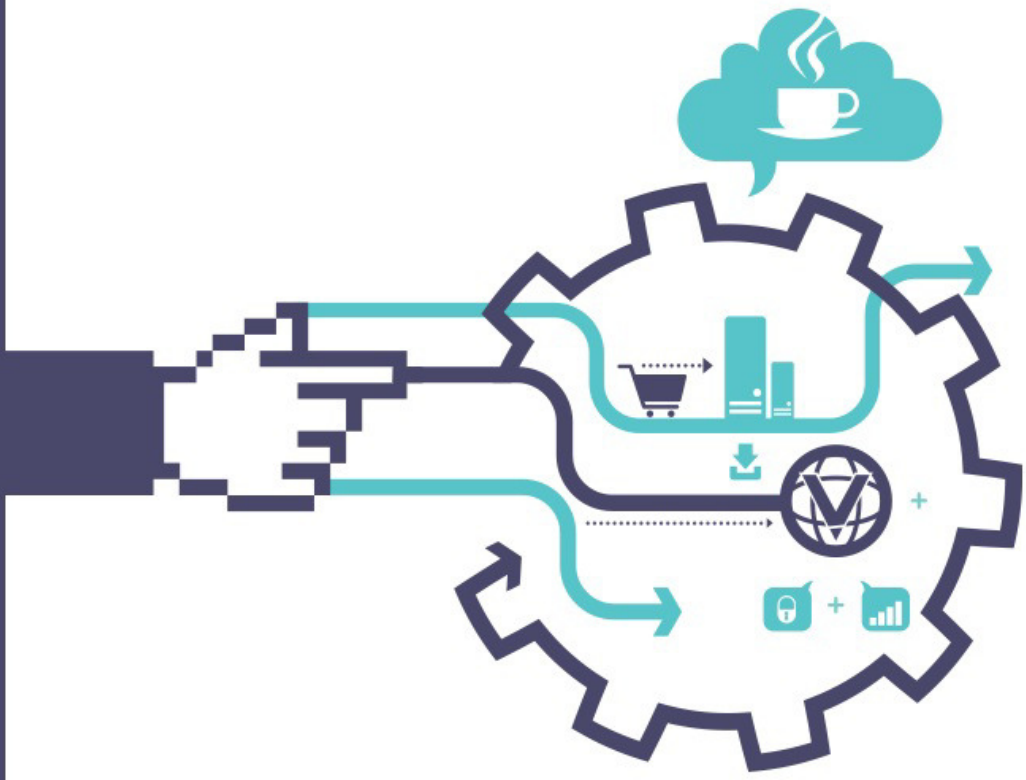
VRaptor

Desenvolvimento Ágil para Web com Java



Casa do Código

LUCAS CAVALCANTI

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-66250-26-8

EPUB: 978-85-66250-80-0

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sumário

Versão: 21.9.11

Capítulo 1

INTRODUÇÃO

1.1 UM POUCO DE HISTÓRIA

Há muito tempo, desenvolver uma aplicação web em Java era uma tarefa muito trabalhosa. Por um lado, tínhamos os Servlets e JSP, nos quais todo o tratamento das requisições era manual e cada projeto ou empresa acabava criando seu próprio *framework* — "O" framework — para trabalhar de uma forma mais agradável. Por outro lado, tínhamos vários frameworks que se propunham a resolver os problemas da dificuldade de se trabalhar com os Servlets, mas que lhe obrigavam a escrever toneladas de XML, arquivos `.properties` e a estender classes ou implementar interfaces para poder agradar a esses frameworks.

Nesse cenário, surgiu a ideia de um framework mais simples, que facilitasse o desenvolvimento web sem tornar o projeto refém das suas classes e interfaces. Surgia o VRaptor, criado pelos irmãos Paulo Silveira e Guilherme Silveira, em 2004 na Universidade de São Paulo. Em 2006, foi lançada a primeira versão estável: o VRaptor 2, com a ajuda do Fabio Kung, do Nico Steppat e vários outros desenvolvedores, absorvendo várias ideias e boas práticas que vieram do Ruby on Rails.

Em 2009, o VRaptor foi totalmente reformulado, levando em

conta a experiência obtida com os erros e acertos da versão anterior e de muitos outros frameworks da época. A versão 3 levava ao extremo os conceitos de Convenção sobre Configuração e Injeção de Dependências, em que todos os comportamentos normais podiam ser mudados facilmente usando anotações ou sobrescrevendo um dos componentes internos — bastando, para tanto, implementar uma classe com a interface desse componente.

Além disso, ele possibilitava criar de uma maneira fácil não apenas aplicações web que rodam no *browser*, mas também serviços web que seguem as ideias de serviços RESTful. Isso facilitava a comunicação entre sistemas e a implementação de AJAX no browser.

Em meados de 2013, motivados pelo grande poder e facilidade de uso do Java EE 7, o VRaptor foi novamente reformulado, desta vez totalmente baseado em CDI, a especificação de injeção de dependências. Com isso, além de todas as vantagens que já existiam no VRaptor 3, temos a possibilidade de usar os recursos nativos do servidor de aplicação, totalmente integrado com os componentes da aplicação.

Como o CDI possui um modo *standalone*, ainda é possível rodar a aplicação em um servidor web, como o Jetty ou o Tomcat, nesse caso precisando adicionar manualmente outros recursos do Java EE, como a JPA. A versão 4 final está disponível desde março de 2014.

1.2 AONDE VOCÊ VAI CHEGAR COM ESTE LIVRO?

O objetivo deste livro é mostrar muito além do uso básico do VRaptor e suas convenções. Mais importante do que aprender essas convenções, é entender a arquitetura interna do VRaptor e como criar suas próprias convenções, adaptando-o para as necessidades específicas do seu projeto. Saber como as coisas funcionam internamente é metade do caminho andado para que você tenha um domínio sobre o framework e se sinta confortável para fazer o que quiser com ele.

Durante os mais de quatro anos do VRaptor 3 e agora no começo do VRaptor 4, tenho respondido as mais variadas dúvidas sobre o seu uso no GUJ (http://www.guj.com.br), o que nos ajudou muito a modelar as novas funcionalidades e descobrir *bugs*. Mas, muito mais do que isso, mostrou-nos todo o poder da extensibilidade: mesmo os problemas mais complexos e necessidades mais específicas dos projetos conseguiram ser resolvidos sobrescrevendo o comportamento do VRaptor usando os meios normais da sua API, ou sobrescrevendo um de seus componentes.

Neste livro, você vai ver o uso básico e esperado, junto com o que considero boas práticas e uso produtivo das ferramentas que esse framework proporciona. Mas também vamos compartilhar vários casos do uso não esperados e aprender como também é fácil implementá-los.

Portanto, este livro é para quem quer entender como funciona o VRaptor e como aproveitar todo o seu poder e sua extensibilidade para tornar o desenvolvimento de aplicações o mais fácil e produtivo quanto possível.

Está pronto para começar?

Capítulo 2

O COMEÇO DE UM PROJETO COM VRAPTOR

Um grande problema no desenvolvimento de um projeto é justamente como (e quando) começá-lo. Isso não só na parte técnica. É preciso definir muito bem qual é o problema a ser resolvido, qual é o perfil do usuário do sistema, como será a usabilidade, onde será instalado, e muitos outros detalhes que devem influenciar nas escolhas que você deve fazer.

Hoje em dia, temos muitas ferramentas — bibliotecas, frameworks, linguagens de programação, servidores etc. — à nossa disposição, o que torna muito difícil definir o que usar. Muitas destas ferramentas são boas e adequadas para resolver o seu problema. Entretanto, mais importante do que saber quando usar uma ferramenta é saber quando não usá-la, ou quando ela vai atrapalhar mais do que ajudar.

2.1 VANTAGENS E DESVANTAGENS

O VRaptor é um framework feito para desenvolver aplicações web e também muito bom para desenvolver APIs HTTP/REST para comunicação entre sistemas. É um framework MVC que dá um suporte muito bom à parte do Modelo — as classes que contêm

a lógica de negócio da sua aplicação.

Com ele, é muito fácil aplicar as melhores práticas de Orientação a Objetos, já que ele não impõe restrições ao *design* das suas classes. Ele também facilita a prática de inversão de controle por injeção de dependências.

É também um dos frameworks mais extensíveis que existe em Java. Por ser construído em cima do CDI, a especificação de injeção de dependências do Java EE, todos os comportamentos são implementados por componentes com interfaces específicas, que podem facilmente ser sobrescritos com anotações do CDI. Se for feito o *deploy* em um servidor de aplicação, como o *Glassfish* ou o *Wildfly*, todos os serviços do Java EE podem ser integrados às classes da aplicação, sem nenhuma configuração adicional.

Além disso, é um framework que nos deixa totalmente livre para escolher sua camada de Visualização. O que também significa que não nos ajudará muito para desenvolver o HTML das páginas, diferente de alguns outros frameworks como, por exemplo, o JSF.

Você pode usar JSP, Velocity, Freemarker para gerar os templates, mas os componentes visuais precisam ser feitos manualmente, ou usando alguma biblioteca externa, como o jQuery UI, Angular.JS, Bootstrap ou ExtJS. Cada vez mais, essas bibliotecas se tornam mais simples de serem usadas, então até mesmo com pouco conhecimento de JavaScript, HTML e CSS, é possível criar uma interface rica e interessante.

2.2 O PROJETO: LOJA DE LIVROS

Nas próximas páginas, desenvolveremos uma loja de livros

online que terá duas partes: a administrativa, na qual serão gerenciados todos os livros, quantidade em estoque etc.; e a parte de vendas, em que serão mostrados todos os livros para que o usuário possa escolhê-los e comprá-los.

Essas duas partes serão feitas em projetos separados, que trocarão dados sempre que necessário. A parte administrativa guardará todos os dados referentes aos livros em um banco de dados. A parte de vendas não terá acesso a esse banco, apenas consultará os dados do livro usando a API da parte administrativa.

A ideia desse projeto é ser simples o suficiente para que consigamos começar a desenvolvê-lo sem muito contexto inicial, mas ao mesmo tempo tratando de diversos aspectos reais do desenvolvimento de aplicações web, como cadastros, uploads de imagens, integração entre sistemas, comunicação com o banco de dados, transações, segurança, entre outros.

2.3 PRÉ-REQUISITOS

O VRaptor 4 tem como pré-requisitos principais a JDK 7 e o CDI 1.1, ou superiores. Portanto, é necessário usar algum dos servidores que suportam essas versões. Caso seja usado um servidor web, como o Jetty ou o Tomcat, é necessário adicionar o Weld 2.x como dependência da aplicação.

Até agora, os servidores testados e suportados são o Glassfish 4, Wildfly 8, Tomcat 7 e Jetty 8, sendo que os dois últimos com a adição do Weld 2.0 e o seu listener no `web.xml` :

```
<listener>
    <listener-class>
      org.jboss.weld.environment.servlet.Listener
```

```
    </listener-class>
</listener>
```

Além disso, existem algumas pequenas configurações necessárias dependendo do servidor usado. Esses ajustes podem ser encontrados na wiki do VRaptor 4 no GitHub, em: http://www.vraptor.org/pt/docs/dependencias-e-pre-requisitos/.

2.4 CRIANDO OS PROJETOS

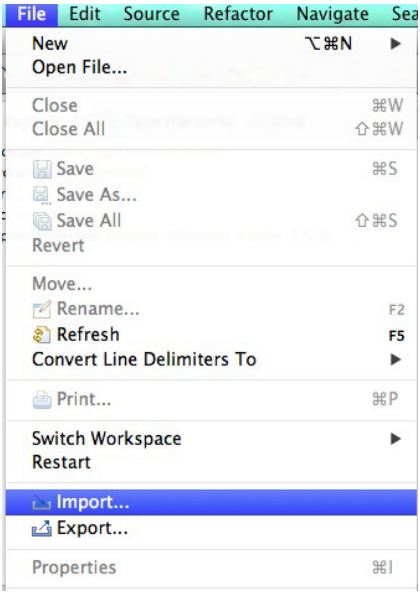
Existem várias formas de criar um projeto com VRaptor, cada uma com seus pré-requisitos e flexibilidades. Mostraremos a seguir as maneiras mais comuns e suas limitações e vantagens, para que você possa escolher a que achar mais adequada.

VRaptor Blank Project

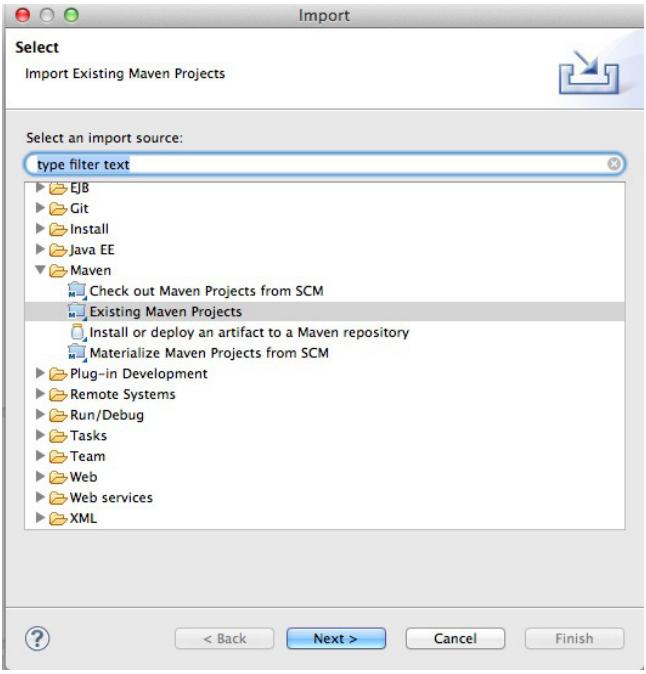
O VRaptor Blank Project é um projeto preparado com o mínimo necessário para rodar o VRaptor, usando o Maven para gerenciar as dependências. Para baixá-lo, vamos à página de *Downloads* do VRaptor: https://bintray.com/caelum/VRaptor4/br.com.caelum.vraptor.

Esse projeto está em um ZIP com o nome `vraptor-blank-project-4.x.y.zip` , no qual `4.x.y` é a última versão do VRaptor e pode ser importada na sua IDE como um projeto maven. No caso do Eclipse Kepler:

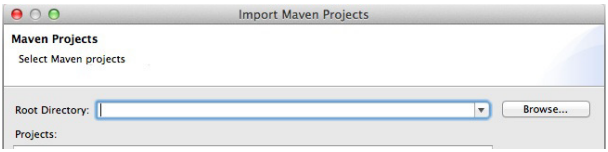
1. Extraia o arquivo `vraptor-blank-project-4.x.y.zip` para uma pasta conhecida.
2. Acesse o menu `File >> Import` .



3. Selecione `Existing Maven Project`.



4. Clique em `Browse` , em `Root directory` .



5. Escolha a pasta `vraptor-blank-project` que foi descompactada.

6. Clique em `Finish` . Um projeto com o nome `vraptor-blank-project` deve aparecer.
7. Clique com o botão direito no projeto e escolha `Run As >>`

`Run On Server`.

8. Escolha algum servidor existente ou crie um.
9. Deve aparecer uma tela dizendo: `"It works!! VRaptor! /vraptor-blank-project/"`.
10. O projeto com o VRaptor está configurado com sucesso!

Zip de distribuição do VRaptor

Também na página de Downloads do VRaptor existe um arquivo chamado `vraptor-4.x.y.zip`, que contém a distribuição completa da última versão do VRaptor. Nesse ZIP, podemos encontrar o `jar` do VRaptor, suas dependências (pasta `lib`), seu javadoc (pasta `apidocs`) e código-fonte (pasta `src`). Assim, já é possível linkar esses artefatos na sua IDE (footnote Eclipse, Netbeans etc.) e facilitar o desenvolvimento.

Para criar um projeto usando esse ZIP, você precisa criar um novo projeto web na sua IDE. No caso do Eclipse:

1. Crie um novo projeto web. Aperte `Ctrl+N` e escolha `Dynamic Web Project`.
2. Escolha um nome de projeto, por exemplo, **livraria**.
3. Selecione como `Target Runtime` um dos servidores compatíveis, como o Tomcat 7.
4. Selecione **3.0** como `Dynamic web module version`.
5. Clique em `Finish`.
6. Abra o ZIP do VRaptor e copie o arquivo `vraptor-4.x.y.jar` para a pasta `WebContent/WEB-INF/lib`.
7. Também no ZIP do VRaptor, abra a pasta `lib/mandatory` e copie todos os `jar`s para `WebContent/WEB-INF/lib`.
8. Copie também os opcionais que deseja usar:

- `lib/fileupload` : caso precise fazer uploads.
- `lib/javaee` : caso esteja usando Jetty ou Tomcat.
- `lib/jodatime` : uma API melhor para datas.
- `lib/serialization` : para gerar JSON ou XML.
- `lib/validation` : para usar validações do Bean Validations, caso esteja no Jetty ou Tomcat.

9. Atualize o projeto (`F5`) e rode esse projeto no servidor.
10. Faça o `Hello World` , seguindo os passos da seção *Criando um Hello World* adiante.
11. Projeto configurado com sucesso!

Esse processo é um pouco mais manual, mas é bem mais flexível e não precisa de nada instalado a mais no sistema.

Maven

Outro bom jeito de começar um projeto Java é usando o Maven, uma ferramenta de build e gerenciamento de dependências. Os `jar` s do VRaptor estão no repositório do Maven, portanto, podemos acrescentá-lo como dependência no Maven.

Para isso, precisamos que ele esteja instalado no computador (http://maven.apache.org/download.html), ou na IDE (http://maven.apache.org/eclipse-plugin.html ou http://maven.apache.org/netbeans-module.html). Para criar um projeto VRaptor com o Maven, siga os passos:

1. Rode o plugin de archetype do Maven. Na linha de comando:

```
mvn archetype:generate
```

No Eclipse: dê `Ctrl+N`, escolha `Maven Project` e aperte `Next` até a tela de seleção do archetype.

2. Selecione o `maven-archetype-webapp` (na linha de comando, você pode digitar isso em vez do número).

3. Escolha um `groupId` (por exemplo, `br.com.casadocodigo`), um `artifactId` (por exemplo, `livraria`) e um pacote (por exemplo, `br.com.casadocodigo.livraria`).

4. No `pom.xml` do projeto criado, adicione a dependência do VRaptor (editando o XML ou pelos wizards do Eclipse):

```
<project ....>
<dependencies>
 ...
 <dependency>
     <groupId>br.com.caelum</groupId>
     <artifactId>vraptor</artifactId>
     <version>4.0.0</version><!-- ou a última versão -->
 </dependency>
</dependencies>
</project>
```

5. Faça o **Hello World**, como na seção *Criando um Hello World* adiante.

6. Projeto criado com sucesso!

O Maven é muito produtivo se você quiser trabalhar do jeito dele. Porém, se precisar fazer algo mais personalizado para seu projeto, prepare-se para se divertir a valer com toneladas de configurações em XML. E também reserve um espaço no seu HD para fazer o backup da internet a cada comando.

2.5 CRIANDO UM HELLO WORLD

Com o projeto criado, podemos ver o VRaptor funcionando rapidamente, apenas usando as suas convenções básicas. O objetivo do VRaptor é se intrometer o mínimo possível nas classes da sua aplicação, então vamos criar uma classe Java normal, que faça um simples "Olá mundo":

```
public class Mundo {

    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

Quando estamos em uma aplicação web, queremos poder executar algum código quando o usuário faz uma requisição ao sistema — ao acessar uma URL no navegador, por exemplo. Para que o método `ola()` seja acessível por uma URL, tudo que precisamos fazer é anotar a classe `Mundo` com `@Controller`:

```
@Controller
public class Mundo {
    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

Seguindo a convenção do VRaptor, podemos acessar esse método pela URL http://localhost:8080/livraria/mundo/ola. As classes que recebem requisições web no nosso sistema têm um papel especial na arquitetura **MVC** — são os **Controllers**. Para deixarmos mais claro esse papel da nossa classe `Mundo`, podemos dizer isso no nome da classe, ou seja, `MundoController`.

O VRaptor exclui o sufixo `Controller` das convenções para

definir a URL. Podemos renomear nossa classe e tudo continuará funcionando:

```
@Controller
public class MundoController {
    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

Dessa forma, continuamos acessando o método `ola()` pela URL http://localhost:8080/livraria/mundo/ola. Esse método imprime `"Olá Mundo!"`, só que no console do servidor. Precisamos mostrar uma tela para o usuário que acessou a URL do método e, seguindo o padrão MVC, geramos essa tela em outro arquivo, por exemplo um JSP.

Essa é a parte **V** do MVC, a visualização. Por padrão, ao final da invocação do método `ola`, o VRaptor redireciona a requisição para o JSP em `/WEB-INF/jsp/mundo/ola.jsp`.

Os JSPs ficam em `/WEB-INF/jsp` para não serem acessíveis diretamente pelo navegador, forçando todas as requisições a passarem pelo ciclo completo do MVC. Depois disso, vem o nome do controller (sem a palavra `Controller`) seguido pelo nome do método e da extensão `jsp`. Todas essas convenções podem ser configuradas e modificadas, e veremos mais disso adiante.

Criando então a JSP correspondente ao método `ola()` do `MundoController`, em `/WEB-INF/jsp/mundo/ola.jsp`:

```
<html>
  <h1>Olá Mundo!!!</h1>
</html>
```

Para ver tudo funcionando, vamos subir o servidor e acessar

http://localhost:8080/livraria/mundo/ola. A página que acabamos de criar deverá aparecer:

E pronto! Temos todo o código necessário para processar uma requisição com o VRaptor. Agora vamos criar os projetos necessários para criar nossa livraria.

2.6 CRIANDO OS PROJETOS DA LIVRARIA

O nosso sistema será dividido em duas partes: uma que cuidará do site onde os usuários comprarão os livros — o projeto `livraria-site` —, e outra que cuidará do cadastro dos livros — o projeto `livraria-admin` . Além disso, usaremos o Tomcat como servidor e o MySQL como banco de dados.

Para criar esses projetos, podemos usar qualquer um dos métodos descritos na seção *Criando os projetos*. Mas para facilitar um pouco a configuração inicial, usaremos o `vraptor-blank-project` , seguindo os passos:

- Descompacte o ZIP do `blank project` para uma pasta conhecida, por exemplo, a `Downloads` .
- Copie a pasta `vraptor-blank-project` gerada para a sua pasta `workspace` do Eclipse, renomeando-a para

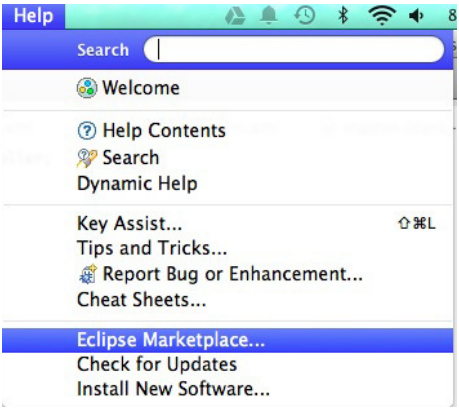
`livraria-site`. Copie novamente a pasta para o `workspace`, desta vez renomeando-a para `livraria-admin`.

- Abra na pasta `livraria-site` o arquivo `pom.xml` em qualquer editor de texto, trocando o `groupId` e `artifactId` das primeiras linhas para `br.com.casadocodigo.livraria` e `livraria-site`, respectivamente. Faça a mesma coisa na pasta `livraria-admin`, desta vez trocando o `artifactId` para `livraria-admin`.
- No Eclipse, vá no menu `File >> Import >> Existing Maven Project`.
- Clique em `Browse` e selecione a pasta `livraria-site` do seu workspace.
- Clique em `Finish`.
- Repita os três últimos passos para a pasta `livraria-admin`.

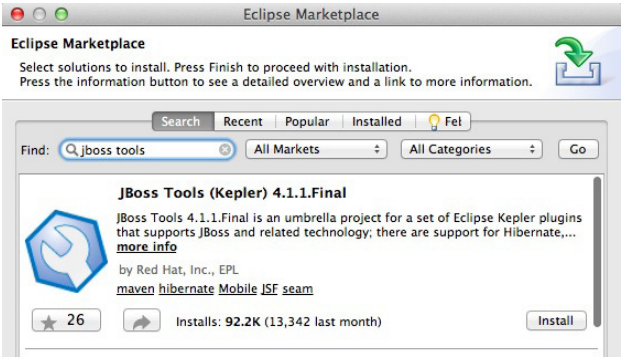
Você também pode acompanhar a evolução do projeto navegando pelos commits desse repositório do GIT: https://github.com/lucascs/livraria-vraptor4/commits/master.

Para rodar esses projetos, vamos usar o Wildfly 8, que é a última versão do JBoss e que contém o CDI e todas as outras implementações de um servidor de aplicação.

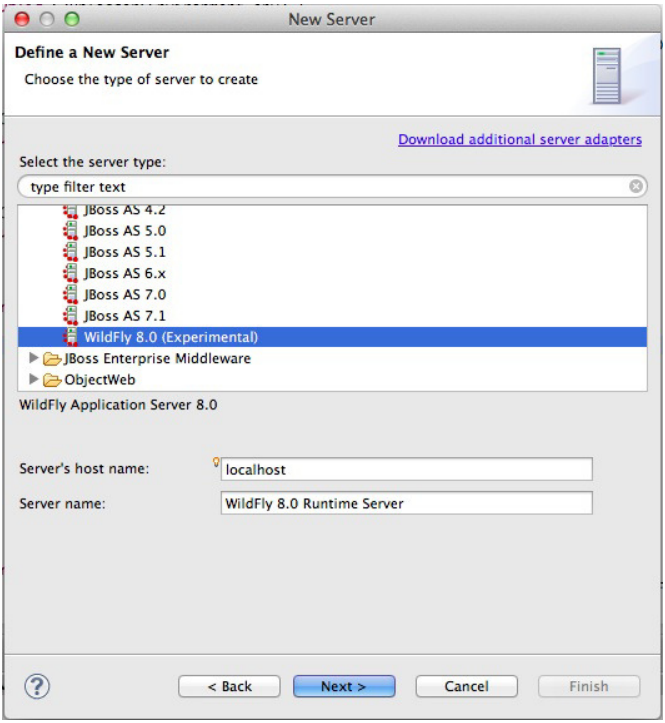
Para isso, no Eclipse (Kepler), vamos instalar o JBossTools. Use o menu `Help >> Eclipse Marketplace`.



Procure por `jboss tools` e instale o do Eclipse Kepler, clicando em `Install` e seguindo o processo até o final.



O próximo passo é adicionar o servidor Wildfly, no menu `File >> New >> Other >> Server .`



Ao clicar em `Next`, escolha em `Home directory` a pasta onde você baixou o Wildfly, ou clique em `Download and install runtime`. Espere o download terminar e coloque a pasta escolhida no campo `Home directory`. Continue clicando em `Next` até aparecerem os projetos da livraria. Adicione `livraria-site` e `livraria-admin`, e clique em `Finish`.

Agora inicie o servidor que acabou de ser criado, na view `Servers` (se ela não apareceu, use `Ctrl + 3` e digite `Servers`). Se tudo der certo, você verá no console que o servidor foi inicializado. Tente acessar as seguintes URLs:

- http://localhost:8080/livraria-site/

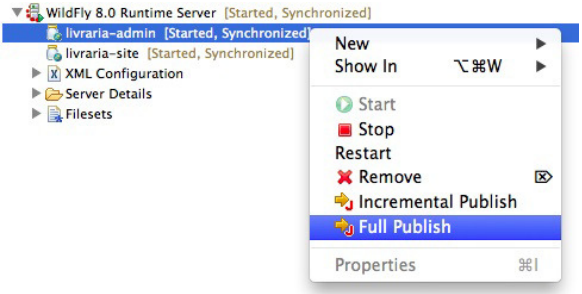
- http://localhost:8080/livraria-admin/

Elas devem apresentar respectivamente as seguintes respostas, vindas do `vraptor-blank-project`:

