos baixar mais dois zips. O primeiro é o Log4J, do grupo Apache. O segundo é o próprio SL4J, que contém o jar que faz o binding do SL4J com o Log4J.

CONFIGURANDO O DRIVER DO MySQL

Apesar de toda facilidade que o Hibernate nos trará, por baixo dos panos ele faz chamadas ao JDBC. Portanto, para que nossa aplicação se conecte no banco de dados, precisamos do driver do MySQL no classpath. O driver é a implementação JDBC que será utilizada. Cada banco de dados possui a sua implementação, e ela deve ser baixada do site do próprio banco de dados. Faça o download do *mysql jdbc driver*.

MAIS SOBRE JDBC E DRIVERS

Para mais informações sobre o JDBC e como funcionam os Drivers, consulte o capítulo inicial da apostila do curso FJ-21.

4.4 CONFIGURANDO O HIBERNATE

Para configurar o Hibernate, podemos utilizar ou um arquivo *.properties* ou um arquivo XML.

O arquivo de properties é mais simples, mais fácil de se criar, mas uma das desvantagens é que ele não consegue configurar tudo que queremos, por exemplo as entidades. As entidades têm que ser configuradas no código, e as outras informações no arquivo de properties.

Já o XML, por mais que seja um pouco mais difícil em relação ao properties, permite que toda a configuração seja feita nele. Por isso faremos nossa configuração no XML. O arquivo XML que o Hibernate procurará será o *hibernate.cfg.xml* e ele deve estar no classpath.

Para nosso caso, vamos seguir a convenção e criar o arquivo *hibernate.cfg.xml* na pasta *src*, dentro do nosso projeto. O conteúdo do arquivo será esse:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.connection.password">
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/fj28
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
  </session-factory>
</hibernate-configuration>
```



```
</property>
<property name="hibernate.dialect">
    org.hibernate.dialect.MySQL5InnoDBDialect
</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<property name="show_sql">true</property>
<property name="format_sql">true</property>
</session-factory>
</hibernate-configuration>
```

As configurações que passamos nesse arquivo são parecidas quando queremos nos conectar a um banco de dados. Para conectar em um banco de dados, precisamos informar qual é o usuário, a senha, algumas informações do banco, como host, porta, etc.

Um detalhe importante da nossa configuração é o banco de dados que foi passado. Na configuração `hibernate.connection.url` foi passado o nome do database que utilizaremos. Para esse caso escolhemos o database `fj28`.

Abaixo segue a descrição de todas as configurações que usamos.

- **hibernate.connection.username** - usuário do banco de dados
- **hibernate.connection.password** - senha do usuário
- **hibernate.connection.url** - chamada de URL ou string de conexão, deve ser configurada de acordo com documentação do banco de dados
- **hibernate.connection.driver_class** - driver que deve ser utilizado
- **hibernate.hbm2ddl.auto** - como o hibernate vai se comportar em relação às tabelas do banco. Com o valor `update` ele vai criar ou modificar tabelas sempre que necessário.
- **hibernate.dialect** - dialeto a ser utilizado para a comunicação com o banco de dados
- **show_sql** - flag que indica se os SQLs gerados devem ser impressos
- **format_sql** - flag que indica se os SQLs devem ser formatados

Existem muitas outras configurações possíveis. Para maiores detalhes, acesse a documentação do Hibernate.

4.5 EXERCÍCIOS: CONFIGURANDO O BANCO

- 1) Todos os jars necessários para usar o hibernate já estão no projeto base que usamos. O `hibernate.cfg.xml` também já existe na pasta `src`. Caso você esteja fazendo essa apostila em casa, baixe os JARs de acordo com as orientações desse capítulo e crie o `hibernate.cfg.xml` similarmente.

- 2) Conecte-se no banco de dados para verificar se o database fj28 existe. Se ele não existir, teremos que criá-lo. Primeiro, abra um terminal pelo ícone no Desktop. Depois digite:

```
mysql -u root
```

Depois de conectado, vamos verificar os bancos de dados disponíveis:

```
mysql> show databases;
```

Se o database fj28 aparecer, digite:

```
mysql> drop database fj28;
```

```
mysql> create database fj28;
```

CAPÍTULO 5

Cadastrando Produtos

No capítulo anterior, configuramos o Hibernate para acessar o banco de dados. Neste capítulo, nosso objetivo é cadastrar produtos.

5.1 MODELANDO UM PRODUTO

Agora que estamos com o ambiente configurado e com o banco de dados pronto, podemos partir para a codificação.

A primeira classe que faremos será a classe `Produto`. Ela representará os produtos que serão cadastrados no sistema, ou seja, quando algum produto for colocado no sistema, nós criaremos uma instância dessa classe.

Nossa classe `Produto` ficará assim:

```
public class Produto {  
  
    private String nome;  
  
    private String descricao;  
  
    private Double preco;  
  
    // getter e setters  
}
```

BIGDECIMAL

Em aplicações normais, não se deve usar doubles para representar números reais (ex. dinheiro, porcentagem) por causa de problemas de arredondamento. Podemos usar o `BigDecimal` que tem uma precisão fixa, portanto os problemas de arredondamento são facilmente contornáveis. Não usaremos `BigDecimal` pois ele é um pouco mais complicado de usar:

```
double a, b, c, d;  
d = a+b/c;
```

com `BigDecimal`:

```
BigDecimal a, b, c, d;  
d = a.add(b.divide(c));
```

5.2 ANOTANDO A CLASSE Produto

A classe `Produto` é, por enquanto, uma classe Java normal. Mas precisamos que instâncias dessa classe sejam persistidas ou recuperadas do banco de dados.

Uma das maneiras de transformar essa classe Java normal em uma classe que pode ser persistida é através de XML. O problema dessa abordagem é a dificuldade e o trabalho de se fazer isso com XML.

Outra maneira, que está sendo bem aceita pela comunidade é o uso de anotações para a configuração. Ao invés de usar um arquivo XML para configurar, basta ir na própria classe e anotá-la. Uma das grandes vantagens, além da simplicidade, é que as anotações passam pelo processo de compilação normal, assim como qualquer outro código Java. Ou seja, a classe com as anotações deve compilar.

Para anotar a classe `Produto`, basta colocar na declaração da classe a anotação `@Entity`, do pacote `javax.persistence`. Essa anotação pertence a especificação da Java Persistence API.

```
@Entity  
public class Produto {  
  
    private String nome;  
  
    private String descricao;  
  
    private Double preco;
```

```
// getter e setters  
}
```

Quando anotamos uma classe com `@Entity`, devemos indicar qual será o campo de chave primária. Para o nosso caso, vamos criar um campo `id` para ser a chave primária da tabela.

Para indicar que o campo `id` será a chave primária, utilizaremos a anotação `@Id`. Um detalhe importante é que o nome da anotação não é devido ao nome do atributo, ou seja, o atributo e a anotação tem o nome `id`, mas a anotação sempre será `@Id`, já o atributo de chave primária pode ser qualquer nome.

Várias vezes não queremos passar o valor do `id` porque o valor desse campo será gerado no banco de dados. Podemos informar isso pro Hibernate através da anotação `@GeneratedValue`. Para o nosso caso, esse campo será do tipo `auto_increment` no banco de dados.

```
@Entity  
public class Produto {  
  
    @Id @GeneratedValue  
    private Long id;  
  
    private String nome;  
  
    private String descricao;  
  
    private Double preco;  
  
    // getter e setters  
}
```

Pronto, agora nossa entidade está devidamente anotada. Só temos que fazer mais uma coisa: avisar o Hibernate dessa entidade, isso porque ele não encontra a entidade automaticamente. Para informar isso ao Hibernate, utilizaremos o arquivo de configuração dele, o `hibernate.cfg.xml`.

O conteúdo do arquivo ficará assim:

```
<hibernate-configuration>  
    <session-factory>  
        <!-- todas as propriedades configuradas anteriormente -->  
  
        <!-- entidades -->  
        <mapping class="br.com.caelum.goodbuy.modelo.Produto" />  
    </session-factory>  
</hibernate-configuration>
```

Repare que temos que utilizar o nome completo da classe, ou seja, o nome da classe mais o pacote.

5.3 EXERCÍCIOS - MODELANDO O PRODUTO

- 1) Crie a classe Produto dentro do pacote `br.com.caelum.goodbuy.modelo`.

```
public class Produto {  
  
    private Long id;  
  
    private String nome;  
  
    private String descricao;  
  
    private Double preco;  
  
    // getter e setters  
}
```

- 2) Anote a classe com as anotações da JPA.

```
@Entity  
public class Produto {  
  
    @Id @GeneratedValue  
    private Long id;  
  
    private String nome;  
  
    private String descricao;  
  
    private Double preco;  
  
    // getter e setters  
}
```

- 3) Configure essa entidade no Hibernate, alterando o arquivo `hibernate.cfg.xml`.

```
<!DOCTYPE hibernate-configuration PUBLIC  
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
    <session-factory>  
        <!-- todas as propriedades configuradas anteriormente -->  
        <!-- -->  
  
        <!-- entidades -->  
        <mapping class="br.com.caelum.goodbuy.modelo.Produto" />  
    </session-factory>  
</hibernate-configuration>
```

```
</session-factory>  
</hibernate-configuration>
```

5.4 ADICIONANDO UM PRODUTO NO BANCO

Para adicionar um produto no banco de dados, precisamos primeiro nos conectar no banco de dados. Quando usamos o JDBC, utilizamos a interface `java.sql.Connection`, que representa uma conexão com o banco de dados.

Mas no nosso caso, que estamos usando o Hibernate, precisamos usar diretamente essa conexão do JDBC? Não, deixaremos isso para o Hibernate. Temos que lembrar que sempre que quisermos fazer algo no banco de dados, utilizaremos o Hibernate.

Para nos comunicarmos com o banco de dados utilizando o Hibernate, precisamos pedir para o Hibernate um objeto que encapsule uma conexão, e com essa conexão, ele faça o que queremos. Esse objeto que precisamos é a `Session`, do pacote `org.hibernate`.

Esse objeto é muito importante para nós porque é ele quem vai encapsular todas as chamadas JDBC, facilitando muito o mecanismo de acesso ao banco de dados.

Nosso primeiro passo para conseguir uma `Session` é instanciar uma classe que representa a configuração do Hibernate, portanto iniciamos instanciando a classe do Hibernate cujo nome é `org.hibernate.cfg.AnnotationConfiguration`.

```
// Cria uma configuração  
AnnotationConfiguration configuration = new AnnotationConfiguration();
```

Depois de criar esse objeto, vamos chamar o método responsável por ler o arquivo `hibernate.cfg.xml`, o `configure()`.

```
// Lê o hibernate.cfg.xml  
configuration.configure();
```

Após configurar, precisamos criar um objeto que cria as sessões, uma fábrica. Esse objeto é uma `SessionFactory`, que pode ser obtida com:

```
SessionFactory factory = configuration.buildSessionFactory();
```

Com a `SessionFactory` em mãos conseguimos criar uma `Session`. O código completo fica:

```
1 public class TesteDeSessao {  
2     public static void main(String[] args) {  
3         AnnotationConfiguration configuration = new AnnotationConfiguration();
```

```
4         configuration.configure();
5
6         SessionFactory factory = configuration.buildSessionFactory();
7         Session session = factory.openSession();
8     }
9 }
```

A partir desse objeto session, basta passarmos uma entidade que ele se encarregará de transformar do modelo orientado a objetos para o modelo relacional.

```
Produto produto = new Produto();
produto.setNome("Prateleira");
produto.setDescricao("Uma prateleira para colocar livros");
produto.setPreco(35.90);
```

```
Transaction tx = session.beginTransaction();
session.save(produto);
tx.commit();
```

Note que utilizamos uma transação para salvar o produto. Essa transação é do pacote `org.hibernate`, e ela é necessária para que o insert seja efetivado.

5.5 EXERCÍCIOS

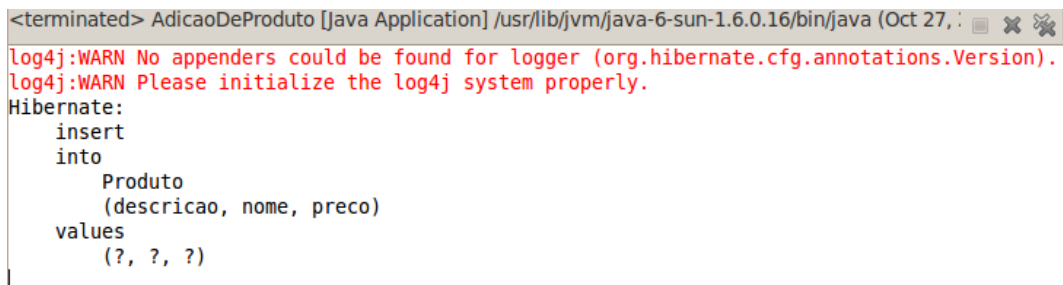
1) Crie a seguinte classe, que adiciona produtos no banco.

```
1 package br.com.caelum.goodbuy.testes;
2
3 import org.hibernate.Session;
4 import org.hibernate.SessionFactory;
5 import org.hibernate.Transaction;
6 import org.hibernate.cfg.AnnotationConfiguration;
7
8 public class AdicaoDeProduto {
9     public static void main(String[] args) {
10         AnnotationConfiguration configuration = new AnnotationConfiguration();
11         configuration.configure();
12
13         SessionFactory factory = configuration.buildSessionFactory();
14         Session session = factory.openSession();
15
16         Produto produto = new Produto();
17         produto.setNome("Prateleira");
```



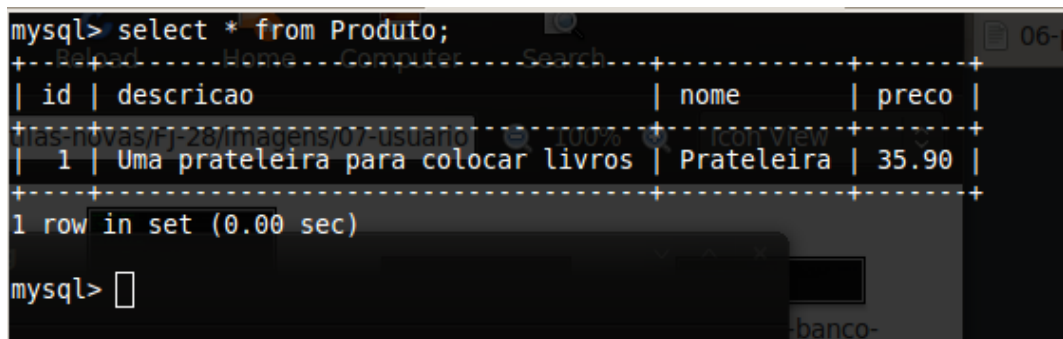
```
18     produto.setDescricao("Uma prateleira para colocar livros");
19     produto.setPreco(35.90);
20
21     Transaction tx = session.beginTransaction();
22     session.save(produto);
23     tx.commit();
24 }
25 }
```

2) Execute a classe que adiciona o produto. Repare no console o sql que foi gerado.



```
<terminated> AdicaoDeProduto [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.16/bin/java (Oct 27, :
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate:
insert
into
    Produto
(descricao, nome, preco)
values
    (?, ?, ?)
```

3) Acesse o banco de dados e verifique se o produto foi inserido com sucesso.



```
mysql> select * from Produto;
+----+-----+-----+-----+
| id | descricao | nome | preco |
+----+-----+-----+-----+
| 1  | Uma prateleira para colocar livros | Prateleira | 35.90 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> 
```

4) (Opcional) Crie outros produtos no sistema.

5.6 OUTRAS OPERAÇÕES COM PRODUTO

Para adicionarmos um produto no banco de dados, utilizamos o método `save()` da sessão. A sessão do Hibernate possui também métodos de atualização, remoção e busca, e a utilização desses métodos é bem parecida com o que fizemos. Para alterar um produto, basta carregarmos esse produto, alterarmos e depois atualizarmos.

```
1 public class AlteracaoDeProduto {
2     public static void main(String[] args) {
3         AnnotationConfiguration configuration = new AnnotationConfiguration();
```

```
4      configuration.configure();
5
6      SessionFactory factory = configuration.buildSessionFactory();
7      Session session = factory.openSession();
8
9      // carrega o produto do banco de dados
10     Produto produto = (Produto) session.load(Produto.class, 1L);
11
12     Transaction tx = session.beginTransaction();
13     produto.setPreco(42.50);
14     session.update(produto);
15     tx.commit();
16 }
17 }
```

E para remover um produto:

```
1 public class RemocaoDeProduto {
2     public static void main(String[] args) {
3         AnnotationConfiguration configuration = new AnnotationConfiguration();
4         configuration.configure();
5
6         SessionFactory factory = configuration.buildSessionFactory();
7         Session session = factory.openSession();
8
9         // carrega o produto do banco de dados
10        Produto produto = (Produto) session.load(Produto.class, 1L);
11
12        Transaction tx = session.beginTransaction();
13        session.delete(produto);
14        tx.commit();
15    }
16 }
```

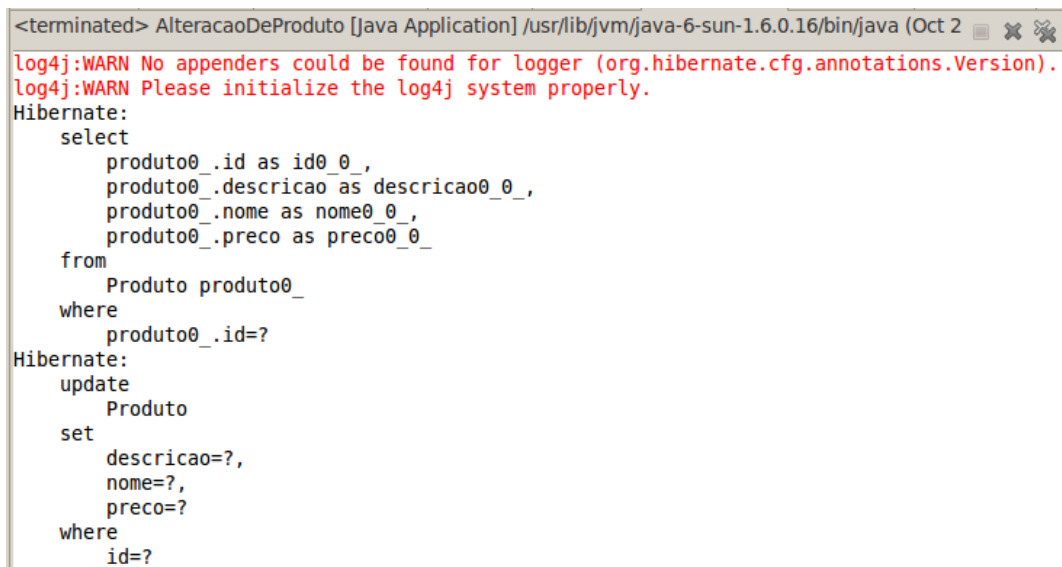
5.7 EXERCÍCIOS - OUTRAS OPERAÇÕES COM PRODUTO

1) Crie a classe de alteração do produto.

```
1 package br.com.caelum.goodbuy.testes;
2
3 public class AlteracaoDeProduto {
4     public static void main(String[] args) {
5         AnnotationConfiguration configuration = new AnnotationConfiguration();
6         configuration.configure();
```

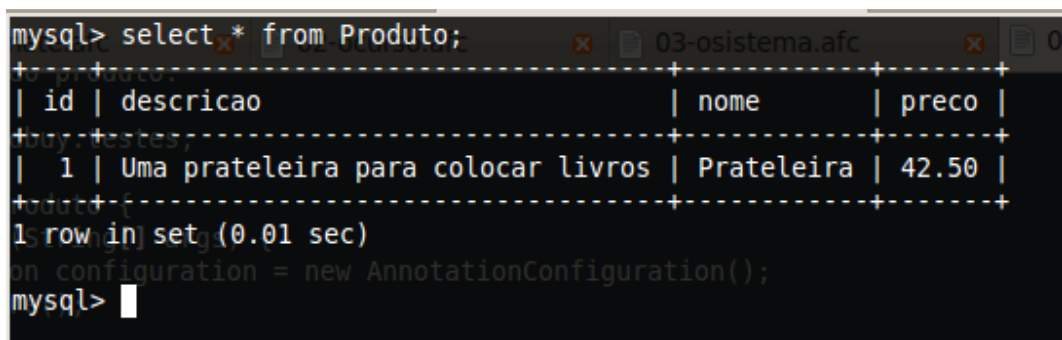
```
7
8     SessionFactory factory = configuration.buildSessionFactory();
9     Session session = factory.openSession();
10
11     // carrega o produto do banco de dados
12     Produto produto = (Produto) session.load(Produto.class, 1L);
13
14     Transaction tx = session.beginTransaction();
15     produto.setPreco(42.50);
16     session.update(produto);
17     tx.commit();
18 }
19 }
```

2) Execute a classe que altera o produto. Repare no console o sql que foi gerado.



```
<terminated> AlteracaoDeProduto [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.16/bin/java (Oct 2
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate:
select
    produto0_.id as id0_0_,
    produto0_.descricao as descricao0_0_,
    produto0_.nome as nome0_0_,
    produto0_.preco as preco0_0_
from
    Produto produto0_
where
    produto0_.id=?
Hibernate:
update
    Produto
set
    descricao=?,
    nome=?,
    preco=?
where
    id=?
```

3) Acesse o banco de dados e verifique se o produto foi alterado com sucesso.

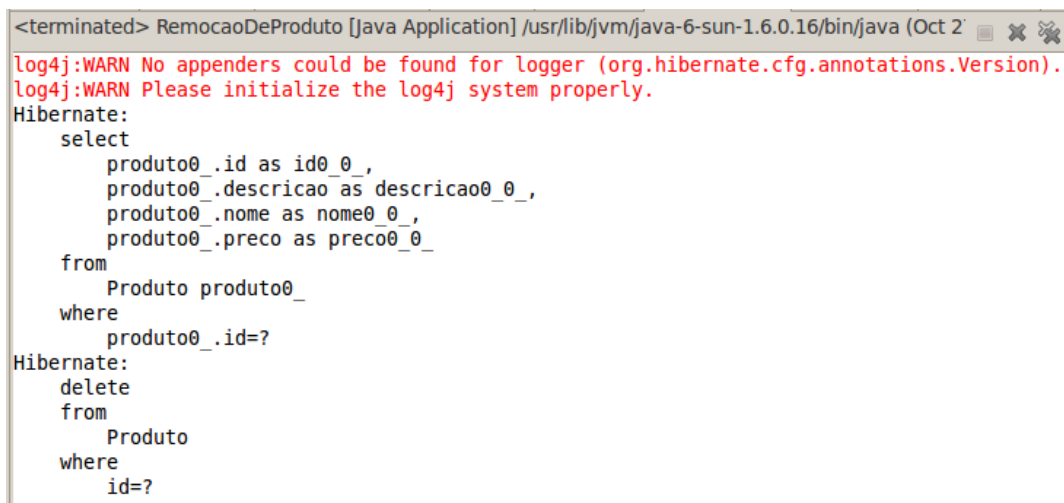


```
mysql> select * from Produto;
+-----+-----+-----+-----+
| id | descricao | nome | preco |
+-----+-----+-----+-----+
| 1 | Uma prateleira para colocar livros | Prateleira | 42.50 |
+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

4) Crie a classe de remoção do produto.

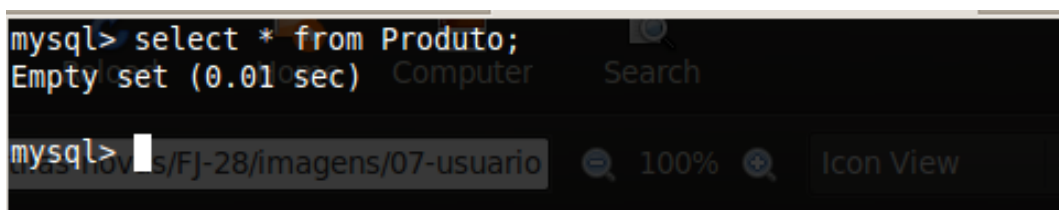
```
1 package br.com.caelum.goodbuy.testes;
2
3 public class RemocaoDeProduto {
4     public static void main(String[] args) {
5         AnnotationConfiguration configuration = new AnnotationConfiguration();
6         configuration.configure();
7
8         SessionFactory factory = configuration.buildSessionFactory();
9         Session session = factory.openSession();
10
11         // carrega o produto do banco de dados
12         Produto produto = (Produto) session.load(Produto.class, 1L);
13
14         Transaction tx = session.beginTransaction();
15         session.delete(produto);
16         tx.commit();
17     }
18 }
```

5) Execute a classe que remove o produto. Repare no console o sql que foi gerado.



```
<terminated> RemocaoDeProduto [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.16/bin/java (Oct 2
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate:
  select
    produto0_.id as id0_0_,
    produto0_.descricao as descricao0_0_,
    produto0_.nome as nome0_0_,
    produto0_.preco as preco0_0_
  from
    Produto produto0_
  where
    produto0_.id=?
Hibernate:
  delete
  from
    Produto
  where
    id=?
```

6) Acesse o banco de dados e verifique se o produto foi alterado com sucesso.



```
mysql> select * from Produto;
Empty set (0.01 sec)

mysql>
```

5.8 DISCUSSÃO EM SALA - MAIS SOBRE O HIBERNATE

- Por que utilizamos o Long e não o long para o campo id
- “Eu sempre fiz com que minhas entidades implementassem Serializable. É mesmo necessário?”
- Vale mesmo a pena utilizar o Hibernate? Não vou perder performance?

Refatorando

6.1 ANALISANDO O CÓDIGO ATUAL

Se olharmos para o nosso projeto agora, temos a seguinte situação:

- Tabela de produtos está no banco de dados
- Hibernate configurado
- Classes de testes para lidar com produtos

Mas será que com o que temos agora podemos cadastrar produtos, conforme o cliente pediu?

Repare na última parte da pergunta, que é muito importante. Se entregarmos o que temos agora para o cliente, ele dirá que isso é inútil e nosso sistema não tem nada ainda.

O que faltou, que é muito importante, é integrar a parte visual (web) com as regras de negócios.

Mas repare que o jeito que estão nossas classes que lidam com um produto no banco de dados. Todo nosso código está em classes de testes, dentro do método `main()`. Se deixarmos dessa maneira ficaria muito difícil de utilizar esse código.

Então o que precisamos fazer é melhorar nosso código, para que possamos utilizar o que foi escrito dentro do `main()`, e também para melhorar a manutenção e a legibilidade.

6.2 REFACTORING

Uma prática bastante comum e difundida no meio da Orientação a Objetos é a chamada Refatoração (Refactoring). Refatorar um programa é melhorar seu código sem alterar sua funcionalidade. A ideia da refatoração não é corrigir bugs, por exemplo, mas melhorar a estrutura de seu código, deixá-lo mais OO, mais legível.

Há diversos tipos de refatorações. Renomear uma variável para um nome mais claro é um exemplo simples. Quebrar um método grande em vários métodos menores é um outro exemplo um pouco mais complicado.

Várias IDEs de Java possuem suporte à refatorações comuns. O Eclipse possui ótimas opções automáticas que utilizaremos nesse curso.

O livro mais famoso sobre refatorações foi escrito por Martin Fowler e chama-se *Refactoring - Improving the Design of existing code*. É um catálogo com dezenas de técnicas de refatoração e instruções de como e quando executá-las.

6.3 APRENDENDO A REFATORAR

Vamos abrir novamente a classe de inserção de produtos.

```
package br.com.caelum.goodbuy.testes;

// import's

public class AdicaoDeProduto {

    public static void main(String[] args) {
        AnnotationConfiguration configuration = new AnnotationConfiguration();
        configuration.configure();

        SessionFactory factory = configuration.buildSessionFactory();
        Session session = factory.openSession();

        Produto produto = new Produto();
        produto.setNome("Prateleira");
        produto.setDescricao("Uma prateleira para colocar livros");
        produto.setPreco(35.90);

        Transaction tx = session.beginTransaction();
        session.save(produto);
        tx.commit();
    }
}
```

Olhando para o método main, podemos dividi-lo em três partes. Uma parte é a aquisição de uma Session, outra é a criação do produto, e a outra é adição do produto no banco de dados. Com base nessa divisão, vamos pedir para o Eclipse nos ajudar a refatorar, para fazer essa divisão de uma maneira segura e automática.

Apenas para ficar claro essa divisão, veja o código novamente, agora com os comentários mostrando onde começa cada parte.

```
1 package br.com.caelum.goodbuy.testes;
2
3 // import's
4
5 public class AdicaoDeProduto {
6
7     public static void main(String[] args) {
8         // Aquisição da sessão
9         AnnotationConfiguration configuration = new AnnotationConfiguration();
10        configuration.configure();
11
12        SessionFactory factory = configuration.buildSessionFactory();
13        Session session = factory.openSession();
14
15        // Criação do produto
16        Produto produto = new Produto();
17        produto.setNome("Prateleira");
18        produto.setDescricao("Uma prateleira para colocar livros");
19        produto.setPreco(35.90);
20
21        // Adição do produto no banco de dados
22        Transaction tx = session.beginTransaction();
23        session.save(produto);
24        tx.commit();
25    }
26 }
```

Selecione apenas a parte da aquisição da sessão, vamos utilizar as teclas de atalho do Eclipse para criar um método com essas instruções. As teclas de atalho são **Alt + Shift + M**, que representa a opção **Extract Method**.

Apertando essas teclas, o Eclipse mostrará uma tela perguntando o nome do método que será criado. O nome desse método será `getSession()`.

Colocando o nome do método e apertando **OK**, o método será criado e a parte do código que usa essas instruções já muda automaticamente, fazendo a chamada ao método recém-criado.

6.4 EXERCÍCIOS

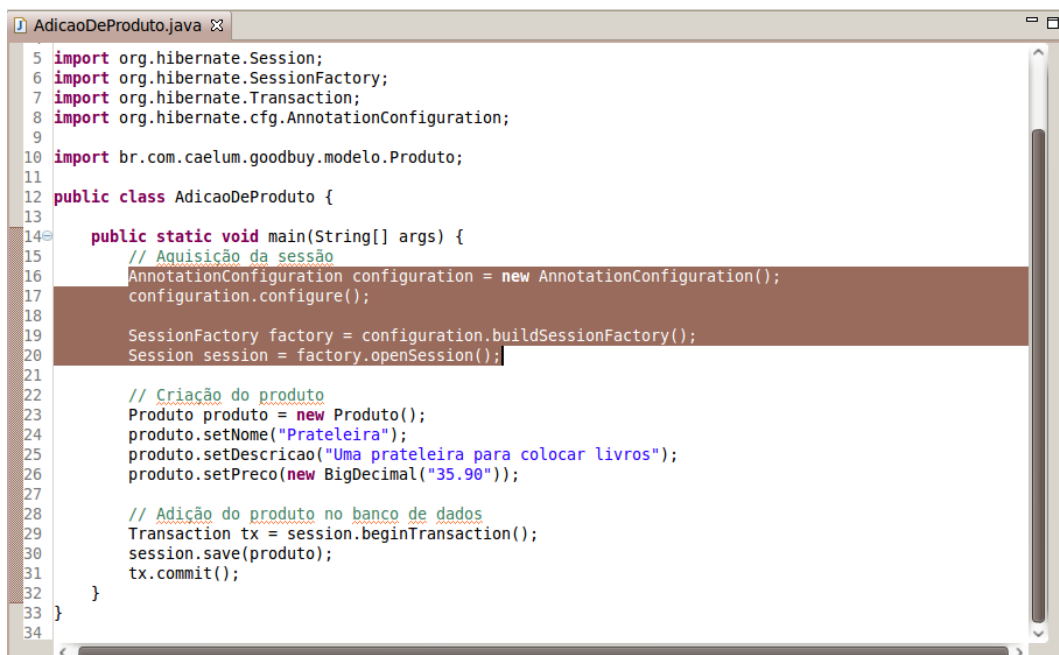
- 1) Altere a classe `AdicaoDeProduto`, que está no pacote `br.com.caelum.goodbuy.testes`, colocando os comentários para deixar claro a divisão das partes.

```
1 package br.com.caelum.goodbuy.testes;
2
```



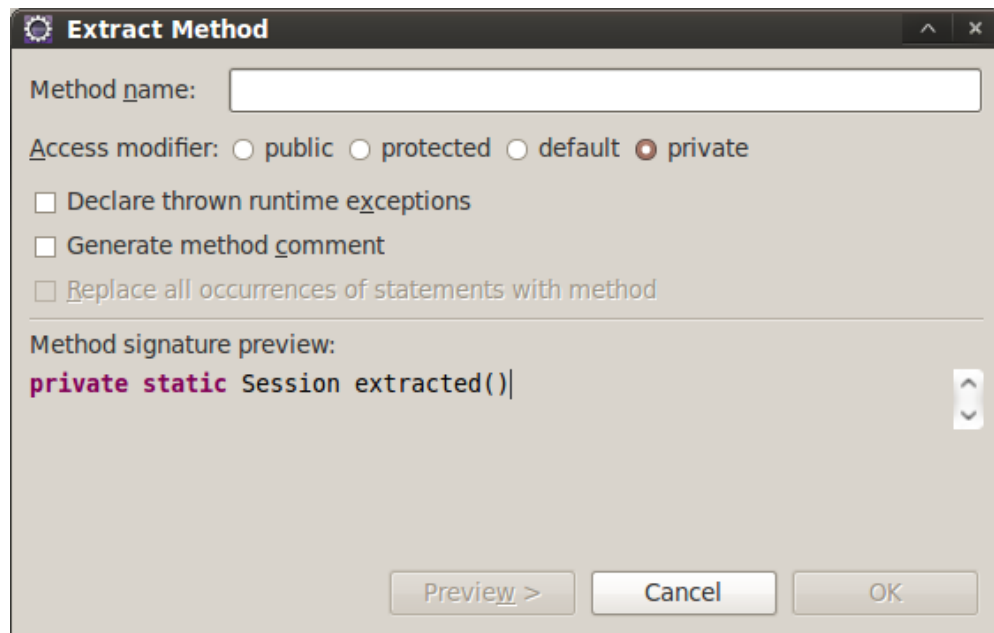
```
3 // import's
4
5 public class AdicaoDeProduto {
6
7     public static void main(String[] args) {
8         // Aquisição da sessão
9         AnnotationConfiguration configuration = new AnnotationConfiguration();
10        configuration.configure();
11
12        SessionFactory factory = configuration.buildSessionFactory();
13        Session session = factory.openSession();
14
15        // Criação do produto
16        Produto produto = new Produto();
17        produto.setNome("Prateleira");
18        produto.setDescricao("Uma prateleira para colocar livros");
19        produto.setPreco(35.90);
20
21        // Adição do produto no banco de dados
22        Transaction tx = session.beginTransaction();
23        session.save(produto);
24        tx.commit();
25    }
26 }
```

2) Selecione as linhas que fazem a aquisição da sessão, conforme a figura abaixo:

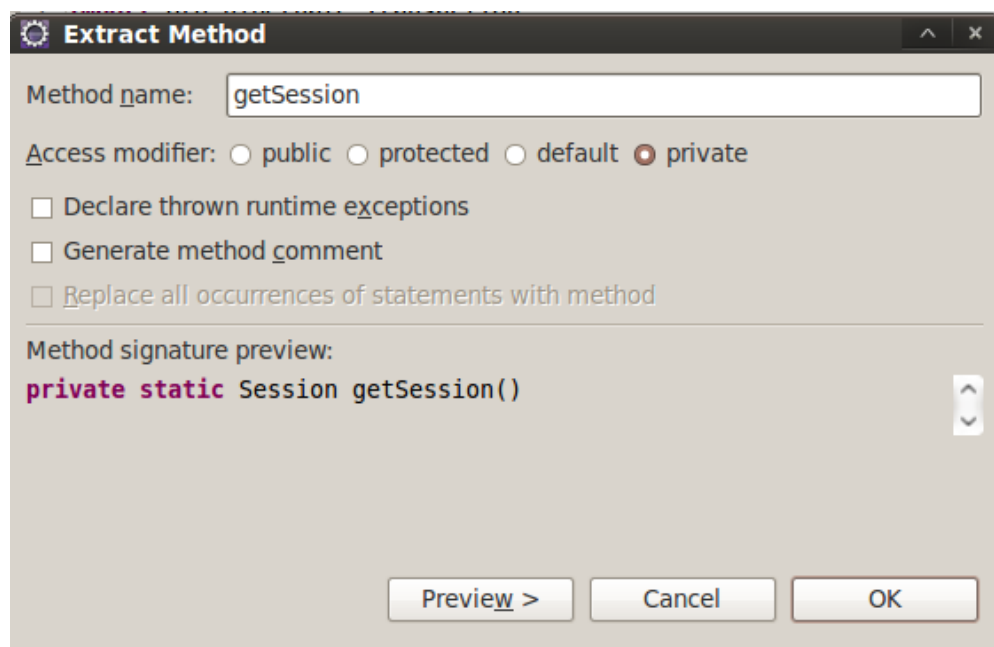


3) Com as linhas selecionadas, pressione as teclas Alt + Shift + M (Extract Method), para criar um método

com essas linhas. Após pressionar essas teclas, aparecerá a seguinte tela:



- 4) A tela que está sendo mostrada está esperando por um nome de método. Digite getSession como nome do método e pressione OK.



- 5) Note como o Eclipse mudou seu código, criando o método getSession e fazendo a chamada a este método. A classe deve estar assim:

```
public class AdicaoDeProduto {
```

```
public static void main(String[] args) {
    // Aquisição da sessão
    Session session = getSession();

    // Criação do produto
    Produto produto = new Produto();
    produto.setNome("Prateleira");
    produto.setDescricao("Uma prateleira para colocar livros");
    produto.setPreco(35.90);

    // Adição do produto no banco de dados
    Transaction tx = session.beginTransaction();
    session.save(produto);
    tx.commit();
}

private static Session getSession() {
    AnnotationConfiguration configuration = new AnnotationConfiguration();
    configuration.configure();

    SessionFactory factory = configuration.buildSessionFactory();
    Session session = factory.openSession();
    return session;
}
```

- 6) Faça o mesmo para as linhas de criação de produto e adição de produto no banco de dados. No final, ficaremos com 3 métodos, um chamado getSession, que já foi feito, outro chamado criaProduto e outro chamado gravaProduto. Lembre-se de utilizar as teclas de atalho que vimos nesse exercício.
- 7) Para conferência, ficaremos com a classe da seguinte forma:

```
1 package br.com.caelum.goodbuy.testes;
2
3 // import's
4
5 public class AdicaoDeProduto {
6
7     public static void main(String[] args) {
8         // Aquisição da sessão
9         Session session = getSession();
10
11         // Criação do produto
12         Produto produto = criaProduto();
13
14         // Adição do produto no banco de dados
```

```
15         gravaProduto(session, produto);
16     }
17
18     private static void gravaProduto(Session session, Produto produto) {
19         Transaction tx = session.beginTransaction();
20         session.save(produto);
21         tx.commit();
22     }
23
24     private static Produto criaProduto() {
25         Produto produto = new Produto();
26         produto.setNome("Prateleira");
27         produto.setDescricao("Uma prateleira para colocar livros");
28         produto.setPreco(35.90);
29         return produto;
30     }
31
32     private static Session getSession() {
33         AnnotationConfiguration configuration = new AnnotationConfiguration();
34         configuration.configure();
35
36         SessionFactory factory = configuration.buildSessionFactory();
37         Session session = factory.openSession();
38         return session;
39     }
40 }
```

6.5 COMENTÁRIOS SÃO SEMPRE NECESSÁRIOS?

Note que o que fizemos foi quebrar um método grande em vários métodos pequenos. Pode parecer inútil esse tipo de alteração, já que não mudamos nada no código, só a forma como está organizado.

Mas repare que o método `main` ficou muito mais legível, intuitivo. Basta dar uma olhada rápida que podemos entender o que ele faz.

Mas agora que o método `main` foi “quebrado” em métodos menores, será que precisamos mesmo de comentários? Olhe novamente para o código, e veja que os nomes dos nossos métodos já dizem muito o que queremos, então nem precisamos mais de comentários.

Já que não precisamos mais dos comentários, vamos retirá-los do nosso código:

```
public class AdicaoDeProduto {

    public static void main(String[] args) {
```

```
    Session session = getSession();

    Produto produto = criaProduto();

    gravaProduto(session, produto);
}
// outros métodos
}
```

6.6 REFATORANDO PARA CRIAR OS DAOs

Olhando novamente para a classe de adição de produtos, podemos notar que ainda não podemos reaproveitar o código de criação de sessão do Hibernate em outras classes. O mesmo acontece com a classe de remoção de usuários.

Podemos então criar uma classe que vai ser responsável pela obtenção de Sessions, para que possamos usá-la sempre que precisarmos de uma. Vamos dar a essa classe o nome de CriadorDeSession, e mover o método getSession() para ela.

Da mesma forma, podemos juntar as operações de adicionar, remover, atualizar produtos em uma classe, que será responsável por acessar os produtos no banco de dados. E a essa classe vamos dar o nome de ProdutoDao.

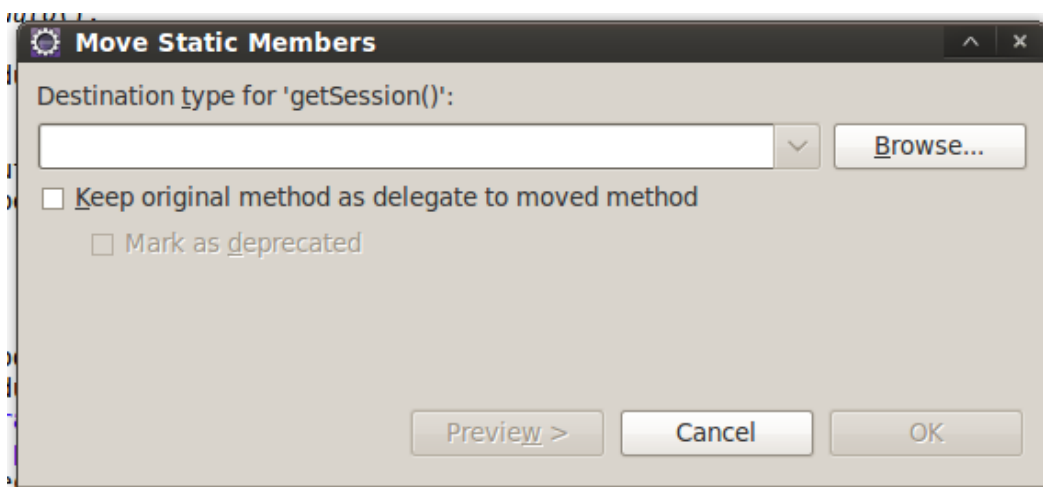
6.7 EXERCÍCIOS

- 1) Crie uma classe chamada CriadorDeSession no pacote `br.com.caelum.goodbuy.infra`. Essa classe será a responsável por criar uma Session.
- 2) Selecione o método getSession da classe AdicaoDeProduto.

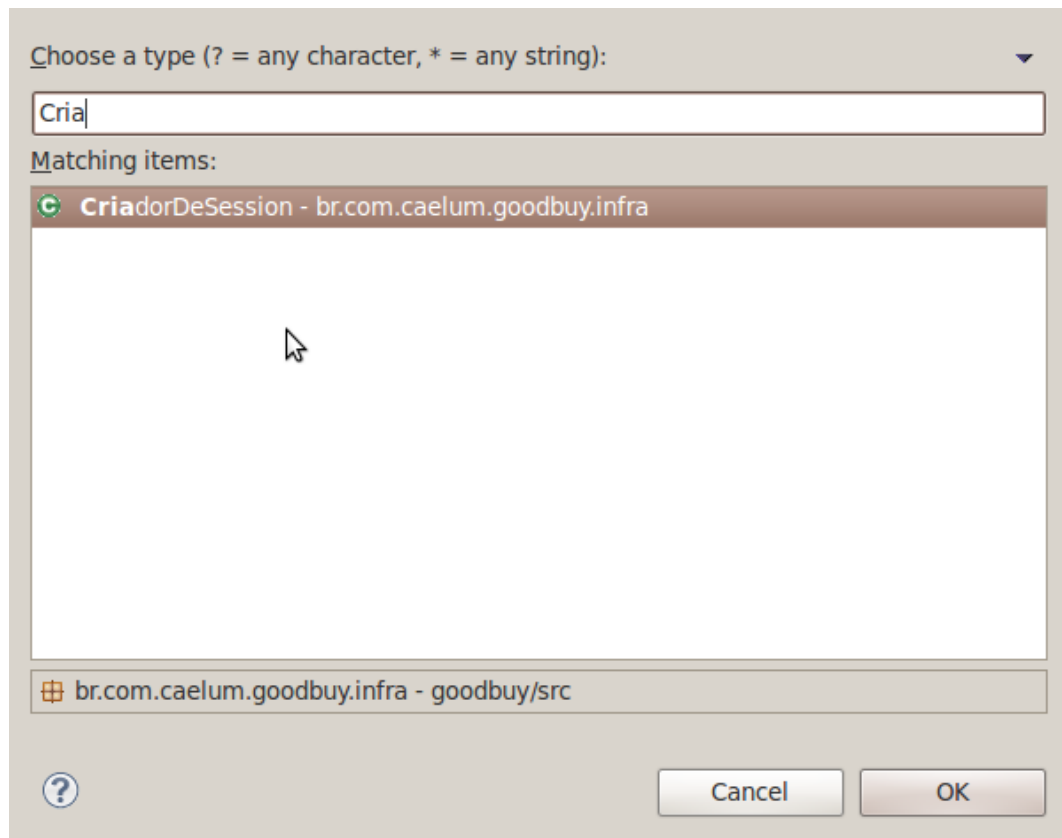
```

11
12 public class AdicaoDeProduto {
13
14     public static void main(String[] args) {
15         Session session = getSession();
16
17         Produto produto = criaProduto();
18
19         gravaProduto(session, produto);
20     }
21
22     private static void gravaProduto(Session session, Produto produto) {
23         Transaction tx = session.beginTransaction();
24         session.save(produto);
25         tx.commit();
26     }
27
28     private static Produto criaProduto() {
29         Produto produto = new Produto();
30         produto.setNome("Prateleira");
31         produto.setDescricao("Uma prateleira para colocar livros");
32         produto.setPreco(new BigDecimal("35.90"));
33         return produto;
34     }
35
36     private static Session getSession() {
37         AnnotationConfiguration configuration = new AnnotationConfiguration();
38         configuration.configure();
39
40         SessionFactory factory = configuration.buildSessionFactory();
41         Session session = factory.openSession();
42         return session;
43     }
44 }
  
```

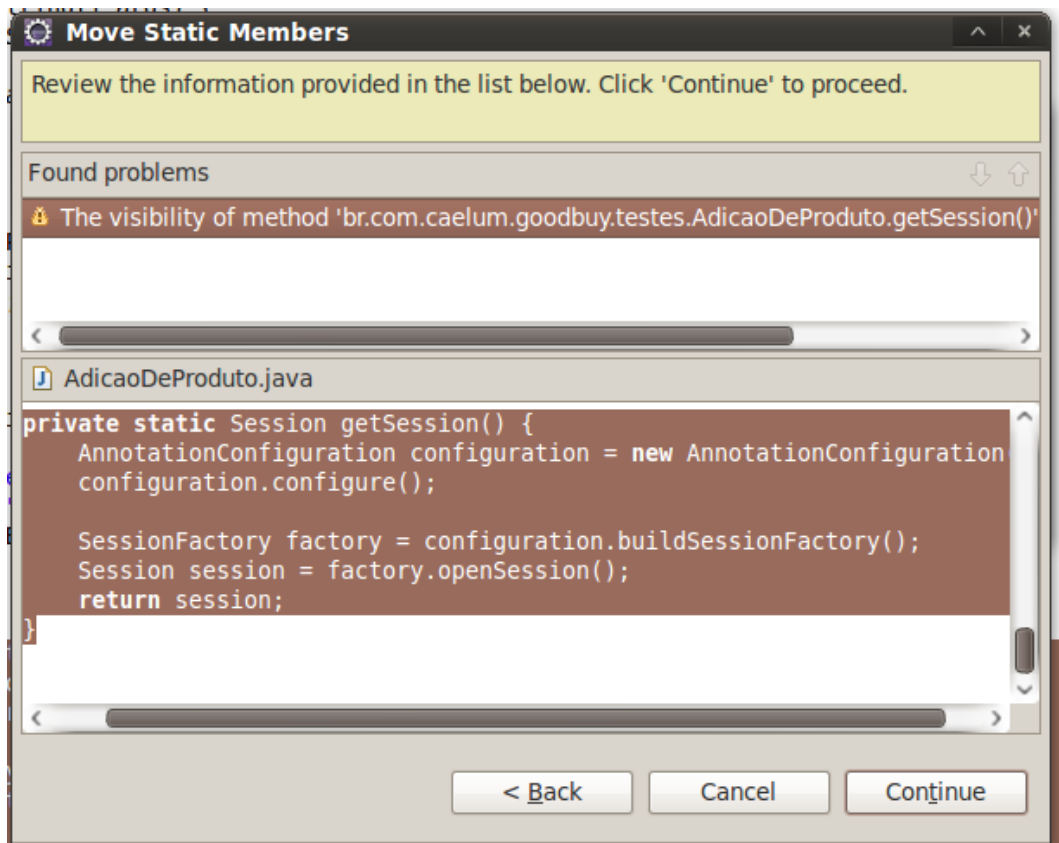
- 3) Com as linhas seleccionadas, pressione as teclas **Alt + Shift + V** (Move Method), para criar um método com essas linhas. Após pressionar essas teclas, aparecerá a seguinte tela:



- 4) A tela que está sendo mostrada está esperando pela classe que ficará com esse método. Pressione o botão **Browse...**, e busque pela classe **CriadorDeSession**, conforme a imagem abaixo:



- 5) Após encontrar a classe `CriadorDeSession`, pressione o botão OK para escolher essa classe, depois OK novamente para confirmar a mudança do método para essa classe.
- 6) Aparecerá uma tela avisando que a visibilidade do método `getSession` será alterada `public`. Confirme essa alteração pressionando o botão Continue.



7) Repare como ficou a classe após essa alteração:

```
public class AdicaoDeProduto {  
  
    public static void main(String[] args) {  
        Session session = CriadorDeSession.getSession();  
  
        Produto produto = criaProduto();  
  
        gravaProduto(session, produto);  
    }  
  
    private static void gravaProduto(Session session, Produto produto) {  
        Transaction tx = session.beginTransaction();  
        session.save(produto);  
        tx.commit();  
    }  
  
    private static Produto criaProduto() {  
        Produto produto = new Produto();  
        produto.setNome("Prateleira");  
        produto.setDescricao("Uma prateleira para colocar livros");  
    }  
}
```



```
        produto.setPreco(35.90);  
        return produto;  
    }  
}
```

- 8) Crie uma classe chamada ProdutoDao no pacote `br.com.caelum.goodbuy.dao`. Essa classe será a responsável por encapsular as chamadas ao Hibernate.
- 9) Mova o método `gravaProduto` para a classe `ProdutoDao`. Utilize as teclas de atalho que vimos nesse exercício.
- 10) Repare como ficou a classe após essa alteração:

```
public class AdicaoDeProduto {  
  
    public static void main(String[] args) {  
        Session session = CriadorDeSession.getSession();  
  
        Produto produto = criaProduto();  
  
        ProdutoDao.gravaProduto(session, produto);  
    }  
  
    private static Produto criaProduto() {  
        Produto produto = new Produto();  
        produto.setNome("Prateleira");  
        produto.setDescricao("Uma prateleira para colocar livros");  
        produto.setPreco(35.90);  
        return produto;  
    }  
}
```

- 11) Vamos refatorar agora a classe `ProdutoDao`, que está no momento assim:

```
public class ProdutoDao {  
  
    public static void gravaProduto(Session session, Produto produto) {  
        Transaction tx = session.beginTransaction();  
        session.save(produto);  
        tx.commit();  
    }  
  
}
```

Primeiro vamos tirar o `static` do método `gravaProduto`.

```
public class ProdutoDao {
```

```
    public void gravaProduto(Session session, Produto produto) {  
        Transaction tx = session.beginTransaction();  
        session.save(produto);  
        tx.commit();  
    }  
  
}
```

- 12) A classe AdicaoDeProduto agora não compila. Vamos trocar o acesso estático por acesso a uma instância do dao:

```
public class AdicaoDeProduto {  
  
    public static void main(String[] args) {  
        Session session = CriadorDeSession.getSession();  
  
        Produto produto = criaProduto();  
  
        new ProdutoDao().gravaProduto(session, produto);  
    }  
  
    //...  
}
```

- 13) O método gravaProduto recebe uma Session. Mas o ProdutoDao deveria ser responsável pelo acesso a dados, então ela mesma deve ser responsável por criar a session. Mude isso no ProdutoDao:

```
public class ProdutoDao {  
  
    public void gravaProduto(Produto produto) {  
        Session session = CriadorDeSession.getSession();  
        Transaction tx = session.beginTransaction();  
        session.save(produto);  
        tx.commit();  
    }  
  
}
```

Mude também a classe AdicaoDeProduto:

```
public class AdicaoDeProduto {  
  
    public static void main(String[] args) {  
  
        Produto produto = criaProduto();  
  
        new ProdutoDao().gravaProduto(produto);  
    }  
}
```

```
    }  
    //...  
}
```

- 14) Por último, não precisamos criar uma Session dentro do método gravaProduto, vamos fazer isso dentro do construtor da classe, e colocar a Session como atributo:

```
public class ProdutoDao {  
  
    private final Session session;  
  
    public ProdutoDao() {  
        this.session = CriadorDeSession.getSession();  
    }  
  
    public void gravaProduto(Produto produto) {  
        Transaction tx = session.beginTransaction();  
        session.save(produto);  
        tx.commit();  
    }  
  
}
```

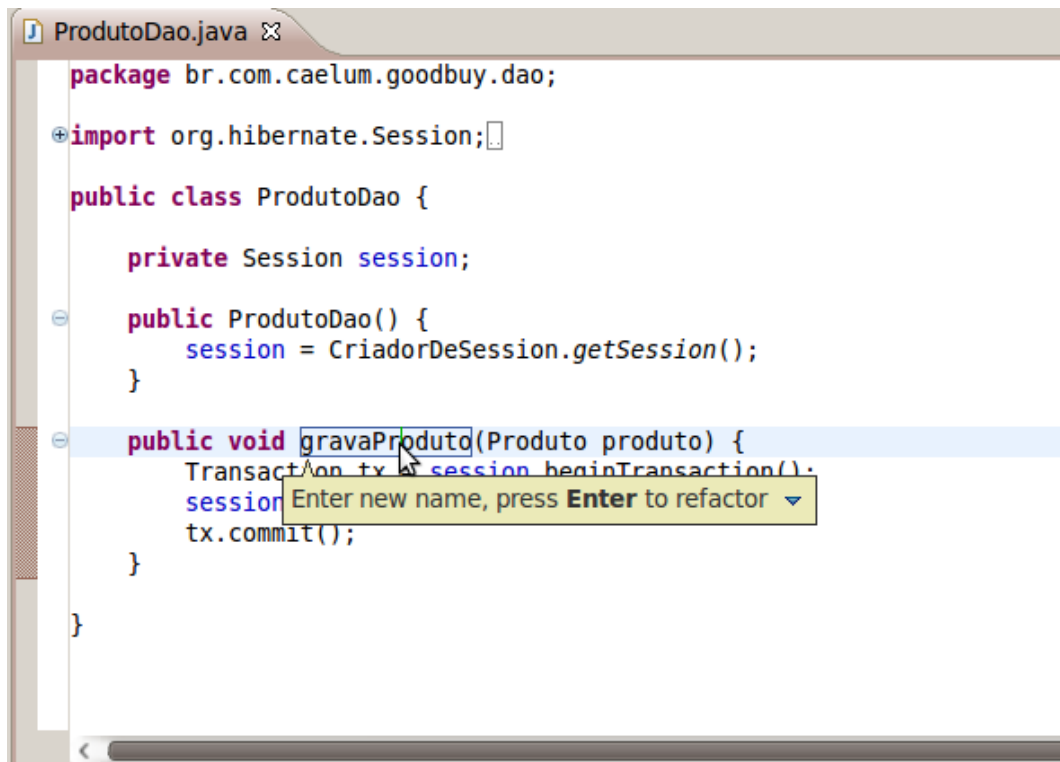
- 15) Agora a chamada do método gravaProduto está correta, porém repare como está um pouco redundante:

```
new ProdutoDao().gravaProduto(produto);
```

Seria melhor se fosse assim:

```
new ProdutoDao().salva(produto);
```

Vamos fazer essa mudança de nome de método utilizando o Eclipse. Pressione as teclas Alt + Shift + R (rename) método da classe ProdutoDao, e o Eclipse pedirá o novo nome para o método.



```
ProdutoDao.java
package br.com.caelum.goodbuy.dao;

import org.hibernate.Session;

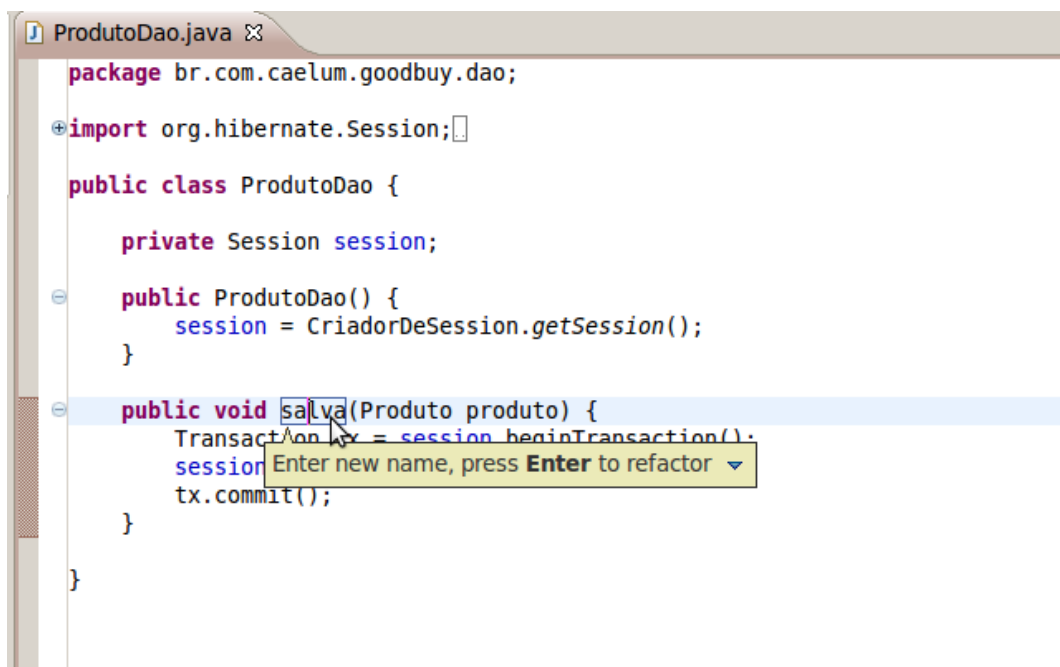
public class ProdutoDao {

    private Session session;

    public ProdutoDao() {
        session = CriadorDeSession.getSession();
    }

    public void gravaProduto(Produto produto) {
        Transaction tx = session.beginTransaction();
        session.save(produto);
        tx.commit();
    }
}
```

Digite salva como novo nome do método.



```
ProdutoDao.java
package br.com.caelum.goodbuy.dao;

import org.hibernate.Session;

public class ProdutoDao {

    private Session session;

    public ProdutoDao() {
        session = CriadorDeSession.getSession();
    }

    public void salva(Produto produto) {
        Transaction tx = session.beginTransaction();
        session.save(produto);
        tx.commit();
    }
}
```

- 16) (Opcional) Refatore as outras classes de teste, `AlteracaoDeProduto` e `RemocaoDeProduto`, para utilizar o `ProdutoDao`.

6.8 DISCUSSÃO EM SALA

- Ouvi falar do padrão Repository. Ainda preciso dos daos?
- O eclipse ajuda no refactoring?

VRaptor

7.1 SOBRE O VRAPTOR

VRaptor 3 é um framework MVC para web focado no desenvolvimento ágil.

Através da inversão de controle e injeção de dependências, ele diminui drasticamente o tempo de trabalho que seria perdido com o código repetitivo: validações, conversões, direcionamentos, ajax e lookups.

7.2 COMO INSTALAR

Para utilizar o VRaptor, basta fazer o download da última versão do VRaptor, e depois fazer o unzip.

Após fazer o unzip, abra a pasta que foi criada e copie os jar's que estão em `lib/mandatory` para a pasta `WEB-INF/lib` do seu projeto, além do `vraptor-3.X.X.jar`. Também é necessário escolher um container de injeção de dependências (falaremos sobre eles mais pra frente no curso), entre o Spring, Google Guice ou Pico container, adicionando todos os jars da pasta `lib/containers/<container>`.

7.3 COMO CONFIGURAR

Uma das grandes vantagens do VRaptor é que ele usa muitas convenções, ou seja, ao invés de termos que configurar tudo, como por exemplo no Struts, não precisamos configurar quase nada.

Seguindo as convenções que o VRaptor usa, podemos criar apenas nossas lógicas (código java) sem precisar escrever xml's ou properties.

A única configuração que precisamos fazer é configurar o controlador do VRaptor. Essa configuração é feita dentro do arquivo `web.xml`, o arquivo de configuração da aplicação.

```
<web-app ...>
  <!-- configura o controlador do VRaptor -->
  <filter>
    <filter-name>vraptor</filter-name>
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
</web-app>
```

Essas configurações já foram feitas no projeto base.

SERVLET 3.0

Se você estiver usando um Servlet Container que implementa a Servlet 3.0 você nem precisa da configuração do filtro no web.xml. Basta colocar o jar do VRaptor na pasta WEB-INF/lib e ele será configurado automaticamente.

7.4 PRIMEIRO EXEMPLO COM O VRAPTOR

Vamos criar nosso primeiro exemplo utilizando o VRaptor. Esse primeiro exemplo será bem simples, para entendermos o funcionamento do VRaptor. Veremos mais adiante exemplos mais complexos e mais elaborados.