```
It works!! VRaptor! /livraria-site/
```

```
It works!! VRaptor! /livraria-admin/
```

Caso essas respostas não apareçam, é possível que os projetos não tenham sido publicados. Abra a aba de `Servers`, expanda o servidor, clique com o botão direito em cada um dos projetos e clique em `Full Publish`.



Tente acessar novamente as URLs e, caso as respostas apareçam, estamos quase prontos para começar a desenvolver os projetos. Vamos aproveitar os recursos do servidor de aplicação e usar a JPA para interagir com o banco de dados.

Para isso, vamos configurar os projetos para usarem a JPA:

- Clique com o botão direito no projeto `livraria-admin` e escolha `Properties`.
- Abra a guia `Project Facets`.
- Selecione `JPA`.

- Clique em `Further configuration available`.
- Na seção `Platform`, escolha `Hibernate 2.1`.
- Na seção `JPA implementation`, escolha o `Type` `Disable Library Configuration`.
- Na seção `Connection`, escolha na lista `DefaultDS`, que já vem por padrão no servidor.
- Clique em `Ok`, e então `Ok` novamente.
- Verifique que o arquivo `persistence.xml` foi criado em `src/main/resources`.
- Repita esses passos para o projeto `livraria-site`.

Agora sim, podemos continuar o desenvolvimento das duas partes do nosso sistema: o `livraria-admin` e o `livraria-site`, que veremos nos próximos capítulos, a começar pelo cadastro de livros no admin.

CAPÍTULO 3

CRIE O SEU PRIMEIRO CADASTRO

Agora que já temos o projeto criado, completamente configurado e já fizemos até um "olá mundo" para garantir que tudo funciona, podemos começar a implementar de fato nossa aplicação. Vamos fazer a criação do cadastro principal, com o qual poderemos gravar e manipular as informações dos livros.

Passaremos por diversas características do framework e veremos todas elas em detalhes no decorrer do livro. O mais importante é que agora você perceba a facilidade e objetividade do framework. Preparado?

3.1 CRIAÇÃO DOS MODELOS

O nosso sistema será uma loja de livros, logo um dos dados mais importantes que teremos de guardar é a coleção de livros do sistema. Cada livro terá um título, uma descrição, um ISBN, um preço e uma data de publicação. Dando uma atenção especial aos dois últimos atributos:

- **Preço:** um valor em dinheiro. Vamos representá-lo com o tipo `BigDecimal` , com o qual podemos manter a precisão

que escolhermos para ele e que também possui algumas facilidades se precisarmos realizar cálculos.

- **Data de publicação:** podemos representar essa data usando as classes já embutidas no Java, `Date` ou `Calendar`. Escolheremos nesse momento a classe `Calendar`, por ser um pouco mais completa e possuir métodos para manipular a data, sem precisarmos fazer contas com *timestamps*.

> **MELHOR ALTERNATIVA PARA AS DATAS**
>
> Tanto a classe `java.util.Date` como `java.util.Calendar` possuem dois problemas fundamentais: a interface dos objetos é muito ruim de se trabalhar, e elas só conseguem representar data e hora. Para resolver isso, existe um projeto chamado Joda-time (http://joda-time.org), que possui representações bem mais completas e flexíveis para abstrações de tempo.
>
> Por exemplo, no nosso caso, só precisamos da data, sem a hora. Então, poderíamos usar a classe `LocalDate` do Joda-time. A partir do Java 8, existe o pacote `java.time` que também trata datas e horas de maneira muito melhor.

Vamos criar a classe que representa esse livro no projeto `livraria-admin`.

```
package br.com.casadocodigo.livraria.modelo;

import java.math.BigDecimal;
```

```
import java.util.Calendar;

public class Livro {
    private String titulo;
    private String descricao;
    private String isbn;
    private BigDecimal preco;
    private Calendar dataPublicacao;

    //getters/setters
}
```

Para cadastrar os livros no nosso sistema, precisamos armazená-los em algum lugar. No momento, a forma com a qual vamos conseguir fazer esse armazenamento não importa muito, portanto, vamos criar uma classe que representa um conjunto de livros — um **repositório**, conforme explicado no livro *Domain Driven Design* do Eric Evans.

Na vida real, guardamos livros em uma estante, então nada mais justo do que usar esse nome para abstrair o lugar onde guardamos livros. Criaremos uma interface com todas as operações que queremos ter, já que ainda não sabemos ainda como será a implementação.

```
package br.com.casadocodigo.livraria.modelo;

public interface Estante {

   void guarda(Livro livro);

   List<Livro> todosOsLivros();
}
```

3.2 CRIANDO O CADASTRO

Para criar um livro agora no nosso projeto web, precisamos

criar uma página com o formulário de adição do livro.

```
<form action="?" method="post">
    ...
</form>
```

Formulários como este podem conter alguma parte mais dinâmica, por exemplo, a lista de categorias em que um livro pode estar. Por esse motivo, sempre que criamos uma página, passamos primeiro por uma classe Java que vai popular os dados necessários para ela.

Criaremos então um controlador, que vai controlar o nosso cadastro de livros. Esse será o nosso `LivrosController`, com um método para fornecer acesso ao formulário.

```
package br.com.casadocodigo.livraria.controlador;

public class LivrosController {

    public void formulario() {}

}
```

Para que o VRaptor passe a gerenciar essa classe como um controlador, precisamos anotá-la com `@Controller`. Isso significa que essa classe controla um dos recursos da aplicação, no caso os livros. Essa classe será o ponto de acesso via web que controlará todas as operações que pudermos fazer com um livro.

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

@Controller
public class LivrosController {

    public void formulario() {}

}
```

A partir desse momento, seguindo as convenções do VRaptor, conseguimos executar o que está dentro do método `formulario`, acessando a URI `/livros/formulario`. Vale lembrar de que a convenção é: `/<nome_do_recurso>/<nome_do_metodo>`. O nome do recurso é o nome da classe sem a palavra `Controller`, **Livros**, com a primeira letra minúscula: `livros`. O nome do método é exatamente o nome do método. O caminho completo no nosso caso seria http://localhost:8080/livraria-admin/livros/formulario.

O método `formulario` não faz nada de interessante ainda, mas vamos usá-lo para chegar até a página do formulário que, segundo a convenção do VRaptor, estará em `WEB-INF/jsp/livros/formulario.jsp`. Essa convenção é parecida com a da URI: `WEB-INF/jsp/<nome_do_recurso>/<nome_do_metodo>.jsp`. O caminho completo para a JSP seria:

```
livraria-admin/src/main/webapp/WEB-INF/jsp/livros/formulario.jsp
```

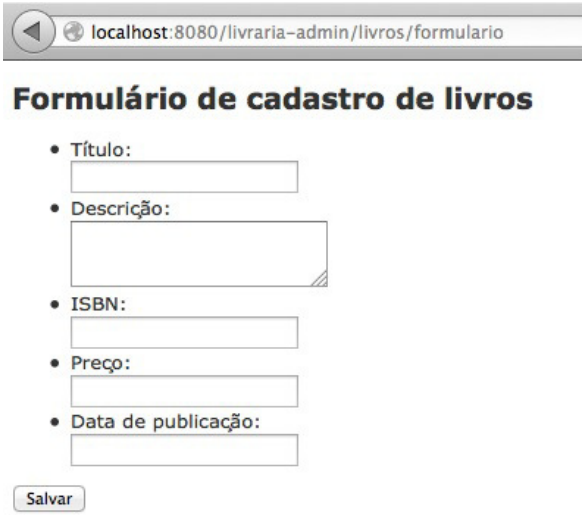
Se o projeto for criado direto pelo Eclipse, sem o maven, em vez de `src/main/webapp`, use `WebContent`.

Podemos seguir com a criação do formulário. Precisamos criar um objeto `livro` a partir dele. Para isso, devemos usar mais uma convenção do VRaptor no nome dos inputs. Precisamos de um nome de variável para o livro criado, por exemplo, `livro`.

Cada input deve ter seu atributo `name` começando com o nome dessa variável, e cada propriedade acessível delimitada por pontos. Para popular o título do livro, usamos `livro.titulo`, para o preço, `livro.preco`, e assim por diante:

```
<form method="post">
  <h2>Formulário de cadastro de livros</h2>
```

```
  <ul>
    <li>Título: <br/>
        <input type="text" name="livro.titulo" /></li>

    <li>Descrição: <br/>
        <textarea name="livro.descricao"></textarea></li>

    <li>ISBN: <br/>
        <input type="text" name="livro.isbn" /></li>

    <li>Preço: <br/>
        <input type="text" name="livro.preco" /></li>

    <li>Data de publicação: <br/>
      <input type="text" name="livro.dataPublicacao" /></li>
  </ul>
  <input type="submit" value="Salvar" />
</form>
```



Assim podemos criar o método do controller que vai receber o post desse formulário. Se quisermos o livro populado, o parâmetro desse método deve se chamar `livro`, que é o prefixo dos inputs

do formulário:

```
@Controller
public class LivrosController {
    public void formulario() {}

    public void salva(Livro livro) {
        Estante estante = new UmaEstanteQualquer();
        estante.guarda(livro);
    }
}
```

Criando a classe UmaEstanteQualquer

Esse código não compila, pois não existe a classe `UmaEstanteQualquer`. Mas é possível rapidamente criá-la usando a sua IDE (por exemplo, o Eclipse ou o Netbeans). No caso do Eclipse, com o cursor em `UmaEstanteQualquer`, aperte `Ctrl + 1` e escolha a opção de criar a classe. A classe criada já implementará `Estante` e já vai estar com os métodos da interface implementados, porém em branco.

Por enquanto, você pode deixar esses métodos em branco. No próximo capítulo, vamos implementá-los fazendo a comunicação com banco de dados e aprendendo como o VRaptor pode nos ajudar nessa tarefa.

Como o método se chama `salva`, vai receber a requisição em `/livros/salva`, de modo que precisamos colocar isso na action do formulário. Essa URI, no entanto, está sem o *context-path*, então podemos usar a tag `<c:url`, ou usar a variável `${pageContext.request.contextPath}` para acrescentá-lo:

```
<form action="<c:url value="/livros/salva"/>" method="post">
```

Ou:

```
<form action="${pageContext.request.contextPath}/livros/salva"
      method="post">
```

Melhor ainda, se você não quer lembrar de qual é a URI de um dos métodos do controller, você pode usar o `linkTo`. Com ele, você passa o controller e o método, e ele retorna a URI correspondente, já com o *context-path*.

Passamos o nome do controller entre colchetes e o nome do método após: `${linkTo[NomeDoController].nomeDoMetodo}`. No nosso caso, ficaria assim:

```
<form action="${linkTo[LivrosController].salva}" method="post">
```

Dessa forma, fica mais claro o que vai ser executado: a ação do formulário é um link para o `LivrosController`, método `salva`. Quando o usuário preencher o formulário e clicar em `Salvar`, o VRaptor pegará cada um dos valores digitados nos inputs (com o prefixo `livro`) e preencherá uma instância de `Livro` com esses valores.

Essa instância será passada para o método `salva`, com o qual guardamos o livro na `Estante`. Após executar o método `salva`, o VRaptor automaticamente redireciona para a página `WEB-INF/jsp/livros/salva.jsp`, na qual podemos indicar que o livro foi salvo:

```
<h2>Livro adicionado com sucesso!</h2>

<p>Veja aqui a
  <a href="${linkTo[LivrosController].lista}">
    lista de todos os livros
```

```
  </a>
</p>
```

Colocamos um link para a lista de todos os livros, assim podemos ver tudo o que foi cadastrado. Agora precisamos criar essa listagem, primeiramente criando o método que colocamos no link:

```
class LivrosController {
    //...
    public void lista() {

    }
}
```

Nesse caso, queremos mostrar a lista de todos os livros, que vamos buscar de algum lugar — da `Estante` onde estamos guardando-os — e queremos deixar essa lista disponível para usar na JSP. Como estamos executando um método na requisição, podemos usar o jeito natural do Java, retornando o valor; no caso, a lista de livros:

```
public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
}
```

Por padrão, o VRaptor deixa o retorno dos métodos do controller disponível no JSP, seguindo outra convenção. Se o retorno do método é um objeto do tipo `Livro`, ele será colocado em uma variável chamada `${livro}`, ou seja, nome da classe com a primeira letra em minúsculo.

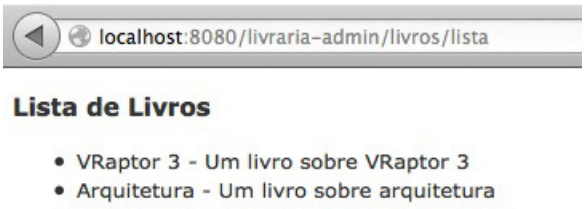
No caso de o retorno ser uma `List<Livro>`, o nome da variável no JSP será `${livroList}`, ou seja, o nome da classe dos elementos da lista com a primeira minúscula, seguido de `List`. Podemos criar o `lista.jsp` dentro de `WEB-INF/jsp/livros`,

que é o JSP padrão para esse método `lista` :

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>${livro.titulo} - ${livro.descricao}</li>
</c:forEach>
</ul>
```

Por enquanto, para ver essa lista preenchida, precisamos modificar a classe `UmaEstanteQualquer` para retornar uma lista de livros.

```
  @Override
  public List<Livro> todosOsLivros() {
    Livro vraptor = new Livro();
    vraptor.setIsbn("123-45");
    vraptor.setTitulo("VRaptor 3");
    vraptor.setDescricao("Um livro sobre VRaptor 3");

    Livro arquitetura = new Livro();
    arquitetura.setIsbn("5678-90");
    arquitetura.setTitulo("Arquitetura");
    arquitetura.setDescricao("Um livro sobre arquitetura");

    return Arrays.asList(vraptor, arquitetura);
  }
```



3.3 COMPLEMENTANDO O CADASTRO

Os livros são o carro-chefe do nosso sistema e, para poder

vendê-los, precisamos que as suas informações sejam completas e de qualidade. Já criamos a listagem de livros e o formulário de criação de um livro. Isso não é o bastante para manter nossa livraria funcionando, pois eles podem sofrer alterações ao longo do tempo, principalmente no seu preço. Portanto, vamos criar a possibilidade de editar os livros existentes.

Vamos modificar a listagem adicionando um link para permitir a alteração de um livro. Colocaremos um link para a edição em cada livro mostrado:

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>${livro.titulo} - ${livro.descricao}
      <a href="${linkTo[LivrosController].edita}">
        Modificar
      </a>
    </li>
</c:forEach>
</ul>
```

Esse link deverá abrir um formulário de edição, que é bem parecido com o formulário de criação, mas precisa vir com os dados do livro que estamos querendo modificar já preenchidos. Para isso, precisamos identificá-lo de alguma forma, por exemplo, usando o ISBN. Para passar esse valor para o método `edita`, usamos a forma que estamos acostumados para receber um valor — através de um parâmetro no método:

```
public class LivrosController {

    public void formulario() {}
    public void edita(String isbn) {}
}
```

Como chamamos o parâmetro do método de `isbn`, podemos

passá-lo no link "Modificar", passando um parâmetro na URI com o mesmo nome — `isbn`. Dessa forma, o VRaptor saberá que o valor passado na URI será passado como parâmetro do método:

```
<a href="${linkTo[LivrosController].edita}?isbn=${livro.isbn}">
    Modificar
</a>
```

Assim, se o ISBN do livro é `123-45`, o link gerado será `/livros/edita?isbn=123-45`. O valor `123-45` será passado como parâmetro do método, e podemos usá-lo para buscar o livro desejado na nossa `Estante`.

Para mostrar essa `Estante` na página, vamos usar a mesma estratégia da listagem: retornar o livro no método `edita`.

```
public Livro edita(String isbn) {
    Estante estante = new UmaEstanteQualquer();
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    return livroEncontrado;
}
```

Criando o método buscaPorIsbn

Novamente, esse código não compila e o método `buscaPorIsbn` fica com erro de compilação. Para criá-lo na interface `Estante`, use o `Ctrl + 1` no Eclipse e escolha a opção de criar esse método.

Esse atalho funciona toda vez que algo está com erro de compilação por não existir, e é apresentada uma solução rápida para o problema (*Quick Fix*). Experimente usá-lo para fazer a classe `UmaEstanteQualquer` compilar.

Queremos agora mostrar os dados desse livro em um formulário, assim vamos poder modificá-lo. Poderíamos simplesmente copiar o `formulario.jsp` para `edita.jsp` , e deixá-o preenchido, mas como toda alteração em um formulário vai ter de ser refeita no outro, vamos aproveitar o mesmo arquivo para os dois formulários. Para isso, precisamos modificar o `formulario.jsp` para que sirva tanto para criar um livro novo quanto para editar um já existente.

No resultado do método `edita` , podemos usar `${livro.titulo}` para mostrar o título do livro, `${livro.preco}` para mostrar o preço, e assim por diante. Para preencher os inputs com esses valores, vamos colocá-los no atributo `value` de cada um:

```
<form action="${linkTo[LivrosController].salva }" method="post">
  <h2>Formulário de cadastro de livros</h2>
  <ul>
    <li>Título: <br/>
        <input type="text" name="livro.titulo"
                        value="${livro.titulo}"/></li>

    <li>Descrição: <br/>
        <textarea name="livro.descricao">${livro.descricao
        }</textarea></li>

    <li>ISBN: <br/>
        <input type="text" name="livro.isbn"
                        value="${livro.isbn}"/></li>

    <li>Preço: <br/>
        <input type="text" name="livro.preco"
                        value="${livro.preco}"/></li>

    <li>Data de publicação: <br/>
      <input type="text" name="livro.dataPublicacao"
                      value="${livro.dataPublicacao}"/>
    </li>
  </ul>
```

```
  <input type="submit" value="Salvar" />
</form>
```

No resultado do método `formulario` não existe um `Livro` retornado, então as expressões `${livro.titulo}`, `${livro.preco}` etc. ficarão em branco, que é o esperado no formulário de adição. Agora só falta falar para o VRaptor usar o `formulario.jsp` como resultado do método `edita`.

Para modificar o resultado padrão de um método, o VRaptor possui a classe `Result`, que podemos receber como parâmetro do método de ação:

```
public Livro edita(String isbn, Result result) {
    //...
}
```

Falaremos mais sobre o `Result` no capítulo *Tomando o controle dos resultados*. Por enquanto, só queremos usar a mesma página do método `formulario`, ou seja, o mesmo **resultado** do método `formulario` desse mesmo controller. Dizemos isso com o método `of` do `Result`:

```
public Livro edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);

    result.of(this).formulario();

    return livroEncontrado;
}
```

Traduzindo, queremos o resultado (`result`) desse controller (`of(this)`) no método formulário (`.formulario()`). Como modificamos o resultado padrão, não podemos mais retornar o `Livro` nesse método, pois o retorno aconteceria após a mudança do resultado.

Retornamos o `livroEncontrado`, porque queremos incluí-lo na página do formulário, ou seja, incluí-lo no resultado do método. Então, podemos usar o `Result` para expressar essa intenção com o método `include`:

```
public void edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();

    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);

    result.of(this).formulario();
}
```

Dessa forma, incluímos o `livroEncontrado` no resultado, da mesma forma que fazíamos no retorno do método, mas **antes** de redirecionar para a página do formulário. Apesar de a variável no código Java ser `livroEncontrado`, esse objeto passará a `jsp` como `${livro}`, que é o nome da classe com a primeira letra minúscula. E a página usada será a `WEB-INF/jsp/livros/formulario.jsp` e não mais a `edita.jsp`. Nesse caso, só modificamos o JSP a ser utilizado — o método `formulario` **não** será executado.

Para conseguirmos ver o formulário preenchido adequadamente, podemos modificar o método `buscaPorIsbn` da classe `UmaEstanteQualquer`, retornando um dos livros:

```
@Override
public Livro buscaPorIsbn(String isbn) {
  return todosOsLivros().get(0);
}
```

Após modificar os dados do formulário e clicar em "Salvar", o VRaptor vai executar o método `salva`, conforme o que está configurado e, após guardar o livro na estante, a página de sucesso é mostrada. Se quisermos continuar cadastrando ou modificando livros, precisamos clicar no link "Ver todos os livros".

Para tornar o processo mais prático, já que conseguimos alterar o resultado padrão, vamos eliminar essa página de sucesso e retornar direto para a listagem de livros, mostrando uma mensagem de sucesso em cima da listagem.

Seguindo a mesma ideia do método `edita`, poderíamos modificar o `salva` para receber o `Result` e mudar o resultado para a listagem:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

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

```
}
```

Mas, como foi dito anteriormente, essa mudança de resultado **não** executaria o método `lista` e, portanto, não mostraria a lista de todos os livros — só a página `lista.jsp` sem a variável `${livroList}` que seria preenchida pelo método `lista`. Se queremos executar o método, costumamos falar que vamos **redirecionar** a execução de um método para outro. No nosso caso, queremos redirecionar para o método `lista`, então a chamada necessária é o `redirectTo`:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

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

Ou seja, como resultado do método `salva`, redirecionaremos para o método `lista` desse mesmo controller. Para indicar que o livro foi salvo, vamos adicionar uma mensagem antes do redirecionamento:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.include("mensagem", "Livro salvo com sucesso!");
    result.redirectTo(this).lista();
}
```

Nesse caso, estamos incluindo a `String` "Livro salvo com sucesso!" no `result`, e usando a variável `${mensagem}`. Assim podemos modificar nossa listagem para mostrar essa mensagem:

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

```
</p>
</c:if>
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>
      ${livro.titulo} - ${livro.descrição}
        <a href="${linkTo[LivrosController].edita}">
          Modificar
        </a>
    </li>
</c:forEach>
</ul>
```

Conseguimos construir até aqui as operações mais importantes do cadastro de livros, praticamente só usando as convenções básicas do VRaptor. Criamos URLs para executar o código do `LivrosController`, usando a anotação `@Controller`. Com os JSPs no caminho certo, conseguimos linkar para os métodos dos controllers usando o `${linkTo[NomeDoController].metodo}`.

Com os inputs de um formulário seguindo a convenção de nomes, conseguimos preencher objetos e usá-los no controller, como vimos no método `salva`. E, por fim, conseguimos alterar o resultado de um método do controller usando o `Result`.

Nos próximos capítulos, veremos que o VRaptor nos ajuda muito mais, não somente no controle das requisições, mas também

na organização dos componentes da sua aplicação. Afinal, o desenvolvimento de uma aplicação vai muito além de simples cadastros. Veremos também como integrar o que fizemos com o banco de dados e outros serviços.

CAPÍTULO 4

ORGANIZAÇÃO DO CÓDIGO COM INJEÇÃO DE DEPENDÊNCIAS

4.1 COMPLETANDO O FUNCIONAMENTO DO CONTROLLER

No capítulo anterior, os métodos do `LivrosController` foram implementados usando uma classe chamada `UmaEstanteQualquer`, para executar as operações:

```
public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
}
```

Isso foi feito porque, neste primeiro momento, para construirmos os métodos do controller, bastava apenas qualquer implementação de `Estante` fornecendo a interface necessária. Porém, para termos o sistema funcionando, precisamos definir uma implementação real de `Estante`, que vai guardar os livros para podermos mostrá-los em seguida.

São várias as maneiras de fazer isso, por exemplo, guardando os livros em arquivos, ou usando algum serviço de armazenamento

de dados. Mas a forma mais comum é usar uma implementação que guarda os livros em um **banco de dados**. Vamos, então, mudar a implementação de `Estante` do controller para a `EstanteNoBancoDeDados`.