Nosso primeiro exemplo será composto de uma classe, que representará as regras de negócio, e uma página de resultado.

Vamos começar pela regra de negócio, um método dentro de uma classe java simples.

```
1 package br.com.caelum.goodbuy;
2
3 public class Mundo {
4
5     public void boasVindas() {
6         System.out.println("olá mundo!");
7     }
8
9 }
```

Repare que essa classe não herda nem implementa nenhuma interface. Essa classe é um exemplo de um POJO. Isso é muito importante porque a classe está bem simples, bem legível.

Agora como fazer para chamar essa lógica? Queremos chamá-la assim pelo browser:

`http://localhost:8080/goodbuy/mundo/boasVindas`

Para chamar essa lógica, temos que anotar nossa classe que contém a regra de negócio, para indicar para o controlador do VRaptor que essa classe deve ser controlada por ele.

A anotação que utilizaremos para indicar isso é a `@Resource`.

Nossa classe ficará assim:

```
1 package br.com.caelum.goodbuy;
2
3 import br.com.caelum.vraptor.Resource;
4
5 @Resource
6 public class Mundo {
7
8     public void boasVindas() {
9         System.out.println("olá mundo!");
10    }
11
12 }
```

O VRaptor vai usar uma convenção para chamar o nosso método: para executar o método `boasVindas()` da classe `Mundo`, posso chamar no browser a URI `/mundo/boasVindas` a partir da nossa aplicação, ou seja, a URI `http://localhost:8080/goodbuy/mundo/boasVindas`.

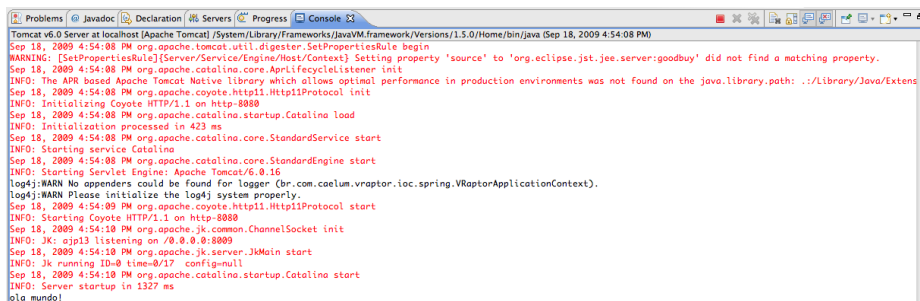
```
1 package br.com.caelum.goodbuy;
2
3 import br.com.caelum.vraptor.Resource;
4
5 @Resource
6 public class Mundo {
7
8     public void boasVindas() {
9         System.out.println("olá mundo!");
10    }
11
12 }
```


7.5 EXERCÍCIOS

- 1) Crie a classe Mundo no pacote `br.com.caelum.goodbuy`.

```
1 package br.com.caelum.goodbuy;  
2  
3 public class Mundo {  
4  
5     public void boasVindas() {  
6         System.out.println("olá mundo!");  
7     }  
8  
9 }
```

- 2) Anote a classe Mundo com `@Resource`.
- 3) Abra um browser, por exemplo o Firefox, e digite a seguinte url: `http://localhost:8080/goodbuy/mundo/boasVindas`
- 4) Ao acessar essa url, é mostrada uma página com erro 404 (página não encontrada). Isso ocorreu porque ainda não criamos a página de resultado. Ela será criada na próxima seção. O importante é verificar o log da aplicação, na view Console do Eclipse, e ver que a mensagem apareceu.



- 5) (Opcional) Crie outro método, parecido com o que fizemos, e acesse pelo browser. Se quiser, esse método pode ser em outra classe.

7.6 REDIRECIONANDO PARA UMA VIEW

Nosso último exemplo executava a lógica que tínhamos criado, mas a página de resultado não tinha sido feita, e recebíamos o erro 404.

Vamos fazer a página de resultado, para que depois de executada a lógica, essa página seja chamada. Esse é o conteúdo da página:

Olá Mundo!

Conforme foi dito anteriormente, o VRaptor prefere convenções do que configurações. A convenção de redirecionamento de páginas após a lógica é a seguinte:

```
/WEB-INF/jsp/{nomeDoResource}/{lógica}.jsp
```

Para nosso caso, temos a seguinte situação:

- {nomeDoResource} = Mundo
- {lógica} = boasVindas()

O VRaptor vai fazer algumas modificações nesses valores. A primeira letra do nome do resource, que para nós é o nome da classe, será passado para minúsculo. O resto do nome continuará igual. Já o nome da lógica continuará igual, mas sem os parênteses. Então o VRaptor vai considerar os seguintes valores:

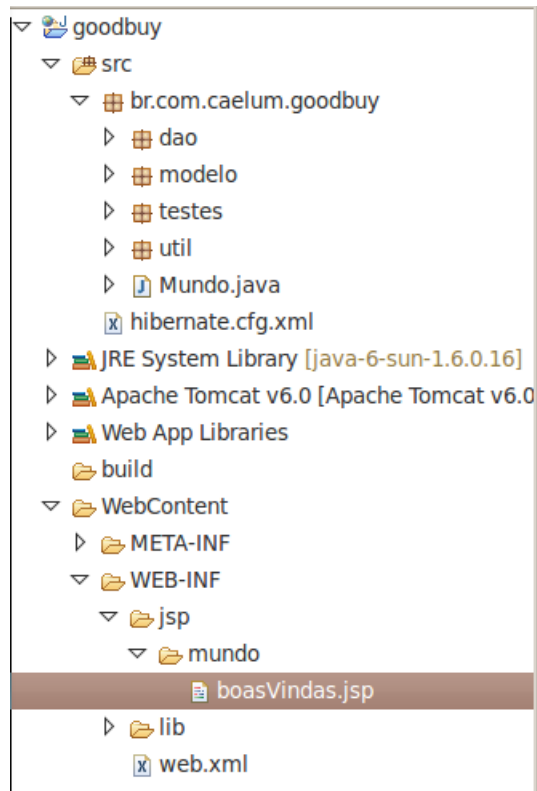
- {nomeDoResource} = mundo
- {lógica} = boasVindas

E a página que ele buscará será a seguinte:

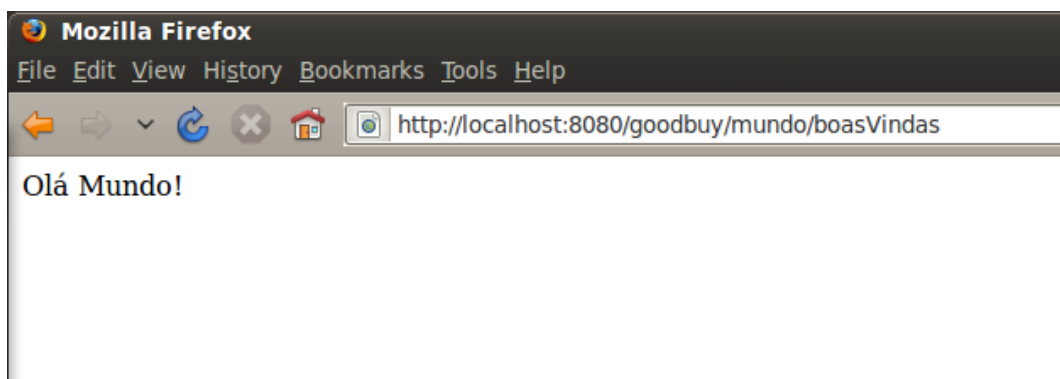
```
/WEB-INF/jsp/mundo/boasVindas.jsp
```

7.7 EXERCÍCIOS

- 1) Crie uma pasta chamada `jsp` dentro da pasta `WEB-INF`. Cuidado, o nome da pasta deve ser com letras minúsculas.
- 2) Crie uma pasta chamada `mundo` dentro da pasta `WEB-INF/jsp`. Cuidado, o nome da pasta deve ser com letras minúsculas.
- 3) Crie um `jsp` chamado `boasVindas.jsp` dentro da pasta `WEB-INF/jsp/mundo`. Seu projeto deve ficar assim:



- 4) Abra o arquivo `boasVindas.jsp` que acabamos de criar e digite o seguinte conteúdo:
`Olá Mundo!`
- 5) Vamos testar nossas alterações. Inicie o Tomcat na view Servers, depois abra um browser e digite a seguinte url: `http://localhost:8080/goodbuy/mundo/boasVindas`
- 6) Verifique se o console mostra a mensagem “olá mundo!”, e verifique se o browser exibe a mensagem `Olá Mundo`.



7.8 DISPONIBILIZANDO INFORMAÇÕES PARA A VIEW

Nosso exemplo até agora só executa algo no servidor e depois a view é chamada.

Mas para muitos casos precisamos que a lógica carregue informações do banco de dados e depois a view exiba essas informações.

No VRaptor 3 foi criada uma característica muito intuitiva de se mandar informações para a view. Quando um método é invocado, por exemplo nosso método `boasVindas()`, podemos fazer com que ele retorne algo, e esse retorno será enviado para a view automaticamente.

As versões anteriores usavam atributos junto com anotações ou getters. O problema dessa abordagem é que a classe ficava muito poluída com atributos, getters ou anotações.

Vamos alterar nosso método para enviar informações para a view:

```
1 package br.com.caelum.goodbuy;
2
3 @Resource
4 public class Mundo {
5
6     public String boasVindas() {
7         return "olá mundo!";
8     }
9
10 }
```

Agora o método `boasVindas()` não imprime nada, apenas retorna uma string. Após a execução desse método, essa string já estará disponível na view.

É muito importante perceber a legibilidade desse método. É uma classe java normal, e um método que retorna uma string.

Para que a view possa imprimir esse valor, vamos utilizar *Expression Language* no nosso jsp. Nosso jsp ficará assim:

```
Mensagem vinda da lógica:
${string}
```

Como utilizamos o retorno do método para disponibilizar informações para a view, não temos como dar um nome significativo para utilizar depois.

O VRaptor usa a convenção de buscar o tipo do retorno, nesse nosso caso uma `String`, e disponibilizar para a view com o mesmo nome, apenas passando a primeira letra para minúsculo. Por esse motivo que foi utilizado a expression language `${string}`.

7.9 DISPONIBILIZANDO COLEÇÕES PARA A VIEW

Podemos utilizar a mesma maneira que fizemos com Strings para disponibilizar coleções para a view. Utilizamos muito as coleções do pacote `java.util`, seja pela facilidade no uso ou pela integração com frameworks

como o Hibernate.

Se nossa lógica criasse uma coleção com informações que vieram do banco de dados, poderíamos retornar essa coleção, assim como fizemos anteriormente. O código ficaria assim:

```
1 package br.com.caelum.goodbuy;
2
3 @Resource
4 public class Mundo {
5
6     public String boasVindas() {
7         return "olá mundo!";
8     }
9
10    public List<String> paises() {
11        List<String> result = new ArrayList<String>();
12        result.add("Brasil");
13        result.add("Portugal");
14        result.add("Japão");
15        result.add("Canadá");
16        result.add("Paraguai");
17        return result;
18    }
19
20 }
```

Na nossa view, podemos imprimir os valores dessa coleção da seguinte forma:

```
Países vindos da lógica:
${stringList}
```

Note que agora o valor passado na expression language foi `${stringList}`. O VRaptor identifica que devolvemos uma lista, e essa lista é genérica, informando que o tipo de informação dentro da lista será `String`.

Para listas, o VRaptor vai adotar a seguinte convenção:

```
{tipoDaLista}List
```

Nosso exemplo era de uma lista de `String`, ou seja, `List<String>`, que virou `${stringList}`.

7.10 EXERCÍCIOS

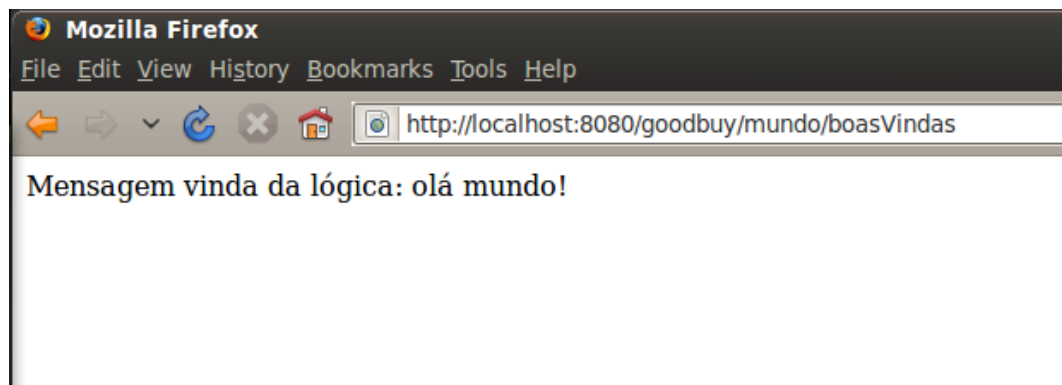
- 1) Altere o método `boasVindas()` da classe `Mundo` para retornar uma `String`.

```
1 package br.com.caelum.goodbuy;
2
3 @Resource
4 public class Mundo {
5
6     public String boasVindas() {
7         return "olá mundo!";
8     }
9
10 }
```

- 2) Altere o jsp `boasVindas.jsp`, que está dentro da pasta `WEB-INF/jsp/mundo`.

Mensagem vinda da lógica:
`${string}`

- 3) Teste a lógica que acabamos de criar, acessando a seguinte url: `http://localhost:8080/goodbuy/mundo/boasVindas` Sua página deve exibir o seguinte resultado:



- 4) Crie um método chamado `paises()` na classe `Mundo`. Esse método deve retornar uma lista de Strings. Seu código ficará assim:

```
public List<String> paises() {
    List<String> result = new ArrayList<String>();
    result.add("Brasil");
    result.add("Portugal");
    result.add("Japão");
    result.add("Canadá");
    result.add("Paraguai");
    return result;
}
```

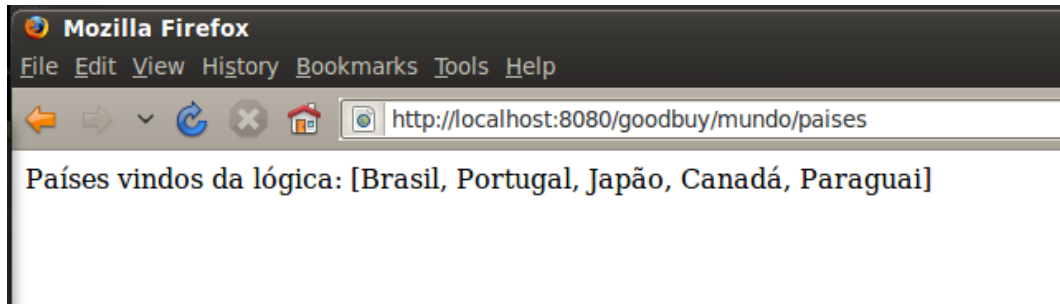
- 5) Crie um jsp em `WEB-INF/jsp/mundo` chamado `paises.jsp`. O conteúdo deve ser esse:

Países vindos da lógica:
`${stringList}`

- 6) Teste a lógica que acabamos de criar, acessando a seguinte url:

`http://localhost:8080/goodbuy/mundo/paises`

Sua página deve exibir o seguinte resultado:



- 7) (Opcional) Crie outro método que mande alguma informação para a view. Dessa vez, tente retornar um valor inteiro ou um valor decimal.
- 8) (Opcional) Crie outro método que retorne um Produto, com algumas informações preenchidas. Tente imprimir essas informações no jsp.
- 9) (Opcional) Na view que exibe os países, itere sobre a coleção e mostre cada país em uma linha. Dica: use o JSTL, tag `c:forEach`.
- 10) (Opcional) Um método só pode retornar uma informação. Mas como faríamos para disponibilizar para a view mais de uma informação?

Criando o Controlador de Produtos

8.1 LISTANDO PRODUTOS

Relembrando o que temos no nosso projeto, criamos o dao de produto para encapsular as chamadas ao Hibernate, e vimos como se usa o VRaptor 3, recebendo e disponibilizando valores da view. Precisamos agora integrar essas duas partes, para que possamos fazer o cadastro do produto pelo browser.

Vamos começar criando nossa classe que fará a lógica de negócios. O nome da classe poderia ser qualquer nome válido, mas usaremos uma convenção do VRaptor: utilizar o sufixo `Controller`.

Vimos que o nome da classe é utilizado pelo VRaptor para registrar esse componente, para que quando fizermos uma chamada do browser, essa classe seja encontrada e algum método seja invocado. Mas utilizando o nome `ProdutosController`, o VRaptor vai mapear essa classe para apenas `/produtos`, removendo o sufixo `Controller` que faz parte da convenção.

Dentro dessa classe, teremos um método para listar todos os produtos. Esse método será invocado pelo VRaptor e o produto em questão será passado com os dados já populados através dos dados do formulário.

Primeiro, vamos listar todos os produtos existentes no banco de dados. Para isso, vamos usar o método `createCriteria` de `Session` que cria uma `Criteria`. Através de um `Criteria`, temos acesso a diversas operações no banco de dados; uma delas é listar tudo com o método `list()`.

Nosso método `listarTudo()` no `ProdutoDao` fica assim:

```
public class ProdutoDao {  
    //...  
    public List<Produto> listarTudo() {  
        return this.session.createCriteria(Produto.class).list();  
    }  
}
```


Com essa listagem pronta, podemos usá-la no `ProdutosController`, e criar uma lógica que lista produtos. Coloque a classe no pacote `br.com.caelum.goodbuy.controller`:

```
package br.com.caelum.goodbuy.controller;

public class ProdutosController {

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

}
```

Olhando para esse método, que no início tínhamos definido para ser nossa regra de negócio, será que temos apenas isso? Será que apenas listamos os produtos?

Na verdade esse método está fazendo mais tarefas do que deveria. Ele possui algumas responsabilidades extras, por exemplo criar o `ProdutoDao`. A única parte que é a regra de negócio para esse caso é a chamada ao `dao.listaTudo()`. O resto é apenas infraestrutura para permitir executar essa tarefa.

Avaliando esse problema, podemos perceber que estamos buscando recursos, nesse caso um `dao`, e portanto nos preocupando demasiadamente com os mesmos. Se buscar um recurso é ruim, seja pela complexidade ou dificuldade no acesso, e também por não fazer parte da regra de negócio, que tal se ao invés de criá-lo, recebêssemos o mesmo?

```
1 package br.com.caelum.goodbuy.controller;
2
3 // import's
4
5 @Resource
6 public class ProdutosController {
7
8     private ProdutoDao dao;
9
10    public ProdutosController(ProdutoDao dao) {
11        this.dao = dao;
12    }
13
14    public List<Produto> lista() {
15        return dao.listaTudo();
16    }
17
18 }
```

Repare como seria mais simples nosso método que executa a regra de negócio, e também como o método só executa a parte que realmente lhe interessa. A busca dos recursos necessários para a execução de nessa lógica de negócios - nesse caso a criação do dao - não interessa pra essa classe.

Um ponto muito importante que temos que notar é que para que essa classe funcione corretamente, precisamos de uma instância de ProdutoDao. Se não tivermos essa instância, é impossível executar nossa regra de negócios, pois não existe construtor que não receba um ProdutoDao.

Como vimos antes, essa lógica redireciona para a jsp /WEB-INF/jsp/produtos/lista.jsp, e como retornamos uma lista de produtos, existe uma variável chamada `{produtoList}` disponível no jsp.

Para podermos mostrar a listagem de um jeito mais fácil, vamos usar uma taglib da JSTL chamada `c:forEach`, que é capaz de iterar sobre uma lista passada.

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

Usando essa taglib, vamos criar uma tabela com todos os produtos do sistema:

```
<table>
  <thead>
    <tr>
      <th>Nome</th>
      <th>Descrição</th>
      <th>Preço</th>
    </tr>
  </thead>
  <tbody>
    <c:forEach items="{produtoList}" var="produto">
      <tr>
        <td>{produto.nome }</td>
        <td>{produto.descricao }</td>
        <td>{produto.preco }</td>
      </tr>
    </c:forEach>
  </tbody>
</table>
```

8.2 QUAIS SÃO MINHAS DEPENDÊNCIAS?

Tudo que um objeto necessita para permitir a execução de seus métodos, isto é, tudo o que ele depende, pode ser passado para o mesmo através de métodos (como os setters, por exemplo), ou do construtor.

Na nossa abordagem, utilizamos o construtor da classe `ProdutosController` para receber nossas dependências e, com isso, ganhamos a garantia de que, se tenho um objeto do tipo `ProdutosController` na minha

mão, posso invocar o método adiciona pois o atributo dao teve seu valor atribuído durante a construção do objeto.

Ao adicionar o construtor customizado, perdemos o construtor padrão, e não podemos mais instanciar essa classe assim:

```
ProdutosController controller = new ProdutosController();
```

Essa característica de, com um objeto em mãos, saber que seus métodos podem ser invocados sem necessidade de nenhuma dependência externa extra, é parte do que foi chamado de Good Citizen: <http://docs.codehaus.org/display/PICO/Good+Citizen>

Em um objeto todas as dependências devem ser passadas durante o processo de construção evitando assim um possível estado inconsistente.

Como poderia então criar o meu ProdutosController?

```
1 ProdutoDao dao = new ProdutoDao();
2 ProdutosController controller = new ProdutosController(dao);
```

Repare que logo após a segunda linha ser executada, tenho a confiança de que o objeto referenciado pela variável controller está preparado para ter seus métodos invocados.

Dessa maneira injetamos todas as nossas dependências durante a instanciação do controlador, um trabalho manual que ficará extremamente repetitivo e propício a erro a medida que aumenta o número de dependências de nossa classe.

8.3 INJEÇÃO DE DEPENDÊNCIAS

Sendo assim, fica muito mais fácil se fizermos uso de uma API que execute todo o processo de injeção automaticamente.

No nosso caso, quem vai instanciar essa classe? Nós ou o VRaptor? **Essa classe será instanciada pelo VRaptor.** Mas se o VRaptor fizer isso, como ele descobrirá sobre interligações entre as dependências?

Uma vez que precisamos da classe ProdutoDao como dependência nossa, precisamos notificar o framework que o mesmo se encarregará de gerenciar as instâncias de ProdutoDao, isto é, nesse caso específico, ele criará uma instância e utilizará como argumento para o construtor. Para fazer isso, basta anotarmos essa classe com @Component.

```
package br.com.caelum.goodbuy.dao;
```

```
// import's
```

```
import br.com.caelum.vraptor.ioc.Component;
```

```
@Component
public class ProdutoDao {
    //...
}
```

Agora a classe `ProdutoDao` também será controlada pelo VRaptor. Dessa forma, quando o VRaptor for instanciar a classe `ProdutosController`, ele verificará que ela depende de um `ProdutoDao`, e criará uma instância da mesma.

O que acabamos de fazer foi criar uma maneira de passar as dependências para onde for necessário, ou seja, um mecanismo de “injetar” as dependências. Por esse motivo, esse conceito é chamado de **Injeção de Dependências**.

O VRaptor está fortemente baseado nesse conceito, uma vez que até ele mesmo utiliza o mesmo para conectar seus componentes internos. O conceito básico por trás de *Dependency Injection (DI)* é que você não deve buscar aquilo que deseja acessar, mas tais necessidades devem ser fornecidas para você.

Isso se traduz, por exemplo, na passagem de componentes através do construtor de seus controladores. Imagine que seu controlador de clientes necessita acessar um dao. Sendo assim, especifique claramente essa necessidade.

TESTANDO SUA APLICAÇÃO

Ao usarmos injeção de dependências, ganhamos uma característica muito boa na nossa aplicação: a testabilidade. Se recebemos nossas dependências no construtor, conseguimos passar implementações falsas ou controladas e, assim, testar unitariamente nossas classes.

Você pode encontrar um conteúdo mais aprofundado sobre testes nos cursos **FJ-22 - Laboratório Java com Testes, JSF, Web Services e Design Patterns** e **PM-87 - Práticas ágeis de desenvolvimento de software**.

8.4 EXERCÍCIOS

1) Abra a classe `ProdutoDao` e adicione o método para listar todos os produtos:

```
public class ProdutoDao {
    //...
    public List<Produto> listaTudo() {
        return this.session.createCriteria(Produto.class).list();
    }
}
```

2) Crie a classe `ProdutosController` no pacote `br.com.caelum.goodbuy.controller`. Crie também o método `lista`, que retorna uma `List<Produto>`.

3) Anote a classe com a anotação `@Resource`.

```
1 package br.com.caelum.goodbuy.controller;
2
3 // import's
4
5 import br.com.caelum.vraptor.Resource;
6
7 @Resource
8 public class ProdutosController {
9
10     public List<Produto> lista() {
11     }
12
13 }
```

4) Crie o construtor que recebe uma instância de `ProdutoDao`, e guarde essa instância em um atributo.

```
1 package br.com.caelum.goodbuy.controller;
2
3 // import's
4 import br.com.caelum.vraptor.Resource;
5
6 @Resource
7 public class ProdutosController {
8
9     private final ProdutoDao dao;
10
11     public ProdutosController(ProdutoDao dao) {
12         this.dao = dao;
13     }
14
15     public List<Produto> lista() {
16     }
17
18 }
```

5) No método `lista`, faça a chamada ao método `listaTudo` do `dao`.

```
1 package br.com.caelum.goodbuy.controller;
2
3 // import's
4
5 import br.com.caelum.vraptor.Resource;
```

```
6
7 @Resource
8 public class ProdutosController {
9
10     private final ProdutoDao dao;
11
12     public ProdutosController(ProdutoDao dao) {
13         this.dao = dao;
14     }
15
16     public List<Produto> lista() {
17         return dao.listaTudo();
18     }
19
20
21 }
```

6) Na pasta WEB-INF/jsp, crie a pasta produtos.

7) Crie o arquivo lista.jsp na pasta /WEB-INF/jsp/produtos.

```
<table>
    <thead>
        <tr>
            <th>Nome</th>
            <th>Descrição</th>
            <th>Preço</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${produtoList}" var="produto">
            <tr>
                <td>${produto.nome }</td>
                <td>${produto.descricao }</td>
                <td>${produto.preco }</td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

8) Acesse a URI que executa o método lista: <http://localhost:8080/goodbuy/produtos/lista> .

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'produtosController':
org.springframework.beans.factory.support.ConstructorResolver.createArgumentArray(ConstructorResolver.java:51)
org.springframework.beans.factory.support.ConstructorResolver.autowireConstructor(ConstructorResolver.java:11)
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createConstructor(AbstractAut
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBeanInstance(AbstractAut
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCap
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$1.run(AbstractAutowireCapableBe
java.security.AccessController.doPrivileged(Native Method)
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapal
org.springframework.beans.factory.support.AbstractBeanFactory$2.getObject(AbstractBeanFactory.java:302)
org.springframework.web.context.request.AbstractRequestAttributesScope.get(AbstractRequestAttributesScope.java
org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:298)
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:185)
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:164)
org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeansOfType(DefaultListableBeanFacto
org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeansOfType(DefaultListableBeanFacto
```

- 9) Deu uma exception! A informação mais importante dela está no final da Stacktrace:

• • •

Caused by: org.springframework.beans.factory

```
.NoSuchBeanDefinitionException:
```

No unique bean of type

[br.com.caelum.goodbuy.dao.ProdutoDao] is defined:

Unsatisfied dependency of type

```
[class br.com.caelum.goodbuy.dao.ProdutoDao]:
```

expected at least 1 matching bean

```
at org.springframework.beans.factory.support...
```

Ou seja, o Spring -- que é usado pelo VRaptor para gerenciar dependências -- não conhece o ProdutoDao. Precisamos indicar para o VRaptor que ele tem que registrar o ProdutoDao no Spring, com a anotação `@Component`.

- 10) Anote a classe ProdutoDao com @Component, para indicar que essa classe é uma dependência e pode ser instanciada pelo VRaptor sempre que necessário.

```
package br.com.caelum.goodbuy.dao;
```

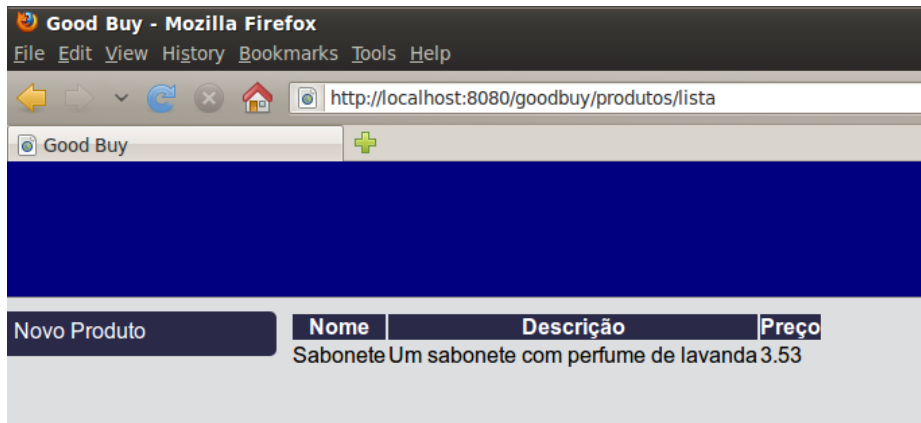
```
// import's
```

```
import br.com.caelum.vraptor.ioc.Component;
```

@Component

```
public class ProdutoDao {
    //...
}
```

- 11) Se acessarmos de novo a URI que executa o método lista: <http://localhost:8080/goodbuy/produtos/lista> veremos a listagem de produtos:



- 12) Abra o arquivo `header.jspf` e procure a div com `id="menu"`. Adicione um link ao menu, para acessar a listagem de produtos:

```
<div id="menu">
    <ul>
        <li><a href="<c:url value="/produtos/lista"/>">Lista Produtos</a></li>
    </ul>
</div>
```

8.5 CADASTRANDO UM PRODUTO

Para cadastrar um produto, basta passarmos esse produto para o método `salva` da classe `ProdutoDao`. Vamos criar então um método no `ProdutoController` para adicionar produtos:

```
1 package br.com.caelum.goodbuy.controller;
2
3 // import's
4
5 @Resource
6 public class ProdutosController {
7
8     public void adiciona(Produto produto) {
9         ProdutoDao dao = new ProdutoDao();
10        dao.salva(produto);
11    }
12
13 }
```


8.6 CRIANDO O FORMULÁRIO HTML

Nossa lógica para criar (salvar) produtos já está pronta. Mas como passar um produto populado para o método adiciona do `ProdutosController`? Uma saída seria aproveitar a injeção de dependências do VRaptor, receber um `HttpServletRequest` no construtor, e pegar os parâmetros que vieram do formulário, e usar os getters e setters do `Produto` para populá-lo. Mas isso é bem chato e repetitivo: imagine se o `Produto` tivesse 20 campos!

Para evitar isso, o VRaptor possui uma convenção para popular os parâmetros da sua lógica. O método adiciona recebe como parâmetro um `Produto` chamado `produto`, então para popular o campo nome desse produto, o VRaptor vai usar um parâmetro da requisição chamado `produto.nome`. Para o campo descricao o parâmetro vai ser `produto.descricao`. Ou seja, a partir do nome do parâmetro do método, podemos usar pontos para setar os campos do parâmetro, desde que você tenha getters e setters para eles.

O que vale é o nome do parâmetro: se o método adiciona fosse

```
public void adiciona(Produto novoProduto) {...}
```

o VRaptor iria usar os parâmetros `novoProduto.nome`, `novoProduto.descricao`, etc para popular os campos do produto.

REFLECTION NO NOME DOS PARÂMETROS

Infelizmente, o Java não realiza reflection em cima de parâmetros, esses dados não ficam disponíveis em bytecode (a não ser se compilado em debug mode, porém é algo opcional). Isso faz com que a maioria dos frameworks que precisam desse tipo de informação criem uma anotação própria para isso, o que polui muito o código (exemplo no JAX-WS, onde é comum encontrar um método como o acima com a assinatura `void add(@WebParam(name="cliente") Cliente cliente)`).

O VRaptor tira proveito do framework Paranamer (<http://paranamer.codehaus.org>), que consegue tirar essa informação através de pré compilação ou dos dados de debug, evitando criar mais uma anotação. Alguns dos desenvolvedores do VRaptor também participam no desenvolvimento do Paranamer.

Agora que sabemos como o VRaptor popula os parâmetros do método, vamos criar um formulário para poder cadastrar produtos. Esse formulário deverá ficar na pasta `WEB-INF/jsp/produtos`, com o nome de `formulario.jsp`.

```
1 <form action="adiciona">
2   <fieldset>
3     <legend>Adicionar Produto</legend>
4
5     <label for="nome">Nome:</label>
```

```
6      <input id="nome" type="text" name="produto.nome"/>
7
8      <label for="descricao">Descrição:</label>
9      <textarea id="descricao" name="produto.descricao"></textarea>
10
11     <label for="preco">Preço:</label>
12     <input id="preco" type="text" name="produto.preco"/>
13
14     <button type="submit">Enviar</button>
15 </fieldset>
16 </form>
```

Como nossas páginas ficam sempre na pasta WEB-INF, não temos como acessar diretamente pelo browser um jsp. Para acessar a página que acabamos de criar, vamos criar um método na classe `ProdutosController` para chamar nossa página, usando a convenção do VRaptor.

```
@Resource
public class ProdutosController {

    // atributo, construtor e métodos

    public void formulario() {
    }

}
```

Pelo browser, chamaremos a seguinte URL:

`http://localhost:8080/goodbuy/produtos/formulario`

Após preencher as informações do formulário e enviar, o VRaptor vai invocar o método `adiciona`, passando o `Produto` populado com os campos que digitamos. E como diz a convenção do VRaptor, precisamos criar o arquivo `adiciona.jsp`, na pasta `WEB-INF/jsp/produtos`. Por enquanto vamos colocar o seguinte conteúdo:

Produto adicionado com sucesso!

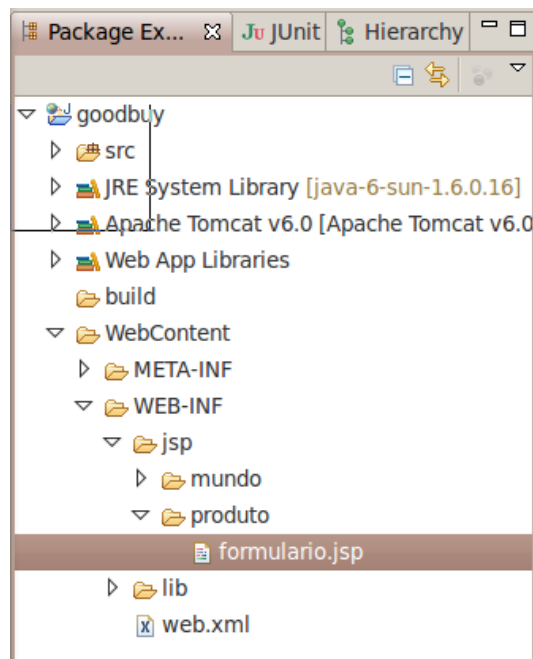
8.7 EXERCÍCIOS

- 1) Crie o método `adiciona` que recebe um `Produto` como parâmetro e faça a chamada ao método `salva` do `dao` passando esse produto.

```
1 package br.com.caelum.goodbuy.controller;
2 public class ProdutosController {
3
```

```
4    //...
5
6    public void adiciona(Produto produto) {
7        dao.salva(produto);
8    }
9
10 }
```

2) Na pasta WEB-INF/jsp/produtos, crie o arquivo formulario.jsp.



3) Abra o arquivo que acabamos de criar e coloque os campos para de digitação, assim como o botão e o formulário.

```
<form action="adiciona">
    <fieldset>
        <legend>Adicionar Produto</legend>

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

        <label for="descricao">Descrição:</label>
        <textarea id="descricao" name="produto.descricao"></textarea>

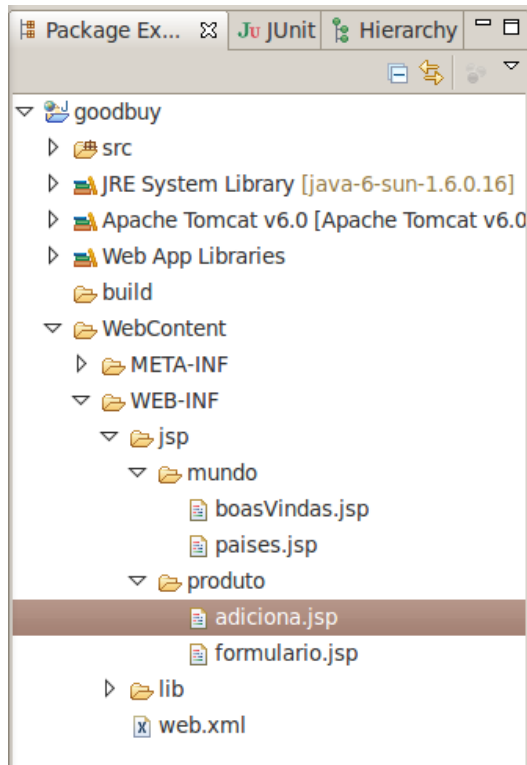
        <label for="preco">Preço:</label>
        <input id="preco" type="text" name="produto.preco"/>

        <button type="submit">Enviar</button>
```

```
</fieldset>
</form>
```

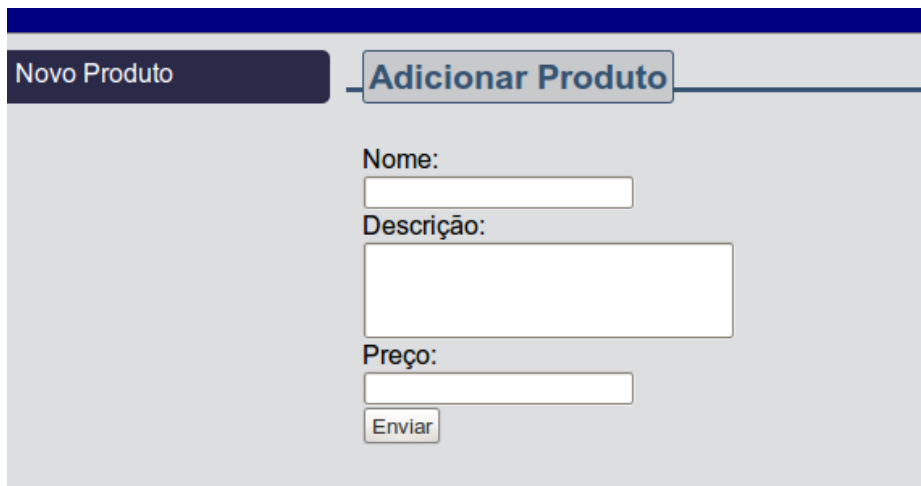
- 4) Na pasta WEB-INF/jsp/produtos, crie o arquivo adiciona.jsp com o conteúdo.

```
<h3>Produto adicionado com sucesso!</h3>
```



- 5) Acesse a listagem de produtos e veja os produtos que existem.
- 6) Acesse a URI do formulário e adicione um produto. <http://localhost:8080/goodbuy/produtos/formulario>
- 7) Volte à listagem, e veja se um usuário foi cadastrado com sucesso.
- 8) Abra o arquivo header.jspf e procure a div com id="menu". Adicione um link dentro do menu para acessar o formulário:

```
<div id="menu">
  <ul>
    <li><a href="<c:url value="/produtos/formulario"/>">
      Novo Produto
    </a></li>
    <li><a href="<c:url value="/produtos/lista"/>">
      Lista Produtos
    </a></li>
  </ul>
</div>
```



The screenshot shows a web interface for adding a new product. It has a header bar with 'Novo Produto' and 'Adicionar Produto'. Below the header, there are three input fields: 'Nome:', 'Descrição:', and 'Preço:'. Each field has a corresponding text input box. At the bottom of the form is an 'Enviar' button.

8.8 REDIRECIONAR PARA LISTAGEM DEPOIS DE ADICIONAR

Uma coisa não aparenta estar muito certa no nosso sistema: Quando adicionamos um produto, redirecionamos para uma página com a mensagem “Produto adicionado com sucesso”. Será que essa página é suficiente? Será que não é melhor mostrar informações mais relevantes?

Após adicionar produtos podemos, por exemplo, mostrar uma lista com os produtos já adicionados. Usando a convenção do VRaptor, poderíamos colocar a lista de produtos no arquivo adiciona.jsp. Mas já temos uma lógica que lista produtos, por que não usá-la? Para isso precisamos falar para o VRaptor que não queremos usar a convenção, queremos ao invés de ir para a página adiciona.jsp, redirecionar para a lógica de listagem.

Para sobrescrever a convenção da view padrão, existe um componente do VRaptor chamado `Result`. Com ele é possível redirecionar para páginas ou para lógicas, ao final do seu método.

Primeiro precisamos de uma instância de `Result`. Vamos dar um `new` nele? Não, e isso nem é possível porque `Result` é uma interface. Como fazer então? Como visto antes, o VRaptor usa injeção de dependências, e como o `Result` é um componente conhecido pelo VRaptor, basta recebê-lo no construtor da classe.

```
import br.com.caelum.vraptor.Result;

@Resource
public class ProdutosController {

    private final ProdutoDao dao;
    private final Result result;

    public ProdutosController(ProdutoDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }
}
```

```
//...  
}
```

Agora que temos essa instância de `Result` podemos usá-la para mudar o resultado da lógica adiciona do `ProdutosController`. O jeito de fazer isso é dizer: “Como resultado, quero redirecionar para a lógica `ProdutosController.lista()`”. Ou em java:

```
result.redirectTo(ProdutosController.class).lista();
```

Dentro do método adiciona isso ficaria:

```
@Resource  
public class ProdutosController {  
  
    private final ProdutoDao dao;  
    private final Result result;  
  
    public ProdutosController(ProdutoDao dao, Result result) {  
        this.dao = dao;  
        this.result = result;  
    }  
  
    public void adiciona(Produto produto) {  
        dao.salva(produto);  
        result.redirectTo(ProdutosController.class).lista();  
    }  
  
    public void formulario() {  
    }  
  
    public List<Produto> lista() {  
        return dao.listaTudo();  
    }  
}
```

Repare que não foi necessário criar arquivos de configuração para fazer essa mudança de convenção, tudo é feito no próprio método. Ao olhar para o método `adiciona` fica claro que o resultado dele não vai ser o `adiciona.jsp`, vai ser a lógica `lista` do `ProdutosController`.

Como o redirecionamento é para uma lógica do mesmo controller você pode ainda usar um comando mais enxuto:

```
result.redirectTo(this).lista();
```

Existem duas maneiras de redirecionar para uma lógica:

- redirecionamento do lado do cliente, usando o método `redirectTo`: o resultado da requisição será um código para o browser fazer outra requisição, para a URL indicada. É uma boa prática usar o redirecionamento do lado do cliente sempre após requisições POST, ou seja, sempre que submetemos formulários prevenindo, assim, que o usuário recarregue a página e resubmeta o formulário.
- redirecionamento do lado do servidor, usando o método `forwardTo`: o servidor vai redirecionar internamente para a lógica especificada, desse modo é possível mudar o resultado de uma lógica sem mudanças do lado do cliente. Não use esse redirecionamento em requisições POST.

Além disso, podemos redirecionar nossa requisição para páginas usando outros métodos:

- `result.forwardTo("/uma/outra/pagina.jsp")`: redirecionamento do lado do servidor para a página especificada.
- `result.redirectTo("/uma/outra/pagina.jsp")`: redirecionamento do lado do cliente para a página especificada.
- `result.of(ProdutosController.class).lista()`: redireciona para a página da lógica especificada, sem executar a lógica. Nesse caso redirecionaria para `/WEB-INF/jsp/produtos/lista.jsp`.

Existem mais redirecionamentos possíveis implementados no VRaptor. Todos eles estão disponíveis como métodos estáticos da classe `Results`, que você pode chamar dentro do método `use` do `Result`, por exemplo:

- `result.use(Results.logic())`: Redireciona usando uma lógica. A chamada que fizemos anteriormente `result.redirectTo(ProdutosController.class).lista()` na verdade é um atalho para `result.use(Results.logic()).redirectTo(ProdutosController.class).lista()`.
- `result.use(Results.page())`: Redireciona usando uma página. A chamada que vimos anteriormente `result.forwardTo("/uma/outra/pagina.jsp")` na verdade é um atalho para `result.use(Results.page()).forwardTo("/uma/outra/pagina.jsp")`.
- `result.use(Results.nothing())`: Não usa nenhuma página como resultado.
- `result.use(Results.http())`: Usa como resultado status codes e Headers HTTP. Ex: `result.use(Results.http()).setStatusCode(404)`
- `result.use(Results.referer())`: Usa como resultado a página que originou a requisição. Usa um Header HTTP chamado Referer que não é obrigatório, e que é removido por alguns Proxies e Firewalls, então cuidado ao usar esse resultado.

8.9 EXERCÍCIOS

- 1) Receba um `Result` no construtor do `ProdutosController` e guarde-o em um atributo.

```
import br.com.caelum.vraptor.Result;

@Resource
public class ProdutosController {

    private final ProdutoDao dao;
    private final Result result;

    public ProdutosController(ProdutoDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }
    //...
}
```

- 2) Mude o resultado da lógica adiciona para redirecionar para a lógica lista.

```
@Resource
public class ProdutosController {

    private final ProdutoDao dao;
    private final Result result;

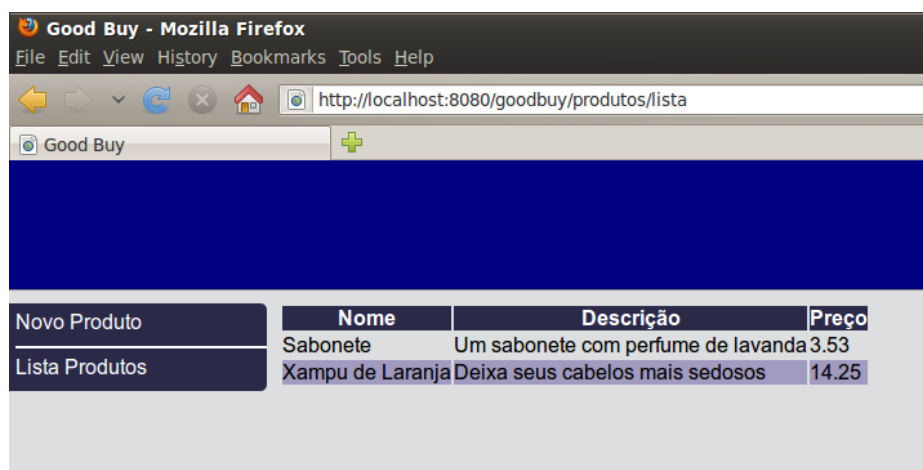
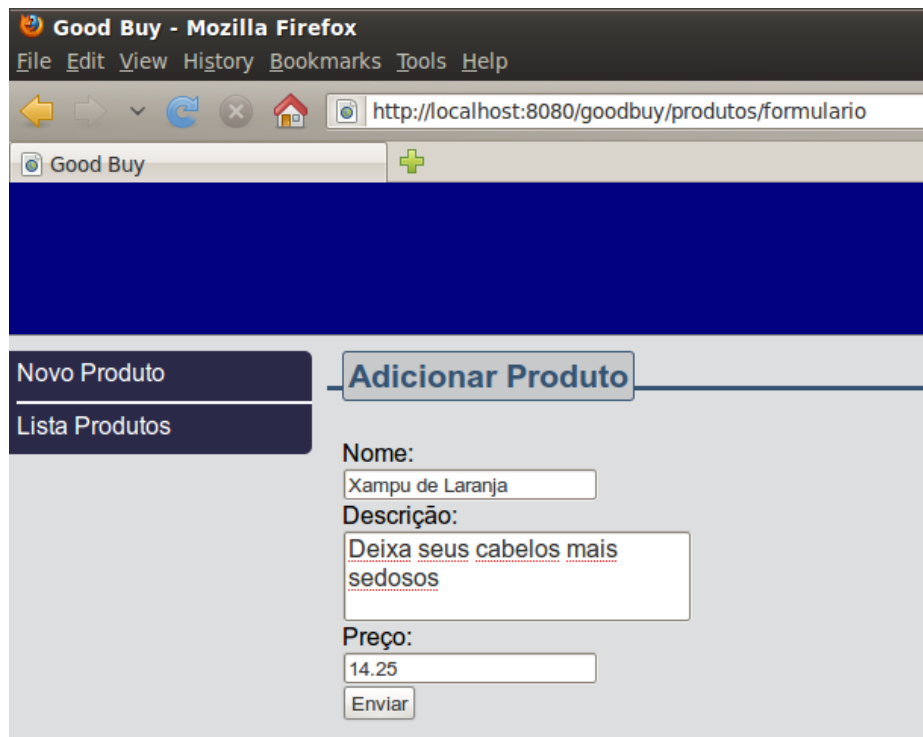
    public ProdutosController(ProdutoDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }

    public void adiciona(Produto produto) {
        dao.salva(produto);
        result.redirectTo(this).lista();
    }

    public void formulario() {
    }

    public List<Produto> lista() {
        return dao.listaTudo();
    }
}
```

- 3) Reinicie o tomcat para que ele carregue as suas mudanças, abra o formulário e adicione mais um produto. Repita essa operação e adicione mais produtos.



Nome	Descrição	Preço
Sabonete	Um sabonete com perfume de lavanda	3.53
Xampu de Laranja	Deixa seus cabelos mais sedosos	14.25

8.10 ATUALIZANDO E REMOVENDO PRODUTOS

Para melhorarmos um pouco mais o nosso sistema, vamos incluir as operações de atualização e remoção de produtos.

Começando pela atualização, primeiro precisamos de uma lógica que mostre um formulário com os dados do produto preenchidos. Vamos dar a essa lógica o nome de edita.

```
public class ProdutosController {
```

```
    public void edita() {...}  
}
```

Precisamos passar para essa lógica o id do produto que queremos editar, para que possamos carregar suas informações do banco de dados. E para passar o Produto carregado para a jsp, basta retorná-lo:

```
public class ProdutosController {  
  
    public Produto edita(Long id) {  
        return dao.carrega(id);  
    }  
}
```

Agora podemos criar o formulário de edição, passando os valores do produto carregado para os inputs:

```
<form action="altera">  
    <fieldset>  
        <legend>Editar Produto</legend>  
  
        <input type="hidden" name="produto.id"  
            value="{produto.id }" />  
  
        <label for="nome">Nome:</label>  
        <input id="nome" type="text"  
            name="produto.nome" value="{produto.nome }"/>  
  
        <label for="descricao">Descrição:</label>  
        <textarea id="descricao" name="produto.descricao">  
            {produto.descricao }  
        </textarea>  
  
        <label for="preco">Preço:</label>  
        <input id="preco" type="text"  
            name="produto.preco" value="{produto.preco }"/>  
  
        <button type="submit">Enviar</button>  
    </fieldset>  
</form>
```

Repare que precisamos também colocar um input hidden com o valor do `produto.id` para que saibamos qual produto será alterado. A action do form é para a lógica altera, então precisamos criá-la no nosso controller. Como usamos os nomes dos campos do formulário do jeito certo, podemos simplesmente receber um Produto como argumento na lógica altera.

```
public class ProdutosController {  
    //...  
    public void altera(Produto produto) {  
        dao.atualiza(produto);  
    }  
}
```

Assim como na lógica adiciona, fizemos um POST num formulário, então ao invés de colocar uma jsp com uma mensagem de “Produto alterado com sucesso”, vamos redirecionar para a listagem de produtos.

```
public class ProdutosController {  
    //...  
    public void altera(Produto produto) {  
        dao.atualiza(produto);  
        result.redirectTo(this).lista();  
    }  
}
```

Agora precisamos de um jeito fácil de editar um produto. Vamos então adicionar um link para editar o produto, na listagem:

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

Vamos também remover produtos no nosso controller. Para remover um produto precisamos apenas do seu id, então basta criar a lógica:

```
public void remove(Long id) {  
    Produto produto = dao.carrega(id);  
    dao.remove(produto);  
}
```

Também não faz muito sentido criar uma página de resultado para a remoção, então vamos redirecionar para a listagem.

```
public void remove(Long id) {  
    Produto produto = dao.carrega(id);  
    dao.remove(produto);  
    result.redirectTo(ProdutosController.class).lista();  
}
```

Assim como criamos o link para a edição, vamos criar um link para a remoção:

```
<c:forEach items="${produtoList}" var="produto">
    <tr>
        <td>${produto.nome }</td>
        <td>${produto.descricao }</td>
        <td>${produto.preco }</td>
        <td><a href="edita?id=${produto.id }">Editar</a></td>
        <td><a href="remove?id=${produto.id }">Remover</a></td>
    </tr>
</c:forEach>
```

8.11 EXERCÍCIOS

- 1) Crie as lógicas para edição de produtos no ProdutosController:

```
public Produto edita(Long id) {
    return dao.carrega(id);
}

public void altera(Produto produto) {
    dao.atualiza(produto);
    result.redirectTo(this).lista();
}
```

- 2) Implemente os métodos no ProdutoDao. Lembre-se de usar o Ctrl+1.

```
public Produto carrega(Long id) {
    return (Produto) this.session.load(Produto.class, id);
}

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

- 3) Crie o formulário de edição no arquivo /WEB-INF/jsp/produtos/edita.jsp.

```
<form action="altera">
    <fieldset>
        <legend>Editar Produto</legend>
        <input type="hidden" name="produto.id"
            value="${produto.id }" />

        <label for="nome">Nome:</label>
```

```

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

<label for="descricao">Descrição:</label>
<textarea id="descricao" name="produto.descricao">
  ${produto.descricao }
</textarea>

<label for="preco">Preço:</label>
<input id="preco" type="text" name="produto.preco"
      value="${produto.preco }"/>

<button type="submit">Enviar</button>
</fieldset>
</form>

```

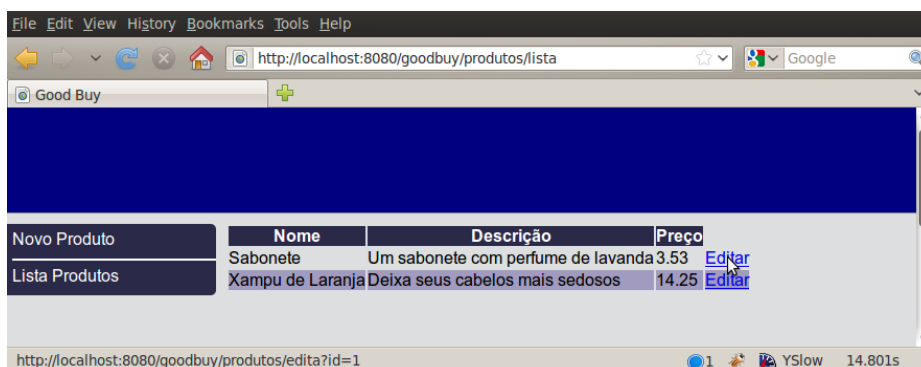
- 4) Modifique a listagem de produtos para incluir um link para a edição, no arquivo /WEB-INF/jsp/produtos/lista.jsp:

```

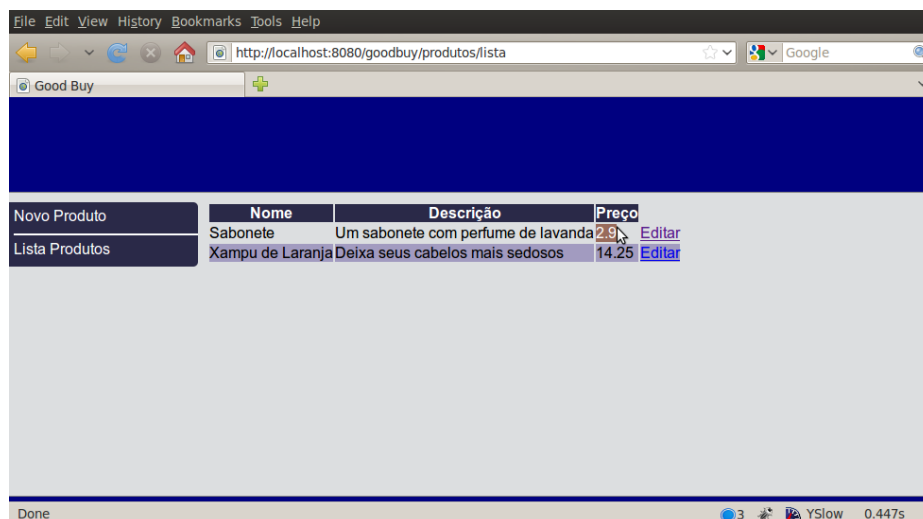
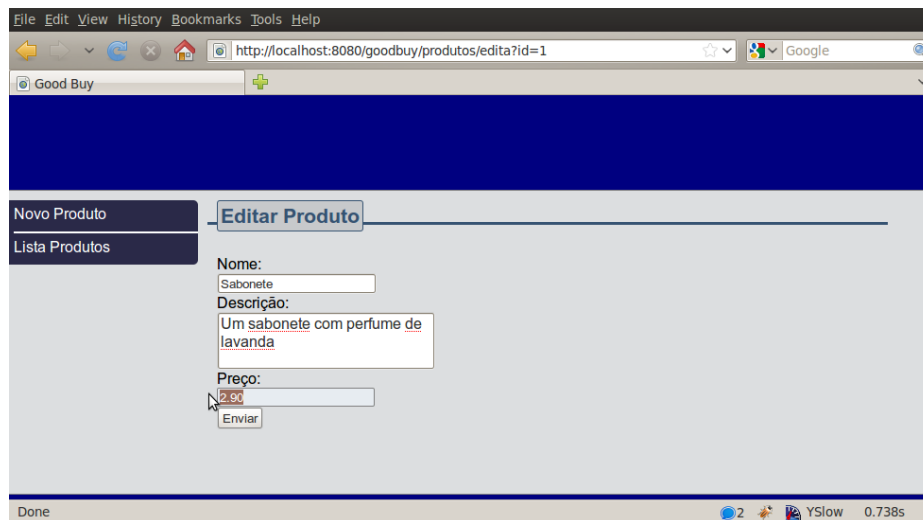
<c:forEach items="${produtoList}" var="produto">
  <tr>
    <td>${produto.nome }</td>
    <td>${produto.descricao }</td>
    <td>${produto.preco }</td>
    <td><a href="edita?id=${produto.id }">Editar</a></td>
  </tr>
</c:forEach>

```

- 5) Reinicie o tomcat e acesse a listagem <http://localhost:8080/goodbuy/produtos/lista>. Escolha um dos produtos e clique em Editar.



Edite o produto e clique em Enviar.



6) Crie agora a lógica de remoção na classe `ProdutosController`:

```
public void remove(Long id) {
    Produto produto = dao.carrega(id);
    dao.remove(produto);
    result.redirectTo(this).lista();
}
```

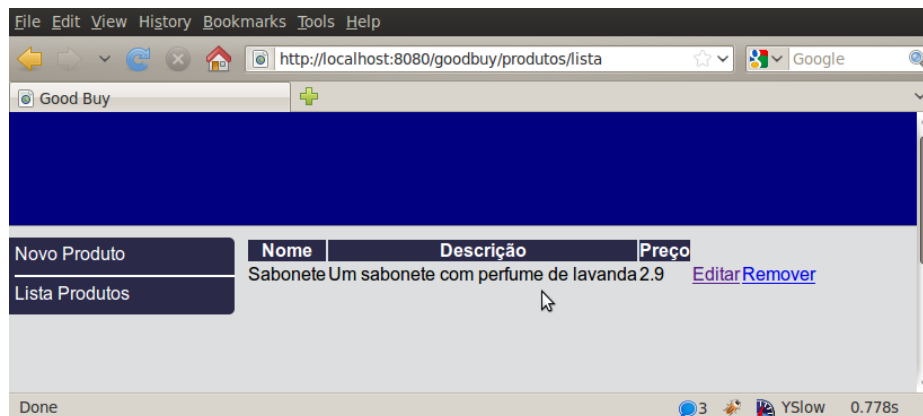
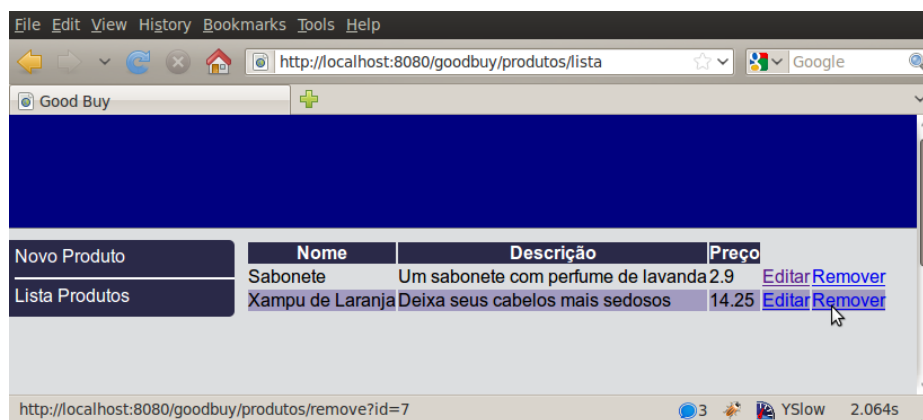
7) Implemente o método `remove` no `ProdutoDao`.

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

8) Modifique a listagem de produtos, adicionando o link para remoção

```
<c:forEach items="${produtoList}" var="produto">
  <tr>
    <td>${produto.nome }</td>
    <td>${produto.descricao }</td>
    <td>${produto.preco }</td>
    <td><a href="edita?id=${produto.id}">Editar</a></td>
    <td><a href="remove?id=${produto.id}">Remover</a></td>
  </tr>
</c:forEach>
```

9) Reinicie o tomcat, acesse a listagem e remova algum produto.



8.12 DISCUSSÃO EM SALA - VRAPTOR

- Mas o mercado usa mais o Struts. Vale a pena trocá-lo?
- Por que tantas convenções?

Refatorando os DAOs

9.1 INJEÇÃO DE DEPENDÊNCIAS NO DAO

Existe algo errado com o `ProdutoDao`: ele está criando uma `Session`, que é uma dependência dele. Usando injeção de dependências, podemos receber a `Session` no construtor, e o VRaptor vai se encarregar de criar essa `Session` e passar pro construtor do dao.