```
public List<Livro> lista() {
    Estante estante = new EstanteNoBancoDeDados();
    return estante.todosOsLivros();
}
```

Porém, para se conectar a um banco de dados e conseguir salvar livros lá, é necessário informar qual é o sistema de banco de dados usado, usuário e senha para fazer a conexão e qual é a base de dados. Logo, poderíamos ter o seguinte código:

```
public List<Livro> lista() {
    Estante estante = new EstanteNoBancoDeDados(BDs.MySQL,
                          "usuario", "senha", "db_livraria");
    return estante.todosOsLivros();
}
```

E se essa estante abriu uma conexão com o banco de dados e eu já terminei de usá-la, eu deveria avisar a estante, se não ela correria o risco de manter a conexão aberta indefinidamente. Então, precisamos colocar o código para fechar essa conexão.

Uma `Estante` não precisa ser fechada, mas a implementação do banco de dados sim, então o método para isso só deveria estar na implementação. Dessa forma, o código ficaria:

```
public List<Livro> lista() {
    Estante estante = null;
    try {
        estante = new EstanteNoBancoDeDados(BDs.MySQL,
                          "usuario", "senha", "db_livraria");
        return estante.todosOsLivros();
    } finally {
        ((EstanteNoBancoDeDados)estante).fechaConexao();
```

```
    }
}
```

Repare como o código ficou bem mais complexo que antes, isso usando apenas a `Estante`. Imagine agora esse código espalhado por cada método do seu sistema que depende do banco de dado. A sua complexidade se dá por uma coisa: a classe `LivrosController` está **criando** a instância de `Estante` para poder trabalhar.

Para isso, o `LivrosController` precisa saber qual é a implementação de `Estante` adequada, quais são as configurações necessárias para criar uma instância dessa implementação e, ao final, saber se precisa sinalizar à instância a hora de fechar osrecursos abertos. Esse tipo de código está longe de ser responsabilidade da `LivrosController`, portanto, não deveria estar aqui.

Além disso, se precisarmos um dia trocar o banco de dados usado, ou passar a usar um serviço de armazenamento, teríamos de passar por todos os pontos do sistema que usam estantes trocando as implementações. E se quisermos guardar os livros só na memória nas máquinas dos desenvolvedores, mas na máquina de produção usar um banco de dados?

Ao criar uma `EstanteNoBancoDeDados` no `LivrosController`, estamos **acoplando** essas duas classes: toda vez que precisarmos alterar a forma de trabalhar da `EstanteNoBancoDeDados`, temos de mudar também a `LivrosController`.

No capítulo anterior, usamos `UmaEstanteQualquer` justamente porque não faz diferença para o `LivrosController`

qual é a implementação da `Estante`, ele só precisa de uma `Estante` pronta para ser usada. Poderíamos **diminuir o acoplamento**, fazendo o `LivrosController` depender apenas do estritamente necessário para ela trabalhar: a **interface** `Estante`.

Alguém precisará criar uma `Estante` e passar para o `LivrosController`, assim o controller pode se concentrar naquilo para o que foi feito: controlar as operações com os livros. Mudamos um cenário no qual o controller ia atrás de criar e gerenciar a vida da `Estante`, para um em que a estante é criada por outra classe e simplesmente passada para o `LivrosController` usar.

Recapitulando, saímos de um controller que estava assim:

```
@Controller
public class LivrosController {

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.include("mensagem", "Livro salvo com sucesso!");
    result.redirectTo(this).lista();
  }

  public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
  }

  public void edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();

    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);
```

```
    result.of(this).formulario();
  }
}
```

E ao trocarmos para a `EstanteNoBancoDeDados`, ficou assim:

```
@Controller
public class LivrosController {

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    Estante estante = null;
    try {
      estante = new EstanteNoBancoDeDados(BDs.MySQL,
          "usuario", "senha", "db_livraria");
      estante.guarda(livro);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }

    result.redirectTo(this).lista();

  }

  public List<Livro> lista() {
    Estante estante = null;
    try {
      estante = new EstanteNoBancoDeDados(BDs.MySQL,
          "usuario", "senha", "db_livraria");
      return estante.todosOsLivros();
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
  }

  public void edita(String isbn, Result result) {
    Estante estante = null;
    try {
      estante = new EstanteNoBancoDeDados(BDs.MySQL,
          "usuario", "senha", "db_livraria");

      Livro livroEncontrado = estante.buscaPorIsbn(isbn);
      result.include(livroEncontrado);
```

```
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
    result.of(this).formulario();
  }
}
```

E para remover a duplicação da criação da estante em todos os métodos, vamos mover isso para o construtor, transformando a `estante` em um atributo da classe:

```
@Controller
public class LivrosController {

  private Estante estante;

  public LivrosController() {
    estante = new EstanteNoBancoDeDados(BDs.MySQL,
        "usuario", "senha", "db_livraria");
  }

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    try {
      estante.guarda(livro);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }

    result.redirectTo(this).lista();

  }

  public List<Livro> lista() {
    try {
      return estante.todosOsLivros();
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
  }

  public void edita(String isbn, Result result) {
```

```
    try {
      Livro livroEncontrado = estante.buscaPorIsbn(isbn);
      result.include(livroEncontrado);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
    result.of(this).formulario();
  }
}
```

Mas não queremos que o `LivrosController` se preocupe com a criação da `Estante`, logo, vamos passar a recebê-la pelo construtor:

```
@Controller
public class LivrosController {

  private Estante estante;

  public LivrosController(Estante estante) {
    this.estante = estante;
  }
  // ...
}
```

E como agora o `LivrosController` não sabe mais qual é a implementação de `Estante` recebida, ele não manda mais a estante fechar a conexão. O código fica:

```
@Controller
public class LivrosController {

  private Estante estante;

  public LivrosController(Estante estante) {
    this.estante = estante;
  }

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    estante.guarda(livro);
```

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

  public List<Livro> lista() {
    return estante.todosOsLivros();
  }

  public void edita(String isbn, Result result) {
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);

    result.of(this).formulario();
  }
}
```

Agora o código do controller só tem o que é necessário. Seu código fica bem mais simples e fácil de entender e manter.

Por outro lado, quem criava uma instância de `LivrosController`, antes só fazia um `new`, como em:

```
LivrosController controller = new LivrosController();

controller.lista();
```

Agora, ele precisa também escolher uma implementação de `Estante`, gerenciar essa implementação, e passar para o controller, antes de poder usá-lo:

```
EstanteNoBancoDeDados estante = null;
try {
  estante = new EstanteNoBancoDeDados(BDs.MySQL,
        "usuario", "senha", "db_livraria");

  LivrosController controller = new LivrosController(estante);

  controller.lista();
} finally {
  estante.fechaConexao();
}
```

Removemos toda essa complexidade do controller, que agora está bem mais fácil de se trabalhar. Entretanto, transferimos essa complexidade para outra classe, que terá de fazer o trabalho sujo.

4.2 INVERSÃO DE CONTROLE: INJEÇÃO DE DEPENDÊNCIAS

A prática que vimos anteriormente é chamada de **inversão de controle**. Ela consiste em extrair trechos mais voltados à infraestrutura das camadas mais externas da aplicação — por exemplo, nossos controllers. Esses trechos são isolados em classes especializadas, que interagem em uma camada mais interna, nas quais têm condições de centralizar o gerenciamento das complexidades que antes ficavam espalhadas por todos o sistema.

Uma das técnicas para inverter o controle é justamente a que usamos, e ela se chama **injeção de dependências**. Nela evitamos que a classe controle a criação e o gerenciamento das suas dependências. Em vez disso, declaramos quais são os componentes necessários para o funcionamento de cada classe e confiamos que as instâncias funcionais dos componentes serão injetadas antes que ela seja usada.

Assim, a responsabilidade de criar e gerenciar os componentes do sistema vai sendo empurrada para camadas inferiores até que a centralizamos em um componente especializado que coordenará a injeção das dependências nos lugares certos. Esse componente especializado é chamado de **container** ou **provedor** (*provider*) de dependências.

Dessa forma, cada componente declara quais são as suas

dependências, se possível como *interfaces* para ficarmos livres para usar qualquer implementação. E registramos esse componentecomo uma implementação disponível para a injeção no container.

Ao tentar instanciar o componente, o container buscará cada dependência e, se necessário, criará uma nova instância dessa dependência, que também tem suas próprias dependências. Esse processo segue até que todas sejam resolvidas e, então, o componente requisitado estará criado e pronto para ser usado.

Uma das filosofias do VRaptor é facilitar e incentivar as melhores práticas de desenvolvimento de software. E para incentivar a **injeção de dependências**, o próprio VRaptor é construído em cima de um container de injeção de dependências: o CDI, que é a especificação do Java responsável pela injeção de dependências. Todos os componentes do sistema que são gerenciados pelo VRaptor podem usar essa técnica e deixar o CDI cuidar da resolução das dependências. Para ver como isso funciona, vamos voltar ao controller.

```
@Controller
public class LivrosController {

    private final Estante estante;

    public LivrosController(Estante estante) {
        this.estante = estante;
    }

    // resto do código, substituindo as criações de
    // estantes por this.estante
}
```

Por padrão no CDI, todas as classes são candidatas à injeção de dependências. Porém, para habilitá-la, precisamos indicar os

pontos de injeção com a anotação `@Inject`. No caso do `LivrosController`, queremos receber injeção no construtor:

```
@Controller
public class LivrosController {
    @Inject
    public LivrosController(Estante estante) {
        this.estante = estante;
    }
    //...
}
```

Ao recebermos uma `Estante` no construtor, estamos declarando que o `LivrosController` não pode ser criado sem antes receber uma `Estante` preenchida e funcionando. Isto é, declaramos que `Estante` é uma dependência.

Como o controller é uma classe gerenciada pelo VRaptor (por causa do `@Controller`), o VRaptor tentará buscar uma implementação de `Estante` candidata a ser injetada, por meio do CDI. Como `Estante` é uma interface, o CDI procurará uma implementação dela que possa ser usada. No momento, temos apenas a implementação `UmaEstanteQualquer`, então será essa a que usaremos.

Construtor sem argumentos

O CDI obriga que todas as classes candidatas à injeção de dependências possuam um construtor sem argumentos, para que ele possa adicionar alguns comportamentos que fazem parte da especificação. Então, se adicionarmos um construtor que recebe as dependências, precisamos também adicionar um construtor sem argumentos para que o CDI possa fazer o seu trabalho:

```
@Controller
public class LivrosController {
    @Inject
    public LivrosController(Estante estante) {
        this.estante = estante;
    }
    /**
     * @deprecated Apenas para o CDI. Não precisa ser públic
o
     */
    LivrosController() {}

    //...
}
```

Alternativamente, podemos remover todos os construtores da classe (o construtor sem argumentos é adicionado automaticamente nesse caso pelo próprio compilador do Java), e usar a injeção por atributos, usando o `@Inject` para indicar isso:

```
@Controller
public class LivrosController {
    @Inject
    private Estante estante;

    // nenhum construtor
}
```

Essa alternativa funciona, mas dificulta a testabilidade dessa classe. Como testes automatizados são uma parte importante do desenvolvimento de software, prefira a injeção por construtor, mesmo com o construtor extra.

Ao criarmos uma segunda implementação para a mesma interface, como a `EstanteNoBancoDeDados` , o CDI não saberá qual implementação usar:

```
Ambiguous dependencies for type Estante with qualifiers @Default
  at injection point [BackedAnnotatedParameter]
    Parameter 1 of [BackedAnnotatedConstructor]
    @Inject public b.c.c.livraria.controlador
                                .LivrosController(Estante)
  at b.c.c.livraria.controlador.LivrosController
                              .<init>(LivrosController.java:0)
  Possible dependencies:
  - Managed Bean
      [class b.c.c.livraria.persistencia.UmaEstanteQualquer]
      with qualifiers [@Any @Default],
  - Managed Bean
      [class b.c.c.livraria.persistencia.EstanteNoBancoDeDados]
      with qualifiers [@Any @Default]
```

Traduzindo: dependências ambíguas para o tipo `Estante`, no primeiro parâmetro do construtor do `LivrosController`. Dependências possíveis: `UmaEstanteQualquer` e `EstanteNoBancoDeDados`.

Em geral, vamos ter apenas uma implementação para cada interface, então não precisamos nos preocupar com esse tipo de erro. Mas caso tenhamos criado uma implementação apenas para teste, que não será usada para código de produção, precisamos anotá-la com `@Alternative`. No nosso caso, precisamos anotar a classe `UmaEstanteQualquer`, ou simplesmente apagá-la (temos os sistemas de controle de versão, como o Git ou o SVN, para restaurá-la caso você mude de ideia).

```
@Alternative
public class UmaEstanteQualquer implements Estante {...}
```

Agora que sabemos que a nossa implementação real de `Estante` usará um banco de dados, precisamos definir como será o acesso a esses dados. Em Java, existem várias maneiras de fazer isso, como usando *JDBC*, que é a especificação de como nos conectamos ao banco de dados e executamos SQL; ou *Hibernate*,

que mapeia os dados de classes Java para tabelas; ou ainda a *JPA*, que é a especificação para fazer tal mapeamento.

Como a forma de acessar o banco de dados é muito diferente, e às vezes muito extensa, em cada uma dessas possibilidades, não queremos deixar esse tipo de código espalhado pelo sistema. Então, encapsulamos esse **acesso** aos **dados** em um **objeto** que nos dará uma interface independente da ferramenta que usamos para isso. O nome que damos a esse tipo de objeto que acessa dados é **DAO** (*Data Access Object*).

A `EstanteNoBancoDeDados` precisará acessar os dados de livros no banco, portanto, ela depende de um `DAO` de livros — um `LivroDAO`. Para declarar essa dependência, fazemos como no controller, recebendo o DAO no construtor:

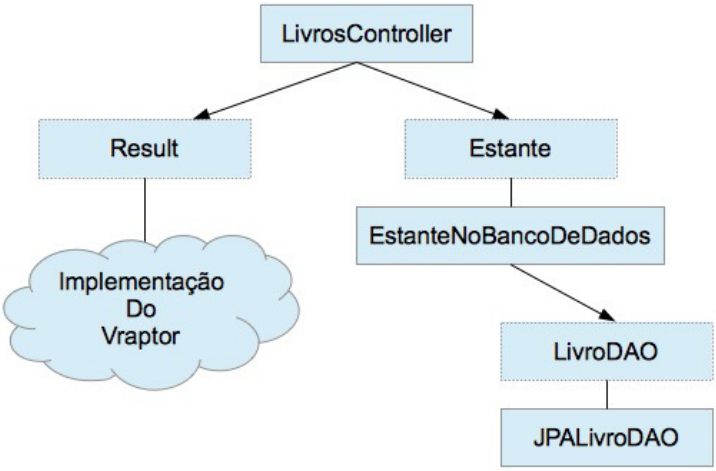
```
public EstanteNoBancoDeDados implements Estante {

    private final LivroDAO dao;

    @Inject
    public EstanteNoBancoDeDados(LivroDAO dao) {
        this.dao = dao;
    }

    /**
     * @deprecated para o CDI
     */
    EstanteNoBancoDeDados() { this(null); }
    //...
}
```

Ao anotar o construtor com `@Inject`, o CDI vai procurar por uma implementação de `LivroDAO` na aplicação. Para que a `EstanteNoBancoDeDados` não fique acoplada à maneira como o `LivroDAO` é implementado, usaremos uma interface. Como estamos usando um servidor de aplicação, o `Wildfly`, uma

escolha natural é usar a `JPA` para fazer o acesso ao banco de dados.

```
public class JPALivroDAO implements LivroDAO {...}
```

Por usar a JPA, esse DAO precisa de um `EntityManager` para que tudo funcione, logo, vamos recebê-lo no construtor. E continuamos indicando os componentes gerenciados até que todas as dependências sejam satisfeitas. Desse modo, o **Controller** é criado e podemos usá-lo para atender uma requisição para, por exemplo, mostrar a página da lista de todos os produtos.



Em um primeiro momento, pode parecer muito trabalhoso declarar todos esses componentes, mas, uma vez criado, um mesmo componente pode ser usado em todas as classes que necessitam dele sem nenhum código adicional, apenas recebendo-o no construtor, ou no atributo anotado com `@Inject` .

4.3 IMPLEMENTANDO A ESTANTE

Agora que sabemos como o VRaptor e o CDI gerenciam as dependências, vamos começar a criar as implementações reais dos nossos componentes. A classe `LivrosController` precisa de uma `Estante` com as seguintes operações:

```
public interface Estante {

    void guarda(Livro livro);
    List<Livro> todosOsLivros();

    Livro buscaPorIsbn(String isbn);
}
```

Como a nossa implementação guardará os livros no banco de dados, criaremos a classe `EstanteNoBancoDeDados`:

```
public class EstanteNoBancoDeDados implements Estante {
    //...
}
```

Ela acessa os dados do livro no banco de dados, logo, podemos usar uma classe que tem essa responsabilidade ser um `DAO` de livros:

```
public class EstanteNoBancoDeDados implements Estante {

    private final LivroDAO dao;

    @Inject
    public EstanteNoBancoDeDados(LivroDAO dao) {
        this.dao = dao;
    }

    /**
     * @deprecated para o CDI
     */
    EstanteNoBancoDeDados() { this(null); }
```

```
    @Override
    public void guarda(Livro livro) {
        this.dao.adiciona(livro);
    }

    @Override
    public List<Livro> todosOsLivros() {
        return this.dao.todos();
    }

    @Override
    public Livro buscaPorIsbn(String isbn) {
        return this.dao.buscaPorIsbn(isbn);
    }
}
```

DIMINUINDO O ACOPLAMENTO

Repare que o código do `LivrosController` não precisa ser mudado agora que adicionamos a implementação de `Estante`. Podemos criar essa implementação usando qualquer tecnologia, com qualquer classe do nosso sistema e pronto! Nenhuma das classes que dependem de `Estante` precisará ser modificada.

Conseguimos isso porque o acoplamento do controller é apenas com a **interface** `Estante`, ou seja, é apenas com **o que** a `Estante` consegue fazer, e não em **como** é feita a implementação. Por esse motivo, prefira receber interfaces como dependência das suas classes, principalmente quando a implementação depende de uma tecnologia ou biblioteca externa, como FTP ou JPA.

Para essa classe funcionar, precisamos de um `LivroDAO` com

a seguinte interface:

```
public interface LivroDAO {
    void adiciona(Livro livro);
    List<Livro> todos();
    Livro buscaPorIsbn(String isbn);
}
```

No nosso sistema, acessaremos o banco de dados com a ajuda da JPA, a especificação do Java para persistência de objetos. Esses objetos persistidos no banco recebem o nome de **entidade**. Para indicar que um `Livro` tem esse papel, anotamos a classe com `@Entity`.

Como estamos lidando com bancos de dados, ao salvar uma entidade, precisamos de um valor que a identifique de forma única, de modo que podemos recuperar o valor salvo de forma fácil. Por isso é obrigatório que um dos campos da entidade esteja anotado com `@Id`, indicando qual é o seu identificador.

No caso do livro, temos o ISBN, que já é um identificador, então poderíamos usá-lo como `@Id`. Mas para conseguirmos distinguir facilmente um livro que já está salvo no banco de um que está sendo criado, vamos criar um atributo id numérico e automaticamente gerado pelo banco, para atuar como identificador. A classe `Livro` ficaria assim:

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

    @Column(unique=true)
    private String isbn;

    private String titulo;
    private String descricao;
```

```
    private BigDecimal preco;
    private Calendar dataPublicacao;

    //getters/setters
}
```

Colocando o id no formulário

Para que o formulário continue funcionando para a edição, precisamos acrescentar um campo que represente o `id`. Como esse campo vai ser automaticamente gerado, não queremos que o usuário altere-o. Por esse motivo, vamos acrescentar um `input` do tipo `hidden` no formulário para guardá-lo:

```
<form ....>
   <input type="hidden" name="livro.id" value="${livro.id}"
/>
   ...
</form>
```

Como o `Livro` é uma entidade, a `JPA` disponibiliza o gerenciador de entidades para realizar as operações de persistência. Esse gerenciador se chama `EntityManager`, e será necessário para que o `LivroDAO`, baseado na JPA, realize o seu trabalho. Ou seja, uma dependência que também será recebida no construtor.

```
public class JPALivroDAO implements LivroDAO {

  private final EntityManager manager;

  @Inject
  public JPALivroDAO(EntityManager manager) {
    this.manager = manager;
  }
```

```
  /**
   * @deprecated para o CDI
   */
  JPALivroDAO() { this(null); }

  @Override
  public void adiciona(Livro livro) {
    if (livro.getId() == null) {
      this.manager.persist(livro);
    } else {
      this.manager.merge(livro);
    }
  }

  @Override
  public List<Livro> todos() {
    return this.manager
            .createQuery("select l from Livro l", Livro.class)
            .getResultList();
  }

  @Override
  public Livro buscaPorIsbn(String isbn) {
    try {
      return this.manager
        .createQuery("select l from Livro l
                      where l.isbn = :isbn", Livro.class)
        .setParameter("isbn", isbn)
        .getSingleResult();
    } catch (NoResultException e) {
      return null;
    }
  }
}
```

Agora nosso `JPALivroDAO` funciona, desde que lhe seja passado um `EntityManager` pronto para ser usado. Mas ainda não registramos nenhum componente que seja um `EntityManager`.

Se tentarmos rodar o sistema do jeito que está, o CDI procurará por um `EntityManager`, mas não vai encontrar. Nesse

caso, acontecerá um erro na aplicação falando que o DAO precisa de um `EntityManager`, porém o CDI não sabe como criá-lo.

Mas se o `EntityManager` é um componente da JPA, como devemos registrá-lo no CDI, sendo que não somos nós que vamos implementá-lo?

```
Caused by: org.jboss.weld.exceptions.DeploymentException:
  WELD-001408:
  Unsatisfied dependencies for type EntityManager with
  qualifiers @Default at injection point
  [BackedAnnotatedParameter]
    Parameter 1 of [BackedAnnotatedConstructor]
      @Inject public b.c.casadocodigo.livraria.persistencia
                                        .JPALivroDAO(EntityManager)
  at br.com.casadocodigo.livraria.persistencia
                        .JPALivroDAO.<init>(JPALivroDAO.java:0)
```

4.4 CRIANDO OBJETOS COMPLICADOS — @PRODUCES

O nosso `LivroDAO` necessita de um `EntityManager` para conseguir acessar o banco de dados e, seguindo a injeção de dependências, recebemos um objeto dessa classe no construtor. Precisamos agora de uma implementação de `EntityManager`, para que o CDI consiga injetar essa dependência.

Mas essa implementação será feita pela nossa aplicação? Não! Queremos que o próprio servidor gerencie essa implementação, já que ela faz parte de uma das suas especificações: a JPA.

No caso em que a implementação da nossa dependência não é uma classe da nossa aplicação, o CDI não consegue saber magicamente como disponibilizá-la para as outras classes — precisamos ensiná-lo a fazer isso. Além disso, criar um

`EntityManager` não é simplesmente dar um `new` em alguma classe determinada. O código para isso é algo parecido com:

```
EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("default");
//...