```
@Component
public class ProdutoDao {

    private final Session session;

    public ProdutoDao(Session session) {
        this.session = session;
    }
    //...
}
```

Mas como falar que a `Session` pode ser usada como dependência?

Poderíamos fazer a mesma alteração que fizemos no `ProdutoDao`, anotando a classe `Session` do pacote `org.hibernate` com `@Component`. Assim, a sessão do Hibernate também seria gerenciada pelo VRaptor. Mas será que podemos anotar uma classe do Hibernate? Só se baixássemos o código fonte e compilássemos na mão, junto com nosso projeto. Daria muito trabalho e seria muito estranho, nosso projeto alterando outro projeto.

Para resolver isso, o VRaptor possui um mecanismo que nos auxilia a passar dependências de outros projetos para nossas classes. Basta criarmos uma classe que implementa a interface `ComponentFactory`. Essa interface define um único método, o `getInstance`.

Criando essa classe, podemos definir a lógica de criação de um determinado objetos, e o VRaptor usará a instância retornada para passar como dependência para outras classes. No nosso caso, podemos criar uma classe que cria sessões do Hibernate.

Já temos uma classe que cria uma classe que nos dá uma instância de `Session`: a `CriadorDeSession`, então para transformá-la em um `ComponentFactory` basta implementar a interface. Mas o método que nos dá a `Session` se chama `getSession`, e a interface espera que o método se chame `getInstance`. E, por último, anotamos a classe com `@Component` para que o VRaptor saiba como criá-la. A classe ficará assim:

```
import br.com.caelum.vraptor.ioc.ComponentFactory;

@Component
public class CriadorDeSession implements ComponentFactory<Session> {

    public Session getInstance() {
        AnnotationConfiguration configuration =
            new AnnotationConfiguration();
        configuration.configure();

        SessionFactory factory = configuration
            .buildSessionFactory();
        Session session = factory.openSession();
        return session;
    }
}
```

Note que utilizamos Generics para informar qual será o tipo de objeto criado por essa classe.

O método `getInstance` ainda está fazendo coisas demais: ele cria uma `AnnotationConfiguration`, configura, constrói uma `SessionFactory`, e só então abre uma sessão. Não seria suficiente só abrir uma sessão a partir da `SessionFactory`? A `SessionFactory` é uma dependência para criar uma `Session`. Então vamos recebê-la no construtor da classe.

```
@Component
public class CriadorDeSession implements ComponentFactory<Session> {

    private final SessionFactory factory;

    public CriadorDeSession(SessionFactory factory) {
        this.factory = factory;
    }

    public Session getInstance() {
        return factory.openSession();
    }
}
```

```
}  
}
```

Podemos usar a mesma idéia e criar um `ComponentFactory` para `SessionFactory`, com o código que removemos da `CriadorDeSession`. Não podemos nos esquecer da anotação `@Component`:

```
@Component  
public class CriadorDeSessionFactory implements  
    ComponentFactory<SessionFactory> {  
  
    public SessionFactory getInstance() {  
        AnnotationConfiguration configuration =  
            new AnnotationConfiguration();  
        configuration.configure();  
  
        SessionFactory factory = configuration  
            .buildSessionFactory();  
        return factory;  
    }  
}
```

Poderíamos continuar esse processo, e criar uma `ComponentFactory` para `AnnotationConfiguration` também, mas vamos parar por aqui. Resumindo o que acabamos de fazer, chegamos a seguinte situação: `ProdutosController` depende de `ProdutoDao` que depende de `Session` que depende de `SessionFactory`.

À primeira vista pode parecer que fizemos muita coisa só pra resolver o problema da `Session`. Mas a vantagem de utilizarmos essa abordagem é que para as nossas próximas lógicas, não precisaremos fazer nada. Se tivermos que criar 50 DAOs no nosso sistema, não precisamos mais se preocupar com `Sessions`, basta receber uma no construtor.

9.2 EXERCÍCIOS

- 1) Faça a classe `ProdutoDao` receber uma `Session` no construtor. Remova o método `getSession`:

```
@Component  
public class ProdutoDao {  
  
    private final Session session;  
  
    public ProdutoDao(Session session) {  
        this.session = session;  
    }  
}
```

```
//...  
}
```

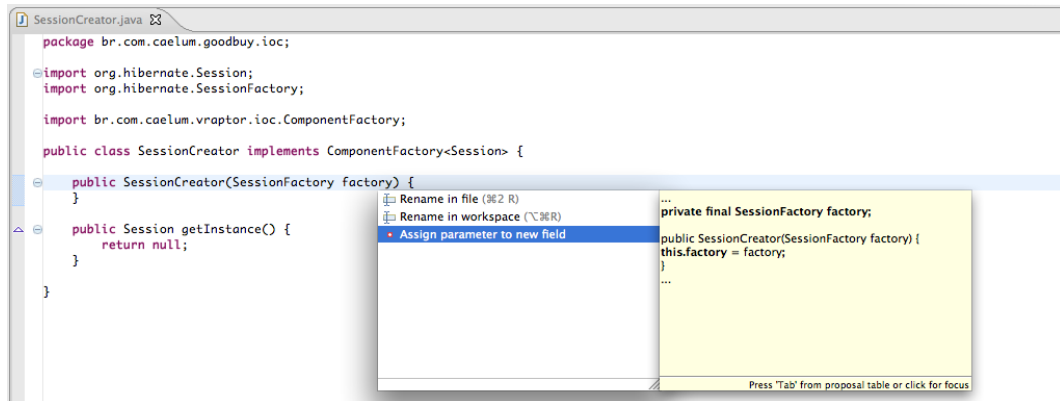
- 2) Modifique a classe `CriadorDeSession`. Faça com que essa classe implemente a interface `ComponentFactory`, passando a `Session` como tipo genérico. Renomeie o método `getSession` para `getInstance`, e remova o `static`.

```
import br.com.caelum.vraptor.ioc.ComponentFactory;  
  
@Component  
public class CriadorDeSession implements ComponentFactory<Session> {  
    public Session getInstance() {  
        AnnotationConfiguration configuration = new AnnotationConfiguration();  
        configuration.configure();  
  
        SessionFactory factory = configuration.buildSessionFactory();  
        Session session = factory.openSession();  
        return session;  
    }  
}
```

- 3) A classe `CriadorDeSession` está fazendo coisas demais. Recorte a parte que cria uma `SessionFactory` e receba uma `SessionFactory` no construtor. A classe deve ficar assim:

```
1 package br.com.caelum.goodbuy.infra;  
2  
3 // import's  
4  
5 public class CriadorDeSession implements ComponentFactory<Session> {  
6  
7     public CriadorDeSession(SessionFactory factory) {  
8     }  
9  
10    public Session getInstance() {  
11        Session session = factory.openSession();  
12        return session;  
13    }  
14  
15 }
```

- 4) Após ter criado o construtor, guarde o parâmetro `factory` em um atributo. Utilize as teclas de atalho do Eclipse. Pressione as teclas `Ctrl + 1` (quick fix) no parâmetro `factory`, e o Eclipse nos mostrará algumas sugestões. Selecione a primeira opção, `Assign parameter to new field`.



- 5) Anote a classe com @Component, para indicar que essa classe é uma dependência e pode ser instanciada pelo VRaptor sempre que necessário.
- 6) Crie a classe CriadorDeSessionFactory no pacote br.com.caelum.goodbuy.infra que implementa a interface ComponentFactory<SessionFactory>.

```

1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 public class CriadorDeSessionFactory implements
6     ComponentFactory<SessionFactory> {
7
8 }
  
```

- 7) A classe não compila, pois falta implementar o método que foi definido na interface. Para isso utilize as teclas de atalho do Eclipse: coloque o cursor no nome da classe e digite Ctrl+1. Selecione a opção *Add unimplemented methods*.

Coloque a criação da fábrica de sessões que você tinha recortado da outra classe nesse método.

```

1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 public class CriadorDeSessionFactory
6     implements ComponentFactory<SessionFactory> {
7
8     public SessionFactory getInstance() {
9         AnnotationConfiguration configuration =
10             new AnnotationConfiguration();
11         configuration.configure();
12
13         SessionFactory factory =
14             configuration.buildSessionFactory();
  
```

```
15         return factory;
16     }
17
18 }
```

- 8) Anote a classe com `@Component`, para indicar que essa classe é uma dependência e pode ser instanciada pelo VRaptor sempre que necessário.
- 9) As classes de teste que tínhamos criado não estão compilando mais, pois mudamos o construtor do `ProdutoDao`. Mude as chamadas, usando as `ComponentFactories` para criar as dependências. Note que na aplicação web, o VRaptor que vai se encarregar executar esse código.

```
SessionFactory factory = new CriadorDeSessionFactory().getInstance();
```

```
Session session = new CriadorDeSession(factory).getInstance();
```

```
ProdutoDao dao = new ProdutoDao(session);
```

- 10) Acesse o sistema e veja que ele continua funcionando como antes.

9.3 ANALISANDO O CÓDIGO

Já conseguimos cadastrar um produto pelo browser e já criamos algumas classes para injeção de dependências para nossas próximas lógicas, mas será que nosso código atual está certo?

Toda vez que precisamos de uma sessão do Hibernate, o VRaptor injeta no dao uma sessão nova. Essa sessão é criada de uma fábrica de sessões, e essa fábrica também é criada. Ou seja, para cada requisição, estamos criando uma fábrica nova. Mas temos que nos lembrar que isso não deveria ser feito. É custoso ter que criar uma fábrica de sessões para cada requisição.

Uma alternativa para resolver esse problema é utilizar um bloco estático dentro do `CriadorDeSessionFactory`. Outra alternativa seria utilizar o *Design Pattern Singleton*, fazendo com que apenas uma instância de determinada classe seja criada.

Mas como estamos utilizando o VRaptor, vamos utilizar um recurso dele para esse controle.

9.4 ESCOPOS DEFINIDOS PELO VRAPTOR

O VRaptor nos permite definir o tempo de vida de instâncias de determinada classe. Esse tempo de vida é importante porque cada componente tem uma responsabilidade, e alguns devem sobreviver por mais tempo que outros.

Alguns componentes devem ter o tempo de vida de uma requisição, ou seja, enquanto estamos atendendo uma solicitação do cliente. Esse caso seria aplicado à sessão do Hibernate, por exemplo.

Mas alguns componentes devem durar mais que uma requisição, devem durar enquanto um cliente estiver logado por exemplo. Esse seria o caso de um carrinho de compras. Para cada produto que um cliente quiser comprar, esse produto deve ser adicionado no mesmo carrinho, e enquanto o cliente estiver usando o sistema o carrinho deve continuar guardando os produtos.

Para alguns casos, um componente deve durar para todas as requisições, e esse componente pode ser utilizado por todos os clientes. Esse caso poderia ser a fábrica de sessões do Hibernate. Essa fábrica ficaria única para toda a aplicação, e qualquer cliente que precisar de uma sessão, usaríamos essa fábrica.

Para cada cenário acima, o VRaptor definiu um tempo de vida. Esse tempo de vida no VRaptor é chamado de **escopo**. O VRaptor define quatro escopos, e para utilizá-los, basta anotarmos nossos componentes com uma das seguintes anotações:

- `@RequestScoped` - o componente será criado a cada requisição. É o escopo padrão dos componentes.
- `@SessionScoped` - o componente será criado uma vez por sessão (HttpSession) de usuário
- `@ApplicationScoped` - o componente será criado uma única vez na aplicação.
- `@PrototypeScoped` - o componente será criado toda vez que for solicitado.

Para nossa classe `CriadorDeSessionFactory`, queremos ter apenas uma instância dessa classe para toda nossa aplicação, então vamos anotá-la com `@ApplicationScoped`.

```
1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 @Component
6 @ApplicationScoped
7 public class CriadorDeSessionFactory implements
8     ComponentFactory<SessionFactory> {
9
10     public SessionFactory getInstance() {
11         AnnotationConfiguration configuration =
12             new AnnotationConfiguration();
13         configuration.configure();
14         return configuration.buildSessionFactory();
15     }
16
17 }
```

Agora o VRaptor vai criar apenas uma instância dessa classe. Mas repare que o método `getInstance` cria toda vez uma fábrica nova. Precisamos refatorar para que seja criada apenas uma fábrica.

Para resolver esse problema, poderíamos criar um atributo do tipo `SessionFactory` e colocar no construtor da classe a criação da fábrica.

```
@Component
@ApplicationScoped
public class CriadorDeSessionFactory implements
    ComponentFactory<SessionFactory>{

    private final SessionFactory factory;

    public CriadorDeSessionFactory() {
        AnnotationConfiguration configuration =
            new AnnotationConfiguration();
        configuration.configure();

        factory = configuration.buildSessionFactory();
    }

    public SessionFactory getInstance() {
        return factory;
    }
}
```

Mas o VRaptor permite a utilização de anotações bem úteis do Java EE 5: o `@PostConstruct` e o `@PreDestroy`. Essas anotações devem ser anotadas em métodos e têm a seguinte semântica:

- os métodos anotados com `@PostConstruct` serão executados assim que o escopo for iniciado, ou seja, no começo da requisição, sessão de usuário ou da aplicação.
- os métodos anotados com `@PreDestroy` serão executados assim que o escopo for finalizado, ou seja, no final da requisição, sessão de usuário ou da aplicação.

Desse modo, podemos usar um método anotado com `@PostConstruct` para criar a fábrica de sessões, e como é uma boa prática fechar todos os recursos que abrimos, devemos criar um método anotado com `@PreDestroy` que vai fechar a fábrica.

Fazendo essas modificações, nossa classe ficaria assim:

```
@Component
@ApplicationScoped
public class CriadorDeSessionFactory implements
    ComponentFactory<SessionFactory> {

    private SessionFactory factory;

    @PostConstruct
    public void abre() {
        AnnotationConfiguration configuration =
```

```
        new AnnotationConfiguration();
        configuration.configure();

        this.factory = configuration.buildSessionFactory();
    }

    public SessionFactory getInstance() {
        return this.factory;
    }

    @PreDestroy
    public void fecha() {
        this.factory.close();
    }
}
```

Note que não precisamos de um bloco estático ou um singleton, pois o VRaptor é que vai se encarregar de criar apenas uma instância da fábrica, e quando o recurso não for mais necessário, o VRaptor vai fechar a fábrica.

9.5 FECHANDO A SESSÃO DO HIBERNATE

Outro problema que temos no nosso código são as sessões do Hibernate. Para cada requisição, abrimos uma sessão e utilizamos sempre que for necessário durante toda a requisição. Mas quando fechamos essa sessão? No nosso caso, **nós não estamos fechando**.

Mas onde colocaríamos o fechamento da sessão? No dao? Na lógica? Mas será que nossa lógica que deveria fechar? Não seria bom que o VRaptor fechasse para nós a sessão no final da requisição?

Utilizando as anotações que vimos na sessão anterior, `@PostConstruct` e `@PreDestroy`, conseguimos fazer um código muito elegante e de fácil entendimento.

Na classe `CriadorDeSession`, podemos criar dois métodos, um para abrir e outro para fechar a sessão. Dessa forma, não precisaremos nos preocupar com esse tipo de tarefa, que não faz parte da regra de negócio.

A classe ficaria assim:

```
1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 @Component
6 public class CriadorDeSession implements ComponentFactory<Session> {
7
```



```
8     private final SessionFactory factory;
9     private Session session;
10
11     public CriadorDeSession(SessionFactory factory) {
12         this.factory = factory;
13     }
14
15     @PostConstruct
16     public void abre() {
17         this.session = factory.openSession();
18     }
19     public Session getInstance() {
20         return this.session;
21     }
22     @PreDestroy
23     public void fecha() {
24         this.session.close();
25     }
26
27 }
```

Note que criamos um atributo chamado `session`, que guardará uma sessão do Hibernate. Temos que lembrar que componentes que controlados pelo VRaptor sempre têm um escopo, mesmo não estando anotado. O escopo padrão para componentes é o de requisição, ou seja, daria na mesma anotar essa classe com `@RequestScoped`. Desse modo, uma sessão vai ser aberta no começo da requisição, e será fechada assim que acabar a requisição

9.6 EXERCÍCIOS

- 1) Anote a classe `CriadorDeSessionFactory` com `@ApplicationScoped`.

```
1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 @Component
6 @ApplicationScoped
7 public class CriadorDeSessionFactory implements
8     ComponentFactory<SessionFactory> {
9     //...
10 }
```

- 2) Refatore essa classe para que seja criado apenas uma instância da fábrica.

```
1 package br.com.caelum.goodbuy.infra;
2
3 // import's
4
5 @Component
6 @ApplicationScoped
7 public class CriadorDeSessionFactory implements
8     ComponentFactory<SessionFactory>{
9
10     private SessionFactory factory;
11
12     @PostConstruct
13     public void abre() {
14         AnnotationConfiguration configuration =
15             new AnnotationConfiguration();
16         configuration.configure();
17
18         this.factory = configuration.buildSessionFactory();
19     }
20
21     public SessionFactory getInstance() {
22         return this.factory;
23     }
24
25     @PreDestroy
26     public void fecha() {
27         this.factory.close();
28     }
29 }
```

- 3) Refatore a classe CriadorDeSession, colocando os métodos de callback e colocando a sessão do Hibernate em um atributo.

```
@Component
public class CriadorDeSession implements ComponentFactory<Session> {

    private final SessionFactory factory;
    private Session session;

    public CriadorDeSession(SessionFactory factory) {
        this.factory = factory;
    }

    @PostConstruct
    public void abre() {
        this.session = factory.openSession();
    }
}
```

```
}  
  
public Session getInstance() {  
    return this.session;  
}  
  
@PreDestroy  
public void fecha() {  
    this.session.close();  
}  
  
}
```

- 4) Reinicie o tomcat, acesse o sistema e veja que tudo continua funcionando
- 5) (Opcional) Coloque `System.out.println` nos métodos dos `ComponentFactories`, para ver a ordem que os métodos são chamados
- 6) (Opcional) Faça em uma folha de papel todo o fluxo de cadastro de um usuário. Esse fluxo pode ser feito de qualquer forma, com flechas e caixas de texto por exemplo. O ponto importante desse exercício é ficar claro como está a arquitetura do seu projeto.

Validando formulários

Quando temos um formulário para adicionar dados no nosso sistema, geralmente precisamos verificar se os dados estão corretos e consistentes. Por exemplo não gostaríamos que o usuário digitasse “vinte e três reais” no campo preço, ou que o campo nome fosse vazio.

Então, antes de salvar esses dados no banco precisamos verificar se tudo está como esperamos. Chamamos isso de validação. Existem duas maneiras de fazer validação:

- Validação do lado do cliente: verificamos os dados antes de mandar os dados para o servidor, assim o feedback da validação é instantâneo.
- Validação do lado do servidor: os dados são enviados para o servidor, e se houver erros, o servidor devolve esses erros para o cliente.

A validação do lado do cliente geralmente é feita usando javascript, impedindo que o formulário seja submetido caso aconteça algum erro. A do lado do servidor é feita em Java, e o VRaptor pode te ajudar a fazê-la.

10.1 VALIDATOR

O VRaptor possui um componente, chamado `Validator`, para ajudar a fazer validações do lado do servidor. Com ele, você pode executar a lógica de validação, e caso haja algum erro, especificar para onde deve ser redirecionada a requisição. Para obter um `Validator` basta recebê-lo no construtor do seu `Controller`:

```
import br.com.caelum.vraptor.Validator;
```

```
@Resource
```

```
public class ProdutosController {
```

```
private final ProdutoDao dao;
private final Result result;
private final Validator validator;

public ProdutosController(ProdutoDao dao, Result result,
    Validator validator) {
    this.dao = dao;
    this.result = result;
    this.validator = validator;
}
//...
}
```

Com essa instância do Validator podemos adicionar erros de validação para o caso em que os dados da requisição não vieram corretos. Geralmente validamos quando estamos adicionando ou atualizando algo no banco, então vamos executar a validação no método adiciona.

Para isso vamos definir primeiro as regras de validação para um produto:

- **nome:** deve ser obrigatório, e ter pelo menos 3 letras.
- **descrição:** deve ser obrigatório, e ter menos de 40 letras.
- **preço:** não pode ser negativo, nem ser de graça.

Para executar essa lógica com o Validator, temos duas maneiras:

- maneira classica: você faz os if's e adiciona as mensagens manualmente:

```
public void adiciona(Produto produto) {
    if (produto.getNome() == null ||
        produto.getNome().length() < 3) {
        validator.add(new ValidationMessage(
            "Nome é obrigatório e precisa ter mais de 3 letras",
            "produto.nome"));
    }
    if (produto.getDescricao() == null ||
        produto.getDescricao().length() > 40) {
        validator.add(new ValidationMessage(
            "Descrição é obrigatória e não pode ter " +
            "mais que 40 letras", "produto.descricao"));
    }
    if (produto.getPreco() <= 0.0) {
        validator.add(new ValidationMessage(
            "Preço precisa ser positivo", "produto.preco"));
    }
    dao.salva(produto);
}
```

```
        result.redirectTo(this).lista();
    }
```

- maneira fluente: você diz o que quer que seja verdade, e caso não seja adiciona uma mensagem internacionalizada:

```
public void adiciona(final Produto produto) {
    validator.checking(new Validations() {{
        that(produto.getNome() != null &&
            produto.getNome().length() >= 3,
            "produto.nome", "nome.obrigatorio");

        that(produto.getDescricao() != null &&
            produto.getDescricao().length() <= 40,
            "produto.descricao", "descricao.obrigatoria");

        that(produto.getPreco() != null && produto.getPreco() > 0.0,
            "produto.preco", "preco.positivo");
    }});
    dao.salva(produto);
    result.redirectTo(this).lista();
}
```

Como a mensagem é internacionalizada, você precisa colocar essas chaves no seu arquivo `messages.properties`:

```
nome.obrigatorio = Nome é obrigatório e precisa ter mais
                  de 3 letras
descricao.obrigatoria = Descrição é obrigatória e não pode ter
                      mais que 40 letras
preco.positivo = Preço precisa ser positivo
```

Em ambas as maneiras você precisa especificar pra qual lógica voltar quando der erro de validação. No nosso caso, ao dar erro queremos voltar pro formulário, então fazemos:

```
validator.onErrorUsePageOf(ProdutosController.class).formulario();
```

A lógica de redirecionamento é a mesma que se você estivesse usando o `result.of()`. Você deve colocar essa chamada logo no final da sua lógica de validação, no momento em que, se houver algum erro, a execução deve parar e voltar para o formulário, mostrando os erros.

Quando existem erros de validação, o VRaptor disponibiliza para a jsp um atributo chamado `${errors}`, que contém a lista dos erros que aconteceram. Então você usar o seguinte código para mostrar as mensagens no seu jsp:

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

```
<li>${error.category} - ${error.message}</li>
</c:forEach>
</ul>
```

Além disso, se quisermos que o formulário volte preenchido quando houver algum erro de validação, precisamos passar como valor dos inputs, o que foi mandado para a requisição:

```
<form action="adiciona">
  <fieldset>
    <legend>Adicionar Produto</legend>

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

    <label for="descricao">Descrição:</label>
    <textarea id="descricao" name="produto.descricao">
      ${produto.descricao }
    </textarea>

    <label for="preco">Preço:</label>
    <input id="preco" type="text" name="produto.preco"
      value="${produto.preco }"/>

    <button type="submit">Enviar</button>
  </fieldset>
</form>
```

10.2 EXERCÍCIOS

- 1) Receba um Validator no construtor do ProdutosController e guarde-o num atributo:

```
import br.com.caelum.vraptor.Validator;

@Resource
public class ProdutosController {

  private final ProdutoDao dao;
  private final Result result;
  private final Validator validator;

  public ProdutosController(ProdutoDao dao, Result result,
    Validator validator) {
    this.dao = dao;
  }
}
```

```
        this.result = result;
        this.validator = validator;
    }
    //...
}
```

- 2) Modifique o método adiciona para que inclua a validação dos campos do produto. Se preferir, faça a validação na forma fluente.

```
public void adiciona(final Produto produto) {
    if (produto.getNome() == null ||
        produto.getNome().length() < 3) {
        validator.add(new ValidationMessage(
            "Nome é obrigatório e precisa ter mais" +
            " de 3 letras", "produto.nome"));
    }
    if (produto.getDescricao() == null ||
        produto.getDescricao().length() > 40) {
        validator.add(new ValidationMessage(
            "Descrição é obrigatória não pode ter mais" +
            " que 40 letras", "produto.descricao"));
    }
    if (produto.getPreco() <= 0) {
        validator.add(new ValidationMessage(
            "Preço precisa ser positivo", "produto.preco"));
    }
    validator.onErrorUsePageOf(ProdutosController.class)
        .formulario();

    dao.salva(produto);
    result.redirectTo(this).lista();
}
```

- 3) Abra o arquivo header.jspf, e adicione o código para mostrar os erros de validação, dentro da div com id="erros".

```
<div id="erros">
    <ul>
        <c:forEach items="${errors}" var="error">
            <li>${error.category} - ${error.message}</li>
        </c:forEach>
    </ul>
</div>
```

- 4) Adicione o código para que o formulário continue preenchido quando der erro de validação:

```
<form action="adiciona">
```



```
<fieldset>
  <legend>Adicionar Produto</legend>

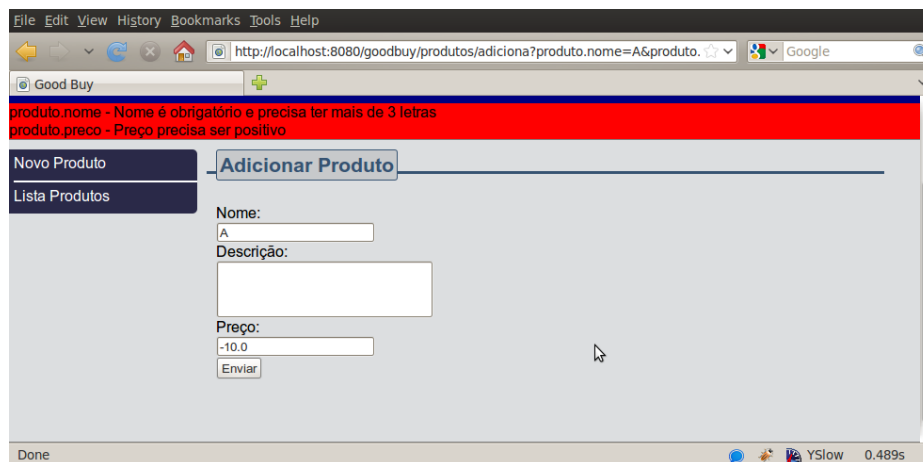
  <label for="nome">Nome:</label>
  <input id="nome" type="text" name="produto.nome"
    value="${produto.nome }"/>

  <label for="descricao">Descrição:</label>
  <textarea id="descricao" name="produto.descricao">
    ${produto.descricao }
  </textarea>

  <label for="preco">Preço:</label>
  <input id="preco" type="text" name="produto.preco"
    value="${produto.preco }"/>

  <button type="submit">Enviar</button>
</fieldset>
</form>
```

5) Abra o browser e tente adicionar um produto inválido.



6) (Opcional) O método atualiza também precisa de validação. Você consegue aproveitar o código do adiciona para fazer isso? Ele deve redirecionar para o mesmo lugar em caso de erro?

10.3 PARA SABER MAIS: HIBERNATE VALIDATOR

Hibernate Validator é um poderoso framework para validação de objetos. Ele já vem com validadores que verificam as regras mais comuns, como campos obrigatórios, tamanhos mínimos e cartões de crédito.

Além disso, o framework também é bastante flexível, já que permite a criação dos seus próprios validadores.

Para definir as regras de validação, basta anotar os campos do nosso modelo que precisam ser validados. Para tal, usaremos as anotações prontas no Hibernate Validator. Alguns exemplos:

- @NotNull, para campos obrigatórios
- @NotEmpty, idem, mas também não aceita String vazia
- @Length, para impor restrições sobre o tamanho (min e max)
- @Min e @Max, para restringir o valor de números
- @Past e @Future, para impor restrições temporais
- @Pattern, para restringir valores usando uma expressão regular
- @Email, conferir validade de e-mail

Munidos dessas anotações, vamos aplicar nossas regras de validação no Produto:

```
import org.hibernate.validator.Length;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotNull;
```

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    @NotNull
    @Length(min=3)
    private String nome;

    @NotNull
    @Length(max=40)
    private String descricao;

    @Min(0)
    private Double preco;
}
```

Com nosso modelo anotado, podemos substituir as validações que fizemos pelo Hibernate Validator. O interface Validator do VRaptor possui um método chamado `validate` que valida qualquer objeto usando o Hibernate Validator. Assim, se você chamar:

```
validator.validate(produto)
```

todos os erros de validação que ocorrerem serão adicionados ao validator automaticamente. Assim podemos substituir as validações que tínhamos feito antes, pelas equivalentes do Hibernate Validator.

10.4 EXERCÍCIOS OPCIONAIS

- 1) Se as anotações não aparecerem, vá para a pasta que você tinha extraído o zip do VRaptor e copie o jar **hibernate-validator-X.X.X.jar** que está na pasta `lib/optional/hibernate` para a pasta `WEB-INF/lib` da aplicação.
- 2) Anote os campos do produto para adicionar nossas regras de validação.

```
import org.hibernate.validator.Length;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotNull;
```

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    @NotNull
    @Length(min=3)
    private String nome;

    @NotNull
    @Length(max=40)
    private String descricao;

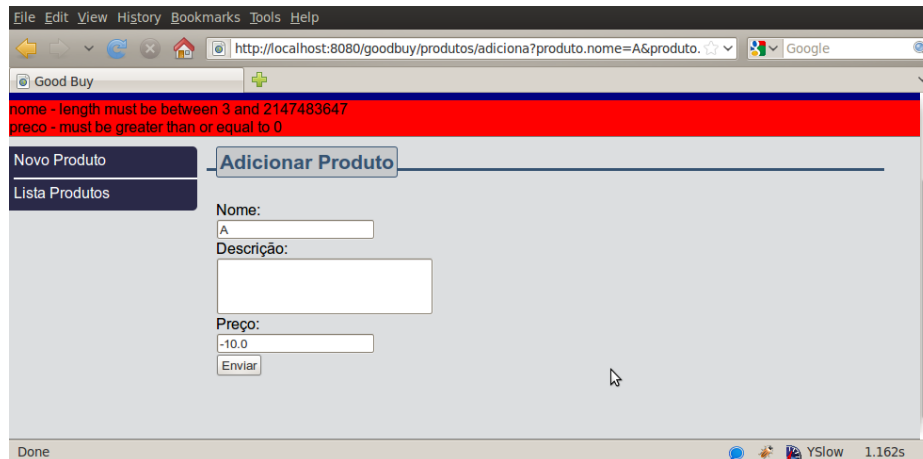
    @Min(0)
    private Double preco;
}
```

- 3) Mude a lógica de validação do método `adiciona` do `ProdutosController` para usar o `Hibernate Validator`:

```
@Resource
public class ProdutosController {
    //...
    public void adiciona(final Produto produto) {
        validator.validate(produto);
        validator.onErrorUsePageOf(ProdutosController.class).formulario();

        dao.salva(produto);
        result.redirectTo(this).lista();
    }
}
```

- 4) Acesse o browser e adicione algum produto inválido, para ver as mensagens de validação.



- 5) (Opcional) Repare que as mensagens são as mensagens padrão e estão em inglês. Para usar uma mensagem em português, precisamos passá-la para as anotações do Hibernate Validator.

```
@NotNull(message="Nome precisa ser preenchido")
@Length(min=3, message="Nome precisa ter mais de 3 letras")
private String nome;
```

Se quisermos usar internacionalização nessas mensagens, basta colocar a chave de internacionalização entre {}:

```
@NotNull(message="{nome.precisa.ser.preenchido}")
@Length(min=3, message="{nome.curto}")
private String nome;
```

e colocar as respectivas mensagens no arquivo messages.properties:

```
nome.precisa.ser.preenchido = Nome precisa ser preenchido
nome.curto = Nome muito curto. Deve ter mais de 3 letras
```

Mude as mensagens

REST

11.1 O QUE É REST

REST é um conjunto de restrições que define um padrão arquitetural com características específicas: benefícios e dificuldades determinadas aparecerão ao implementar um sistema seguindo o padrão REST.

11.2 CARACTERÍSTICAS E VANTAGENS

Permitir o endereçamento dos recursos do seu sistema de uma forma padronizada implica em algumas vantagens do REST: as URIs são bookmarkable, o conteúdo é mais suscetível a cache e os componentes agindo entre o cliente e o servidor possuem uma visão melhor do que está sendo enviado de um lado para o outro permitindo que os mesmos tomem decisões específicas para aumentar a escalabilidade ou a segurança do seu sistema.

Utilizar o protocolo HTTP não somente como uma maneira de transmitir dados mas também como um protocolo de aplicação permite maior visibilidade para os componentes intermediários.

Conteúdo hipermídia permite a navegabilidade de um cliente através do seu sistema, além de diminuir o acoplamento entre o cliente e o servidor, que costuma ser muito alto em aplicações orientadas a serviço.

O tópico de hipermídia é bem amplo e não será abordado no curso, sugerimos a visita ao site do restfulie-java para entender mais como ele funciona e suas vantagens em um sistema: <http://github.com/caelum/restfulie-java>

11.3 O TRIÂNGULO DO REST

Uma requisição web tem três tipos de componentes importantes: substantivos, verbos e tipos de conteúdo (Content Types). Uma aplicação RESTful costuma reforçar mais os substantivos do que os verbos, além de conter uma gama de tipos de conteúdo determinados. A vantagem de seguir isso é que conseguimos aproveitar toda a estrutura que o protocolo HTTP proporciona.

SUBSTANTIVOS: RECURSOS

Substantivos são os nomes dos recursos do seu sistema. Quando fazemos requisições Web, precisamos falar o caminho da mesma, a URI (*Unified Resource Identifier*), ou seja, o identificador único de um recurso.

Então ao criar as URIs do nosso sistema devemos levar em conta que elas representam recursos, não ações. Em sistemas REST, nossas URIs devem conter apenas substantivos, que são nossos recursos: `/produtos/adiciona` não é boa, pois contém um verbo e não está identificando um recurso, mas sim uma operação. Para representar a adição de um produtos podemos usar a URI `/produtos` com o método HTTP POST, que representa que estamos adicionando alguma informação no sistema.

VERBOS: OPERAÇÕES

Uma das características mais importantes de REST é que você tenha um conjunto pequeno e fixo de operações bem definidas, gerando uma interface uniforme. Desse modo, para duas aplicações conversarem elas não precisam implementar diversas operações: basta usar as operações definidas. Perceba que adicionar um produto e adicionar um usuário no sistema são operações equivalentes.

O protocolo HTTP possui sete operações -- os métodos GET, POST, PUT, DELETE, HEAD, OPTIONS e TRACE. Duas delas (GET e POST) já são bem conhecidas e utilizadas: GET quando clicamos em links ou digitamos o endereço no navegador, e POST quando preenchemos formulários de cadastro. Cada método tem uma semântica diferente e juntando o método à URI deveríamos conseguir representar todas as ações do nosso sistema. As semânticas principais são:

- **GET** - recupera informações sobre o recurso identificado pela URI. Ex: listar produtos, visualizar o produto 45. Uma requisição GET não deve modificar nenhum recurso do seu sistema, ou seja, não deve ter nenhum efeito colateral, você apenas recupera informações do sistema.
- **POST** - adiciona informações usando o recurso da URI passada. Ex: adicionar um produto. Pode adicionar informações a um recurso ou criar um novo recurso.
- **PUT** - adiciona (ou modifica) um recurso na URI passada. Ex: atualizar um produto. A diferença fundamental entre um PUT e um POST é que no POST a URI significa o lugar que vai tratar a informação, e no PUT significa o lugar em que a informação será armazenada.
- **DELETE** - remove o recurso representado pela URI passada. Ex: remover um produto.
- **HEAD, OPTIONS e TRACE** - recuperam metadados da URI passada. Respectivamente o Header, quais métodos são possíveis e informações de debug.

CONTENT TYPE: REPRESENTAÇÃO

Quando fazemos uma aplicação não trafegamos um recurso pela rede, apenas uma representação dele. Por exemplo, quando queremos adicionar um produto ao sistema, passamos para o servidor uma representação do produto: dados do formulário. E quando pedimos uma lista de produtos para o servidor ele responde com uma representação HTML desses produtos. Mas não precisa ser restrito a isso: poderíamos adicionar um produto ao sistema via uma representação em XML, e o servidor poderia nos devolver uma lista de produtos em formato JSON.

Resumindo: nossas URIs devem representar recursos, as operações no recurso devem ser indicadas pelos métodos HTTP e podemos falar qual é o formato em que conversamos com o servidor com o Content Type. Nas próximas seções vamos ver como aplicar essas idéias usando o VRaptor.

11.4 MUDANDO A URI DA SUA LÓGICA: @PATH

Como vimos anteriormente, o VRaptor possui uma convenção para gerar as URIs das suas lógicas. Mas, se você está fazendo seu sistema seguindo REST, essa convenção não é muito boa: métodos geralmente contêm verbos, e não é bom colocar verbos nas URIs. Então precisamos sobrescrever essa convenção do VRaptor, e podemos fazer isso facilmente usando a anotação `@Path`. Por exemplo, se quisermos que o método `lista` do `ProdutosController` responda à URI `/produtos`, basta anotá-lo:

```
import br.com.caelum.vraptor.Path;

@Resource
public class ProdutosController {

    @Path("/produtos")
    public List<Produto> lista() {
        return dao.listaTudo();
    }
}
```

Dessa forma, o método `lista` não vai mais estar acessível pela URI `/produto/lista`, ele apenas poderá ser acessado pela URI `/produtos`.

Com a anotação `@Path` ainda é possível usar *templates* para extrair parâmetros a partir da URI. Por exemplo, no nosso método `edita`, precisamos passar um *query parameter* para falar qual é o produto que queremos editar:

```
/produto/edita?id=10
```

As URIs devem identificar recursos (além de não conter verbos, mas vamos tirá-los em breve), e o `id` é algo que **identifica** o recurso que queremos acessar, então ficaria melhor se passássemos esse `id` **dentro** da URI, como por exemplo:

/produto/10/edita

ou seja, uma URI que representa a edição do produto de id 10. Para extrair esse 10 da URI, podemos colocar um template no nosso @Path. Para isso basta colocar o nome do parâmetro que queremos extrair entre chaves, dentro da URI do @Path, e então receber o parâmetro como argumento do método:

```
@Path("/produto/{id}/edita")
public Produto edita(Long id) {
    return dao.carrega(id);
}
```

Assim é possível criar URIs mais representativas, que identificam verdadeiramente recursos específicos.

11.5 MUDANDO O VERBO HTTP DOS SEUS MÉTODOS

Podemos também mudar o(s) verbo(s) HTTP que os métodos dos nossos controllers respondem. Por padrão, você pode acessar seus métodos usando qualquer um dos verbos HTTP. Mas, como vimos, cada verbo HTTP tem uma semântica diferente, e devemos levar em conta essas semânticas quando acessamos nossas lógicas. Se uma operação não segue a semântica de algum dos verbos, não deve aceitar o mesmo.

Por exemplo, nosso método `lista` é uma operação idempotente: não tem nenhum efeito colateral, posso chamá-lo várias vezes e, se o banco de dados não mudou, o resultado vai ser o mesmo. Logo faz sentido que o verbo HTTP para acessar esse método seja o **GET**. Para fazer essa restrição, basta anotar o método com @Get:

```
import br.com.caelum.vraptor.Path;
import br.com.caelum.vraptor.Get;

@Resource
public class ProdutosController {

    @Path("/produtos")
    @Get
    public List<Produto> lista() {
        return dao.listaTudo();
    }
}
```

ou ainda o atalho:

```
import br.com.caelum.vraptor.Get;
```



```
@Resource
public class ProdutosController {

    @Get("/produtos")
    public List<Produto> lista() {
        return dao.listaTudo();
    }
}
```

Se quisermos permitir outros verbos HTTP, podemos usar as anotações `@Post`, `@Put`, `@Delete`, `@Trace` e `@Head`. A partir do momento que colocamos uma anotação de verbo HTTP no nosso método, ele não poderá ser acessado pelos outros verbos. Então se tentarmos fazer uma requisição `POST /produtos`, ela não vai cair no método `lista`, a requisição vai retornar um status **405** (Método não suportado), que quer dizer que existe um recurso nessa URI, mas ele não suporta o verbo HTTP da requisição.

Assim, podemos colocar mais de um método que responda à mesma URI, desde que os verbos HTTP sejam diferentes. Por exemplo, podemos fazer com que o método `adiciona` também responda à URI `/produtos`. Mas o método `adiciona` não é idempotente: se eu repetir duas vezes uma requisição a esse método, vou adicionar dois produtos ao sistema. Estamos adicionando produtos, mas não sabemos qual vai ser a URI dele, então podemos colocar o verbo **POST**.

```
import br.com.caelum.vraptor.Path;
import br.com.caelum.vraptor.Get;
import br.com.caelum.vraptor.Post;

@Resource
public class ProdutosController {

    @Post("/produtos")
    public void adiciona(Produto produto) {
        //...
    }

    @Get("/produtos")
    public List<Produto> lista() {
        return dao.listaTudo();
    }
}
```

11.6 REFATORANDO O PRODUTOSCONTROLLER

O nosso `ProdutosController` é a classe responsável por controlar um recurso do sistema: o `Produto`. Nele podemos realizar várias operações em cima dos `Produtos`, então podemos padronizar as URIs que vão exe-

cutar tais operações. Vamos remover todos os verbos das URIs, e usar o verbo HTTP para determinar qual será a operação, assim podemos criar URIs que representam recursos.

A URI `/produtos` representa a lista de todos os produtos do sistema. E as operações possíveis são:

```
GET /produtos => recupera a lista de todos os
                  produtos. Método lista.
POST /produtos => adiciona um produto na lista de todos os
                  produtos. Método adiciona.
```

As URIs do tipo `/produtos/42` representam um produto específico, no caso de id 42. As operações possíveis são:

```
GET /produtos/4 => mostra o produto de id 4. Método edita.
PUT /produtos/10 => atualiza o produto de id 10. Método atualiza.
DELETE /produtos/3 => remove o produto de id 3. Método remove.
```

Nem todas as URIs precisam representar recursos fixos, mas elas precisam semanticamente representar o mesmo tipo de recurso. Por exemplo a URI `/produtos/ultimo` pode representar o último produto adicionado e a URI `/produtos/maisVendidos` pode representar uma lista dos produtos mais vendidos, que são recursos bem definidos. No nosso caso precisamos de uma URI para identificar o formulário de adição de um produto. Vamos então usar a seguinte:

```
GET /produtos/novo => mostra o formulário para adicionar um novo
                    produto. Método formulario.
```

Com essa especificação de operações no recurso Produto, podemos usar as anotações vistas anteriormente para deixar o `ProdutosController` de acordo com ela:

```
@Resource
public class ProdutosController {

    @Get("/produtos/novo")
    public void formulario() {...}

    @Get("/produtos/{id}")
    public Produto edita(Long id) {...}

    @Put("/produtos/{produto.id}")
    public void altera(Produto produto) {...}

    @Post("/produtos")
    public void adiciona(final Produto produto) {...}
```

```
@Delete("/produtos/{id}")
public void remove(Long id) {...}

@Get("/produtos")
public List<Produto> lista() {...}

}
```

Note no método altera que o parâmetro extraído é o **produto.id**. O VRaptor vai se comportar da mesma forma que se esse `produto.id` tivesse vindo da requisição: vai popular o campo `id` do produto com o valor extraído.

Agora que mudamos as URIs dos métodos, precisamos atualizar as nossas jsps para usar as URIs novas. Podemos usar a tag **c:url** para poder usar as URIs absolutas, a partir do nosso nome de contexto.

Primeiro na jsp do formulário, precisamos mudar a action do form para `/produtos`, e o method para `POST`.

```
<form action="<c:url value="/produtos"/>" method="POST">
```

Na listagem de produtos, temos um link para a edição, vamos mudá-lo:

```
<a href="<c:url value="/produtos/${produto.id}"/>">Editar</a>
```

Ainda existe o link de Remoção, mas o método `remove` do nosso controller só aceita o verbo `DELETE`! Como fazer uma requisição com o verbo `DELETE` de dentro da sua página? Infelizmente os browsers atuais só conseguem fazer requisições `GET` (através de links e formulários) e `POST` (através de formulários). Para conseguir usar os outros verbos, podemos fazer duas coisas:

- mudar o verbo `HTTP` da requisição via javascript
- passar um parâmetro adicional, especificando qual deveria ser o método da requisição. No VRaptor, esse parâmetro deve se chamar **_method**.

Então, para criar o link de remoção, precisamos criar um formulário (não podemos usar um `<a href>` pois isso geraria uma requisição `GET` que não pode ter efeitos colaterais), que passa o parâmetro `_method=DELETE`. Podemos ainda fazer com que o botão do formulário pareça um link, para que a página de listagem pareça a mesma de antes:

```
<!-- Esse pedaço de css já está adicionado no projeto base -->
<style type="text/css">
.link {
    text-decoration: underline;
    border: none;
    background: none;
    color: blue;
```

```
        cursor: pointer;
    }
</style>

<!-- ... -->

<td>
    <form action="<c:url value="/produtos/${produto.id}"/>" method="POST">
        <button class="link" name="_method" value="DELETE">Remover</button>
    </form>
</td>
```

Por último precisamos mudar o formulário de edição, colocando a action correta, e mudando o método para PUT. Mas como colocar `method="PUT"` no nosso formulário não funciona, precisamos passar o parâmetro `_method`:

```
<form action="<c:url value="/produtos/${produto.id }"/>" method="POST">
    <fieldset>
        <legend>Editar Produto</legend>

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

        <label for="descricao">Descrição:</label>
        <textarea id="descricao" name="produto.descricao">
            ${produto.descricao }
        </textarea>

        <label for="preco">Preço:</label>
        <input id="preco" type="text" name="produto.preco"
            value="${produto.preco }"/>
        <!--          vvvvvvvv          vvv -->
        <button type="submit" name="_method" value="PUT">
            Enviar
        </button>
    </fieldset>
</form>
```

Não precisamos mais passar o input hidden com `produto.id`, porque essa informação já está na URI!

BUTTON E O INTERNET EXPLORER

Em algumas versões do Internet Explorer, não é possível usar o name e value do button para mandar parâmetros na requisição. Nesse caso você precisa trocar o button por:

```
<input type="hidden" name="_method" value="PUT"/>
<input type="submit" value="Enviar"/>
```

11.7 EXERCÍCIOS

- 1) Anote os métodos do ProdutosController para seguirmos a nossa especificação de Produto:

Recurso Produto:

GET /produtos => recupera a lista de todos os produtos.

Método lista.

POST /produtos => adiciona um novo produto.

Método adiciona.

GET /produtos/4 => mostra o produto de id 4.

Método edita.

PUT /produtos/10 => atualiza o produto de id 10.

Método atualiza.

DELETE /produtos/3 => remove o produto de id 3.

Método remove.

GET /produtos/novo => mostra o formulário para adicionar um novo produto. Método formulario.

- 2) Abra o formulario.jsp e mude a action do form:

```
<form action="<c:url value="/produtos"/>" method="POST">
```

- 3) Abra o edita.jsp e mude a action e o método do form. Lembre-se que você não precisa mais do input hidden do produto.id.

```
<form action="<c:url value="/produtos/${produto.id }"/>"
method="POST">
  <!-- ... -->
  <button type="submit" name="_method" value="PUT">
    Enviar
  </button>
</form>
```

- 4) Modifique o lista.jsp, colocando o link para Edição e o “link” para Remoção:

```
<td><a href="<c:url value="/produtos/${produto.id}"/>">
    Editar
</a></td>
<td>
    <form action="<c:url value="/produtos/${produto.id}"/>"
        method="POST">
        <button class="link" name="_method" value="DELETE">
            Remover
        </button>
    </form>
</td>
```

5) Abra o `header.jspf` e modifique os links do menu:

```
<div id="menu">
    <ul>
        <li><a href="<c:url value="/produtos/novo"/>">
            Novo Produto
        </a></li>
        <li><a href="<c:url value="/produtos"/>">
            Lista Produtos
        </a></li>
    </ul>
</div>
```

AJAX e efeitos visuais

12.1 O QUE É AJAX?

AJAX (*Asynchronous Javascript and XML*) é um conjunto de técnicas de desenvolvimento Web para executar tarefas do lado do cliente, por exemplo modificar pedaços de uma página sem ter que carregar ela inteira. Na verdade o mecanismo é muito simples. De acordo com alguma ação um javascript envia uma requisição ao servidor como se fosse em background. Na resposta dessa requisição vem um XML que o javascript processa e modifica a página segundo essa resposta.

É o efeito que tanto ocorre no gmail e google maps.

Há várias ferramentas para se trabalhar com Ajax em Java. DWR e Google Web Toolkit são exemplos famosos de ferramentas que te auxiliam na criação de sistemas que usam AJAX.

Nós utilizaremos o VRaptor no servidor para nos ajudar com as requisições Ajax. No cliente, usaremos a biblioteca JQuery que é totalmente escrita em JavaScript, portanto independente de linguagem do servidor.

Há vários frameworks javascript disponíveis no mercado além do JQuery. Apenas para citar os mais famosos: Prototype/Script.aculo.us, Yahoo User Interface (YUI), Dojo. Usaremos o JQuery devido a sua extrema simplicidade de uso, muito boa para quem não domina javascript mas quer usar recursos de Ajax.

12.2 UM POUCO DE JQUERY

A biblioteca JQuery é baseada em um conceito de encadeamento (**chaining**). As chamadas de seus métodos são encadeadas uma após a outra, o que cria código muito simples de ser lido.

O ponto de partida do JQuery é a função \$ que seleciona elementos DOM a partir de seletores CSS. Para selecionar um nó com id “teste” por exemplo, fazemos:

```
$('#teste')
```

ou para selecionar os elementos que possuem a classe produto:

```
$('.produto')
```

SELETORES

O JQuery suporta XPath e CSS 3, com seletores avançadíssimos (além dos clássicos id e class).
Veja mais aqui: <http://docs.jquery.com/Selectors>

Depois de selecionar o(s) elemento(s) desejado(s), podemos chamar métodos para as mais variadas coisas. O Hello World do JQuery mostra como exibir e esconder um div específico:

```
$('#meuDiv').show()  
$('#meuDiv').hide()
```

Um outro ponto forte do JQuery é que ele possui diversos plugins que fazem várias coisas interessantes como validação de formulários, autocomplete, efeitos visuais, *drag 'n' drop*, etc. Uma lista de plugins disponíveis pode ser encontrada em <http://plugins.jquery.com/>. Uma documentação mais completa sobre o JQuery se encontra em <http://docs.jquery.com/> e uma documentação visual em <http://www.visualjquery.com/>.

MAIS SOBRE JAVASCRIPT

Nesse curso não daremos foco no desenvolvimento de código javascript, boas práticas e características dessa linguagem, apenas usaremos plugins do JQuery. Mas o código javascript é uma parte bastante importante do desenvolvimento Web, e precisamos ter com ele os mesmos cuidados que temos com o código java.

Um conteúdo aprofundado sobre javascript, além de CSS e HTML, pode ser encontrado no curso **WD-43 | Desenvolvimento Web com HTML, CSS e JavaScript**.

12.3 VALIDANDO FORMULÁRIOS COM O JQUERY

Quando fizemos a validação do formulário de adição de produtos, precisávamos enviar uma requisição para o servidor, executando algumas lógicas para decidir se os dados estavam corretos, para então responder para o usuário que os seus dados não estão válidos. Isso pode ser evitado se fizermos a validação do lado do cliente.

Além de ser bem mais rápido, evita que seja feita uma requisição desnecessária para o servidor.

Existe um plugin do JQuery chamado **Validation** (<http://bassistance.de/jquery-plugins/jquery-plugin-validation/>) que torna a validação do lado do cliente bem simples. Para usá-lo você precisa importar os javascripts no seu jsp:

```
<script type="text/javascript" src="../../../jquery-1.3.2.min.js"></script>
<script type="text/javascript" src="../../../jquery.validate.min.js"></script>
```

Você também precisa falar qual vai ser o form que vai ser validado. Para isso você pode adicionar um id ao seu formulário, e usar um seletor do JQuery para torná-lo "validável":

```
<form id="produtosForm" action="<c:url value="/produtos"/>"
      method="POST">
  ...
```

```
<script type="text/javascript">
  $('#produtosForm').validate();
</script>
```

Assim você pode adicionar as restrições aos seus campos, lembrando que o nome do produto é obrigatório e tem que ter o tamanho maior que 3, a descrição também é obrigatória e o tamanho tem que ser menor que 40, e o preço maior que zero:

```
<form action="<c:url value="/produtos"/>" method="POST">
  <fieldset>
    <legend>Adicionar produtos</legend>

    <label for="nome">Nome:</label>
    <input id="nome" class="required" minlength="3"
      type="text" name="produto.nome" value="{produto.nome }"/>

    <label for="descricao">Descrição:</label>
    <textarea id="descricao" class="required" maxlength="40"
      name="produto.descricao">{produto.descricao }</textarea>

    <label for="preco">Preço:</label>
    <input id="preco" min="0"
      type="text" name="produto.preco" value="{produto.preco }"/>

    <button type="submit">Enviar</button>
  </fieldset>
</form>
```

Se você não gosta de colocar mais atributos nos seus campos de formulário, você ainda pode definir todas as regras de validação de uma vez só, usando os **names** dos inputs, dentro da parte rules:

```
<script type="text/javascript">
    $('#produtosForm').validate({
        rules: {
            "produto.nome": {
                required: true,
                minlength: 3
            },
            "produto.descricao": {
                required: true,
                maxlength: 40
            },
            "produto.preco": {
                min: 0.0
            }
        }
    });
</script>
```

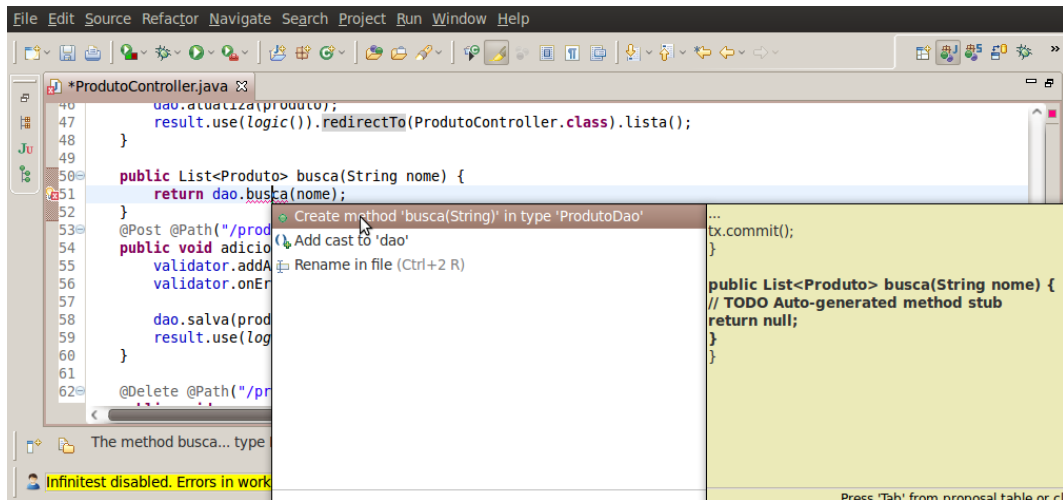
12.4 CRIANDO A BUSCA DE PRODUTOS

Como todo loja virtual que se preze, precisamos criar uma funcionalidade de busca de produtos. Para isso vamos criar uma lógica que efetua a busca no banco, dentro do `ProdutosController`:

```
@Resource
public class ProdutosController {
    //...

    public List<Produto> busca(String nome) {
        return dao.busca(nome);
    }
}
```

Repare que o nosso dao ainda não possui o método `busca`, mas podemos usar o eclipse para criar esse método pra nós. Para isso, coloque o cursor em cima do erro de compilação e aperte `Ctrl+1`. No menu que apareceu selecione **Create method 'busca(String)' in type 'ProdutoDao'**.



Agora podemos modificar o método criado no ProdutoDao para realmente buscar os produtos que contém a string passada no nome. Para isso usaremos a API de Criteria do Hibernate:

```
@Component
public class ProdutoDao {

    //...
    public List<Produto> busca(String nome) {
        return session.createCriteria(Produto.class)
            .add(Restrictions.ilike("nome", nome, MatchMode.ANYWHERE))
            .list();
    }
}
```

Traduzindo essa chamada: “crie uma Criteria de produtos, com a restrição de que o nome contenha a string passada em qualquer lugar, ignorando maiúsculas e minúsculas”.

Agora precisamos criar o jsp que mostra os resultados da busca, em /WEB-INF/jsp/produto/busca.jsp. Podemos aproveitar o jsp de listagem para mostrar a tabela:

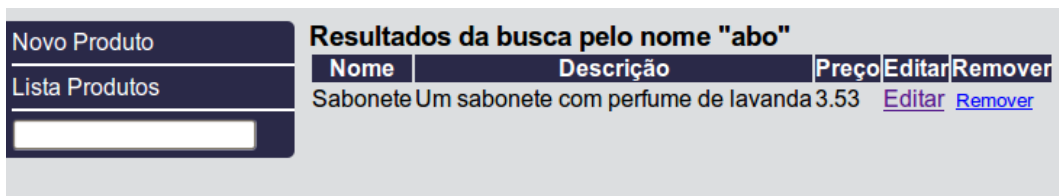
```
<h3>Resultados da busca pelo nome <b>"${nome }"</b></h3>
<%@ include file="lista.jsp" %>
```

Precisamos também passar o nome que foi buscado, então podemos mudar o método busca do ProdutosController:

```
public List<Produto> busca(String nome) {
    result.include("nome", nome);
    return dao.busca(nome);
}
```

Por último, vamos criar um formulário para poder acessar essa busca. Abra o arquivo `header.jspf` e modifique o `div menu`:

```
<div id="menu">
  <ul>
    <li><a href="<c:url value="/produtos/novo"/>">Novo Produto</a></li>
    <li><a href="<c:url value="/produtos"/>">Lista Produtos</a></li>
    <li><form action="<c:url value="/produto/busca"/>">
      <input name="nome"/>
    </form>
    </li>
  </ul>
</div>
```



Para que não fique um input perdido no menu, sem que as pessoas saibam o que ele faz, podemos usar um plugin do JQuery bem simples chamado Puts (<http://github.com/cairesvs/Puts>) . Esse plugin coloca um texto em cima do input, e quando a pessoa clica nele, o texto some. Assim podemos colocar o texto “Busca de produtos por nome” no nosso input. Para usar esse plugin, precisamos baixar o javascript e importá-lo na nossa página, e então usar o método `puts()` para colocar nosso texto.

```
<script type="text/javascript"
  src="<c:url value="/javascripts/jquery.puts.js"/>"></script>
...
<li><form action="<c:url value="/produto/busca"/>">
  <input id="busca" name="nome"/>
</form>
<script type="text/javascript">
  $("#busca").puts("Busca produtos por nome");
</script>
</li>
```

Assim temos o seguinte resultado:

Novo Produto
 Lista Produtos

Resultados da busca pelo nome "abo"

Nome	Descrição	Preço	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover

12.5 MELHORANDO A BUSCA: AUTOCOMPLETE

Podemos ainda melhorar nossa busca, mostrando dicas de produtos já existentes enquanto o usuário está digitando. Queremos um resultado parecido com este:

Novo Produto
 Lista Produtos

Sabonete(3.53)
 Sapato(132.00)
 Sabre de luz(120.00)
 Abridor de garrafas(2.50)

Resultados da busca pelo nome "ab"

Nome	Descrição	Preço	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover
Sabre de luz	faz uóon quando você mexe	120.00	Editar	Remover
Abridor de garrafas	Abre garrafas	2.50	Editar	Remover

Para isso vamos usar um plugin do JQuery chamado AutoComplete (<http://bassistance.de/jquery-plugins/jquery-plugin-autocomplete/>) que faz *autocomplete* de campos de texto. Esse plugin faz uma requisição ajax para uma URL que retorna um JSON com os dados que vão aparecer no autocomplete.

Por causa desse plugin, precisamos criar uma lógica que gere um JSON para nós. No VRaptor, gerar JSON é bem simples:

```
import static br.com.caelum.vraptor.view.Results.*;
//...
@Get("/produtos/busca.json")
public void buscaJson(String nome) {
    result.use(json()).from(dao.busca(nome)).serialize();
}
```

Ou seja, você passa para o método `from()` o que você quer serializar em JSON. Pode ser um objeto qualquer, mas no nosso caso vai ser uma lista. Ao chamarmos a URI desse método no browser, passando um nome qualquer, por exemplo `http://localhost:8080/goodbuy/produtos/busca.json?nome=a` é retornado um JSON parecido com:

```
{"list": [  
  {  
    "id": 1,  
    "nome": "Sabonete",  
    "descricao": "Um sabonete com perfume de lavanda",  
    "preco": "3.53"  
  },  
  {  
    "id": 3,  
    "nome": "Sapato",  
    "descricao": "um sapato de couro",  
    "preco": "132.00"  
  }  
]}
```

Mas como vamos buscar os produtos apenas por nome, não precisamos de todas essas informações, só precisamos de uma lista de nomes, talvez com o preço. Para isso podemos excluir as outras propriedades:

```
import static br.com.caelum.vraptor.view.Results.*;  
//...  
@Get @Path("/produtos/busca.json")  
public void buscaJson(String nome) {  
    result.use(json()).from(dao.busca(nome))  
        .exclude("id", "descricao")  
        .serialize();  
}
```

Gerando o seguinte json:

```
{"list": [  
  {  
    "nome": "Sabonete",  
    "preco": "3.53"  
  },  
  {  
    "nome": "Sapato",  
    "preco": "132.00"  
  }  
]}
```

Podemos também remover o {"list": } usando o método `.withoutRoot()`:

```
result.use(json()).withoutRoot()  
    .from(dao.busca(nome))
```

```
.exclude("id", "descricao")
.serialize();
```

que gera o json:

```
[
  {
    "nome": "Sabonete",
    "preco": "3.53"
  },
  {
    "nome": "Sapato",
    "preco": "132.00"
  }
]
```

Agora estamos prontos para usar o plugin de *autocomplete*. Para isso, abra o arquivo “header.jspf”, e adicione o javascript e o css do plugin. Use o mesmo lugar do plugin Puts:

```
<link
  href=<c:url value="/javascripts/jquery.autocomplete.css"/>"
  rel="stylesheet" type="text/css" media="screen" />
<script type="text/javascript"
  src=<c:url value="/javascripts/jquery.autocomplete.min.js"/>"></script>
...
<script type="text/javascript">
  $("#busca").puts("Busca produtos por nome");
  $("#busca").autocomplete('/goodbuy/produtos/busca.json');
</script>
```

Isso deveria ser o suficiente, mas o plugin AutoComplete espera que você mande os dados separados por pipe(|) ou um dado por linha. Como queremos usar JSON, precisamos configurar isso no plugin:

```
$("#busca").autocomplete('/goodbuy/produtos/busca.json', {
  dataType: "json", // pra falar que vamos tratar um json
  parse: function(produtos) { // para tratar o json
    // a função map vai iterar por toda a lista,
    // e transformar os dados usando a função passada
    return $.map(produtos, function(produto) {
      return {
        // todos os dados do produto
        data: produto,
        // o valor lógico do produto
        value: produto.nome,
      }
    });
  }
});
```

```
        // o que vai aparecer ao selecionar
        result: produto.nome
    });
},
// o que vai aparecer na lista de autocomplete
formatItem: function(produto) {
    return produto.nome + "(" + produto.preco + ")";
}
});
```

Além disso, o plugin passa como parâmetro da requisição o que você digitou no input, numa variável chamada `q` então você precisa modificar a lógica de busca para o parâmetro se chamar `q`:

```
@Get("/produtos/busca.json")
public void buscaJson(String q) {
    result.use(json()).withoutRoot()
        .from(dao.busca(q))
        .exclude("id", "descricao")
        .serialize();
}
```

PARA SABER MAIS: C:URL DENTRO DE JAVASCRIPT

Repare que estamos passando o nome de contexto (`goodbuy`) nas URLs do javascript acima. Isso é uma péssima prática, pois o nome de contexto depende de como você fez o deploy de sua aplicação. Do jeito que fizemos, somos obrigados a sempre fazer o deploy da aplicação com o nome “goodbuy”.

Para corrigir isso, podemos usar a tag `c:url` dentro do javascript:

```
$("#busca").autocomplete('<c:url value="/produtos/busca.json"/>');
```

É possível usar qualquer tag JSTL ou Expression Language dentro dos javascripts das suas jsp's. Mas cuidado, pois seus javascripts podem ficar mais difíceis de entender.

12.6 EXERCÍCIOS

- 1) Modifique os formulários de adição e edição de produtos para incluir a validação do lado do cliente. Os javascripts do plugin já estão importados no projeto base. Se preferir use a outra forma de validação, passando as opções para o método `validate()`.