EntityManager manager = factory.createEntityManager();
```

Para instruir o CDI a executar esse código toda vez que precisar de um `EntityManager`, precisamos colocá-lo dentro de um método anotado com `@Produces`, que retorna o `EntityManager`. Ou seja, é o método que vai produzir essa dependência, e será invocado sempre que necessário. Esse método pode estar em qualquer classe da aplicação, mas para mantê-lo organizado, vamos criar uma classe especializada em fazer isso:

```
public class FabricaDaJPA {
  @Produces
  public EntityManager criaEntityManager() {
    EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("default");
    //...

    return factory.createEntityManager();
  }
}
```

Recapitulando, toda vez que pedimos a injeção de uma dependência com o `@Inject`, o CDI vai procurar por alguma classe que sirva de implementação dessa dependência. Além disso, ele procurará por algum método que esteja anotado com `@Produces` e retorne uma implementação dessa dependência, como é o nosso caso com o `EntityManager`.

Quando o CDI precisar de um `EntityManager`, ele vai instanciar a `FabricaDaJPA` e executar o método

`criaEntityManager` . Mas em que ocasiões ele passará por esse processo?

4.5 TEMPO DE VIDA DOS COMPONENTES — ESCOPO

Quando estamos em um ambiente de injeção de dependências, temos de indicar como os componentes serão criados, para que o nosso *container* possa fazer isso quando necessário. Entretanto, não é só **como** que importa, precisamos saber também **quando** os componentes serão **criados** e quando eles não serão mais necessários, ou seja, o momento de serem **destruídos**.

Isso é o que chamamos de **escopo** do componente, o tempo em que ele será usado no sistema e após o qual sua instância poderá ser jogada fora.

Durante um escopo, todas as classes que dependerem de um componente receberão a mesma instância deste componente. Um escopo bastante simples é o seguinte: toda vez que um objeto precisar de uma instância do meu componente, eu crio uma nova e entrego para essa classe. Quando esse objeto morrer, a instância do componente morre junto, ou seja, sua vida depende da vida do objeto que depende dessa instância. O CDI chama esse escopo de **dependente**, e é o escopo padrão, pois é a situação normal no Java quando nós mesmos criamos objetos e os passamos para outras classes.

Em um ambiente web, toda interação com o usuário, como cliques em links ou submissões de formulários, se dá através de **requisições**. Logo, um escopo natural é o de **requisição**: todo

objeto que precisar ser criado durante uma requisição será destruído ao final dela, ou seja, após a resposta ser devolvida para o usuário.

O CDI possui cinco escopos implementados, todos eles definidos por anotações do pacote `javax.enterprise.context` :

- `@Dependent` : escopo dependente. Uma instância será criada para cada objeto que precisar dessa dependência, isto é, não será compartilhada. O tempo de vida dessa instância depende da vida do objeto que precisa dela, ou seja, ela é jogada fora junto com o objeto que a recebeu como dependência. É o escopo padrão, mas você pode usar esta anotação caso queira deixar o escopo explícito.

- `@RequestScoped` : escopo de requisição. Cada requisição terá uma instância diferente desse componente, que será compartilhada entre todos os objetos que dependerem desse componente. Ao final da requisição, o objeto é destruído.

- `@SessionScoped` : escopo de sessão. A instância será criada por usuário do sistema, atrelada à sua sessão HTTP (`HttpSession`). Ao final da sessão, seja por inatividade ou por chamada explícita, o objeto será destruído. Os objetos nesse escopo precisam implementar `Serializable` , para que o container consiga passivá-lo (guardar no disco) e reativá-lo (ler do disco).

- `@ConversationScoped` : escopo de conversação. É um escopo intermediário entre o de requisição e o de sessão. Você pode definir o início e o fim do escopo

explicitamente, recebendo como dependência uma `Conversation` e chamando os métodos `begin` e `end`.

- `@ApplicationScoped`: escopo de aplicação. Apenas uma instância do componente será criada na aplicação. Também chamado de *singleton*, em alusão ao *Design Pattern* com o mesmo nome.

Durante este livro, você aprenderá situações em que cada um dos escopos se encaixam melhor e quando usar cada um deles.

Para atribuir escopos aos componentes da aplicação, precisamos anotar a sua implementação:

```
@ApplicationScoped
public class ColetorDeEstatisticasNoMemcached
  implements ColetorDeEstatisticas {...}
```

Existe uma particularidade quando estamos trabalhando com métodos anotados com `@Produces`, pois a classe e o resultado do método podem ter escopos diferentes:

```
@ApplicationScoped
public class GeradorDeIds {
  private Long ultimoIdGerado = 0;

  @Produces
  @Dependent
  public Id novoId() {
    return new Id(ultimoIdGerado++);
  }
}
```

Ou seja, só existirá uma instância do `GeradorDeIds` durante toda a execução da aplicação. Mas a cada vez que um componente precisar de um `Id`, o método `novoId` será executado, e cada componente terá um id diferente.

No caso da nossa `FabricaDaJPA`, estamos usando o seguinte código:

```
public class FabricaDaJPA {
  @Produces
  public EntityManager criaEntityManager() {
    EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("default");
    //...

    return factory.createEntityManager();
  }
}
```

Como não especificamos o escopo, vale o padrão: toda vez que alguém pedir a injeção de um `EntityManager`, será criada uma instância da `FabricaDaJPA` e o método `criaEntityManager` será invocado. Só que cada invocação também cria uma `EntityManagerFactory`, que é um objeto de criação cara, que deveria ser único na aplicação inteira, isto é, seu escopo deveria ser o de aplicação.

Para aproveitar a mesma `factory` durante toda a execução da aplicação, precisamos criá-la de forma separada:

```
public class FabricaDaJPA {

  @Produces
  @ApplicationScoped
  public EntityManager criaEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("default");
  }

  @Produces
  public EntityManager criaEntityManager() {
    return factory.createEntityManager();
  }
}
```

Mas, ao fazer isso, o método `criaEntityManager` não

compila, pois a variável `factory` não está definida. Precisamos passar essa `factory` de algum jeito para o método. Já vimos que é possível receber a injeção de dependências no construtor e direto no atributo. Porém, no caso de um método `@Produces`, também podemos receber dependências como argumento do método:

```
  @Produces
  public EntityManager
  criaEntityManager(EntityManagerFactory factory) {
    return factory.createEntityManager();
  }
```

Nesse caso, não é necessário o `@Inject`, pois o CDI já sabe que, se tiver de executar esse método, precisa passar os parâmetros. Podemos receber qualquer dependência da aplicação, como se estivéssemos recebendo no construtor.

```
public class FabricaDaJPA {

  @Produces
  @ApplicationScoped
  public EntityManager criaEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("default");
  }

  @Produces
  public EntityManager
  criaEntityManager(EntityManagerFactory factory) {
    return factory.createEntityManager();
  }
}
```

4.6 CALLBACKS DE CICLO DE VIDA

Criamos a `FabricaDaJPA` que cria `EntityManager`, mas existe um fato importante sobre eles: o `EntityManager` abre conexões com o banco de dados e elas **precisam** ser fechadas. Para

isso, precisamos chamar o seu método `close()` assim que acabarmos de usá-lo, para evitar o vazamento dessas conexões.

Mas **quem** vai chamar esse método? Alguma classe que depende do `EntityManager`? Não podemos fazer desse modo, pois várias outras classes podem estar usando esse mesmo `manager`.

A regra de ouro para recursos que precisam ser fechados/liberados, como conexões, é a seguinte: se uma classe abriu/adquiriu esse recurso, ela é a responsável por fechá-lo/liberá-lo. Por esse motivo, se a `FabricaDaJPA` criou o `EntityManager`, ela deveria fechá-lo. Vamos criar um método para isso:

```
public class FabricaDaJPA {

    public void fechaManager() {
        manager.close();
    }
}
```

O método está criado, mas **quando** ele deve ser chamado? Ele deve ser chamado quando acabarmos de usar o `manager`, ou seja, quando a instância do manager puder ser jogada fora. Conseguimos fazer isso com o callback `@Disposes` do CDI.

Se criarmos um método que recebe uma instância anotada com esse callback em qualquer classe, ele será invocado logo antes de a instância ser jogada fora, possibilitando a liberação dos recursos abertos, por exemplo:

```
public class FabricaDaJPA {
    //...
    public void fechaManager(@Disposes EntityManager manager) {
        manager.close();
    }
}
```

Podemos fazer a mesma coisa com o `EntityManagerFactory`, que também precisa ser fechado:

```
public class FabricaDaJPA {
    //...
    public void fechaManager(@Disposes EntityManager manager) {
        manager.close();
    }

    public void
    fechaFactory(@Disposes EntityManagerFactory factory) {
        factory.close();
    }
}
```

Como o `EntityManagerFactory` é `@ApplicationScoped`, será fechado apenas uma vez, quando o servidor estiver sendo parado. Já o `EntityManager` será fechado várias vezes, sempre que alguém terminar de usá-lo.

Usamos o `@Disposes`, pois estamos lidando com um objeto do qual não temos controle. Entretanto, se estivermos dentro da nossa própria implementação, podemos usar outro *callback*, o `@PreDestroy`, que faz a mesma coisa, com a diferença de que se refere à mesma instância em que o método é executado:

```
public class ClienteHTTP {
  private List<Conexao> conexoes;

  @PreDestroy
  public void fecha() {
    for (Conexao conexao : conexoes) {
      conexao.fecha();
    }
  }
}
```

4.7 JPA DENTRO DE UM SERVIDOR DE

APLICAÇÃO

Se estivermos dentro de um servidor de aplicação, como é o nosso caso com o Wildfly, por exemplo, podemos usar o `EntityManager` e o `EntityManagerFactory`, que são gerenciados pelo próprio servidor, usando as anotações `PersistenceContext` e `PersistenceUnit`, respectivamente:

```
@PersistenceContext
private EntityManager manager;

@PersistenceUnit
private EntityManagerFactory factory;
```

Uma limitação é que isso só pode ser feito em atributos. Mas se quisermos usar injeção por construtor como no resto do sistema, podemos usar uma versão mais simples da `FabricaDaJPA`:

```
public class FabricaDaJPA {
  @PersistenceContext
  private EntityManager manager;

  @Produces
  public EntityManager getManager() {
    return manager;
  }
}
```

Não precisamos nos preocupar com o `@Disposes`, pois o servidor vai tomar conta de tudo. Mas para isso funcionar, precisamos que o código esteja em um contexto transacional, requerido por qualquer operação no banco de dados. Fazemos isso com a anotação `@Transactional`, em cima de cada método ou classe que vai fazer operações no banco de dados.

No nosso caso, o `JPALivroDAO` vai usar o banco em todos os métodos, então anotamos a classe:

```
@Transactional
public class JPALivroDAO implements LivroDAO {...}
```

Se quisermos fazer uma operação maior, por exemplo em um controller, que envolva várias escritas no banco de dados, podemos colocar todas elas dentro da mesma transação usando a anotação no método:

```
@Controller
public class FilialController {
   private EstoqueDao estoqueDao;

   @Transactional
   public void transfereLivro(Livro livro, int quantidade,
                              Filial origem, Filial destino) {

      estoqueDao.removeDoEstoque(livro, quantidade, origem);
      estoqueDao.adicionaAoEstoque(livro, quantidade, destino);

   }
}
```

Pronto, agora temos o nosso criador de `EntityManager` implementado e, como ele era o componente que faltava para o `JPALivroDAO` funcionar, conseguimos subir o servidor e começar cadastrar os livros diretamente no banco de dados.

Fora do servidor de aplicação

No caso de rodar a aplicação em um servidor web, como o Tomcat ou o Jetty, precisamos usar a versão completa da `FabricaDaJPA`, que cria e gerencia o `EntityManager`. Além disso, o `@Transactional` não vai funcionar, portanto, você precisa controlar as transações manualmente, por exemplo, abrindo e fechando a transação em cada método do DAO:

```
@Override
public void adiciona(Livro livro) {
  this.manager.getTransaction().begin();
  this.manager.persist(livro);
  this.manager.getTransaction().commit();
}
```

Existe uma solução melhor para isso, que veremos no capítulo *Cuidando da infraestrutura do sistema: interceptors*.

4.8 OUTROS TIPOS DE INJEÇÃO DE DEPENDÊNCIA E @POSTCONSTRUCT

O VRaptor incentiva a injeção de dependências via construtor, pois facilita a testabilidade e melhora a semântica: não é possível criar o objeto sem passar todos os argumentos do construtor, logo tudo que está no construtor já é uma dependência da classe. Mas existem outras maneiras de fazer injeção de dependências, isto é, outros pontos de injeção nos quais podemos colocar o `@Inject`:

- **Setter:** bem comum no Spring, você cria um setter para a

dependência e indica que ele precisa ser invocado.

```
@Inject
public void setDao(LivroDAO dao) {
   this.dao = dao;
}
```

- **Atributo:** usado nos EJBs, a dependência é injetada diretamente no atributo, via reflection.

```
@Inject
private Estante estante;
```

- **Por método de inicialização:** um método que recebe de uma vez todas as dependências da classe.

```
@Inject
public void init(Estante estante, LivroDAO dao) {...}
```

Em todos esses outros tipos, se quisermos executar algum código na inicialização, não podemos usar oconstrutor: as dependências ainda não foram preenchidas. Nesse caso, podemos ter um método anotado com o callback `@PostConstruct`. Este será chamado logo após todas as dependências serem preenchidas, quando o objeto está pronto para ser usado.

```
@Inject
public void setDependencia(Dependencia dependencia) {...}

@PostConstruct
public void inicializa() {
    logger.info("Classe inicializada");
    dependencia.fazAlgo();
}
```

CAPÍTULO 5

TOMANDO O CONTROLE DOS RESULTADOS

Temos agora no nosso sistema o cadastro de livros totalmente funcional, com todas as páginas e os livros sendo salvos e recuperados do banco de dados. Nesse cadastro, vimos que o VRaptor tem a convenção de retornar para uma JSP com o mesmo nome do método executado, na pasta com o nome do controller dentro de `/WEB-INF/jsp`.

Essa é uma convenção muito interessante no caso geral em que o resultado da requisição é uma página HTML, mas nem sempre é isso que queremos. E para esses cenários, vamos precisar sobrescrever essa convenção.

5.1 REDIRECIONANDO PARA OUTRO MÉTODO DO MESMO CONTROLLER

Um dos resultados possíveis que já vimos é reutilizar a página de outro método, como fizemos no método `edita`:

```
public void edita(String isbn, Result result) {
  //...
  result.of(this).formulario();
}
```

Nessa linha, estamos dizendo para o VRaptor usar o resultado do (`result.of`) método `formulário` desse mesmo objeto (`(this).formulario()`). Consequentemente, o JSP usado será o `/WEB-INF/jsp/livro/formulario.jsp` .

Outra mudança da convenção que vimos foi no método `salva` que, ao final da requisição, redireciona para a listagem:

```
public void salva(Livro livro, Result result) {
  //...

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

Ou seja, o resultado será um redirecionamento para (`result.redirectTo`) o método `lista` desse mesmo controller (`(this).lista()`).

Essa classe `Result` é o componente do VRaptor responsável pela personalização do resultado final da execução do método do controller. Além de receber no método, podemos recebê-lo no construtor da classe, principalmente se formos usar o `Result` na maioria dos métodos:

```
@Controller
public class LivrosController {
  public Estante estante;
  public Result result;

  public LivrosController(Estante estante, Result result) {
    this.estante = estante;
    this.result = result;
  }
  //...
}
```

Através disso, o próprio VRaptor se encarregará de instanciar e disponibilizar o objeto `Result` para nós. Na realidade, o `Result`

é um componente, como vimos no capítulo anterior, mas que já vem implementado dentro do próprio VRaptor.

5.2 DISPONIBILIZANDO VÁRIOS OBJETOS PARA AS JSPS

Outra convenção que vimos é sobre o retorno do método. Por exemplo, no método `lista`:

```
public List<Livro> lista() {
  return this.estante.todosOsLivros();
}
```

Essa lista de livros será disponibilizada em um atributo da requisição chamado `livroList`, acessível na JSP por `${livroList}`. Mas e se, além dessa lista de livros, quisermos acrescentar também o livro em promoção do dia?

Não podemos simplesmente retornar dois objetos, pois isso não é válido em Java, então precisamos novamente do `Result` para alterar essa convenção, com o método `include`. Com ele, podemos adicionar quantos objetos forem necessários, e dar nomes diferentes da convenção para eles:

```
public List<Livro> lista() {
  this.result.include("promocao", this.estante.promocaoDoDia());
  return this.estante.todosOsLivros();
}
```

Para não misturar as convenções, podemos usar o `result` para a lista de livros também:

```
public void lista() {
  this.result.include("promocao", this.estante.promocaoDoDia());
  this.result.include("livros", this.estante.todosOsLivros());
}
```

Dessa forma, a lista de livros não ficará mais acessível por `${livroList}`, mas apenas por `${livros}`. Uma sobrecarga desse método não recebe como parâmetro o nome, como foi usado no método `edita` do controller:

```
public void edita(String isbn) {
  Livro livroEncontrado = this.estante.buscaPorIsbn(isbn);
  result.include(livroEncontrado);

  result.of(this).formulario();
}
```

Nesse caso, o VRaptor usará a convenção para nomes: nome da classe com a primeira letra minúscula. Ou seja, o atributo se chamará `livro`. Essa convenção não funcionará, no entanto, se tentarmos usar uma lista:

```
List<Livro> livros = this.estante.todosOsLivros();
result.include(livros);
```

Não é possível descobrir o tipo genérico da variável `livros`, pois o Java o apaga em tempo de execução. Logo, a variável aqui seria apenas `list` em vez de `livroList`. Para dar um nome melhor, devemos usar a versão do método `include` que recebe uma string:

```
List<Livro> livros = this.estante.todosOsLivros();
result.include("livros", livros);
```

5.3 MAIS SOBRE REDIRECIONAMENTOS

Vimos como redirecionar para outros métodos do mesmo controller, mas isso não resolve todos os nossos problemas. Por exemplo, se estivermos no `LivrosController`, invocando um método chamado `comprar` que, ao final da compra, redireciona

para a página inicial do sistema. Para isso, podemos usar o redirecionamento que recebe a classe do controller em vez de `this`:

```
public void comprar(Livro livro) {
  //todo o processo de comprar o livro
  result.redirectTo(HomeController.class).paginaInicial();
}
```

Isso funciona para todos os três tipos de redirecionamento que o VRaptor suporta e que podem ser usados ao final do método do controller. Esses redirecionamentos são:

- `result.of(this).metodo()` ou `result.of(UmController.class).metodo()`: a página `jsp` do método indicado será renderizada, sem executar o método. Útil quando queremos compartilhar a mesma JSP entre dois ou mais métodos.
- `result.forwardTo(this).metodo()` ou `result.forwardTo(UmController.class).metodo()`: executa o método indicado até o final e usa o seu resultado. Esse redirecionamento é transparente para o usuário, pois a URL que permanece no browser é a do primeiro método chamado. É um redirecionamento do tipo *FORWARD*, que é executado do lado do servidor.
- `result.redirectTo(this).metodo()` ou `result.redirectTo(UmController.class).metodo`: redireciona para o método indicado do lado do cliente, ou seja, a requisição volta para o browser e a URL é trocada para a do novo método. É um redirecionamento do tipo *REDIRECT*, que é executado do lado do cliente.

Redirecionamento no cliente versus Redirecionamento no servidor

Os redirecionamentos do tipo *REDIRECT*, ou seja, do lado do cliente, voltam para o browser e executam uma nova requisição limpa. São bastante recomendados quando queremos usar como resultado de um método (por exemplo, o método `salva` do controller) outro método (por exemplo, o método `lista`).

Se o usuário recarregar a página, a requisição que vai direto para o método `lista` , não passará mais pelo método `salva` , assim o navegador não pedirá para ele submeter o formulário novamente.

Já os redirecionamentos do tipo *FORWARD*, isto é, do lado do servidor, executam o segundo método dentro da mesma requisição, reaproveitando todo o estado dessa requisição. Devem ser usados quando quisermos executar as duas lógicas na mesma requisição e quando, ao recarregar a página, quisermos executar as duas lógicas novamente.

5.4 OUTROS TIPOS DE RESULTADO

Acabamos de ver o uso básico do `Result` para as operações mais comuns do desenvolvimento de uma aplicação web. No entanto, existem outros tipos de mudanças de resultado que são bastante úteis em determinadas situações.

Por exemplo, se tomarmos o método `edita` do `LivrosController`:

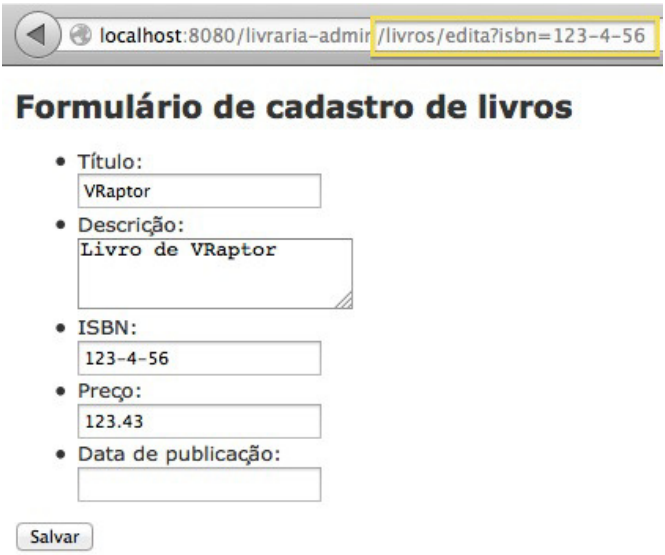
```
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  result.include(livroEncontrado);

  result.of(this).formulario();
}
```

O que fazer se for passado um `isbn` que não pertence a nenhum livro? Poderíamos criar uma página diferente para isso e redirecionar para ela:

```
public void naoEncontrado() {}

public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.forwardTo(this).naoEncontrado();
  } else {
    result.include(livroEncontrado);
    result.of(this).formulario();
  }
}
```

Mas, em aplicações web, já existe uma página padrão para redirecionarmos quando tentamos acessar algo que não existe: a página 404. Se quisermos redirecionar para essa página, podemos usar o método `notFound` do `Result`. Assim, será usada a página 404 configurada no sistema.

```
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.notFound();
  } else {
    result.include(livroEncontrado);
    result.of(this).formulario();
  }
}
```



Customizando as páginas de erro

É possível customizar as páginas de erro 404 ou 500 padrão do site, colocando no arquivo `web.xml` :

```
<error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
</error-page>
```

Desse modo, podemos mostrar uma página de 404 com a cara do sistema em vez da página padrão do Tomcat ou do servidor escolhido.

Além desse método, existe o `use` do `result` , que nos permite usar diversos tipos de resultado, da seguinte forma:

```
result.use(umTipoDeResultado).configuracaoDesseResultado();
```

Os tipos de resultado que já vêm implementados no VRaptor estão disponíveis através de métodos estáticos na classe `Results` . Eles são:

- `Results.http()` : possibilita a alteração das partes HTTP da resposta, como *status code*, *headers*, ou até mesmo retornar um corpo da requisição. Por exemplo:

  ```
  result.use(Results.http()).body("Deu tudo certo");
  ```

  Assim, em vez de ir para uma JSP, a resposta da requisição será o texto "Deu tudo certo".

- `Results.status()` : possibilita a alteração do *status code*

da requisição. Por exemplo:

```
result.use(Results.status())
      .forbidden("Você não está autorizado a acessar
                  esse conteúdo");
```

- `Results.json()` : retorna os dados de um objeto serializados em JSON. Por exemplo:

```
result.use(Results.json()).from(livro).serialize();
```

Nesse caso a resposta será algo como:

```
{ "livro": {
    "titulo": "VRaptor 3",
    "descricao": "Um livro legal sobre VRaptor",
    "isbn": "12345-6"
} }
```

- `Results.jsonp()` : semelhante ao resultado de JSON, mas com a opção de passar um callback de JSONP.

- `Results.xml()` : semelhante ao resultado de JSON, mas serializa o objeto em XML.
- `Results.representation()` : tenta decidir, de acordo com o que veio na requisição, se o objeto passado será serializado em XML, JSON ou se será redirecionado para a JSP. O uso é similar ao resultado de XML e de JSON.

- `Results.message()` : serializa uma mensagem no formato desejado, dada a chave da tradução da mensagem:

```
result.use(Results.message())
  .from("alerta", "usuario.nao.salvo", usuario.getNome())
  .as(Results.json());
```

Assim, a mensagem será serializada em JSON, com a categoria "alerta" e o texto com a chave

`usuario.nao.salvo` do arquivo `messages.properties`:

```
usuario.nao.salvo = O usuário {0} não foi salvo
```

- `Results.nothing()`: retorna uma resposta vazia. Existe um atalho para isso no próprio `result`:

```
result.use(Results.nothing());
//ou
result.nothing();
```

- `Results.referer()`: possibilita o redirecionamento de volta para a página que gerou a requisição. Esse resultado usa o *Header Referer* da requisição, que em geral é enviado pelos navegadores, com a URL da página atual. No entanto, esse header não é obrigatório e, se não estiver presente na requisição, esse resultado gerará um erro.

- `Results.page()`: agrupa os redirecionamentos para páginas, aqueles que não executam métodos de controllers. Por exemplo:

```
result.use(Results.page()).of(UmController.class).metodo(
);
// que é o mesmo que:
result.of(UmController.class).metodo();
```

- `Results.logic()`: agrupa os redirecionamentos para métodos de um controller. Por exemplo:

```
result.use(Results.logic()).redirectTo(UmController.class
)
                           .metodo();
// que é o mesmo que:
result.redirectTo(UmController.class).metodo();
```

Todos esses resultados possuem mais opções, você pode

explorá-las usando o *autocomplete* da sua IDE. No caso do Eclipse, você pode fazer, por exemplo:

```
result.use(Results.status()).<Ctrl + Espaço>
```

E você consegue ver a lista de métodos que esse resultado disponibiliza.

Capítulo 6

VALIDANDO O SEU DOMÍNIO

Nosso sistema possui um cadastro de livros com todos os dados editáveis no formulário implementado pelo `LivrosController`. No entanto, não podemos vender qualquer livro cadastrado nesse formulário. No momento, é possível cadastrar um livro sem título ou sem preço, que não pode ser colocado à venda no site.

Poderíamos ficar em todas as partes do sistema verificando se os dados do livro (ou qualquer outro modelo do sistema) estão corretos antes de usá-los, mas isso deixaria o sistema bastante complicado, cheio de `if`s de 'segurança'. O melhor a fazer é simplesmente não guardar um `Livro` se ele não satisfizer algumas restrições consideradas necessárias para que o `Livro` seja usado no sistema.

Assim, no resto do código não precisamos nos preocupar em checar os dados do livro: se um livro chegou para nós, ele está pronto para ser usado. Para implementar essas restrições, por exemplo, a obrigatoriedade do título do livro, vamos modificar o método `salva` do nosso `LivrosController`:

```
public void salva(Livro livro) {
```

```
  if (livro.getTitulo() == null) {
    //não deixa salvar!
  }
  estante.guarda(livro);

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

O que significa não deixar salvar? Esse método `salva` é invocado a partir do formulário de cadastro do `Livro` e, se existe algum campo inválido, precisamos avisar para a pessoa que está preenchendo-o o que está errado. Voltando para o método:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    //volta para o formulário dizendo que o título é obrigatório
  }
  estante.guarda(livro);

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

Se, no entanto, existirem mais validações:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
  //volta para o formulário dizendo que o título é obrigatório
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
  //volta para o formulário dizendo que o preço é obrigatório e
  //deve ser positivo
  }
  estante.guarda(livro);

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

Não podemos voltar para o formulário a cada campo que estiver errado, se não vai acontecer o seguinte: o usuário preenche o formulário, clica em "Salvar" e vê que o título está inválido; então

ele arruma o título e clica novamente em "Salvar" e a página fala que o preço está inválido; e assim por diante para cada campo do livro. Melhor seria acumular todos os erros que acontecerem e mostrar todos de uma vez no formulário, se houver algum erro para ser mostrado. Se não existirem erros, podemos guardar o livro e continuar com o processo.

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    //adiciona o erro "título é obrigatório"
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    //adiciona o erro "preço é obrigatório e deve ser positivo"
  }
  if (houverem erros) {
    //volta para o formulário mostrando os erros
  }
  estante.guarda(livro);

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

Poderíamos controlar tudo isso manualmente, somente usando o `result` para mudar o fluxo. Mas como essa tarefa de validar se o objeto a ser salvo é bem comum, existe um componente do VRaptor especializado em executar essas validações: o `Validator`.

Nele, podemos adicionar mensagens de erro de validação e, se houver erros, redirecionar a requisição de volta para o formulário, ou seja, exatamente o que nós queríamos. Para ter acesso a esse componente, vamos recebê-lo no construtor:

```
@Controller
public class LivrosController {

  private Estante estante;
```

```
  private Result result;
  private Validator validator;

  public LivrosController(Estante estante, Result result,
                          Validator validator) {
    this.estante = estante;
    this.result = result;
    this.validator = validator;
  }
}
```

Para adicionar uma validação, usamos o método `add` do `Validator`, passando uma `Message`. Uma das mensagens possíveis é a `SimpleMessage`, com a qual passamos o campo ou categoria a que se refere a mensagem e o seu texto:

```
  if (livro.getTitulo() == null) {
    validator.add(new SimpleMessage("titulo",
                                    "título é obrigatório"));
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    validator.add(new SimpleMessage("preco",
          "preço é obrigatório e deve ser positivo"));
  }
```

Para redirecionar para o formulário, fazemos algo bem parecido com os redirecionamentos do `Result`. No `Validator`, falamos que, se houver erros, queremos redirecionar para o formulário:

```
validator.onErrorRedirectTo(this).formulario();
```

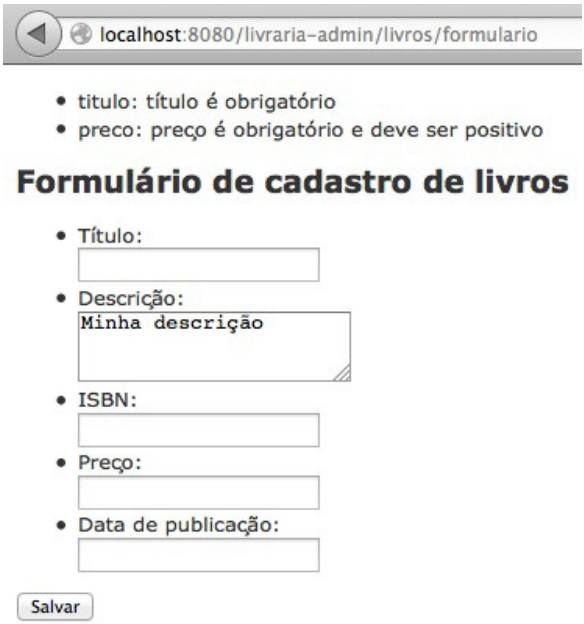
Isso segue a mesma lógica do `Result`: estamos redirecionando para esse mesmo controller, no método `formulario`. Ao voltar para o formulário, todos os erros que ocorreram vão estar disponíveis na variável `${errors}`. Para mostrar os erros no formulário, podemos fazer:

```
<ul class="errors">
  <c:forEach items="${errors}" var="error">
    <li>
      <!-- o campo em que ocorreu o erro, ou o tipo do erro -->
      ${error.category}:

      <!-- a mensagem de erro de validação -->
      ${error.message}
    </li>
  </c:forEach>
</ul>
```

O nosso método `salva` ficaria assim:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    validator.add(
      new SimpleMessage("titulo", "título é obrigatório")
    );
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    validator.add(new SimpleMessage("preco",
        "preço é obrigatório e deve ser positivo"));
  }
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

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

Nada que está abaixo da linha do `onErrorRedirectTo` será executado caso exista algum erro de validação.



6.1 INTERNACIONALIZAÇÃO DAS MENSAGENS

Em `Livro` , temos título e preço como campos obrigatórios. Podemos dizer também que o ISBN é obrigatório, já que é o nosso identificador do livro. Se formos parar para pensar, cada classe que formos salvar no banco terá alguns campos obrigatórios e, em todos eles, vamos fazer uma validação com a mensagem bem parecida:

```
if (livro.getTitulo() == null) {
  validator.add(new ValidationMessage("título é obrigatório",
                                      "titulo");
}
```

O que acontece agora se for preciso alterar a mensagem dos

campos obrigatórios de "campo é obrigatório" para "campo deve ser preenchido"? Vamos ter de passar pelo sistema todo mudando as mensagens de campo obrigatório. E se precisarmos mostrar as mensagens ora em português, ora em espanhol e ora em inglês? O processo de geração das mensagens ficaria bem complicado e espalhado pelo sistema.

Para resolver esse tipo de problema, o Java possui uma classe chamada `ResourceBundle`, que foi pensada para separar esse tipo de mensagens do meio do código, além de possibilitar a internacionalização (**i18n**) delas. Para fazer isso, precisamos criar um conjunto de arquivos `.properties`, um para cada língua que formos suportar no sistema.

```
messages.properties       => as mensagens na língua padrão
messages_en.properties    => as mensagens em inglês
messages_es.properties    => as mensagens em espanhol
messages_pt_BR.properties => as mensagens em português do Brasil.
```

Se usarmos os arquivos exatamente nesse padrão, começando com `messages` e colocando-os no *classpath*, o VRaptor possibilita usar as mensagens desse bundle, com a classe `I18nMessage`. Nela, passamos primeiro o campo em que ocorreu o erro, e depois a chave da mensagem:

```
if (livro.getTitulo() == null) {
  validator.add(new I18nMessage("titulo", "campo.obrigatorio"));
}
if (livro.getPreco() == null) {
  validator.add(new I18nMessage("preco", "campo.obrigatorio"));
}
if (livro.getIsbn() == null) {
  validator.add(new I18nMessage("isbn", "campo.obrigatorio"));
}
```

E no arquivo `messages.properties` (ou em alguma das

línguas):

```
campo.obrigatorio = deve ser preenchido
```

Podemos ainda passar parâmetros para a mensagem, usando os próximos argumentos do construtor do `I18nMessage` e usando `{0}`, `{1}`, `{2}` etc. no arquivo de mensagens, representando cada um dos parâmetros adicionais, na ordem dos argumentos.

```
if (livro.getTitulo() == null) {
  validator.add(
    new I18nMessage("titulo", "campo.obrigatorio", "título"));
}

if (livro.getPreco() == null) {
  validator.add(
    new I18nMessage("preco", "campo.obrigatorio", "preço"));
} else if (livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
  validator.add(
    new I18nMessage("preco", "campo.maior.que", "preço", 0));
}

if (livro.getIsbn() == null) {
  validator.add(new I18nMessage("isbn", "campo.obrigatorio",
                                "isbn"));
}
```

O `messages.properties` fica:

```
campo.obrigatorio = {0} deve ser preenchido
campo.maior.que = {0} deve ser maior que {1}
```

Assim, se forem geradas, as mensagens ficariam:

```
título deve ser preenchido
preço deve ser preenchido
preço deve ser maior que 0
isbn deve ser preenchido
```

Por outro lado, se digitarmos um texto qualquer nos campos "Preço" e "Data de publicação" do formulário, receberemos as

mensagens:

```
preco: ???is_not_a_valid_number???
dataPublicacao: ???is_not_a_valid_date???
```

Esses erros são adicionados automaticamente pelo VRaptor se o valor mandado na requisição não puder ser convertido para o tipo necessário. No nosso caso, converter um texto qualquer para `BigDecimal` ou `Calendar`. O texto que está entre `???` é a chave de i18n que podemos usar para gerar a mensagem desses erros. Podemos colocar uma mensagem com essa chave no `messages.properties`, usando `{0}` para incluir o valor inválido:

```
is_not_a_valid_number = "{0}" não é um número válido
is_not_a_valid_date = "{0}" não é uma data válida
```

E receber as mensagens:

```
preco: "dez reais" não é um número válido
dataPublicacao: "hoje a noite" não é uma data válida
```

Esses erros são adicionados antes de o controller ser executado e, caso você não tenha usado o `validator` para dizer para onde ir em caso de erro, receberá a exception:

```
There are validation errors and you forgot to specify where
to go.
Please add in your method something like:

validator.onErrorUse(page()).of(AnyController.class)
                            .anyMethod();

or any view that you like.
If you didn't add any validation error, it is possible that a
conversion error had happened.
```

6.2 VALIDAÇÃO FLUENTE

Outra forma de usar as mensagens internacionalizadas, fugindo um pouco dos `ifs`, é usando a validação fluente do VRaptor. Nela, declaramos as condições, e caso não sejam satisfeitas, o erro de validação é adicionado.

Esse erro pode ser adicionado usando a `SimpleMessage` ou a `I18nMessage`. Temos duas maneiras de declarar as condições. Uma é bem parecida com os `ifs`:

```
validator.addIf(livro.getTitulo() == null,
    new I18nMessage("titulo", "campo.obrigatorio", "título"));

BigDecimal preco = livro.getPreco();
validator.addIf(preco == null,
    new I18nMessage("preco", "campo.obrigatorio", "preço"));

validator.addIf(preco != null &&
                preco.compareTo(BigDecimal.ZERO) < 0,
    new I18nMessage("preco", "campo.maior.que", "preço", 0));

validator.addIf(livro.getIsbn() == null,
    new I18nMessage("isbn", "campo.obrigatorio", "isbn"));

validator.onErrorRedirectTo(this).formulario();
```

Ou seja, cada mensagem será adicionada se a respectiva condição for verdadeira. Na segunda maneira, especificamos o que queremos que seja verdade. Nesse caso, queremos garantir que o título esteja preenchido, então usamos o método `ensure`:

```
validator.ensure(livro.getTitulo() != null,
    new I18nMessage("titulo", "campo.obrigatorio", "título"));

BigDecimal preco = livro.getPreco();
validator.ensure(preco != null,
    new I18nMessage("preco", "campo.obrigatorio", "preço"));

validator.ensure(preco != null &&
                 preco.compareTo(BigDecimal.ZERO) > 0,
    new I18nMessage("preco", "campo.maior.que", "preço", 0));
```

```
validator.ensure(livro.getIsbn() != null,
    new I18nMessage("isbn", "campo.obrigatorio", "isbn"));

validator.onErrorRedirectTo(this).formulario();
```

Repare que sempre é necessário especificar o que fazer caso haja erros e, nesse caso, redirecionamos para o `formulario`.

6.3 ORGANIZANDO MELHOR AS VALIDAÇÕES COM O BEAN VALIDATIONS

Embora o `Validator` do VRaptor ajude bastante, acabamos repetindo bastante código ao declarar validações das maneiras vistas anteriormente. Por exemplo, toda vez que um campo for obrigatório, teremos as linhas:

```
if (objeto.getCampo() == null)
  validator.add(new I18nMessage("campo", "campo.obrigatorio",
                                "campo"));

//ou
that(objeto.getCampo() != null, "campo", "campo.obrigatorio",
                                "campo");
```

É a mesma coisa para outros tipos de validação, como ver se um número é maior do que zero, ver se uma string tem no máximo 10 caracteres, ver se uma data está no passado etc. Essas validações estão presentes em quase todo tipo de objeto que vai ser salvo no banco de dados.

Por esse motivo, em vez de ficar repetindo esses `ifs`, poderíamos falar que um determinado campo é obrigatório, e alguém deve se encarregar de fazer o `if` para nós. Também é o mesmo para um campo que deve ser maior que zero, ou um

campo que tem de estar no passado.

Para resolver esse problema, existe uma especificação do Java chamada **Bean Validations**. Com ela, usamos anotações em cima dos campos para declarar as validações a serem aplicadas. No caso do `Livro`, as validações seriam:

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

  @NotEmpty
  private String isbn;

  @NotEmpty
  private String titulo;

  @NotNull @DecimalMin("0.0")
  private BigDecimal preco;

  @Past
  private Calendar dataPublicacao;

  private String descricao;
  //...
}
```

E os `ifs` para verificar se o campo está válido e adicionar a mensagem de validação serão feitos automaticamente. Tudo o que temos de fazer é indicar ao VRaptor que queremos validar o `Livro`, anotando o parâmetro com `@Valid`:

```
public void salva(@Valid Livro livro) {
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

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

Ou, se preferir, chamando explicitamente o método

validate do Validator:

```
public void salva(Livro livro) {
  validator.validate(livro);
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

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

Se o método recebe mais de um parâmetro, podemos validar cada um deles:

```
public void modifica(@Valid Livro livro,
                     @Valid Editora editora,
                     @NotEmpty String motivo) {
  validator.onErrorRedirectTo(this).formulario();
  //...
}
```

Exemplos de anotações de validação do Bean Validations

A especificação Bean Validations possui algumas anotações já padronizadas para fazermos as validações. Porém, cada implementação está livre para adicionar novas anotações, sendo que é possível criá-las na nossa própria aplicação. Se usarmos o Hibernate Validator, algumas das anotações são:

- **Para objetos em geral:** `@Null` e `@NotNull`, para garantir que um objeto seja ou não nulo, respectivamente.
- **Para strings:** `@Size(min=0, max=20)`, para validar o tamanho; `@Pattern(regexp="[0-9]*")`, para casar padrões; `@Email` e `@CreditCardNumber`, para garantir que a `String` represente um e-mail ou um

número de cartão de crédito.

- **Para números:** `@Min(2)`, `@Max(30)` e `@Range(min=1, max=40)`, para definir limites para um número inteiro; `@DecimalMin("0.01")` e `@DecimalMax("999.99")`, para definir limites para um número decimal; e `@Digits(integer=6, fraction=2)`, para definir a precisão da parte inteira e da fracionária de um número decimal.
- **Para datas:** `@Past` e `@Future`, para datas no passado ou no futuro.
- **Para booleans:** `@AssertTrue` e `@AssertFalse`, para garantir que o boolean é verdadeiro ou falso.

Ainda é possível usar bibliotecas de terceiros, como por exemplo o **Caelum Stella**, que nos dá validações brasileiras, como o `@CPF` e o `@CNPJ`.

Assim, podemos concentrar a maioria das validações do nosso modelo no próprio modelo. Mas nem sempre é possível usar uma validação do *Bean validations*, como se quisermos saber se já existe um livro salvo no banco com o mesmo ISBN. Nesse caso, precisamos nos conectar ao banco e fazer uma consulta por livros com um determinado ISBN.

Entretanto, o nosso modelo `Livro` não possui esse acesso. Logo, devemos fazer essa validação usando as formas vistas anteriormente.

As mensagens geradas com o Bean Validations são em inglês. Se quisermos sobrescrever a mensagem de validação, temos três

opções:

- Por mensagem direta:

```
@NotNull(message = "Título precisa ser preenchido")
private String titulo;
```

- Por chave de validação:

```
@NotNull(message = "{campo.obrigatorio}")
private String titulo;
```

Essa chave precisa estar no bundle `ValidationMessages.properties` .

- Genericamente para um tipo de anotação de validação:

```
//Se o import é esse:
import javax.validation.constraints.DecimalMin;
```

Podemos adicionar no `ValidationMessages.properties` :

```
javax.validation.constraints.DecimalMin =
                                    Deve ser maior que {v
alue}
```

É possível interpolar na mensagem os valores que estão dentro da anotação de validação. Pudemos usar o `{value}` , pois usamos na anotação.

```
@DecimalMin("0.0")
//que é equivalente a
@DecimalMin(value="0.0")
```

6.4 BOAS PRÁTICAS DE VALIDAÇÃO

O Bean Validations é ótimo para implementar a maioria das validações. Logo, idealmente um método do controller que salva

algo no banco deveria ser algo como:

```
public void salva(@Valid Livro livro) {
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

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

Ou seja, manda validar o objeto; retorna para o formulário em caso de erro; estando tudo ok, salva o objeto; redireciona para alguma página de sucesso. Um método simples de entender e de dar manutenção. Qualquer validação adicional pode ser feita diretamente no `Livro` com as anotações do Bean Validations sem precisar alterar o controller.

Mas isso não vai resolver todas as validações possíveis, pois só conseguimos validar facilmente os dados de algum dos atributos do `Livro`, então temos de executar a validação manualmente:

```
public void salva(@Valid Livro livro) {
  validator.addIf(estante.jaExisteNoBanco(livro.getIsbn()),
    validator.add(new I18nMessage("isbn", "isbn.duplicado"));

  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

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

Quanto mais validações adicionamos nos métodos dos controllers, mais difícil é entender o que está acontecendo. Para manter o controller o mais simples possível, podemos criar novos `validator`s, por exemplo, o `LivroValidator`, que fará as validações específicas do livro.

```
public class LivroValidator {
  private Validator validator;
  private Estante estante;
  private Editoras editoras;

  public LivroValidator(Validator validator,
          Estante estante, Editoras editoras) {
    this.validator = validator;
    this.estante = estante;
    this.editoras = editoras;
  }

  public void validate(Livro livro) {
    validator.validate(livro);

    if (estante.existeLivroComTitulo(livro.getTitulo())) {
      validator.add(new I18nMessage("titulo", "ja.existe"));
    }

    if (editoras.concorrentes().contains(livro.getEditora())) {
      validator.add(new I18nMessage("editora",
            "nao.pode.ser.editora.concorrente"))
    }
  }

  //gere os delegate methods do Validator usando a sua IDE!
  public <T> T onErrorRedirectTo(T controller) {
    return validator.onErrorRedirectTo(controller);
  }
}
```

Dessa forma, nosso método do controller volta a ser simples, bastando receber um `LivroValidator` em vez do `Validator` do VRaptor.

```
@Resource
public class LivrosController {
  private LivroValidator validator;

  public LivrosController(/*...,*/ LivroValidator validator) {
    //...
    this.validator = validator;
  }
```

```
  public void salva(Livro livro) {
    validator.validate(livro);

    validator.onErrorRedirectTo(this).formulario();

    estante.guarda(livro);

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

De um modo geral, prefira controllers com menos código, focados em controlar a requisição, e mantenha lógicas de negócio em componentes especializados, ou nos próprios modelos (como no `Livro` ou na `Estante`).

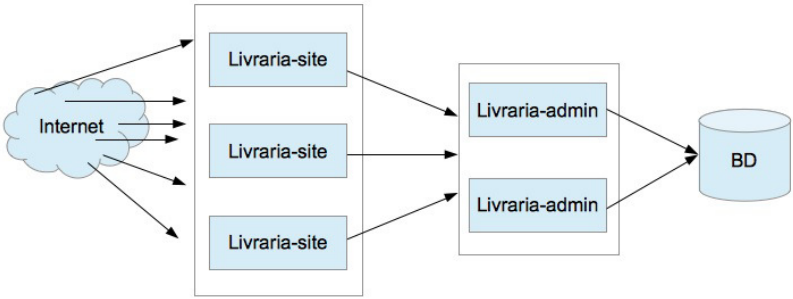
CAPÍTULO 7

INTEGRAÇÃO ENTRE SISTEMAS USANDO O VRAPTOR

Completamos o nosso cadastro de livros, com todas as validações necessárias para conseguirmos vendê-los sem problemas. Essa parte do sistema será instalada em uma rede protegida, com acesso restrito às pessoas que podem cadastrar os livros.

A segunda parte do nosso sistema é a que vai vender os livros cadastrados, o site da livraria. Ela será instalada em outro servidor, e cuidará apenas da apresentação dos livros para a compra. Para isso, ela precisará dos dados dos livros cadastrados pelo `admin` do sistema.

Existem diversas formas de integrar esses dois sistemas, desde jogar arquivos em uma pasta da rede até usar Web Services, e essa integração depende muito de como os sistemas foram implementados. No nosso caso, como os dois usam VRaptor, existe uma forma mais natural. Antes de falar sobre ela, vamos ver com mais detalhes como será essa integração.



Nossos dois sistemas são o `livraria-admin`, que cuidará do cadastro dos livros, e o `livraria-site`, que mostrará os livros em um site para vendê-los. O `livraria-site` não terá acesso ao banco de dados, logo, para conseguir os dados dos livros terá de consultar o `livraria-admin`. Com isso, conseguimos evoluir as duas partes do sistema independentemente, tanto no código quanto no deploy. Ao atualizar o `livraria-admin`, não precisamos reiniciar o `livraria-site`, e vice-versa.

Além disso, os dois sistemas têm requisitos diferentes. O `livraria-site` será acessado por milhares de pessoas ao mesmo tempo, enquanto o `livraria-admin` será acessado apenas por algumas pessoas que têm acesso ao cadastro. Este precisa de acesso ao banco de dados, de um ambiente transacional, de controle de acesso, enquanto o `livraria-site` ficará aberto ao público geral, servindo páginas que mostram os livros e guardando os carrinhos de compras dos usuários. São responsabilidades bem diferentes que deveriam ser feitas por sistemas diferentes.

O grande problema agora é: como passar os dados dos livros de um sistema para o outro? O VRaptor é um framework Web, que nos possibilita executar operações dentro do sistema a partir de

URLs no nosso navegador.

Por exemplo, ao acessar http://localhost:8080/livraria-admin/livros/lista, caímos no método `lista` do `LivrosController`. Ao final dele, é mostrada uma página que foi gerada a partir do `/WEB-INF/jsp/livros/lista.jsp`. O ponto de partida é uma requisição a uma URL que começa com `http://`, ou seja, estamos usando um protocolo chamado HTTP.

Em geral, usamos bastante esse protocolo quando estamos navegando na internet, junto com a sua versão mais segura, o HTTPS, para acessarmos páginas dos sites que visitamos. No entanto, podemos usar o mesmo HTTP para acessar outros dados, além de páginas Web, como imagens, arquivos para download, músicas, enfim, qualquer dado relevante para nós.

Usando o protocolo HTTP, podemos disponibilizar os dados dos livros no `livraria-admin`. Para isso, precisamos criar uma URL que executará um código e nos dará esses dados de que precisamos. O jeito de fazer isso no VRaptor é criar um método em um controller:

```
@Controller
public class IntegracaoController {

  public void listaLivros() {
    // ...
  }
}
```

Temos a URL `/integracao/listaLivros` que executará o método e, por padrão, redirecionará para o JSP correspondente. Geralmente usamos JSPs para criar páginas HTML, mas nesse caso queremos retornar apenas os dados dos livros.

O que significa retornar os dados dos livros? Esses dados precisam ser passados de alguma forma para o outro sistema.

No mundo ideal, poderíamos passar uma `List<Livro>` de um sistema para o outro. Porém, essa lista é um objeto que reside na memória do servidor do `livraria-admin`, e não é possível passar exatamente esse objeto para o `livraria-site`, que é outra aplicação, está em outro servidor, em outra JVM.

7.1 SERIALIZANDO OS OBJETOS

Se não podemos passar o mesmo objeto de um lado para o outro, o melhor que podemos fazer é extrair os dados do objeto em um formato qualquer e lê-los no outro sistema, transformando-os de volta em objetos. Chamamos esse processo, respectivamente, de serialização e deserialização do objeto em questão.

Para fazer isso, precisamos definir um formato que as duas aplicações entendam, e assim consigam escrever e ler o objeto. O formato pode ser qualquer coisa, por exemplo, um arquivo texto formatado posicionalmente, um arquivo binário em um formato que você mesmo inventou, ou o objeto serializado com o próprio Java (com `Serializable` s e `Object` streams).

Um bom formato, no entanto, é um que seja suportado facilmente nos dois sistemas, de preferência sem que seja preciso escrever o seu *parser*. Melhor ainda se puder ser consumido em qualquer linguagem de programação, assim ficaríamos livres para escrever o site como bem entendermos, como em *Ruby on Rails*, *Play* no *Scala* ou *ASP*.

Um dos formatos que atende isso bem é o bom e velho XML.

Conseguimos ler e gerar esse formato em qualquer linguagem de programação que seja usável em projetos de verdade. Um livro representado em XML seria algo parecido com:

```
<livro>
  <isbn>85-336-0292-8</isbn>
  <titulo>O Senhor dos Anéis</titulo>
  <autor>J. R. R. Tolkien</autor>
  <preco>130.00</preco>
</livro>
```

Por ser um formato tão conhecido e usado, o VRaptor possui uma forma fácil de serializar qualquer objeto em XML. O que queremos é mudar o resultado padrão e gerar o XML, então usamos o `Result` :

```
@Controller
public class IntegracaoController {

  private Estante estante;
  private Result result;

  @Inject
  public IntegracaoController(Estante estante, Result result) {
    this.estante = estante;
    this.result = result;
  }
  @Deprecated IntegracaoController() {}

  public void listaLivros() {
    List<Livro> livros = estante.todosOsLivros();

    result.use(Results.xml()).from(livros,"livros").serialize();
  }
}
```

Assim, ao acessarmos a URL `/integracao/listaLivros` , não veremos mais uma página HTML, e sim um XML parecido com:

```
<livros>
```

```
  <livro>
    <id>1310</id>
    <isbn>85-336-0292-8</isbn>
    <titulo>O Senhor dos Anéis</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>130.00</preco>
  </livro>
  <livro>
    <id>42</id>
    <isbn>9789728839130</isbn>
    <titulo>O Guia dos Mochileiros das Galáxias</titulo>
    <autor>Douglas Adams</autor>
    <preco>90.00</preco>
  </livro>
</livros>
```

Explicando um pouco melhor a chamada:

```
// Usar como resultado um xml
result.use(Results.xml())
  // a partir do objeto livros, com o nome "livros"
  .from(livros, "livros")
  // serializa o objeto e joga na resposta.
  .serialize();
```

Esse método `serialize()` pode parecer um pouco estranho em um primeiro momento, mas existe, pois o VRaptor tem uma convenção para serializar o objeto. Por padrão, só são serializados atributos simples do objeto, como números, datas, enums e Strings.

Se o objeto tiver algum atributo que seja, por exemplo, um outro objeto da aplicação ou uma lista, para serializarmos esse objeto, precisamos explicitamente incluí-lo na serialização. Por exemplo, se tivermos a classe `Autor`:

```
enum Pais { BRASIL, ESTADOS_UNIDOS, REINO_UNIDO  }
class Autor {

  private String nome;
  private Calendar dataNascimento;
```

```
  private Integer numeroDeLivros;
  private Pais naturalidade;

  private List<Livro> livros;
  private Livro ultimoLivro;

}
```

Ao serializarmos um autor da forma padrão:

```
Autor autor = // busca do banco
result.use(Results.xml()).from(autor).serialize();
```

Só serão serializados os atributos simples:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1892-01-03</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
</autor>
```

Se quisermos serializar, por exemplo, o último livro, precisamos pedir para incluí-lo:

```
result.use(Results.xml()).from(autor).include("ultimoLivro")
                                     .serialize();
```

Assim, a serialização ficaria:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1892-01-03</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>13.00</preco>
  </ultimoLivro>
</autor>
```

A regra de somente serializar atributos simples vale para o

objeto incluído também, ou seja, o último livro só terá os atributos simples. Se quisermos incluir um atributo específico do livro, fazemos `.include("ultimoLivro", "ultimoLivro.umAtributo")` . Se quisermos também serializar a lista de livros desse autor, incluímos seu respectivo campo:

```
result.use(Results.xml()).from(autor)
      .include("ultimoLivro", "livros").serialize();
```

A serialização seria:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1876-04-05</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <livros>
    <livro>
      <id>233</id>
      <isbn>85-336-1165-X</isbn>
      <titulo>Silmarillion</titulo>
      <autor>J. R. R. Tolkien</autor>
      <preco>13.00</preco>
    </livro>
    <livro>
      <id>1310</id>
      <isbn>8533613377</isbn>
      <titulo>O Senhor dos Anéis - A sociedade do anel</titulo>
      <autor>J. R. R. Tolkien</autor>
      <preco>60.00</preco>
    </livro>
    <!-- outros livros -->
  </livros>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>13.00</preco>
  </ultimoLivro>
</autor>
```

Uma outra possibilidade é pedir para serializar a árvore inteira

de objetos, isto é, todos os atributos, recursivamente, com o método `recursive` :

```
result.use(Results.xml()).from(autor).recursive().serialize();
```

Nada será excluído da serialização. Muito cuidado, pois isso pode gerar respostas gigantes e pesadas:

```
<autor>
  ...
  <ultimoLivro>
    ...
    <editora>
      ...
      <colecoes>
        <colecao>
          ...
          <lojas>
            ...
              ...
          </lojas>
        </colecao>
      </colecoes>
    </editora>
  </ultimoLivro>
</autor>
```

Ou pior, se existisse um ciclo nessa árvore de objetos, por exemplo, se o livro tivesse uma referência para o autor, a recursão seria infinita:

```
<autor>
  <name>J. R. R. Tolkien</name>
  ...
  <ultimoLivro>
    <name>Senhor dos Anéis</name>
    ...
    <autor>
      <name>J. R. R. Tolkien</name>
      ...
      <ultimoLivro>
        <name>Senhor dos Anéis</name>
```

```
...
<autor>
  <name>J. R. R. Tolkien</name>
  ...
  <ultimoLivro>
    <name>Senhor dos Anéis</name>
    ...
```

Para evitar isso, acontecerá um erro caso exista um ciclo na árvore de objetos. Em todo caso, pense muito bem antes de usar o `.recursive()`, pois a resposta pode ficar maior que o necessário.

Caso a configuração do XML fique muito complicada, considere criar uma classe que expõe somente os dados que você quer serializar — um DTO (*Data Transfer Object*, do livro PoEAA do Martin Fowler, ou http://martinfowler.com/eaaCatalog/dataTransferObject.html).

Voltando ao resultado anterior, sem o `recursive()`, cada livro possui o nome do autor, que é um pouco redundante, já que estamos serializando o livro. Se quisermos remover o atributo `autor` da serialização, podemos usar o método `.exclude`:

```
result.use(Results.xml())
      .include("ultimoLivro")
      .exclude("ultimoLivro.autor")
      .serialize();
```

Assim, a resposta ficaria:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1876-04-05</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <preco>13.00</preco>
```

```
  </ultimoLivro>
</autor>
```

7.2 RECEBENDO OS DADOS NO SISTEMA CLIENTE

Uma das vantagens de expor um serviço da maneira anterior é que o cliente desse serviço pode ser qualquer tipo de sistema, escrito em qualquer linguagem de programação com a qual é possível executar uma requisição HTTP e consumir XML. E como HTTP é um dos protocolos mais usados e XML é um dos formatos mais adotados, praticamente todas as linguagens de programação saberão trabalhar com eles.

No nosso caso, o sistema cliente será uma aplicação Java também escrita usando o VRaptor: o `livraria-site`. Antes de começar a integração, vamos escrever a página inicial do site, que mostrará os livros em uma vitrine. No projeto `livraria-site`, criaremos o `HomeController`:

```
@Controller
public class HomeController {

  public void inicio() {
  }

}
```

Precisamos mostrar na página dessa lógica uma lista de livros, que será consumida do serviço do `livraria-admin`. Devemos colocar o código que consome o serviço dentro do método `inicio`?

Precisaremos também de dados de livros na página que mostra o livro, na página do carrinho de compras, na página de finalização

da compra e em várias outras. Então não vale a pena repetir esse código por todo canto.

Para evitar essa repetição, vamos criar um componente do sistema responsável por acessar os dados dos livros, como fizemos no `livraria-admin`, um **repositório** de livros. A diferença é que, agora, o acesso aos dados se dá pelo serviço disponibilizado pelo `livraria-admin`, e não pelo banco de dados.

Podemos utilizar aqui também a `Estante`, nosso repositório de livros. Mas para não causar confusões, usaremos outra abstração para um conjunto de livros: um **Acervo**, que nos dará acesso aos livros do sistema, que estão cadastrados no `livraria-admin`.

Vamos criar a interface `Acervo` no projeto `livraria-site`.

```
public interface Acervo {

  List<Livro> todosOsLivros();

}
```

Precisamos também ter o `Livro` do lado do site, já que trabalhamos com objetos, e não com XMLs, no resto do sistema. A classe `Livro` não precisa ser necessariamente idêntica nos dois sistemas, já que alguns dados podem não fazer sentido para o site, por exemplo, se guardássemos preço de custo, margem de lucro, quantidade em estoque ou outras coisas.

No nosso caso, podemos copiar a classe `Livro` do admin para o site, pois vamos usar todos os dados do livro. Podemos, no entanto, remover as anotações da JPA, já que não vamos salvá-lo no banco de dados.

Vamos agora implementar a *home* do site, disponibilizando a lista de livros para a JSP:

```
@Controller
public class HomeController {

  private Acervo acervo;
  private Result result;

  @Inject
  public HomeController(Acervo acervo, Result result) {
    this.acervo = acervo;
    this.result = result;
  }
  @Deprecated HomeController() {}

  public void inicio() {
    this.result.include("livros", acervo.todosOsLivros());
  }

}
```

E no JSP padrão (`WEB-INF/jsp/home/inicio.jsp` dentro de `livraria-site/src/main/webapp`) acrescentar em alguma parte a lista dos livros:

```
<h3>Veja as últimas ofertas para você:</h3>

<ul class="livros">
  <c:forEach items="${livros}" var="livro">
    <li class="livro">${livro.titulo} - R$ ${livro.preco}</li>
  </c:forEach>
</ul>
```

7.3 CONSUMINDO OS DADOS DO ADMIN

Com a página inicial implementada, precisamos agora alimentá-la com os dados dos livros, que virão do `livraria-admin` . Precisamos implementar o `Acervo` consumindo os

serviços desse sistema.

Vamos criar a classe `AcervoNoAdmin`, como um componente que implementa `Acervo`:

```
package br.com.casadocodigo.livraria.site.servico;

public class AcervoNoAdmin implements Acervo {

  @Override
  public List<Livro> todosOsLivros() {
    //...
  }
}
```

A `livraria-admin` possui um serviço que criamos para retornar a lista de todos os livros. Ele foi criado usando o protocolo HTTP, que é o que usamos na web. Precisamos de algo que consiga consumir um serviço HTTP na URL que nos retorna o XML dos livros.

Se pensarmos nos serviços web tradicionais, teríamos de criar um *endpoint*, que geraria um documento com todas as operações possíveis. E na aplicação cliente, precisaríamos criar várias classes para conseguir consumir esse serviço.

No nosso caso, estamos usando o próprio VRaptor para expô-lo, como se fôssemos utilizá-lo no navegador, usando o protocolo HTTP. Para isso, tudo o que precisamos é de um cliente HTTP, que existe em qualquer linguagem de programação que formos usar. Em Java, existe um jeito fácil de fazer uma requisição HTTP e receber o resultado, utilizando a classe `java.net.URL`:

```
URL url = new URL("http://localhost:8080/livraria-admin" +
  "/integracao/listaLivros");
InputStream resposta = url.openStream();
```

Ou seja, passando a URL do serviço que nos retorna a lista de livros, conseguimos executar uma requisição a essa URL, recebendo a resposta como um `InputStream`. No entanto, muitas coisas podem dar errado no meio do caminho. Por exemplo, a URL pode estar incorreta, pode estar apontando para um lugar que não existe, o servidor pode estar fora do ar, a rede pode ter caído etc. Por esse motivo, esse código lança exceções que precisam ser tratadas.

```
try {
  URL url = new URL("http://localhost:8080/livraria-admin" +
    "/integracao/listaLivros");
  InputStream resposta = url.openStream();
} catch (MalformedURLException | IOException e) {
  e.printStackTrace();
}
```

Esse é um código de muito baixo nível para estar no `Acervo`, que é uma abstração de alto nível. Vamos criar um componente para nos ajudar a fazer requisições HTTP, retornando-nos uma `String` com a resposta. Esse componente será o `ClienteHTTP`, com a implementação usando `URL`.

```
package br.com.casadocodigo.livraria.site.servico;

public interface ClienteHTTP {
  String get(String url);
}

public class URLClienteHTTP implements ClienteHTTP {

  @Override
  public String get(String url) {
    try {
      URL servico = new URL(url);
      InputStream resposta = servico.openStream();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
```

```
      e.printStackTrace();
    }
  }
}
```

Precisamos agora converter essa resposta de `InputStream` para `String`. Existem vários jeitos de fazer isso, em várias bibliotecas diferentes. Vamos usar a classe `com.google.common.io.CharStreams`, da biblioteca **Google Guava** que já é dependência do VRaptor e contém vários utilitários que melhoram vários aspectos do Java.

Essa classe consegue transformar um `Reader` em uma `String`. Para transformar o `InputStream` em um `Reader`, usamos o `InputStreamReader` do próprio Java.

```
Reader reader = new InputStreamReader(resposta);
String respostaEmString = CharStreams.toString(reader);
```

Integrando com o resto do código, ficaria:

```
public String get(String url) {
  try (
    InputStream resposta = new URL(url).openStream();
    Reader reader = new InputStreamReader(resposta);
  ) {
    return CharStreams.toString(reader);
  } catch (MalformedURLException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Esse código ainda não compila, pois precisamos decidir o que fazer caso ocorram exceptions. Temos dois tipos de `Exception` sendo lançadas aqui: `MalformedURLException`, quando a URL não é válida; e `IOException`, quando aconteceu algum erro de

comunicação com o servidor durante a requisição.

A `MalformedURLException`, no nosso caso, seria culpa do programador, já que as URLs virão do código. Logo, podemos supor que elas só vão acontecer em tempo de desenvolvimento e simplesmente a lançarmos dentro de alguma `RuntimeException`.

```
} catch (MalformedURLException e) {
  throw new IllegalArgumentException("A url " + url +
        " está inválida, corrija-a!", e);
}
```

Já a `IOException` foge do nosso controle e, se faz parte do nosso processo se recuperar de alguma forma caso o servidor esteja fora do ar, precisamos deixar isso no contrato do nosso cliente HTTP, isto é, da interface `ClienteHTTP`. Lançar `IOException` talvez seja genérico demais — podemos optar por criar uma exceção mais específica, como `ServidorIndisponivelException`:

```
public class
ServidorIndisponivelException extends RuntimeException {
  public ServidorIndisponivelException(String url, Exception e){
    super("Erro ao fazer requisição ao servidor na url " +
          url, e);
  }
}

public interface ClienteHTTP {
  public String get(String url) throws
                                  ServidorIndisponivelException;
}
```

E na hora de fazer o `catch`, usamos a nossa exceção:

```
} catch (IOException e) {
  throw new ServidorIndisponivelException(url, e);
}
```

A `ServidorIndisponivelException` está como `RuntimeException` para não obrigarmos o usuário do `ClienteHTTP` a tratá-la. Mas deixamos a exceção declarada na interface, para que o usuário saiba que tem a opção de se recuperar desse caso específico.

Se quisermos obrigar a sempre tratar esse caso, trocamos o `extends` para `Exception`. Porém, cuidado. Isso pode gerar muitos métodos que, ou relançam essa exceção declarando o `throws ServidorIndisponivelException`, ou fazem um `try..catch` relançando a exceção dentro de uma `RuntimeException`, o que torna o código bem chato de manter e evoluir.

7.4 TRANSFORMANDO O XML EM OBJETOS

Com o nosso `ClienteHTTP` implementado, agora nosso código da `AcervoNoAdmin` está assim:

```
public class AcervoNoAdmin implements Acervo {

  private ClienteHTTP http;

  @Inject
  public AcervoNoAdmin(ClienteHTTP http) {
    this.http = http;
  }
  @Deprecated AcervoNoAdmin() {}

  @Override
  public List<Livro> todosOsLivros() {
    String url = "http://localhost:8080/livraria-admin" +
        "/integracao/listaLivros";
    String resposta = http.get(url);

    return null;
  }
```

```
}
```

A resposta será um XML parecido com:

```
<livros>
  <livro>
    <id>1310</id>
    <isbn>8533613377</isbn>
    <titulo>O Senhor dos Anéis</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>130.00</preco>
  </livro>
  <livro>
    <id>42</id>
    <isbn>9789728839130</isbn>
    <titulo>O Guia dos Mochileiros das Galáxias</titulo>
    <autor>Douglas Adams</autor>
    <preco>90.00</preco>
  </livro>
</livros>
```

Precisamos agora ler essa resposta, que contém um XML com a lista de livros, e transformá-la em objetos `List<Livro>`, para podermos usar esses dados no restante do sistema. Existem várias bibliotecas que fazem isso para nós, sendo uma delas o **XStream**.

Ela já vem como dependência do VRaptor, e é a biblioteca que ele usa para gerar e consumir XMLs (e JSONs). A ideia do XStream é que ele seja bem simples de usar, com o mínimo de configurações.

```
XStream xstream = new XStream();
Object object = xstream.fromXML(resposta);
```

A convenção do XStream para conseguir deserializar a resposta em um objeto é que o nó raiz tenha o nome da classe. Ou seja, se quisermos deserializar um `Livro`, o nó raiz do XML deveria ser `<br.com.casadocodigo.livraria.modelo.Livro>`. A partir

daí, sempre que possível, ele tenta usar os atributos dos objetos como nomes das tags.

Para conseguir consumir o XML que o VRaptor gerou, precisamos configurar isso no XStream, falando que o nó raiz `<livros>` é uma lista e que o nó de cada elemento da lista, `<livro>`, é um `Livro`. Fazemos isso com o método `alias`:

```
XStream xstream = new XStream();
xstream.alias("livros", List.class);
xstream.alias("livro", Livro.class);
```

O VRaptor usa o XStream para fazer a serialização em XML, adicionando conversores para alguns tipos, como o `Calendar`. É conveniente usar os mesmos conversores, ou pelo menos os mesmos formatos, na serialização e na deserialização. Para seguir o padrão do VRaptor, podemos receber como dependência a classe `XStreamBuilder`, e usá-la para criar o `XStream`:

```
public class AcervoNoAdmin implements Acervo {

  private ClienteHTTP http;
  private XStreamBuilder builder;

  @Inject
  public AcervoNoAdmin(ClienteHTTP http,XStreamBuilder builder){
    this.http = http;
    this.builder = builder;
  }
  @Deprecated AcervoNoAdmin() {}

  @Override
  public List<Livro> todosOsLivros() {
    XStream xstream = builder.xmlInstance();
    xstream.alias("livros", List.class);
    xstream.alias("livro", Livro.class);
    //...
  }
}
```

Ao pedirmos para o XStream consumir o XML, temos como resposta uma `List<Livro>` :

```
List<Livro> livros = (List<Livro>) xstream.fromXML(xml);
```

Nosso método completo ficaria:

```
@Override
public List<Livro> todosOsLivros() {
  String xml = http.get("http://localhost:8080/livraria-admin" +
      "/integracao/listaLivros");

  XStream xstream = builder.xmlInstance();
  xstream.alias("livros", List.class);
  xstream.alias("livro", Livro.class);

  List<Livro> livros = (List<Livro>) xstream.fromXML(xml);
  return livros;
}
```

Com isso, se conhecermos a URL da lógica do admin que retorna a lista de livros, conseguimos os dados dos livros novamente em objetos Java. Então, podemos trabalhar com eles normalmente no resto do sistema.

7.5 GERENCIANDO CONFIGURAÇÕES DIFERENTES ENTRE AMBIENTES — ENVIRONMENT

Acabamos de escrever um código que chama o serviço de lista dos livros no `livraria-admin` . Mas para isso, estamos fazendo uma suposição forte: que esse serviço estará localizado em http://localhost:8080/livraria-admin. Isso significa que obrigatoriamente o `livraria-site` e o `livraria-admin` devem estar na mesma máquina. Será que isso é razoável?

Enquanto estamos desenvolvendo a aplicação, ou seja, em **ambiente de desenvolvimento**, muito provavelmente estaremos mesmo na mesma máquina. Mas e quando formos instalar o sistema real no **ambiente de produção**, isso será verdade?

No começo do capítulo, falamos justamente que separamos o sistema em dois serviços para podermos colocar vários servidores para o `livraria-site`, que terá muitos usuários, e poucos servidores para o `livraria-admin`, ou seja, serão servidores diferentes. Isto é, nesse ambiente, o endereço do `livraria-admin` será diferente, por exemplo, `http://admin.livraria.com.br`.

Poderíamos substituir essa URL na classe `AcervoNoAdmin` logo antes de gerar o WAR que vai para produção. Porém, existe uma solução melhor, que não precisa confiar na nossa memória: **configurações**.

URLs dos serviços, base de dados, servidor de e-mails ou senhas são exemplos de diferenças entre os ambientes de desenvolvimento e de produção. Para gerenciar essas diferenças, precisamos extrair esses valores do meio do código e jogá-los em outro lugar, como um arquivo de configuração `.properties`:

```
url.admin = http://localhost:8080/livraria-admin

base.dados = livraria_dev

email.usuario = lucas
email.senha = VRaptor4!
```

Assim, para fazer o *deploy* do sistema em produção, basta substituir esse arquivo antes de gerar o WAR.

Para facilitar esse processo de gerenciar configurações, o

próprio VRaptor pode gerenciá-las para você. Para isso, você pode criar o arquivo `development.properties` na pasta `src/main/resources` (ou qualquer pasta que seja jogada no classpath), e colocar as configurações vistas anteriormente. Para o ambiente de produção, podemos criar o arquivo `production.properties`, também na pasta `src/main/resources`, com as configurações desse ambiente:

```
url.admin = http://admin.livraria.com.br

base.dados = livraria_prod

email.usuario = livraria
email.senha = $3nH453<rE74
```

Por padrão, o ambiente é `development`. Se quisermos mudá-lo, temos três opções:

- **Variável de ambiente `VRAPTOR_ENV`:** se o servidor for subido com essa variável de ambiente setada, esse será o ambiente.

```
#no caso do linux ou mac
cd meu_servidor_favorito/
VRAPTOR_ENV=production bin/startup.sh
```

- **Parâmetro da JVM `br.com.caelum.vraptor.environment`:** normalmente, cada servidor possui uma maneira de setar parâmetros da JVM, em geral em arquivos `.xml` ou `.properties`.

```
JAVA_OPTIONS=<outras opçoes> -Dbr.com.caelum.vraptor
                              .environment=production

java -Dbr.com.caelum.vraptor.environment=production -jar
                                        start.jar
```

- **Parâmetro de contexto**

`br.com.caelum.vraptor.environment` : pode ser configurado no `web.xml` ou em algum outro XML de contexto do servidor.

```
<context-param>
  <param-name>br.com.caelum.vraptor.environment</param-na
me>
  <param-value>production</param-value>
</context-param>
```

Podemos usar qualquer nome de ambiente, como `test` , `homologacao` ou `staging` . E o arquivo de configuração correspondente será o `<nome do ambiente>.properties` .

Para acessar essas configurações, podemos receber uma String como dependência, anotada com `@Property` :

```
@Inject
public AcervoNoAdmin(
          ClienteHTTP http,
          XStreamBuilder builder,
          @Property("admin.url") String adminUrl) {
  this.http = http;
  this.builder = builder;
  this.adminUrl = adminUrl;
}
```

Ou se não for injeção por construtor:

```
@Inject
@Property("admin.url")
private String adminUrl;
```

Então podemos usar essa configuração no método:

```
@Override
public List<Livro> todosOsLivros() {
  String xml = http.get(adminUrl + "/integracao/listaLivros");
  //...
```

Colocamos dentro da anotação `@Property` exatamente a chave da configuração que queremos receber, e o valor injetado será o valor dessa chave correspondente ao arquivo de configuração do ambiente atual, por exemplo no `development.properties`. Se a chave não existir no arquivo do ambiente, o VRaptor lançará uma exception:

```
java.util.NoSuchElementException: Key admin.url not found in
                                  environment development
```

Além de receber propriedades desses arquivos, podemos receber a classe `Environment` como dependência e verificar qual é o ambiente atual:

```
@Inject
private Environment env;

//...
env.getName() // => o nome do ambiente
env.isDevelopment() // se o ambiente é development
env.isProduction() // se o ambiente é production
env.get("admin.url") // pega a configuracao admin.url
env.get("admin.url", "valor padrão") // mesmo que get, mas
                                     // retorna o valor padrão
                                     // se a config não existir

env.getResource("/abc.txt") // pega o arquivo abc.txt que está
                            // na pasta <nome do env>/
                            // ex. development/abc.txt
```

Com isso, conseguimos gerenciar as configurações que são diferentes entre os ambientes de desenvolvimento, teste e produção de forma fácil e clara, recebendo-as como dependência das nossas classes.