```
<form id="produtosForm" action="<c:url value="/produtos"/>"
  method="POST">
  <fieldset>
    <legend>Adicionar produtos</legend>

    <label for="nome">Nome:</label>
    <input id="nome" class="required" minlength="3"
      type="text" name="produto.nome"
      value="{produto.nome }"/>

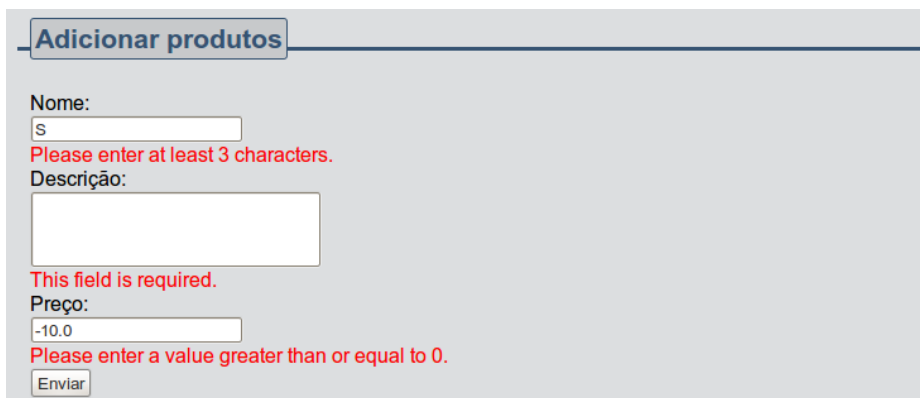
    <label for="descricao">Descrição:</label>
    <textarea id="descricao" class="required"
      maxlength="40" name="produto.descricao">
      ${produto.descricao }
    </textarea>

    <label for="preco">Preço:</label>
    <input id="preco" min="0" type="text"
      name="produto.preco" value="{produto.preco }"/>

    <button type="submit">Enviar</button>
  </fieldset>
</form>

<script type="text/javascript">
  $(' #produtosForm').validate();
</script>
```

- 2) Tente adicionar um produto inválido.



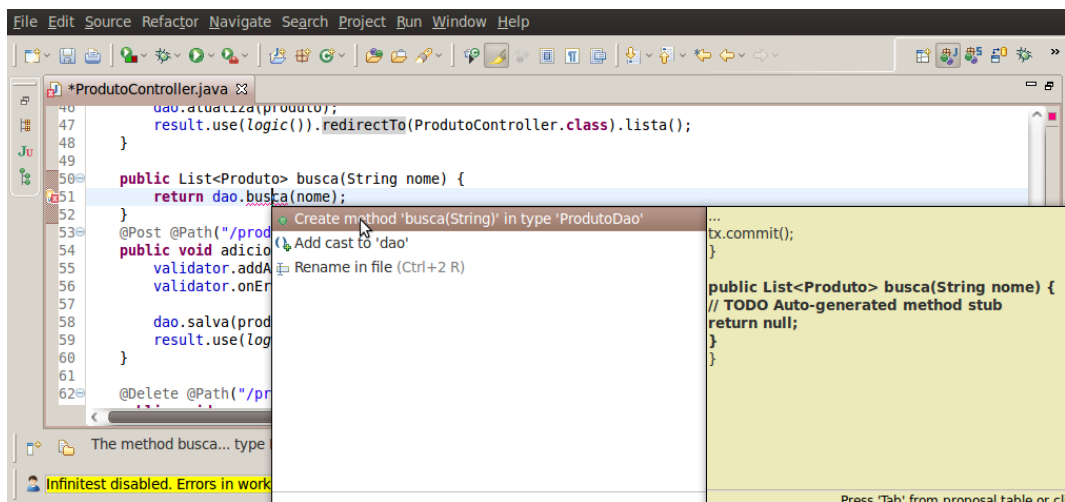
The screenshot shows a web form titled "Adicionar produtos". It contains three input fields: "Nome:" with the value "S" and a red error message "Please enter at least 3 characters."; "Descrição:" which is empty and has a red error message "This field is required."; and "Preço:" with the value "-10.0" and a red error message "Please enter a value greater than or equal to 0.". There is an "Enviar" button at the bottom.

- 3) (Opcional) As mensagens de erro estão em inglês. Tente procurar na documentação do plugin como fazer para que as mensagens fiquem em português.
- 4) Crie um método em `ProdutosController` para a listagem de produtos.

```
@Resource
public class ProdutosController {
    //...

    public List<Produto> busca(String nome) {
        result.include("nome", nome);
        return dao.busca(nome);
    }
}
```

- 5) Use o atalho Ctrl+1 para criar o método busca no ProdutoDao.



- 6) Implemente o método busca no ProdutoDao.

```
@Component
public class ProdutoDao {
    //...

    public List<Produto> busca(String nome) {
        return session.createCriteria(Produto.class)
            .add(Restrictions.ilike("nome", nome, MatchMode.ANYWHERE))
            .list();
    }
}
```

- 7) Crie o JSP de resultado da busca, em /WEB-INF/jsp/produto/busca.jsp

```
<h3>Resultados da busca pelo nome <b>"${nome }"</b></h3>
<%@ include file="lista.jsp" %>
```

- 8) Abra o arquivo header.jspf e modifique o menu para incluir um formulário de busca:

```
<div id="menu">
  <ul>
    <li><a href="<c:url value="/produtos/novo"/>">Novo Produto</a></li>
    <li><a href="<c:url value="/produtos"/>">Lista Produtos</a></li>
    <li><form action="<c:url value="/produto/busca"/>">
      <input name="nome"/>
    </form>
    </li>
  </ul>
</div>
```

9) Faça buscas usando esse formulário.

Resultados da busca pelo nome "Sab"				
Nome	Descrição	Preço	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover
Sabre de luz	faz uóon quando você mexe	120.00	Editar	Remover

10) Modifique o formulário de busca para usar o plugin Puts e deixar uma mensagem dentro do input.

```
<div id="menu">
  <ul>
    <li><a href="<c:url value="/produtos/novo"/>">Novo Produto</a></li>
    <li><a href="<c:url value="/produtos"/>">Lista Produtos</a></li>
    <li><form action="<c:url value="/produto/busca"/>">
      <input id="busca" name="nome"/>
    </form>
    <script type="text/javascript">
      $("#busca").puts("Busca produtos por nome");
    </script>
    </li>
  </ul>
</div>
```

11) Recarregue a página atual e veja que a mensagem apareceu no input.

Novo Produto Lista Produtos <input type="text" value="Busca produtos por nome"/>	Resultados da busca pelo nome "abo" <table border="1"> <tr> <th>Nome</th> <th>Descrição</th> <th>Preço</th> <th>Editar</th> <th>Remover</th> </tr> <tr> <td>Sabonete</td> <td>Um sabonete com perfume de lavanda</td> <td>3.53</td> <td>Editar</td> <td>Remover</td> </tr> </table>	Nome	Descrição	Preço	Editar	Remover	Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover
Nome	Descrição	Preço	Editar	Remover							
Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover							

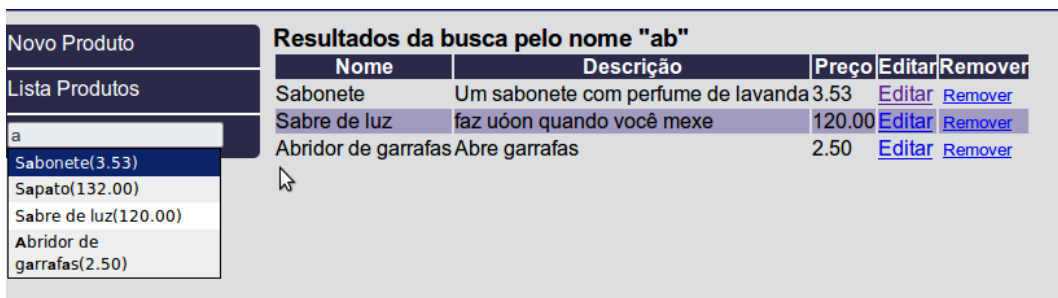
- 12) Crie o método de busca que retorna JSON no ProdutosController.

```
@Get("/produtos/busca.json")
public void buscaJson(String q) {
    result.use(json()).withoutRoot()
        .from(dao.busca(q))
        .exclude("id", "descricao")
        .serialize();
}
```

- 13) Adicione o código javascript para que o autocomplete funcione.

```
$("#busca").autocomplete('/goodbuy/produtos/busca.json', {
    dataType: "json",
    parse: function(produtos) {
        return $.map(produtos, function(produto) {
            return {
                data: produto,
                value: produto.nome,
                result: produto.nome
            };
        });
    },
    formatItem: function(produto) {
        return produto.nome + "(" + produto.preco + ")";
    }
});
```

- 14) Recarregue a página e teste o autocomplete digitando algo no campo de busca.



The screenshot shows a web application interface. On the left, there is a sidebar with a search input field containing the letter 'a'. Below the input field, a dropdown menu is open, displaying a list of product names with their prices in parentheses: Sabonete(3.53), Sapato(132.00), Sabre de luz(120.00), and Abridor de garrafas(2.50). On the right, there is a table titled "Resultados da busca pelo nome 'ab'". The table has five columns: Nome, Descrição, Preço, Editar, and Remover. The table contains three rows of data:

Nome	Descrição	Preço	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	Editar	Remover
Sabre de luz	faz uóon quando você mexe	120.00	Editar	Remover
Abridor de garrafas	Abre garrafas	2.50	Editar	Remover

12.7 PARA SABER MAIS: REPRESENTATION

É possível usar a mesma lógica para a busca normal e a busca por json, usando a view representation() do VRaptor.

```
public void busca(String nome) {  
    result.include("nome", nome);  
    result.use(representation())  
        .from(dao.busca(q), "produtos")  
        .exclude("id", "descricao")  
        .serialize();  
}
```

Assim, se a requisição for feita passando o content-type “application/json”, será a lista serializada em json, se for passado o content type “application/xml”, será serializada em xml, senão o VRaptor redireciona para a jsp respectiva, com a variável `${produtos}` disponível.

Criando o Carrinho de Compras

13.1 O MODELO DO CARRINHO

Ainda não é possível comprar os produtos na nossa loja virtual. Precisamos de algum jeito fazer com que o usuário selecione os produtos que deseja comprar. O jeito mais comum de fazer isso é criar um carrinho de compras, guardando os produtos que foram selecionados.

Vamos criar uma classe que representa o carrinho de compras. Essa classe vai conter uma lista dos produtos escolhidos e o valor total:

```
public class Carrinho {  
  
    private List<Item> itens = new ArrayList<Item>();  
  
    private Double total = 0.0;  
  
    // getters e setters  
}
```

Além disso precisamos criar uma classe que representa um item do carrinho: um produto com a quantidade selecionada.

```
public class Item {  
  
    private Produto produto;  
  
    private Integer quantidade;
```

```
// getters e setters  
}
```

Pergunta importante: precisamos guardar esse carrinho no banco? O usuário não comprou ainda os produtos, então não faz muito sentido guardar no banco de dados. Esse carrinho precisa ficar disponível enquanto o usuário estiver navegando no sistema, então podemos apenas colocar o carrinho na sessão. Como vimos antes, para que um componente seja único durante a sessão do usuário basta anotá-lo com `@SessionScoped`, então basta modificar a classe do carrinho para incluir as anotações:

```
@Component  
@SessionScoped  
public class Carrinho {  
  
    private List<Item> itens = new ArrayList<Item>();  
  
    private Double total = 0.0;  
  
    //getters e setters  
}
```

Agora podemos receber esse carrinho no construtor de algum controller, com a certeza de que ele será único durante a sessão do usuário.

13.2 CONTROLANDO O CARRINHO DE COMPRAS

Agora que temos o modelo do Carrinho, vamos criar uma maneira de adicionar produtos. O jeito mais simples é modificar a listagem de produtos adicionando um botão para comprar:

```
<table>  
    <thead>  
        <tr>  
            ...  
            <th>Comprar</th>  
            ...  
        </tr>  
    </thead>  
    <tbody>  
        <c:forEach items="{produtoList}" var="produto">  
            <tr>  
                ...  
                <td>  
                    <!-- Adicionando o produto no carrinho de compras -->  
                    <form action="{c:url value='/carrinho/'}" method="POST">
```

```

        <input type="hidden" name="item.produto.id"
                value="{produto.id }"/>
        <input class="qtde" name="item.quantidade" value="1"/>
        <button type="submit">Comprar</button>
    </form>
</td>
...
</tr>
</c:forEach>
</tbody>
</table>

```

Nome	Descrição	Preço	Comprar	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	1 <input type="button" value="Comprar"/>	Editar	Remover
Sapato	um sapato de couro	132.00	1 <input type="button" value="Comprar"/>	Editar	Remover
Sabre de luz	faz uóon quando você mexe	120.00	1 <input type="button" value="Comprar"/>	Editar	Remover
Abridor de garrafas	Abre garrafas	2.50	1 <input type="button" value="Comprar"/>	Editar	Remover

O novo formulário vai adicionar um item ao carrinho, passando a quantidade e o id do produto. Precisamos agora criar um controlador que trate das operações no carrinho de compras, que seja capaz de adicionar um item ao carrinho. Esse controlador vai se chamar CarrinhoController:

```

package br.com.caelum.goodbuy.controller;

import br.com.caelum.vraptor.Resource;

@Resource
public class CarrinhoController {

}

```

Vamos criar o método que responde à URI colocada no formulário de compra:

```

@Resource
public class CarrinhoController {

    @Post("/carrinho")
    public void adiciona(Item item) {

    }

}

```

Para adicionar o item ao carrinho, temos que receber um Carrinho no construtor, assim o VRaptor vai passar para o CarrinhoController o carrinho da sessão do usuário:


```
@Resource
```

```
public class CarrinhoController {  
  
    private final Carrinho carrinho;  
  
    public CarrinhoController(Carrinho carrinho) {  
        this.carrinho = carrinho;  
    }  
  
    @Post("/carrinho")  
    public void adiciona(Item item) {  
        carrinho.adiciona(item);  
    }  
}
```

Não existe ainda o método `adiciona` no `Carrinho`, então use o `Ctrl+1` para criar o método. Ao adicionar um item no carrinho precisamos atualizar o total:

```
public class Carrinho {  
  
    private List<Item> itens = new ArrayList<Item>();  
  
    private Double total = 0.0;  
  
    public void adiciona(Item item) {  
        itens.add(item);  
        total += item.getProduto().getPreco() * item.getQuantidade();  
    }  
  
    //...  
}
```

Repare que estamos usando o preço do produto para atualizar o total do carrinho, mas no formulário só estamos passando o id do produto. Precisamos carregar as outras informações do Produto no banco de dados, usando o `ProdutoDao`:

```
@Resource
```

```
public class CarrinhoController {  
  
    private final Carrinho carrinho;  
    private final ProdutoDao dao;  
  
    public CarrinhoController(Carrinho carrinho, ProdutoDao dao) {  
        this.carrinho = carrinho;  
    }  
}
```

```
        this.dao = dao;
    }

    @Post @Path("/carrinho")
    public void adiciona(Item item) {
        dao.recarrega(item.getProduto());
        carrinho.adiciona(item);
    }
}
```

Para implementar o método `recarrega` no `ProdutoDao`, vamos usar um método da `Session` que busca as informações no banco e coloca no próprio objeto passado. Esse método se chama `refresh`.

```
@Component
public class ProdutoDao {
    //...
    public void recarrega(Produto produto) {
        session.refresh(produto);
    }
}
```

Por último, precisamos ir para alguma página após adicionar algum item no carrinho. Por ora, vamos voltar pra página de listagem de produtos. Repare que como o redirecionamento é para uma lógica de outro controller, precisamos usar a classe deste controller:

```
@Resource
public class CarrinhoController {

    private final Carrinho carrinho;
    private final ProdutoDao dao;
    private final Result result;

    public CarrinhoController(Carrinho carrinho,
        ProdutoDao dao, Result result) {
        this.carrinho = carrinho;
        this.dao = dao;
        this.result = result;
    }

    @Post("/carrinho")
    public void adiciona(Item item) {
        dao.recarrega(item.getProduto());
        carrinho.adiciona(item);
    }
}
```

```
        result.redirectTo(ProdutosController.class).lista();
    }
}
```

13.3 VISUALIZANDO OS ITENS DO CARRINHO

Já conseguimos adicionar itens no carrinho, mas não é possível ainda visualizar o que tem dentro dele. Primeiro, vamos adicionar uma lugar em todas as páginas que mostra o estado atual do carrinho: quantos itens ele tem e qual é o valor total. Mas como vamos acessar o carrinho da sessão em todas as nossas páginas? Será que precisamos incluir o carrinho no Result em todas as lógicas? Não, isso não é necessário.

Quando criamos um componente `SessionScoped`, o VRaptor coloca a instância desse componente como um atributo da sessão, usando a convenção de nomes padrão. Ou seja, para o Carrinho, existirá um atributo da sessão chamado **carrinho**, então você consegue acessar o carrinho nas suas jsps usando a variável `${carrinho}`.

Vamos, então, abrir o arquivo `header.jspf` e adicionar dentro da div header uma div do carrinho:

```
<div id="header">

    <div id="carrinho">
        <h3>Meu carrinho:</h3>
        <c:if test="${empty carrinho or carrinho.totalDeItens eq 0 }">
            <span>Você não possui itens no seu carrinho</span>
        </c:if>
        <c:if test="${carrinho.totalDeItens > 0 }">
            <ul>
                <li>
                    <strong>Itens:</strong> ${carrinho.totalDeItens }
                </li>
                <li>
                    <strong>Total:</strong>
                    <fmt:formatNumber type="currency"
                        value="${carrinho.total }"/>
                </li>
            </ul>
        </c:if>
    </div>
</div>
```

Estamos usando a expressão `${carrinho.totalDeItens}` mas classe Carrinho não possui um getter para esse total de itens. Então vamos adicionar:

Capítulo 13 - Criando o Carrinho de Compras - Visualizando os itens do carrinho - Página 133

```

    <td>${item.produto.nome }</td>
    <td>${item.produto.descricao }</td>
    <td>
      <fmt:formatNumber type="currency"
        value="${item.produto.preco }"/>
    </td>
    <td>${item.quantidade }</td>
    <td>
      <fmt:formatNumber type="currency"
        value="${item.quantidade * item.produto.preco }"/>
    </td>
  </tr>
</c:forEach>
</tbody>
<tfoot>
  <tr>
    <td colspan="2"></td>
    <th colspan="2">Total:</th>
    <th>
      <fmt:formatNumber type="currency"
        value="${carrinho.total }"/>
    </th>
  </tr>
</tfoot>
</table>

```

Itens do seu carrinho de compras				
Nome	Descrição	Preço	Qtde	Total
Sabre de luz	faz uóon quando você mexe	R\$ 120,00	1	R\$ 120,00
Sabonete	Um sabonete com perfume de lavanda	R\$ 3,53	5	R\$ 17,65
Total:				R\$ 137,65

Para acessar essa visualização, vamos criar um link na div do carrinho:

```

<div id="header">
  <div id="carrinho">
    <h3><a href="<c:url value="/carrinho"/>">Meu carrinho:</a></h3>

```

Agora que temos uma página de visualização, podemos redirecionar para ela quando adicionamos algum produto ao carrinho.

```

@Resource
public class CarrinhoController {
  //...
  @Post("/carrinho")

```

```
public void adiciona(Item item) {  
    dao.recarrega(item.getProduto());  
    carrinho.adiciona(item);  
  
    result.redirectTo(this).visualiza();  
}  
}
```

13.4 REMOVENDO ITENS DO CARRINHO

Para completar as funcionalidades do carrinho de compras, vamos modificar a visualização do carrinho para ser possível remover itens:

```
<table>  
...  
<tbody>  
<c:forEach items="${carrinho.itens}" var="item" varStatus="s">  
    <tr>  
        ...  
        <td>  
            <form action="<c:url value="/carrinho/${s.index}"/>" method="POST">  
                <button class="link" name="_method" value="DELETE">  
                    Remover  
                </button>  
            </form>  
        </td>  
    </tr>  
</c:forEach>  
</tbody>  
...  
</table>
```

Vamos criar, também, o método que remove itens do carrinho no controller:

```
public class CarrinhoController {  
    //...  
    @Delete("/carrinho/{indiceItem}")  
    public void remove(int indiceItem) {  
        carrinho.remove(indiceItem);  
        result.redirectTo(this).visualiza();  
    }  
}
```

E implementar o método que remove o item do carrinho, atualizando o total:

```
public class Carrinho {  
    //...  
    public void remove(int indiceItem) {  
        Item removido = itens.remove(indiceItem);  
        total -= removido.getProduto().getPreco() * removido.getQuantidade();  
    }  
}
```

13.5 EXERCÍCIOS

- 1) Crie os modelos Carrinho e Item:

```
package br.com.caelum.goodbuy.modelo;  
  
@Component  
@SessionScoped  
public class Carrinho {  
  
    private List<Item> itens = new ArrayList<Item>();  
  
    private Double total = 0.0;  
  
    //getters e setters  
}  
  
package br.com.caelum.goodbuy.modelo;  
  
public class Item {  
  
    private Produto produto;  
  
    private Integer quantidade;  
    //getters e setters  
}
```

- 2) Modifique a listagem de produtos para incluir um botão para comprar o produto:

```
<table>  
    <thead>  
        <tr>  
            ...  
            <th>Comprar</th>  
            ...  
        </tr>  
    </thead>
```

```
<tbody>
  <c:forEach items="${produtoList}" var="produto">
    <tr>
      ...
      <td>
        <!-- Adicionando o produto no carrinho de compras -->
        <form action="<c:url value="/carrinho"/>" method="POST">
          <input type="hidden" name="item.produto.id"
            value="${produto.id }"/>
          <input class="qtde" name="item.quantidade" value="1"/>
          <button type="submit">Comprar</button>
        </form>
      </td>
      ...
    </tr>
  </c:forEach>
</tbody>
</table>
```

3) Adicione o div do carrinho no cabeçalho da página (arquivo header.jspf)

```
<div id="header">

  <div id="carrinho">
    <h3><a href="<c:url value="/carrinho"/>">Meu carrinho:</a></h3>
    <c:if test="${empty carrinho or carrinho.totalDeItens eq 0 }">
      <span>Você não possui itens no seu carrinho</span>
    </c:if>
    <c:if test="${carrinho.totalDeItens > 0 }">
      <ul>
        <li><strong>Itens:</strong> ${carrinho.totalDeItens }</li>
        <li><strong>Total:</strong>
          <fmt:formatNumber type="currency" value="${carrinho.total }"/></li>
      </ul>
    </c:if>
  </div>
</div>
```

4) Crie o CarrinhoController, com um método para adicionar itens no carrinho.

```
package br.com.caelum.goodbuy.controller;
```

```
@Resource
```

```
public class CarrinhoController {
```

```
    private final Carrinho carrinho;
```

```
    private final ProdutoDao dao;
```



```

private final Result result;

public CarrinhoController(Carrinho carrinho,
    ProdutoDao dao, Result result) {
    this.carrinho = carrinho;
    this.dao = dao;
    this.result = result;
}

@Post("/carrinho")
public void adiciona(Item item) {
    dao.recarrega(item.getProduto());
    carrinho.adiciona(item);

    result.redirectTo(ProdutosController.class).lista();
}

}

public class Carrinho {
    //...
    public void adiciona(Item item) {
        itens.add(item);
        total += item.getProduto().getPreco() *
            item.getQuantidade();
    }
    public Integer getTotalDeItens() {
        return itens.size();
    }
}

public class ProdutoDao {
    //...
    public void recarrega(Produto produto) {
        session.refresh(produto);
    }
}

```

5) Compre alguns produtos e veja o carrinho sendo atualizado no header.

					Meu carrinho: Itens: 3 Total: R\$ 131,03
Nome	Descrição	Preço	Comprar	Editar	Remover
Sabonete	Um sabonete com perfume de lavanda	3.53	1 <input type="button" value="Comprar"/>	Editar	Remover
Sapato	um sapato de couro	132.0	1 <input type="button" value="Comprar"/>	Editar	Remover
Sabre de luz	faz uóon quando você mexe	120.0	1 <input type="button" value="Comprar"/>	Editar	Remover
Abridor de garrafas	Abre garrafas	2.5	1 <input type="button" value="Comprar"/>	Editar	Remover

- 6) Crie a lógica e o jsp que visualiza os itens do carrinho. Mude também o redirecionamento do método adiciona para a nova lógica.

```
@Resource
public class CarrinhoController {
    //...
    @Get("/carrinho")
    public void visualiza() {
    }
    @Post("/carrinho")
    public void adiciona(Item item) {
        dao.recarrega(item.getProduto());
        carrinho.adiciona(item);

        result.redirectTo(this).visualiza();
    }
}

/WEB-INF/jsp/carrinho/visualiza.jsp

<h3>Itens do seu carrinho de compras</h3>

<table>
    <thead>
        <tr>
            <th>Nome</th>
            <th>Descrição</th>
            <th>Preço</th>
            <th>Qtde</th>
            <th>Total</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${carrinho.itens}" var="item"
            varStatus="s">
            <tr>
                <td>${item.produto.nome }</td>
                <td>${item.produto.descricao }</td>
                <td>
                    <fmt:formatNumber type="currency"
                        value="${item.produto.preco }"/>
                </td>
                <td>${item.quantidade }</td>
                <td>
                    <fmt:formatNumber type="currency"
                        value="${item.quantidade * item.produto.preco }"/>
                </td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

```

        </td>
      </tr>
    </c:forEach>
  </tbody>
  <tfoot>
    <tr>
      <td colspan="2"></td>
      <th colspan="2">Total:</th>
      <th>
        <fmt:formatNumber type="currency"
          value="${carrinho.total }"/>
      </th>
    </tr>
  </tfoot>
</table>

```

7) Adicione mais produtos ao carrinho.

Nome	Descrição	Preço	Qtde	Total
Sabre de luz	faz uóon quando você mexe	R\$ 120,00	1	R\$ 120,00
Sabonete	Um sabonete com perfume de lavanda	R\$ 3,53	5	R\$ 17,65
Total:				R\$ 137,65

8) Crie a lógica de remover produtos do carrinho e adicione o botão de remoção na visualização do carrinho.

```

public class CarrinhoController {
    //...
    @Delete("/carrinho/{indiceItem}")
    public void remove(int indiceItem) {
        carrinho.remove(indiceItem);
        result.redirectTo(this).visualiza();
    }
}

public class Carrinho {
    public void remove(int indiceItem) {
        Item removido = itens.remove(indiceItem);
        total -= removido.getProduto().getPreco() * removido.getQuantidade();
    }
}

/WEB-INF/jsp/carrinho/visualiza.jsp

<table>
...
<tbody>

```

```
<c:forEach items="${carrinho.itens}" var="item" varStatus="s">
  <tr>
    ...
    <td>
      <form action="<c:url value="/carrinho/${s.index}"/>" method="POST">
        <button class="link" name="_method" value="DELETE">Remover</button>
      </form>
    </td>
  </tr>
</c:forEach>
</tbody>
...
</table>
```

9) Adicione e remova itens do carrinho.

Autenticação

14.1 CRIANDO USUÁRIOS

No nosso sistema, atualmente, qualquer um pode adicionar, editar ou remover produtos. Será que isso é o desejável? Não seria melhor apenas habilitar essas funcionalidades para os administradores do sistema?