7.6 APROVEITANDO MELHOR O PROTOCOLO HTTP ─ REST

Quando pensamos em serviços web tradicionais, os *Web Services SOAP*, para cada operação que o servidor suportar, é necessário criar pelo menos 4 classes no cliente, sem contar as classes de modelo intermediárias que serão trafegadas de um sistema para o outro. Isso porque precisamos ensinar o cliente a fazer cada uma das operações, qual é o formato de entrada, qual é o formato de saída, entre outras coisas. Tudo isso é definido dentro de um documento chamado de WSDL (sigla para *Web Services Description Language*).

Já no nosso caso, estamos usando o protocolo HTTP para implementar o serviço, e não precisamos criar classes para ensinar o sistema cliente a executar uma operação no sistema servidor. Tudo o que precisamos foi de um cliente HTTP. Isso acontece porque o HTTP já define um conjunto pequeno de operações possíveis dentro desse protocolo.

Com isso, basta que o cliente saiba executar essas operações e o servidor saiba entender essas operações fixas. Assim, nenhuma classe de infraestrutura é necessária para que um sistema se comunique com o outro.

Apesar de esse conjunto de operações ser pequeno, são operações universais, que são suficientes para implementarmos qualquer tipo de serviço. Essas operações, também chamadas de método ou verbo, são: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, `TRACE` e, recentemente, `PATCH`.

Mas como representar a operação de listar todos os livros do sistema, sendo que eu só tenho esses 8 métodos HTTP disponíveis? A ideia do HTTP é que esses métodos devem ser obrigatoriamente executados em um **recurso**, que é qualquer entidade, dado ou

personagem do sistema.

Esse recurso é representado por uma URI (*Unified Resource Identifier*, ou identificador único de um recurso). Uma URL é um tipo de URI, que também representa qual é o local onde podemos encontrar o recurso (o **L** é de *Location*, ou seja, local).

Cada método tem uma semântica muito bem definida, possibilitando que, em conjunção com um recurso, consigamos representar qualquer operação possível do nosso sistema. Usamos dois desses métodos constantemente enquanto navegamos na web: o `GET`, ao clicarmos em links ou digitarmos endereços diretamente no navegador; e o `POST`, ao submetermos formulários.

A ideia é que, se usarmos corretamente as semânticas dos métodos, podemos aproveitar toda a infraestrutura da internet, como proxies, load balancers e outros intermediários entre o cliente e o servidor.

A semântica dos métodos HTTP são:

- `GET` — Retorna as informações de um recurso. Por exemplo, `GET /livros` pode devolver todos os livros do sistema, e `GET /livro/1234`, as informações do livro de `id` igual a `1234`. Esse método **não pode** alterar o estado do recurso em questão e é idempotente, ou seja, pode ser executado quantas vezes for necessário sem que isso afete a resposta.

  Usando esse fato, os proxies podem fazer o *cache* da resposta, evitando a sobrecarga do servidor. Robôs de busca podem indexar todo o conteúdo de um site seguindo

todos os links que encontram em uma página, já que eles executam o método `GET` na URL do link, e não alteram nada no site em questão.

- `POST` — Acrescenta informações em um recurso. Por exemplo, ao fazermos um `POST /livros`, passando os dados de um livro, estamos criando um livro novo no sistema. Esse método modifica o estado recurso em questão, podendo criar novos recursos. Ele não é idempotente, ou seja, se fizermos dois `POST`s em `/livros`, mesmo que estejamos passando os mesmos dados, criaremos dois livros.

- `PUT` — Substitui as informações de um recurso. Se fizermos `PUT /livro/1234`, passando dados desse livro, atualizamos os seus atributos. Esse método modifica o recurso indicado, substituindo **todas** as suas informações pelas passadas no corpo da requisição. Se o recurso não existir, ele poderá ser criado.

- `PATCH` — Tem a mesma semântica do `PUT`, exceto que podemos passar apenas a parte das informações que desejamos alterar no recurso, que deve existir. Esse método é mais recente e ainda não é suportado em todos os servidores.

- `DELETE` — Remove o recurso. Por exemplo, `DELETE /livro/321` remove o livro de `id 321`. Também é uma operação idempotente, ou seja, remover duas vezes o recurso deve ter o mesmo efeito de fazer isso uma única vez.

- `HEAD` — Parecido com o `GET`, mas retorna apenas os headers da resposta.
- `OPTIONS` — Retorna as operações possíveis e, possivelmente, algumas metainformações do recurso.
- `TRACE` — Para fazer *debugging*.

Pensando na web "humana", dificilmente usamos todos esses métodos, principalmente porque, pela especificação do HTML, os navegadores só suportam os métodos `GET` e `POST`. Mas mesmo usando esses dois métodos, é bastante importante usarmos as suas semânticas: `GET` para operações que não modificam recursos, e `POST` para as que modificam, assim os intermediários podem trabalhar da maneira correta.

O protocolo HTTP vai muito além de apenas métodos e recursos. E se estamos implementando um serviço que usa as características do HTTP, dizemos que é um serviço REST (ou *RESTful web service*). Isso vai desde usar métodos e recursos da maneira correta até fazer negociação de conteúdo e usar hipermídia.

É um assunto bastante extenso, que não será abordado em detalhes neste livro, mas podemos dar os primeiros passos nessa direção com a ajuda do VRaptor.

7.7 USANDO MÉTODOS E RECURSOS DA MANEIRA CORRETA

Quando criamos um controller no VRaptor, existe uma convenção que gera a URL de cada método desse controller. Por

exemplo, no nosso `LivrosController`:

```
@Controller
public class LivrosController {
   public void lista() {...}
   public void salva(Livro livro) {...}
   public void edita(String isbn) {...}
}
```

As URLs dos métodos `lista`, `salva` e `edita` terminam em `/livros/lista`, `/livros/salva` e `/livros/edita`, respectivamente. É bastante usual darmos os nomes dos métodos de um objeto usando verbos, afinal, estamos executando ações. Mas, pensando em REST, URLs (ou URIs) deveriam denotar recursos, que são substantivos, e não verbos.

Entretanto, se estamos usando substantivos, onde ficam as ações? Nos métodos (ou verbos) HTTP.

No método `lista`, estamos retornando as informações de todos os livros, então o recurso pode ser `/livros`. A lista não altera nada no sistema, portanto, o método HTTP é o `GET`. Para alterarmos a URL de um método do controller, podemos usar a anotação `@Path` do VRaptor:

```
@Path("/livros")
public void lista() {...}
```

A partir do momento em que fazemos isso, a URL `/livros/lista` já não corresponde mais a esse método, que deverá ser acessado por `/livros` somente. Por padrão, o VRaptor aceita qualquer verbo HTTP nos métodos (mesmo nos que não tem o `@Path`), logo, se quisermos indicar que esse método só pode ser acessado via `GET`, usamos a anotação `@Get`:

```
@Get @Path("/livros")
```

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

As anotações de métodos HTTP também recebem a URI como parâmetro. Podemos fazer simplesmente:

```
@Get("/livros")
public void lista() {...}
```

Já o método `salva` acrescenta um `Livro` novo ao sistema, assim, não podemos usar o `GET`. O recurso ainda é a lista dos livros do sistema, então podemos usar `POST /livros` para acessar esse método. Assim como temos o `@Get` para requisições `GET`, usamos o `@Post` para requisições `POST`:

```
@Post("/livros")
public void salva(Livro livro) {...}
```

Repare que tanto o método `lista` quanto o método `salva` usam a mesma URI, mas selecionamos qual método será invocado dependendo do método HTTP da requisição.

No método `edita`, não estamos mais tratando de todos os livros do sistema, mas sim de um livro específico. Para indicar isso em uma URI, precisamos que ela contenha uma informação que identifique unicamente o livro, no nosso caso, o `isbn`. Esse método mostra um formulário com os dados do `Livro` para a edição, então não estamos alterando nada no sistema, ou seja, podemos usar o método `GET`.

Como podemos editar qualquer um dos livros do sistema, precisamos de uma URI para cada um deles. Não é viável declararmos todas as URIs possíveis no método do controller. Precisamos declarar um *template* de URI que vai cair no método `edita`, ou seja, uma URI que contém uma variável no meio, que será o ISBN do livro.

Para declarar uma variável no meio de uma URI no VRaptor, usamos `{}` com o nome da variável capturada dentro. No caso do método `edita`, ficaria:

```
@Get("/livros/{isbn}")
public void edita(String isbn) {...}
```

Desse modo, ao acessarmos a URI `/livros/12345`, o VRaptor invocará o método `edita`, capturando o valor `12345` na variável `isbn`. Como o método recebe um parâmetro com esse nome, o valor `12345` será passado para o método.

Como o `isbn` faz parte da URI, não é possível invocá-lo via HTTP sem passar um `isbn`, assim não precisamos verificar se esse parâmetro foi passado mesmo na requisição. Nosso método `edita` está assim:

```
@Get("/livros/{isbn}")
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.notFound();
  } else {
    result.include(livroEncontrado);
    result.of(this).formulario();
  }
}
```

O que acontece se passarmos um `isbn` que não existe, por exemplo `GET /livros/bazinga`? O mundo inteiro já conhece uma resposta apropriada para um endereço que não existe na web: a famosa página 404. Esse número nada mais é do que um dos status codes possíveis do protocolo HTTP, e significa que o recurso não foi encontrado. É exatamente o nosso caso, por isso que simplesmente podemos redirecionar para a página 404 do sistema caso um livro com o ISBN passado não for encontrado, usando o

método `notFound` do `Result`.

Se tivermos um método `altera` no controller, podemos também receber o identificador do objeto na URI, já populando o objeto, usando a mesma convenção de nomes que a dos parâmetros de formulário. Nesse caso, como estamos alterando os dados do `Livro`, podemos usar o método `PUT`:

```
@Put("/livros/{livro.isbn}")
public void altera(Livro livro) {...}
```

Assim, o `livro` virá com o `isbn` populado com o valor passado na URI, e o resto dos atributos populados usando o corpo da requisição. Da mesma forma, se quisermos remover o livro, podemos usar o método `DELETE`:

```
@Delete("/livros/{livro.isbn}")
public void remove(Livro livro) {...}
```

Passando a pensar nas URIs como recursos fica muito mais simples para um cliente do sistema consumir os serviços web. Não é mais necessário conhecer a URI de todos os métodos do serviço, somente conhecer as correspondentes aos recursos é o suficiente.

Sabendo que temos o recurso `/livros`, o cliente sabe que `GET /livros` significa obter os dados dos livros, e `POST /livros` significa criar um livro novo. Se o recurso é um livro específico, `/livros/1234`, o cliente sabe que: para obter os dados do livro, deve usar `GET`; para editar os dados, deve usar `PUT` (ou `PATCH`, se for editar apenas alguns campos); e para remover o livro, deve usar `DELETE`.

Alterando o link para a edição de livros

Na listagem de livros, estamos usando o seguinte link:

```
<a href="${linkTo[LivrosController].edita }?isbn=${livro.isb
n}">
  Modificar
</a>
```

Usando o recurso de uma maneira REST, estamos passando o ISBN diretamente na URL, e não como parâmetro. Nesse caso, podemos passar parâmetros para o método do controller que serão usados na formação da URL. Fazemos isso usando colchetes em vez de parênteses:

```
<a href="${linkTo[LivrosController].edita(livro.isbn)}">
  Modificar
</a>
```

Essa passagem de parâmetros só será efetiva se ele for usado diretamente na URL.

Resolvendo conflitos de rotas

Ao criar a rota do método edita, usamos `@Get("/livros/{isbn}")` . O problema é que a rota para o método formulário é, implicitamente, `@Path("/livros/formulario")` , que também casa com o padrão `@Get("/livros/{isbn}")` , com o ISBN valendo `formulario` .

Isso, a princípio, vai gerar um 404, pois se o método formulário não tem nada anotado, a sua rota tem menos prioridade. Mas caso

ele estivesse anotado com `@Get("/livros/formulario")`, receberíamos um erro:

```
There are two rules that matches the uri '/livros/formulario'
  with method GET:
[[FixedMethodStrategy:
  /livros/formulario LivrosController.formulario() [GET]],
[FixedMethodStrategy:
  /livros/{isbn} LivrosController.edita(String) [GET]]]
with same priority. Consider using @Path priority attribute.
```

Ou seja, ele fala que o método `formulario` e o método `edita` conseguem tratar a URI `/livros/formulario`, com a mesma prioridade. Na última frase, ele dá a dica de usar o atributo `priority` da anotação `@Path` para resolver essa ambiguidade. Para isso, devemos mudar o método `edita` para:

```
@Get @Path(value="/livros/{isbn}", priority=Path.LOWEST)
public void edita(String isbn) {...}
```

Assim, essa rota tem a menor prioridade, e a `/livros/formulario` será usada para o método `formulario` do controller. Caso o ISBN fosse um número, essa configuração não seria necessária:

```
@Get("/livros/{isbn}")
public void edita(Long isbn) {...}
```

Nesse caso, como o VRaptor sabe que o ISBN é um número (`Long`), ele sabe que a última parte do caminho tem de conter apenas dígitos. Dessa forma, `/livros/1234` cai no método `edita`, mas `/livros/abcd` dá 404 direto.

7.8 USANDO REST NO NAVEGADOR

Pensando em REST, estamos facilitando a vida do cliente que

vai consumir os serviços da nossa aplicação. Mas nem sempre esses clientes são outras aplicações. Quando uma pessoa acessa o sistema usando um navegador, ela é o cliente do sistema, usando o navegador como cliente HTTP. Por isso, podemos aplicar as ideias do REST à web humana também, para implementarmos as interações com o usuário usando HTML e JavaScript.

Uma limitação forte dos navegadores é que só conseguimos fazer requisições `GET` e `POST` nativamente. Isso significa que, se quisermos fazer um formulário para alterar um livro, não conseguimos fazer:

```
<form action="/livros/1234" method="PUT">
```

Mas para simular o método `PUT`, o VRaptor (assim como alguns outros frameworks web) suporta receber um parâmetro a mais na requisição, indicando qual é o método real que queremos executar. No VRaptor, devemos deixar o formulário como `POST` e passar um parâmetro chamado `_method` com valor `PUT`:

```
<form action="/livros/1234" method="post">
  <input type="hidden" name="_method" value="PUT"/>
```

CAPÍTULO 8

DOWNLOAD E UPLOAD DE ARQUIVOS

8.1 ENVIANDO ARQUIVOS PARA O SERVIDOR: UPLOAD

Quando implementamos o cadastro de livros, ficou faltando uma parte importante para podermos mostrar os livros na nossa vitrine virtual: a capa dos livros. Ela será uma imagem cujo upload vamos fazer para o servidor, para depois podermos baixá-la.

Precisamos colocar um campo no formulário do livro para incluirmos a capa. Como esse formulário conterá um campo para upload, precisamos mudar o enctype para `multipart/form-data` . Além disso, criaremos o campo de upload como uma tag `<input>` com `type="file"` :

```
<form action="${linkTo[LivrosController].salva}"
    method="post" enctype="multipart/form-data">
  ...
  Capa: <input type="file" name="capa" />
  ...
</form>
```

Precisamos acrescentar a biblioteca `commons-fileupload` e a `commons-io` , para que o VRaptor consiga tratar uploads

corretamente. No nosso caso, em que estamos usando o Maven para gerenciar dependências, precisamos acrescentar as seguintes linhas no `pom.xml`, na seção de `<dependencies>`:

```
<dependency>
  <groupId>commons-fileupload</groupId>
  <artifactId>commons-fileupload</artifactId>
  <version>1.3.1</version>
</dependency>
```

Do lado do controller, para recebermos a capa, precisamos acrescentar um parâmetro a mais no método `salva`, com o mesmo nome do input e com o tipo `UploadedFile`:

```
public class LivrosController {

  //...
  @Post("/livros")
  public void salva(@Valid Livro livro, UploadedFile capa) {

  }

}
```

Esse `UploadedFile` nos dá o conteúdo do arquivo e algumas informações que podemos usar para salvá-lo corretamente:

```
capa.getFile(); // um InputStream com o conteúdo do arquivo
capa.getFileName(); // o nome do arquivo
capa.getContentType(); // o tipo do arquivo. Ex: image/png
capa.getSize(); // o tamanho do arquivo.
```

Para salvar esse arquivo, temos muitas opções. Podemos escolher uma pasta do servidor e salvar todas as capas nela. Para isso, podemos usar um servidor de arquivos, como um FTP, ou um serviço externo, como o Dropbox ou o S3 da Amazon. Podemos ainda salvar esse arquivo direto no banco de dados.

Para não se preocupar com isso do lado do controller, vamos

criar uma interface responsável por salvar arquivos, o `Diretorio`, onde poderemos salvar uma imagem e recuperá-la depois.

```
public interface Diretorio {

  URI grava(Arquivo arquivo);

  Arquivo recupera(URI chave);

}
```

O que vamos gravar é um `Arquivo` que guardará o conteúdo do arquivo junto com algumas metainformações e nos retornará uma `URI`, identificando o local onde o arquivo foi armazenado. Para recuperá-lo, basta passar essa `URI`, que recebemos de volta o `Arquivo` gravado. A classe `Arquivo` conterá o nome do arquivo, o conteúdo, o tipo e a data de modificação:

```
public class Arquivo {

  private String nome;
  private byte[] conteudo;
  private String contentType;
  private Calendar dataModificacao;

  public Arquivo(String nome, byte[] conteudo,
    String contentType, Calendar dataModificacao) {
    this.nome = nome;
    this.conteudo = conteudo;
    this.contentType = contentType;
    this.dataModificacao = dataModificacao;
  }

  //getters
}
```

Assim, podemos receber o `Diretorio` no construtor do `LivrosController` e guardar o arquivo, salvando a `URI` da imagem no `Livro`:

```
@Inject
public LivrosController(Estante estante, Diretorio imagens,
    Result result, Validator validator) {
  this.estante = estante;
  this.imagens = imagens;
  this.result = result;
  this.validator = validator;
}

@Transactional
@Post("/livros")
public void salva(@Valid Livro livro, UploadedFile capa)
    throws IOException {
  validator.onErrorRedirectTo(this).formulario();

  if (capa != null) {
    URI imagemCapa = imagens.grava(new Arquivo(
      capa.getFileName(),
      ByteStreams.toByteArray(capa.getFile()),
      capa.getContentType(),
      Calendar.getInstance()));

    livro.setCapa(imagemCapa);
  }

  estante.guarda(livro);

  result.include("mensagem", "Livro salvo com sucesso!");
  result.redirectTo(this).lista();
}
```

Para conseguirmos salvar corretamente a `URI` no `Livro`, vamos adaptá-la para salvar como `String`, usando o getter e o setter:

```
public class Livro {
  //...
  private String capa;

  public URI getCapa() {
    if (capa == null) return null;
    return URI.create(capa);
  }
```

```
  public void setCapa(URI capa) {
    this.capa = capa == null ? null : capa.toString();
  }
}
```

Para salvar de fato a imagem da capa, precisamos de uma implementação real do `Diretorio`. Para que não seja necessário criar uma infraestrutura adicional, vamos usar o banco de dados para salvar os dados da imagem. Criaremos o `DiretorioNoBD` que, para funcionar, é preciso adaptar a classe `Arquivo` para virar uma entidade:

```
@Entity
public class Arquivo {

  @Id @GeneratedValue
  private Long id;

  @Lob
  private byte[] conteudo;

  // para a JPA não reclamar
  Arquivo() {}

  // resto dos campos e getters
}

public class DiretorioNoBD implements Diretorio {

  private EntityManager manager;

  @Inject
  public DiretorioNoBD(EntityManager manager) {
    this.manager = manager;
  }
  @Deprecated DiretorioNoBD() {}

  @Override
  public URI grava(Arquivo arquivo) {
    return null;
  }
```

```
  @Override
  public Arquivo recupera(URI chave) {
    return null;
  }

}
```

Para implementar o método `grava`, vamos salvar o arquivo no banco de dados. A `URI` que vamos adotar como resposta será do tipo `bd://<id da imagem no bd>`, afinal, estamos gravando o arquivo no banco de dados. Assim, se o arquivo tiver o id 1234, a `URI` será `bd://1234`. O método `grava` ficaria:

```
@Transactional
@Override
public URI grava(Arquivo arquivo) {
  manager.persist(arquivo);

  return URI.create("bd://" + arquivo.getId());
}
```

Com isso, podemos acessar o formulário de livros e cadastrar a capa para alguns deles.

8.2 RECUPERANDO OS ARQUIVOS SALVOS: DOWNLOAD

Se quisermos mostrar a capa do livro na listagem, precisamos conseguir o conteúdo da imagem a partir da `URI` que salvamos. No HTML, para mostrar uma imagem, usamos a tag `img`:

```
<img src="<url da imagem>" />
```

Precisamos de uma URL da aplicação que nos retorne a capa do livro, e de um método de controller que vai retornar a imagem. Vamos, então, criar o método `capa` no `LivrosController`,

respondendo pela `URL /livros/<isbn>/capa` :

```
@Get("/livros/{isbn}/capa")
public void capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);
  // retorna a imagem
}
```

Agora, para retornar os dados da imagem, não basta incluir a imagem no `result` . Precisamos que a resposta inteira da requisição seja a imagem. Para isso, precisamos fazer o **download** dela. Para fazer um download usando o VRaptor, precisamos que o método retorne a interface `Download` :

```
public Download capa(String isbn) {
```

Existem algumas implementações de `Download` que podemos usar, de acordo com o que temos em mãos. No nosso caso, temos a classe `Arquivo` , que precisamos recuperar usando a `URI` salva no livro:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());
}
```

O `Arquivo` nos dá o conteúdo usando um `byte[]` , portanto, podemos usar a implementação `ByteArrayDownload` que, além do `byte[]` , também recebe o `ContentType` e o nome do arquivo:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());
```

```
  return new ByteArrayDownload(capa.getConteudo(),
      capa.getContentType(), capa.getNome());
}
```

Precisamos implementar o método `recupera` do nosso diretório de imagens, recuperando o arquivo, uma vez dada a `URI`. Para isso, podemos verificar se a URI é de banco de dados e buscar o `Arquivo` do banco pelo id:

```
@Override
public Arquivo recupera(URI chave) {
  if (chave == null) return null;

  // scheme é o protocolo. No caso de bd:// é o bd
  if (!chave.getScheme().equals("bd")) {
    throw new IllegalArgumentException(chave +
        " não é uma URI de banco de dados");
  }

  // authority é o que vem depois do bd://
  Long id = Long.valueOf(chave.getAuthority());
  return manager.find(Arquivo.class, id);
}
```

Assim, podemos modificar a listagem de livros para incluir a capa:

```
<c:forEach items="${livroList}" var="livro">
    <li>
      <img src="${linkTo[LivrosController].capa(livro.isbn) }"
            width="70" height="100">
      ${livro.titulo} - ${livro.descricao} -
    <a href="${linkTo[LivrosController].edita(livro.isbn) }">
      Modificar
    </a>
  </li>
</c:forEach>
```

Apesar de a tela renderizar sem problemas, se olharmos o log do servidor, veremos vários erros do tipo:

```
Caused by: java.lang.NullPointerException
  at br.com.casadocodigo.livraria.controlador.
      LivrosController.capa(LivrosController.java:89)
```

Isso porque alguns livros não possuem capa, logo, o arquivo retornado pelo `DiretorioNoBD` vem nulo. Para evitar esses erros, podemos retornar que a imagem não existe no controller, ou seja, retornar um 404:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());

  if (capa == null) {
    result.notFound();
    return null;
  }

  return new ByteArrayDownload(capa.getConteudo(),
      capa.getContentType(), capa.getNome());
}
```

No caso de mudarmos o `result` , podemos retornar `null` para bloquear o download. Dessa forma, a página renderiza sem problemas, e o console do servidor fica sem erros.

8.3 OUTRAS IMPLEMENTAÇÕES DE

DOWNLOAD

Além da implementação `ByteArrayDownload`, que consegue gerar um `Download` a partir de um `byte[]`, temos implementações que usam `InputStream` e `File` para gerar o `Download`. Todas elas possuem um construtor que recebe o conteúdo, o `ContentType` e o nome do arquivo:

```
byte[] capa = //...
return new ByteArrayDownload(capa, "image/png", "capa.png");

InputStream relatorio = //...
return new InputStreamDownload(relatorio, "application/pdf",
    "relatorio.pdf");

File notas = //...
return new FileDownload(notas, "text/plain", "notas.txt");
```

Por padrão, o arquivo é passado para o browser, que pode escolher entre mostrar a caixa de diálogo de download, ou simplesmente mostrar o arquivo dentro do browser. Para imagens, textos e PDFs, os browsers vão simplesmente mostrar os arquivos na tela.

Se quiser forçar o download, todas as implementações de `Download` possuem um construtor que recebe um boolean para forçá-lo. Por exemplo, para forçar o download das notas:

```
File notas = //...
return new FileDownload(notas, "text/plain", "notas.txt", true);
```

Se quisermos, podemos implementar `Download` para um tipo específico do nosso sistema. Por exemplo, se usarmos a classe `Arquivo` para representar arquivos em vários modelos do sistema, podemos criar uma implementação `ArquivoDownload`, delegando para o `ByteArrayDownload`:

```
public class ArquivoDownload implements Download {

  private Arquivo arquivo;
  public ArquivoDownload(Arquivo arquivo) {
    this.arquivo = arquivo;
  }

  public void write(HttpServletResponse response)
  throws IOException {
    Download download =
      new ByteArrayDownload(arquivo.getConteudo(),
          arquivo.getContentType(), arquivo.getNome());

    download.write(response);
  }
}
```

Em seguida, podemos simplificar o código do controller:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());

  if (capa == null) {
    result.notFound();
    return null;
  }
  return new ArquivoDownload(capa);
}
```

Podemos usar essa mesma ideia para gerar relatórios em PDF ou CSV, por exemplo, sem precisarmos nos preocupar com detalhes de como jogar esses arquivos na resposta da requisição, para que o browser faça o download ou simplesmente renderize diretamente o conteúdo.