Então vamos criar um sistema de login para a nossa aplicação, começando pelo modelo de usuários:

```
package br.com.caelum.goodbuy.modelo;

public class Usuario {

    private String login;

    private String senha;

    private String nome;

    //getters e setters
}
```

É uma boa idéia guardar os usuários no banco de dados, então vamos adicionar as anotações do hibernate. Não há a necessidade de criar um campo id para usuários, pois o login já será um identificador único.

```
package br.com.caelum.goodbuy.modelo;

import javax.persistence.Entity;
import javax.persistence.Id;
```

```
@Entity
public class Usuario {

    @Id
    private String login;

    private String senha;

    private String nome;

    //getters e setters
}
```

Como estamos adicionando uma entidade nova, precisamos colocá-la no `hibernate.cfg.xml`. Ainda vamos adicionar a propriedade `hibernate.hbm2ddl.auto` com o valor `update`, assim o hibernate fará as mudanças nas tabelas do banco quando necessário.

```
<hibernate-configuration>
    <session-factory>
        <!--...-->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <mapping class="br.com.caelum.goodbuy.modelo.Produto" />
        <mapping class="br.com.caelum.goodbuy.modelo.Usuario" />
    </session-factory>
</hibernate-configuration>
```

Com o modelo pronto já é possível criar as lógicas de cadastro e de login de usuários. Vamos começar pelo cadastro, criando o controlador de usuários com uma lógica para mostrar o formulário:

```
package br.com.caelum.goodbuy.controller;

import br.com.caelum.vraptor.Resource;

@Resource
public class UsuariosController {

    public void novo() {

    }

}
```

E então o jsp com o formulário, em `/WEB-INF/jsp/usuarios/novo.jsp`, com todos os campos obrigatórios:

```
<form id="usuariosForm" action="<c:url value="/usuarios"/>"
  method="POST">
  <fieldset>
    <legend>Criar novo usuário</legend>

    <label for="nome">Nome:</label>
    <input id="nome" class="required"
      type="text" name="usuario.nome" value="{usuario.nome }"/>

    <label for="login">Login:</label>
    <input id="login" class="required"
      type="text" name="usuario.login" value="{usuario.login }"/>

    <label for="senha">Senha:</label>
    <input id="senha" class="required" type="password"
      name="usuario.senha"/>

    <label for="confirmacao">Confirme a senha:</label>
    <input id="confirmacao" equalTo="#senha" type="password"/>

    <button type="submit">Enviar</button>
  </fieldset>
</form>

<script type="text/javascript">
  $('#usuariosForm').validate();
</script>
```

Precisamos também criar um link para esse cadastro. Para isso criaremos uma div no cabeçalho que mostrará as informações do usuário. Abra o arquivo `header.jspf` e modifique a div header:

```
<div id="header">
  <div id="usuario">
    Você não está logado.
    <a href="<c:url value="/usuarios/novo"/>">
      Cadastre-se
    </a>
  </div>
  ...
</div>
```

Para completar o cadastro, vamos criar a lógica que adiciona o usuário de fato, validando se o login escolhido ainda não existe no sistema:

```
@Resource
public class UsuariosController {
```

```
private final UsuarioDao dao;
private final Result result;
private final Validator validator;

public UsuariosController(UsuarioDao dao, Result result,
    Validator validator) {
    this.dao = dao;
    this.result = result;
    this.validator = validator;
}
@Post("/usuarios")
public void adiciona(Usuario usuario) {
    if (dao.existeUsuario(usuario)) {
        validator.add(new ValidationMessage("Login já existe",
            "usuario.login"));
    }
    validator.onErrorUsePageOf(UsuariosController.class).novo();

    dao.adiciona(usuario);

    result.redirectTo(ProdutosController.class).lista();
}
//...
}
```

O UsuarioDao ainda não existe. Use o Ctrl+1 para criar o dao e os seus métodos:

```
@Component
public class UsuarioDao {

    private final Session session;

    public UsuarioDao(Session session) {
        this.session = session;
    }

    public boolean existeUsuario(Usuario usuario) {
        Usuario encontrado = (Usuario) session.createCriteria(Usuario.class)
            .add(Restrictions.eq("login", usuario.getLogin()))
            .uniqueResult();
        return encontrado != null;
    }

    public void adiciona(Usuario usuario) {
        Transaction tx = this.session.beginTransaction();
```



```
        this.session.save(usuario);  
        tx.commit();  
    }  
  
}
```

14.2 EFETUANDO O LOGIN

Criamos o cadastro, mas o usuário ainda não consegue fazer o login. Primeiro vamos criar um link para acessar o formulário de login, no cabeçalho:

```
<div id="header">  
    <div id="usuario">  
        Você não está logado. <a href="<c:url value="/login"/>">Login</a>  
        <a href="<c:url value="/usuarios/novo"/>">Cadastre-se</a>  
    </div>  
</div>
```

e a respectiva lógica e formulário:

```
@Resource  
public class UsuariosController {  
    //...  
    @Get("/login")  
    public void loginForm() {  
  
    }  
}  
  
<form action="<c:url value="/login"/>" method="POST">  
    <fieldset>  
        <legend>Efetue o login</legend>  
  
        <label for="login">Login:</label>  
        <input id="login" type="text" name="usuario.login"/>  
  
        <label for="senha">Senha:</label>  
        <input id="senha" type="password" name="usuario.senha"/>  
  
        <button type="submit">Login</button>  
    </fieldset>  
</form>
```

Quando o usuário faz o login, precisamos guardar a informação de que ele já está logado. A melhor forma é guardar a informação de login na sessão, que mantém os dados enquanto o usuário estiver navegando

pela aplicação. Para isso, vamos criar uma classe que guarda o usuário logado. Essa classe será acessada nos jsps para acessar as informações do usuário, então adicionamos alguns getters para expor as informações relevantes.

```
@Component
@SessionScoped
public class UsuarioWeb {

    private Usuario logado;

    public void login(Usuario usuario) {
        this.logado = usuario;
    }

    public String getNome() {
        return logado.getNome();
    }

    public boolean isLogado() {
        return logado != null;
    }
}
```

Então na nossa lógica de login, podemos colocar o usuário logado dentro da classe acima, depois de verificar que o usuário digitou o login e senha certos.

```
@Resource
public class UsuariosController {

    private final UsuarioWeb usuarioWeb;
    //...
    public UsuariosController(UsuarioDao dao, Result result,
        Validator validator, UsuarioWeb usuarioWeb) {
        //...
        this.usuarioWeb = usuarioWeb;
    }

    @Post("/login")
    public void login(Usuario usuario) {
        Usuario carregado = dao.carrega(usuario);
        if (carregado == null) {
            validator.add(
                new ValidationMessage("Login e/ou senha inválidos",
                    "usuario.login"));
        }
    }
}
```

```
        validator.onErrorUsePageOf(UsuariosController.class)
            .loginForm();

        usuarioWeb.login(carregado);

        result.redirectTo(ProdutosController.class).lista();
    }
}
```

Para carregar o usuário, crie o método no UsuarioDao que busca um usuário por login e senha:

```
@Component
public class UsuarioDao {

    //...

    public Usuario carrega(Usuario usuario) {
        return (Usuario) session.createCriteria(Usuario.class)
            .add(Restrictions.eq("login", usuario.getLogin()))
            .add(Restrictions.eq("senha", usuario.getSenha()))
            .uniqueResult();
    }
}
```

E para mostrar que o usuário está logado mesmo, vamos modificar o cabeçalho:

```
<div id="header">
    <div id="usuario">
        <c:if test="${usuarioWeb.logado}">
            Olá, ${usuarioWeb.nome }!
            <a href="<c:url value="/logout"/>">Logout</a>
        </c:if>
        <c:if test="${empty usuarioWeb or not usuarioWeb.logado}">
            Você não está logado.
            <a href="<c:url value="/login"/>">Login</a>
            <a href="<c:url value="/usuarios/novo"/>">Cadastre-se</a>
        </c:if>
    </div>
    ...
</div>
```

Por último vamos adicionar a lógica de logout:

```
public class UsuariosController {
    //...
    @Path("/logout")
    public void logout() {
        usuarioWeb.logout();
        result.redirectTo(ProdutosController.class).lista();
    }
}

public class UsuarioWeb {
    //...

    public void logout() {
        this.logado = null;
    }
}
```

14.3 RESTRINGINDO FUNCIONALIDADES PARA USUÁRIOS LOGADOS

Como falamos no começo do capítulo, não é legal que todo mundo consiga adicionar, remover e editar produtos. Vamos, então, fazer com que só usuários logados consigam acessar essas funcionalidades. Um primeiro passo é retirar os links para elas quando o usuário não está logado:

/header.jspf:

```
<div id="menu">
    <ul>
        <c:if test="${usuarioWeb.logado }">
            <li><a href="<c:url value="/produtos/novo"/>">
                Novo Produto
            </a></li>
        </c:if>
        ...
    </ul>
</div>
```

/WEB-INF/jsp/produto/lista.jsp

```
<table>
    ...
    <tbody>
        <c:forEach items="${produtoList}" var="produto">
            <tr>
                ...
                <c:if test="${usuarioWeb.logado }">
```

```
<td>
    <a href=
        "<c:url value="/produtos/${produto.id}"/>"
        Editar
    </a>
</td>
<td>
    <form action=
        "<c:url value="/produtos/${produto.id}"/>"
        method="POST">
        <button class="link" name="_method"
            value="DELETE">
            Remover
        </button>
    </form>
</td>
</c:if>
</tr>
</c:forEach>
</tbody>
</table>
```

Agora os usuários que não estão logados não conseguem mais ver os links das ações que ele não pode executar. Mas será que isso é o suficiente? O que impede o usuário de digitar na barra de endereços do browser: `http://localhost:8080/goodbuy/produtos/novo` ? Nada! Ele só precisa conhecer qual é a URI. Precisamos, de algum jeito, impedir que os usuários acessem certas URIs se eles não estiverem logados, monitorar todas as ações do usuário para que quando ele acessar uma URI proibida, redirecionar para uma página de erro, ou melhor, para o login.

14.4 INTERCEPTOR

Um Interceptor no VRaptor é como se fosse um Servlet Filter: ele pode interceptar requisições, executando algo antes e/ou depois da sua lógica. Mais ainda, ele pode impedir que sua lógica seja executada, redirecionando a requisição para outro lugar. Isso é exatamente o que a gente queria: verificar se o usuário está logado antes de ir pro controller, e se ele não estiver, redirecionar para o login.

Para implementar um Interceptor do VRaptor precisamos de duas coisas: anotar a classe com `@Intercepts` e implementar a interface `Interceptor`:

```
@Intercepts
public class AutenticacaoInterceptor implements Interceptor {
    ...
}
```

Essa interface tem dois métodos:

- `public boolean accepts(ResourceMethod method)` - decide se vai ou não interceptar a requisição atual. o parâmetro `method` representa qual é o método java que será executado na requisição, o método do seu controller. Com esse objeto você tem acesso à classe do controller e ao método java dele (`java.reflect.Method`) para poder, por exemplo, ver qual é o nome do método, ou se ele tem alguma anotação
- `public void intercept(InterceptorStack stack, ResourceMethod method, Object resourceInstance) throws InterceptionException` { - intercepta a requisição. O parâmetro `stack` possibilita continuar a execução normal da requisição e, portanto, executar de fato a lógica de negócios. O parâmetro `method` é o mesmo do método `accepts`, e o `resourceInstance` é o controller instanciado.

Como qualquer classe registrada no VRaptor, você pode receber qualquer componente da sua aplicação (e do VRaptor) pelo construtor do seu interceptor. No nosso caso precisamos verificar se o usuário está logado, e essa informação está no `UsuarioWeb`.

```
@Intercepts
public class AutorizacaoInterceptor implements Interceptor {

    private final UsuarioWeb usuario;

    public AutorizacaoInterceptor(UsuarioWeb usuario) {
        this.usuario = usuario;
    }

}
```

Mas só precisamos executar a lógica de autenticação caso o usuário não esteja logado, então vamos implementar o método `accepts`:

```
public boolean accepts(ResourceMethod method) {
    return !this.usuario.isLogado();
}
```

Para o método `intercepts`, sabemos já que o usuário não está logado, então vamos redirecionar para a lógica de login. Para isso precisamos do `Result`.

```
@Intercepts
public class AutorizacaoInterceptor implements Interceptor {

    private final UsuarioWeb usuario;
    private final Result result;
```

```
public AutorizacaoInterceptor(UsuarioWeb usuario, Result result) {
    this.usuario = usuario;
    this.result = result;
}

public boolean accepts(ResourceMethod method) {
    return false;
}

public void intercept(InterceptorStack stack, ResourceMethod method,
    Object resourceInstance) throws InterceptionException {
    result.redirectTo(UsuariosController.class).loginForm();
}
}
```

Não é o caso, mas se quiséssemos continuar a requisição normalmente, poderíamos fazer:

```
stack.next(method, resourceInstance);
```

Mas temos um pequeno problema: se o usuário não estiver logado ele não vai conseguir acessar nada no sistema, nem o login! Na verdade, só queremos proibir que o usuário adicione e modifique produtos, então nosso interceptor só pode executar para essas operações.

Poderíamos até colocar no Interceptor uma lista dos métodos que serão interceptados, mas assim toda vez que adicionarmos uma operação nova que precise de autenticação precisaríamos mudar o interceptor.

Um jeito mais legal de fazer isso é marcar os métodos que precisam de autenticação. Para isso podemos criar uma anotação para ser colocada nos métodos:

```
//a anotação vai ficar disponível em tempo de execucao
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD) // anotação para métodos
public @interface Restrito {

}
```

Assim, podemos anotar os métodos restritos (adiciona, atualiza, remove, formulario e edita do Produtos-Controller):

```
@Resource
public class ProdutosController {

    @Restrito
```

```
public void formulario() {}

@Restrito
public Produto edita(Long id) {...}

@Restrito
public void altera(Produto produto) {...}

@Restrito
public void adiciona(final Produto produto) {...}

@Restrito
public void remove(Long id) {...}

...
}
```

E do lado do Interceptor, apenas fazemos:

```
public boolean accepts(ResourceMethod method) {
    return !usuario.isLogado() && method.containsAnnotation(Restrito.class);
}
```

Pronto. Nosso sistema de autenticação está pronto. Poderíamos trocar a anotação `@Restrito` por uma, por exemplo, `@Liberado` caso tenha bem mais operações liberadas do que restritas, depende do seu sistema.

Apêndice - Download e Upload

Nesse capítulo, vamos adicionar imagens aos produtos, para uma melhor visualização na listagem. Para isso, precisamos fazer o upload dessas imagens para o servidor, e depois fazer o download para mostrá-las na listagem.

15.1 EXERCÍCIOS

- 1) Modifique a página de edição de Produtos (/WEB-INF/jsp/produtos/edita.jsp), para incluir um formulário de Upload de imagem. Esse formulário vai submeter para uma lógica nova que vamos criar a seguir.

```
<!--...-->
<form action=
    "<c:url value="/produtos/${produto.id }/imagem"/>"
    method="POST" enctype="multipart/form-data">
    <fieldset>
        <legend>Upload de Imagem</legend>
        <input type="file" name="imagem" />

        <button type="submit">Enviar</button>
    </fieldset>
</form>
```

- 2) Crie o Controller abaixo, que vai tratar dos uploads e downloads de imagens.

```
package br.com.caelum.goodbuy.controller;

import br.com.caelum.vraptor.interceptor.multipart.UploadedFile;
```

```
@Resource
public class ImagensController {

    @Post("/produtos/{produto.id}/imagem")
    public void upload(Produto produto, UploadedFile imagem) {

    }
}
```

- 3) Só estamos interessados em fazer uploads de imagens. Então vamos validar se o upload foi uma imagem mesmo. O ContentType de imagens começa com image:

```
@Post("/produtos/{produto.id}/imagem")
public void upload(Produto produto,
    final UploadedFile imagem) {
    validator.checking(new Validations() {{
        if (that(imagem, is(notNullValue()), "imagem",
            "imagem.nula")) {
            that(imagem.getContentType(),
                startsWith("image"), "imagem", "nao.eh.imagem");
        }
    }});
    validator.onErrorRedirectTo(ProdutosController.class)
        .edita(produto.getId());
}
```

- 4) Coloque as mensagens internacionalizadas no messages.properties:

```
imagem.nula = Digite uma imagem
nao.eh.imagem = Selecione uma imagem
```

- 5) Crie o componente do VRaptor que será responsável por guardar as imagens no servidor. Esse componente vai criar uma pasta fixa onde todas as imagens serão guardadas. Para isso, precisamos conseguir o caminho de uma pasta do servidor, e a classe que consegue nos dar essa informação é o ServletContext.

```
package br.com.caelum.goodbuy.imagens;

import java.io.File;
import javax.servlet.ServletContext;
import br.com.caelum.vraptor.ioc.Component;

@Component
public class Imagens {

    private File pastaImagens;

    public Imagens(ServletContext context) {
```

```
String caminhoImagens = context.getRealPath("/WEB-INF/imagens");
pastaImagens = new File(caminhoImagens);
pastaImagens.mkdir();
}
}
```

- 6) Crie um método que salve a imagem do upload na pasta padrão. Colocaremos uma extensão fixa para facilitar. O IOUtils vem do projeto commons-io e copia um InputStream para algum OutputStream

```
package br.com.caelum.goodbuy.imagens;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.commons.io.IOUtils;

@Component
public class Imagens {

    //...
    public void salva(UploadedFile imagem, Produto produto) {
        File destino = new File(pastaImagens, produto.getId() + ".imagem");

        try {
            IOUtils.copy(imagem.getFile(), new FileOutputStream(destino));
        } catch (IOException e) {
            throw new RuntimeException("Erro ao copiar imagem", e);
        }
    }
}
```

- 7) Use a classe Imagens para salvar a imagem que veio no upload. E a classe Result para voltar para o formulário de edição.

```
package br.com.caelum.goodbuy.controller;

@Resource
public class ImagensController {

    private final Validator validator;
    private final Imagens imagens;
    private final Result result;

    public ImagensController(Validator validator,
        Imagens imagens, Result result) {
        this.validator = validator;
    }
}
```

```
        this.imagens = imagens;
        this.result = result;
    }

    @Post("/produtos/{produto.id}/imagem")
    public void upload(Produto produto,
        final UploadedFile imagem) {
        validator.checking(new Validations() {{
            if (that(imagem, is(notNullValue()), "imagem",
                "imagem.nula")) {
                that(imagem.getContentType(),
                    startsWith("image"), "imagem", "nao.eh.imagem");
            }
        }});
        validator
            .onErrorRedirectTo(ProdutosController.class)
            .edita(produto.getId());

        imagens.salva(imagem, produto);
        result.redirectTo(ProdutosController.class)
            .edita(produto.getId());
    }
}
```

8) Crie a lógica para fazer o download da imagem no ImagensController:

```
@Get("/produtos/{produto.id}/imagem")
public File download(Produto produto) {
    return imagens.mostra(produto);
}
```

9) Crie o método mostra na classe Imagens:

```
public File mostra(Produto produto) {
    return new File(pastaImagens, produto.getId() + ".imagem");
}
```

10) Mude o formulário de edição para incluir a imagem do produto. Repare que estamos usando nosso controller para mostrar a imagem. Coloque essa imagem na listagem de produtos também.

```
"
    width="100" height="100"/>
```

Apêndice - Integrando VRaptor e Spring

16.1 COMO FAZER A INTEGRAÇÃO?

Para usar os componentes do Spring no VRaptor basta configurá-los no `applicationContext.xml` da maneira que você já está acostumado a fazer quando trabalha com o Spring. Todos os componentes declarados no `applicationContext.xml` estarão disponíveis para os componentes do VRaptor, e vice-versa.

Um detalhe da integração é que se você usar algum listener do spring para carregar o `applicationContext.xml` (como o `ContextLoaderListener`), o VRaptor só consegue incluir os seus componentes após um reload do contexto, então se você precisar de um componente do VRaptor dentro de um do Spring, o do VRaptor precisa ser marcado como opcional:

```
@Autowired(optional=true)
public void setComponenteDoVRaptor(ComponenteDoVRaptor c) {...}
```

Se não houver o listener do Spring, e o `applicationContext.xml` estiver no classpath, o `optional` não é necessário.

16.2 INTEGRANDO O TRANSACTION MANAGER DO SPRING

Na nossa aplicação estamos controlando as transações manualmente, abrindo e fechando-as dentro de alguns métodos do DAO. Mas isso não é bom, principalmente se você precisar fazer mais de uma operação dentro da mesma transação.

O Spring possui um componente que controla transações, e podemos usá-lo para transferir esse controle para o Spring. Esse componente depende indiretamente de uma `SessionFactory`, que no momento é gerenciada pelo VRaptor.

Para passar a `SessionFactory` como dependência para o componente de transações do Spring, não podemos mais usar o `CriadorDeSessionFactory`, precisamos criar a `SessionFactory` do jeito do Spring, por exemplo com o componente `AnnotationSessionFactoryBean`.

Ao usar o `SessionFactory` do Spring, não podemos mais criar `Sessions` do jeito que fazíamos antes, apenas chamando o `openSession`, pois o Spring gerencia as `Sessions` de uma maneira que está acoplada a todos os componentes que precisam usá-la. Por causa disso precisaremos adaptar o nosso `CriadorDeSession`.

Nos exercícios abaixo mostraremos como integrar esse componente de transações como a nossa aplicação.

16.3 EXERCÍCIOS: TRANSACTION MANAGER

- 1) Crie o arquivo `applicationContext.xml` dentro da pasta `src`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

</beans>
```

- 2) Inclua o componente que gerencia as transações, baseado em anotações do Spring.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <tx:annotation-driven />

</beans>
```

- 3) Esse componente de transações é genérico: funciona tanto com a JTA direto, com Hibernate/JPA, com JDBC, etc. Então para especificar qual dos gerenciadores de transação vai ser utilizado, você precisa adicionar um bean na configuração. No nosso caso, vamos usar transações do Hibernate:

```
<beans xmlns...>

    <tx:annotation-driven />
```

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

```
</beans>
```

- 4) O transactionManager precisa de uma SessionFactory configurada. Já temos uma no VRaptor, mas ela não poderá ser usada. Remova a anotação @Component do CriadorDeSessionFactory, e adicione o bean que cria uma SessionFactory no Spring:

```
<beans xmlns:...>
  <!--...-->
  <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.
              annotation.AnnotationSessionFactoryBean">
    <property name="configLocation">
      <value>classpath:/hibernate.cfg.xml</value>
    </property>
  </bean>
</beans>
```

- 5) Ao usar a SessionFactory do Spring estamos presos ao seu controle de Sessions. Isso significa que não podemos fazer o sessionFactory.openSession para obter Sessions. O Spring controla isso para você, então você é obrigado a usar componentes do Spring para conseguir uma Session, para poder usar a mesma que está participando da transação. Mas ao invés de mudar todo o nosso sistema para usar as sessions do Spring, vamos modificar o nosso CriadorDeSession para usar-las.

Por causa do jeito que o Spring trabalha, temos que pedir uma Session para ele no momento que vamos usá-la, e não na criação do DAO, por exemplo. Então vamos usar um Proxy Dinâmico para poder passar uma Session no construtor do DAO, mas mesmo assim só pedir a Session pro Spring na hora de chamar algum método.

Modifique o CriadorDeSession.

```
import net.vidageek.mirror.dsl.Mirror;

import org.springframework.orm.hibernate3.SessionFactoryUtils;

import br.com.caelum.vraptor.proxy.MethodInvocation;
import br.com.caelum.vraptor.proxy.Proxifier;
import br.com.caelum.vraptor.proxy.SuperMethod;

@Component
public class CriadorDeSession implements ComponentFactory<Session> {

    private final SessionFactory factory;
```

```
private final Proxifier proxifier;
private Session session;

public CriadorDeSession(SessionFactory factory, Proxifier proxifier) {
    this.factory = factory;
    this.proxifier = proxifier;
}

@PostConstruct
public void abre() {
    this.session = proxifier.proxify(Session.class,
        new MethodInvocation<Session>() {
            public Object intercept(Session proxy,
                Method method, Object[] args,
                SuperMethod superMethod) {
                Session sessionDoSpring =
                    SessionFactoryUtils
                        .doGetSession(factory, true);
                return new Mirror().on(sessionDoSpring)
                    .invoke().method(method).withArgs(args);
            }
        });
}

public Session getInstance() {
    return this.session;
}

@PreDestroy
public void fecha() {
    this.session.close();
}
}
```

- 6) Para usar o controle de transações do Spring, remova a abertura e o fechamento de transações dos métodos do DAO, e anote o método com `@Transactional`. Repare que nem todos os métodos precisam de transações, só os que modificam o banco.

```
import org.springframework.transaction.annotation.Transactional;

@Component
public class ProdutoDao {

    @Transactional
    public void salva(Produto produto) {
        //Transaction tx = session.beginTransaction();
        session.save(produto);
    }
}
```



```
        //tx.commit();
    }

    public Produto carrega(Long id) {
        return (Produto) session.load(Produto.class, id);
    }

    @Transactional
    public void atualiza(Produto produto) {
        session.update(produto);
    }

    public List<Produto> listaTudo() {
        return this.session.createCriteria(Produto.class).list();
    }

    @Transactional
    public void remove(Produto produto) {
        session.delete(produto);
    }

    public List<Produto> busca(String nome) {
        return session.createCriteria(Produto.class)
            .add(Restrictions.ilike("nome", nome, MatchMode.ANYWHERE))
            .list();
    }

    public void recarrega(Produto produto) {
        session.refresh(produto);
    }
}
```

- 7) O Spring transactional usa o Spring AOP para gerenciar as transações. O Spring AOP tem uma restrição forte: para poder decorar suas classes ele usa proxies da Cglib. Esses proxies precisam que sua classe tenha um construtor sem argumentos, ou que as dependências da sua classe sejam sempre interfaces. E como programar voltado a interfaces é uma boa prática, pois diminui o acoplamento de uma classe com as suas dependências, vamos optar pela segunda solução.

Renomeie a classe ProdutoDao para HibernateProdutoDao. Use o Ctrl+Shift+R no nome da classe.

Extraia uma interface chamada ProdutoDao, selecionando todos os métodos. Com o cursor no nome da classe digite Ctrl+3 e então Extract Interface. Ou Alt+Shift+T e selecione Extract Interface. Marque o checkbox: "Use supertype where possible"

Apêndice: Mudando a View Padrão: Velocity

No VRaptor, a view padrão de uma lógica é uma JSP. Mas nem sempre queremos usar JSP para gerar nossas páginas. Veremos então como usar uma outra *Template Engine* para gerar nossos HTMLs: o Velocity.

17.1 EXERCÍCIOS: CONFIGURANDO O VELOCITY

- 1) Entre no site de downloads do velocity (<http://velocity.apache.org/download.cgi>) e baixe os zips velocity-x.x.x.zip e velocity-tools-x.x.x.zip. Copie os seguintes jars para o seu WEB-INF/lib:
 - no velocity-x.x.x.zip copie os jars velocity-x.x.x.jar e velocity-x.x.x-dep.jar
 - no velocity-tools-x.x.x.zip copie todos os jars da pasta lib
- 2) Abra o web.xml e adicione a servlet do Velocity.

```
<servlet>
  <servlet-name>velocity</servlet-name>
  <servlet-class>
    org.apache.velocity.tools.view.servlet
      .VelocityViewServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>velocity</servlet-name>
  <url-pattern>*.vm</url-pattern>
</servlet-mapping>
```

17.2 EXERCÍCIOS: MUDANDO O TEMPLATE ENGINE DE UMA ÚNICA LÓGICA

- 1) Se você quiser usar o Velocity em uma única ou poucas lógicas, você não precisa de muita coisa. Com a Servlet do Velocity configurada, para criar uma página com o velocity basta criar um arquivo com a extensão **.vm**. Por exemplo, crie o arquivo `olamundo.vm` na pasta `WebContent`.

```
<html>
  <body>
    Olá mundo.
  </body>
</html>
```

- 2) Acesse a página do Velocity diretamente: `http://localhost:8080/goodbuy/olamundo.vm`
- 3) Crie um controller simples para redirecionar pra esse arquivo. Note que os objetos incluídos no `Result` ficam acessíveis no template do Velocity.

```
public class TesteController {
    private Result result;

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

    @Path("/teste")
    public void teste() {
        result.include("mensagem", "Estou usando o Velocity");
        result.forwardTo("/olamundo.vm");
    }
}
```

e modifique o `olamundo.vm` para mostrar a mensagem incluída:

```
<html>
  <body>
    Olá mundo. $mensagem.
  </body>
</html>
```

- 4) Acesse a URL que cai nesse método: `http://localhost:8080/goodbuy/teste`

17.3 EXERCÍCIOS: MUDANDO O RESULTADO DE TODAS AS LÓGICAS PARA VELOCITY

- 1) O VRaptor 3 tem uma convenção para chamar a página de resultado de uma lógica: a página em `/WEB-INF/jsp/<nomeDoController>/<nomeDoMetodo>.jsp`. Mas e se quisermos mudar a convenção para outra coisa? Por exemplo não usar jsp, e sim velocity?

Para isso precisamos sobrescrever um comportamento do VRaptor. Quase todos os componentes do VRaptor podem ser sobrescritos por componentes da sua aplicação, pois ele utiliza Injeção de Dependência para ligar seus componentes internos. Se você criar uma implementação de uma interface interna do VRaptor, e anotá-la com `@Component`, o VRaptor vai usar a sua implementação, e não mais a padrão.

Queremos mudar o padrão da página para usar Velocity, então podemos mudar o padrão para algo do tipo: `/WEB-INF/velocity/<nomeDoController>/<nomeDoMetodo>.vm`.

O componente responsável por essa convenção é o `PathResolver`, que tem uma implementação padrão chamada `DefaultPathResolver`. Essa implementação padrão contém métodos `protected` que podem ser sobrescritos, tornando fácil a mudança da convenção. Então para usar a convenção nova basta criar a classe abaixo, que estende de `DefaultPathResolver`.

```
package br.com.caelum.goodbuy.vraptor;

import javax.servlet.http.HttpServletRequest;

import br.com.caelum.vraptor.view.AcceptHeaderToFormat;
import br.com.caelum.vraptor.view.DefaultPathResolver;

@Component
public class VelocityPathResolver extends DefaultPathResolver {

    public VelocityPathResolver(HttpServletRequest request,
                                AcceptHeaderToFormat acceptHeaderToFormat) {
        super(request, acceptHeaderToFormat);
    }

    @Override
    protected String getPrefix() {
        // o retorno padrão desse método é /WEB-INF/jsp/, que é a página
        // onde suas views serão procuradas
        return "/WEB-INF/velocity/";
    }

    @Override
    protected String getExtension() {
        // o retorno padrão é jsp, que a extensão da página da sua view
```

```
        return "vm";  
    }  
  
}
```

2) Crie a pasta /WEB-INF/velocity, e dentro dela a pasta teste. Mova o arquivo olamundo.vm para a pasta teste e o renomeie para teste.vm.

3) Remova o redirecionamento para a página olamundo.vm de dentro do TesteController:

```
public class TesteController {  
    private Result result;  
  
    public TesteController(Result result) {  
        this.result = result;  
    }  
  
    @Path("/teste")  
    public void teste() {  
        result.include("mensagem", "Estou usando o Velocity");  
    }  
}
```

4) Acesse a url do método teste (<http://localhost:8080/goodbuy/teste>), e veja que continua funcionando: ele vai redirecionar automaticamente para a página /WEB-INF/velocity/teste/teste.vm.

5) Veja que agora nenhuma das outras páginas do sistema estão acessíveis. Agora qualquer chamada vai dar 404, pois ele estará usando a nova convenção. O comportamento do VRaptor agora é o da sua classe, e não mais a padrão dele!

Índice Remissivo

AnnotationConfiguration, 24

Criteria, 57

Driver, 17

Escopo, 87

Hibernate, 16

Hibernate Annotations, 16

Hibernate Core, 16

Injeção de Dependências, 61

JPA, 16

Log4J, 16

Logging, 16

MySQL, 15

ORM, 16

POJO, 49

Result, 70

SL4J, 16

Validação, 93