CAPÍTULO 9

CUIDANDO DA INFRAESTRUTURA DO SISTEMA: INTERCEPTORS

Nos capítulos anteriores, vimos ferramentas que nos ajudam no dia a dia do desenvolvimento da nossa aplicação. Usaremos injeção de dependências, controllers, o `Result` e os JSPs durante todo o desenvolvimento, a cada nova funcionalidade do sistema.

Mas existem algumas tarefas que atingem todo o sistema, ou pelo menos uma grande parte dele. Tarefas como controle de acesso, controle de transações, log de erros etc. Neste capítulo, veremos como implementá-las usando interceptors.

9.1 EXECUTANDO UMA TAREFA EM VÁRIOS PONTOS DO SISTEMA: TRANSAÇÕES

Quando estamos trabalhando com alguns bancos de dados relacionais, por exemplo, o MySQL InnoDB ou o Oracle, só conseguimos realizar alguma mudança nos dados se estivermos dentro do ambiente controlado que garante a atomicidade, a consistência, o isolamento e a durabilidade (ACID) da operação: a **Transação**. A consequência disso é que, no nosso sistema, toda vez

que quisermos alterar os dados precisamos abrir e fechar (*commitar*) uma transação, para que a alteração realmente aconteça no banco.

Na seção *JPA dentro de um servidor de aplicação*, resolvemos esse problema pontualmente, no próprio DAO, em um caso em que não estejamos usando a JPA no servidor de aplicação:

```
@Override
public void adiciona(Livro livro) {
  this.manager.getTransaction().begin();
  this.manager.persist(livro);
  this.manager.getTransaction().commit();
}
```

Se seguirmos essa estratégia, a cada método do DAO que altera os dados, teríamos de repetir o código de iniciar transações. Além disso, esse código só está considerando o caminho feliz, em que o dado consegue ser salvo no banco.

Transações podem falhar por vários motivos, como violações de restrições do banco (como colunas `NOT NULL`), por referências inválidas, por falhas de conexão etc. Portanto, precisamos tratar esse caso e fazer o **rollback** da transação.

É um código bastante complicado e que estaria espalhado por todo o sistema. Para evitar essa duplicação de código, vamos pensar no papel do `LivroDAO` no sistema: acessar e modificar os dados de um livro no banco. O controle de transações é uma infraestrutura para que a modificação dos dados ocorra. Não faz parte da lógica que o `LivroDAO` deveria executar, pois não é sua responsabilidade. Além disso, se quisermos adicionar vários livros na mesma transação, já não podemos usar esse código.

Nesse caso, devemos **inverter o controle** e passar o código de

controle das transações para uma classe especializada que conseguirá cuidar disso por todo o sistema. No capítulo *Organização do código com injeção de dependências*, vimos uma das formas de inversão de controle: injeção de dependências.

No entanto, injeção de dependências não vai resolver o nosso problema, pois a transação não é um objeto que podemos receber no construtor, e sim um código que precisa ser executado **em volta** da nossa lógica de negócios: um **aspecto** da nossa aplicação.

Aspecto é todo requisito ou código de infraestrutura que precisamos executar em vários pontos da nossa aplicação, como controle de transação, controle de segurança, logging e auditoria. É um conceito bem importante que até gerou um termo quando usado: *Programação Orientada a Aspectos* (ou **AOP**). Na prática, não conseguimos desenvolver um sistema inteiro somente usando aspectos, mas, nos casos em que eles se aplicam, possibilitam soluções bastante elegantes.

O VRaptor possui uma ferramenta para implementar aspectos: os **interceptors**. Eles funcionam como os Servlet Filters, possibilitando executar alguma lógica antes e depois da requisição. A diferença é que, com os interceptors, podemos receber como dependência qualquer componente da aplicação, então podemos aproveitar o que já está pronto. Podemos pensar que um interceptor vai executar logo antes do método do controller e logo depois dele.

Para criar um interceptor, basta criar uma classe anotada com `@Intercepts`. No nosso caso, queremos criar um interceptor que cuidará das transações, o `TransacoesInterceptor`, no projeto `livraria-admin`.

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

@Intercepts
public class TransacoesInterceptor {

}
```

Precisamos executar um código que vai rodar em volta da requisição: abrir a transação no começo e *commitá-la* no final, se tudo der certo. Se acontecer alguma exceção, faremos o *rollback*. Nesse caso, podemos criar um método anotado com `@AroundCall`.

```
@Intercepts
public class TransacoesInterceptor {
  @AroundCall
  public void trataTransacao() {
     //inicia a transação
     //executa o controller
     //commita ou dá rollback
  }
}
```

Para cuidar das transações, precisamos de um `EntityManager`. Logo, devemos recebê-lo no construtor.

```
@Intercepts
public class TransacoesInterceptor implements Interceptor {

  private EntityManager manager;

  @Inject
  public TransacoesInterceptor(EntityManager manager) {
    this.manager = manager;
  }
  @Deprecated TransacoesInterceptor() {}

  @AroundCall
  public void trataTransacao() {

    //abre a transação antes da execução
    manager.getTransaction().begin();
```

```
    //continua a execução do método do controller
    // como?

    //commita a transação após a execução
    manager.getTransaction().commit();
  }

}
```

Repare que precisamos falar para o VRaptor executar o método do controller entre o *begin* e o *commit*. Para isso, podemos receber no método `trataTransacao` um `SimpleInterceptorStack`, que representa a pilha de interceptors, e chamar o método `next()`:

```
@AroundCall
public void trataTransacao(SimpleInterceptorStack stack) {
  manager.getTransaction().begin();

  //continua a execução do método do controller
  stack.next();

  manager.getTransaction().commit();
}
```

Para completar a lógica da transação, precisamos fazer o *rollback* caso haja algum problema. Um dos jeitos de fazer isso é envolvendo o código por um `try..finally`. E se, ao final da execução, a transação continuar ativa, é porque ela não conseguiu ser *commitada* com sucesso, portanto precisamos fazer o `rollback()`:

```
@AroundCall
public void trataTransacao(SimpleInterceptorStack stack) {
  try {
    manager.getTransaction().begin();

    stack.next();
```

```
    manager.getTransaction().commit();
  } finally {
    if (manager.getTransaction().isActive()) {
      manager.getTransaction().rollback();
    }
  }
}
```

Poderíamos também fazer um `catch(Exception e)` nesse bloco e, só então, fazer o *rollback*. Mas esse interceptor não tem condição de tratar a exceção e acabaria apenas relançando-a.

Uma boa regra é, se a classe não vai fazer nada de útil com a exceção, além de relançá-la, é melhor nem a capturar. Desse modo, podemos passar essa responsabilidade de tratar a exceção para outra classe (um outro interceptor talvez), ou simplesmente deixar aparecer a página de erro 500 para o usuário.

9.2 CONTROLANDO OS MÉTODOS INTERCEPTADOS

Se não falarmos o contrário, o `TransacoesInterceptor` passará por **todos** os métodos de controller, ou seja, todas as requisições web. Mas nem sempre isso é o desejável.

Embora essa solução seja suficiente para um sistema pouco acessado, para um sistema com muitos acessos, criar transações a cada requisição pode ser um problema e deixá-lo mais lento do que deveria.

Transações são necessárias para realizar operações que modificam o banco de dados, mas não são obrigatórias quando fazemos apenas operações de leitura. Nem são necessárias quando estamos apenas mostrando um formulário.

Por esse motivo, precisamos implementar um método anotado com `@Accepts` que retorne `boolean` indicando se o método precisa ou não de transação. Esse método deve receber um `ControllerMethod`, que representa o método do controller que será executado nessa requisição. Essa interface nos possibilita acessar:

- **O método:** `method.getMethod()` retorna um `java.lang.reflect.Method`, que é a classe que representa um método Java. Com ele, podemos verificar atributos como nome, parâmetros e notações presentes no método.
- **O controller:** `method.getController().getType()` retorna um `Class<?>`, que é a classe que representa uma classe Java. Com ele, podemos verificar atributos como nome, pacote e anotações presentes na classe.
- **Alguma anotação:** `method.containsAnnotation(UmaAnotacao.class)`, retorna um `boolean` dizendo se o método contém a anotação indicada.

Uma forma de implementar esse método `@Accepts` é listar todos os métodos de controller que queremos interceptar e verificar se o `ControllerMethod` é um deles. Mas essa lista pode ficar bastante extensa à medida que o projeto vai crescendo, tornando-o bem difícil de manter e evoluir.

Uma maneira de melhorar esse registro é criando uma anotação para indicar que o método será interceptado — nesse caso, se o método deverá ser executado dentro de uma transação. Podemos criar a anotação `@Transacional` para esse fim.

Para isso, primeiramente devemos criar a classe da anotação, usando a palavra chave `@interface` :

```
public @interface Transacional {

}
```

Em seguida, precisamos dizer em que lugares essa anotação será válida, usando a anotação `@Target` . Os lugares válidos vão de parâmetros e construtores até classes e pacotes. No nosso caso, só precisamos dessa anotação para métodos, então escolhemos esse valor na configuração do `@Target` :

```
@Target(ElementType.METHOD)
public @interface Transacional {

}
```

Outra configuração necessária é a retenção da anotação, ou seja, em que momento ela poderá ser lida. Isso é definido com a anotação `@Retention` , cujos valores possíveis são:

- `SOURCE` , com o qual a anotação só vale como documentação;
- `CLASS` , com que a anotação pode ser lida durante o processo de compilação;
- `RUNTIME` , com o qual a anotação pode ser lida durante a execução da aplicação.

Como precisamos ler essa anotação no interceptor, usamos a retenção de `RUNTIME` .

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Transacional {

}
```

Agora podemos anotar os métodos desejados, por exemplo, o método `salva` do `LivrosController`:

```
@Transacional
@Post("/livros")
public void salva(Livro livro) {...}
```

E para implementar o método `@Accepts`, podemos usar o método `containsAnnotation` do `ControllerMethod`, passando a anotação `Transacional`:

```
@Accepts
public boolean ehTransacional(ControllerMethod method) {
  return method.containsAnnotation(Transacional.class);
}
```

Assim, todos os métodos de controller que estiverem anotados com `@Transacional` serão interceptados pelo `TransacoesInterceptor` e, portanto, rodarão dentro de transações. Essa é a maneira mais fácil de configurar os métodos interceptados. Como esse é um caso comum, o VRaptor facilita ainda mais esse uso, com a anotação `@AcceptsWithAnnotations`.

Assim, em vez de implementar o método `@Accepts`, anotamos a classe do interceptor, indicando as anotações por onde ele deve passar:

```
@Intercepts
@AcceptsWithAnnotations(Transacional.class)
public class TransacoesInterceptor {
  @AroundCall
  public void trataTransacao(...){...}
}
```

No caso específico de transações, podemos usar a nossa interface REST para definir quais métodos podem ser interceptados. Se usamos os métodos HTTP corretamente,

sabemos que as requisições `GET` não podem modificar os dados do servidor, então não devem rodar dentro de transações.

Por outro lado, as requisições `POST`, `PUT` e `DELETE` supostamente modificam o estado de algum recurso. Portanto, devem rodar dentro de transação, dispensando a criação de uma nova anotação para esse fim:

```
@Intercepts
@AcceptsWithAnnotations({Post.class, Put.class, Delete.class})
public class TransacoesInterceptor {
  @AroundCall
  public void trataTransacao(...){...}
}
```

USANDO INTERCEPTORS PARA REDIRECIONAR A REQUISIÇÃO

Os interceptors não são obrigados a chamar o `stack.next()`, que continua a execução deles até executar o método do controller. Podemos, por exemplo, fazer um interceptor de segurança, que redireciona para a página de login caso o usuário não esteja logado:

```
@AroundCall
public void intercept(SimpleInterceptorStack stack) {
  if (usuarioEstaLogado()) {
    stack.next();
  } else {
    result.redirectTo(LoginController.class).login();
  }
}
```

Ordem relativa entre interceptors

A ordem em que os interceptors são executados é determinada pela ordem em que eles são lidos pelo VRaptor. Essa ordem não é fixa, nem tem regra predefinida. Isso funciona bem na maioria dos casos, em que os interceptors são independentes entre si, mas caso um precise necessariamente rodar após o outro, é possível usar os atributos `before` e `after` para forçar uma ordem.

```
@Intercepts(before=LoggerInterceptor.class,
            after=ExceptionInterceptor.class)
public class TransacoesInterceptor {
  ...
}
```

Nesse caso, a ordem de execução dos interceptors será: `ExceptionInterceptor -> TransacoesInterceptor -> LoggerInterceptor`.

Outro exemplo de interceptor: controle de usuários

No nosso sistema, o `livraria-admin` cuidará dos cadastros dos livros que vão ser vendidos no site. Esse cadastro pode alterar totalmente como um livro vai ser apresentado no site, logo, só pode ser feito por pessoas qualificadas para tal. Por esse motivo, vamos controlar o acesso de pessoas em geral a ele, criando uma forma de elas se identificarem para realizarem as operações desejadas.

Esse controle de acesso geralmente se dá em dois passos:

primeiro, precisamos saber quem é o usuário que está acessando o sistema e, depois, saber se ele tem permissão de realizar a operação requerida. Chamamos esses dois passos de **Autenticação** e **Autorização**, respectivamente.

A base para esse controle de acesso é criar uma classe que representa o usuário do sistema:

```
public class Usuario {

  private String login;
  private String senha;
  // outros atributos, getters e setters

}
```

Dessa forma, identificamos o usuário que está interagindo com o sistema. Outra parte importante é conseguir saber se ele tem permissão para determinadas operações. Esse controle pode ser tão complexo quanto se quiser, mas no nosso caso teremos só dois tipos de usuários: **admins**, que podem alterar os livros; e **não admins**, que só podem visualizá-los.

Para isso, vamos acrescentar um campo `admin` na classe `Usuario`:

```
public class Usuario {

  private String login;
  private String senha;

  private boolean admin;
  // outros atributos, getters e setters

}
```

Ao tentar acessar o sistema, precisamos pedir ao usuário que se identifique para continuar, ou seja, que faça o login. Como não

queremos ficar pedindo a toda requisição para que ele se logue, precisamos guardar os seus dados na sessão do usuário, após o login. Para isso, podemos usar um componente de escopo de sessão, representando o usuário logado no sistema:

```
@SessionScoped
public class UsuarioLogado implements Serializable {

  private Usuario usuario;

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

  public boolean isLogado() {
    return this.usuario != null;
  }

  public Usuario getUsuario() {
    return usuario;
  }

  public void desloga() {
    this.usuario = null;
  }
}
```

Dessa forma, podemos criar um controller para apresentar o formulário de login e realizar a autenticação. Esse controller buscará o usuário em algum lugar, como por exemplo, no banco de dados ou no LDAP.

Para abstrair o lugar onde ficarão os usuários, vamos usar a interface `RegistroDeUsuarios` . Após encontrar o usuário, vamos guardar as suas informações no `UsuarioLogado` .

```
@Controller
public class LoginController {

  private RegistroDeUsuarios usuarios;
```

```
  private UsuarioLogado logado;
  private Result result;
  private Validator validator;

  @Inject
  public LoginController(RegistroDeUsuarios usuarios,
                         UsuarioLogado logado,
                         Result result,
                         Validator validator) {
    this.usuarios = usuarios;
    this.logado = logado;
    this.result = result;
    this.validator = validator;
  }
  @Deprecated LoginController() {}

  @Get("/login")
  public void formulario() {}

  @Post("/login")
  public void login(String login, String senha) {
    Usuario usuario = usuarios.comLoginESenha(login, senha);
    validator.ensure(usuario != null, new I18nMessage("usuario",
                                  "login.ou.senha.invalidos"));
    validator.onErrorRedirectTo(this).formulario();

    logado.loga(usuario);

    // ou a página inicial
    result.redirectTo(LivrosController.class).lista();
  }

  @Get("/logout")
  public void logout() {
    logado.desloga();
    result.redirectTo(this).formulario();
  }
}
```

Vou omitir aqui qual vai ser a implementação do `RegistroDeUsuarios`. Use a mais apropriada para a sua situação. Vou omitir também o formulário de login, por brevidade.

Com nosso `LoginController` criado, precisamos redirecionar para ele sempre que um usuário tentar acessar o cadastro de livros. Poderíamos ir em cada um dos métodos e verificar se o usuário está logado, mas já temos uma ferramenta para fazer esse tipo de código: os **interceptors**.

Nesse interceptor, se o usuário não estiver logado, vamos redirecioná-lo para o formulário de login. Para isso, precisaremos do `UsuarioLogado` e do `Result`.

```
@Intercepts
public class AutenticacaoInterceptor {

  private UsuarioLogado usuario;
  private Result result;

  @Inject
  public AutenticacaoInterceptor(UsuarioLogado usuario,
  Result result) {
    this.usuario = usuario;
    this.result = result;
  }
  @Deprecated AutenticacaoInterceptor() {}

  @AroundCall
  public void autentica(SimpleInterceptorStack stack) {
    if (usuario.isLogado()) {
      stack.next();
    } else {
      result.redirectTo(LoginController.class).formulario();
    }
  }
}
```

Falta implementar o método `@Accepts`. Para isso, poderíamos criar uma anotação `@Restrito`, por exemplo, para identificar as páginas que precisam de login, ou criar uma anotação `@Liberado` para as páginas que não precisam de login, dependendo do número de páginas que serão controladas.

No nosso caso, o admin inteiro vai precisar de login, então o `accepts` seria `true`. Mas, assim, até o formulário de login passaria no interceptor e nunca seria executado. Vamos, então, aceitar todos os controllers, menos o de login:

```
@Accepts
public boolean ehRestrito(ControllerMethod method) {
  return !method.getController().getType()
                                .equals(LoginController.class);
}
```

Com isso, nosso interceptor de autorização já está pronto. Precisamos ainda ver se o usuário logado tem permissão de executar a operação desejada. Para isso, vamos criar outro interceptor, que cuida da autorização.

Esse interceptor precisa obrigatoriamente rodar depois do `AutenticacaoInterceptor`. E caso o usuário não tenha permissão, redirecionar para uma página falando isso. O HTTP já tem um status de erro para isso, o *401 — Unauthorized*, vamos usá-lo.

```
@Intercepts(after=AutenticacaoInterceptor.class)
public class AutorizacaoInterceptor {

  private UsuarioLogado usuario;
  private Result result;

  @Inject
  public AutorizacaoInterceptor(UsuarioLogado usuario,
  Result result) {
    this.usuario = usuario;
    this.result = result;
  }
  @Deprecated AutorizacaoInterceptor() {}

  @Accepts
  public boolean ehRestrito(ControllerMethod method) {
    return !method.getController().getType().
```

```
                equals(LoginController.class);
  }

  @AroundCall
  public void autoriza(SimpleInterceptorStack stack,
                       ControllerMethod method) {
    if (podeAcessar(method)) {
      stack.next();
    } else {
      result.use(Results.http()).sendError(401,
                                "Você não está autorizado!");
    }
  }

  private boolean podeAcessar(ControllerMethod method) {
    return // ???
  }
}
```

Precisamos definir a estratégia de autorização no método `podeAcessar`. Para facilitar, poderíamos criar uma anotação `@Admin` para lógicas que só podem ser feitas por administradores, ou `@Livre` para lógicas que podem ser acessadas por todos, dependendo da quantidade de lógicas que caem em cada caso. Ou ainda, podemos usar uma convenção: tudo que é `@Get` é liberado para todo mundo, pois não modifica nenhum dado (ou pelo menos não deveria), e todo o resto só é acessível por admin.

Para facilitar, vamos adotar essa convenção: o usuário pode acessar o método se ele for `@Get`; ou se ele for `admin`, que pode acessar tudo:

```
public boolean podeAcessar(ControllerMethod method) {
  return method.containsAnnotation(Get.class) ||
    usuario.getUsuario().isAdmin();
}
```

Dessa forma, conseguimos controlar o acesso ao cadastro de livros só para usuários logados, com as modificações aos livros

feitos somente por admins. Existem diversas ferramentas que auxiliam esse controle de acessos, como a especificação *JAAS* e o *Spring Security*, que cobrem todos os casos possíveis de autenticação e autorização, e já possuem formas padronizadas de autenticar com LDAP, AD e outras tecnologias.

Podemos integrá-las de forma transparente ao VRaptor, criando componentes que realizam essa integração. Podemos também usar os plugins `vraptor-auth` e `SACI-VRaptor`, que podem ser acessados a partir do diretório de plugins do VRaptor: https://github.com/caelum/vraptor-contrib.

Mais um exemplo de interceptor: auditoria

Até agora, vimos interceptors que precisam controlar de alguma forma se o método do controller vai ser executado ou não. Mas nem sempre precisamos desse tipo de controle.

Por exemplo, quando precisarmos apenas logar a cada acesso ao sistema, para fins de auditoria, sabendo qual foi o usuário que acessou. Nesse caso, podemos usar outro tipo de método de um interceptor, o `@BeforeCall`:

```
@Intercepts
public class AuditoriaInterceptor {
  private @Inject Logger logger;
  private @Inject Usuario usuario;
  private @Inject HttpServletRequest request;

  @BeforeCall
  public void loga() {
    logger.info("O usuário {} acessou {}",
      usuario.getNome(),
      request.getRequestURI());
  }
}
```

Assim, o método `loga` será chamado logo antes da execução do método do controller, sem a necessidade de receber e gerenciar a `stack` . Da mesma forma, podemos usar um método anotado com `@AfterCall` , que será executado após a chamada ao método do controller.

Capítulo 10

MELHORANDO O DESIGN DA APLICAÇÃO: CONVERSORES E TESTES

Na Orientação a Objetos, temos um objeto como uma abstração que une dados (atributos) a comportamentos (métodos). Mas quando estamos criando uma aplicação que salva dados no banco de dados, tendemos a criar muitas classes que não têm nenhum comportamento: apenas um punhado de atributos com getters e setters, que são apenas acessores desses atributos, e o mapeamento da JPA.

Quando fazemos isso, algumas lógicas ficam espalhadas, muitas vezes em classes `*Util` , quando poderíamos agrupá-las com os dados em que elas são aplicadas em classes pequenas e especializadas. Veremos neste capítulo como tratar esse tipo de classe e integrá-la facilmente ao processamento da requisição.

Além disso, criar essas classes pequenas melhora o design da aplicação e deveria ser feito constantemente. Uma boa ferramenta para isso são os testes automatizados, que ajudam a identificar partes do sistema que estão ficando complexas demais. Se usamos o ciclo do *Test Driven Development/Design* (TDD), os próprios

testes ajudam a melhorar o design da aplicação. Também veremos neste capítulo como usar TDD em conjunto com os componentes do VRaptor.

10.1 POPULANDO OBJETOS COMPLEXOS NA REQUISIÇÃO: CONVERSORES

No capítulo *Crie o seu primeiro cadastro*, vimos como criar o cadastro de livros, com um formulário capaz de preencher todos os atributos de um `Livro`. Os atributos preenchidos foram:

```
public class Livro {

  private String isbn;
  private String titulo;
  private String descricao;
  private BigDecimal preco;
  private Calendar dataPublicacao;

}
```

Ao criar a classe `Livro`, escolhemos o tipo `BigDecimal` para representar o preço, pois ele consegue representar um valor decimal sem perder precisão. Mas será que esse `BigDecimal` é o suficiente?

Imagine que a nossa livraria cresceu e agora vende livros importados. Alguns dos livros virão dos Estados Unidos, com o preço em dólar, outros da Inglaterra, com o preço em libras, e outros ainda da França, com o preço em euro. Agora, ao salvar um livro no sistema com o preço valendo `15.90`, como sabemos se ele está em reais, dólares, libras ou euros?

Da maneira que representamos o preço na classe `Livro`, não conseguimos saber isso. Precisamos, portanto, guardar o valor da

moeda correspondente ao preço do livro. Para isso, vamos criar um `enum` com as moedas suportadas pelo sistema:

```
public enum Moeda {
  REAL, DOLAR, EURO, LIBRA
}
```

E adicionar um atributo que guarda a moeda na classe `Livro`:

```
public class Livro {
  //...
  private Moeda moeda;
  private BigDecimal preco;
  //...
}
```

Agora, ao consultarmos o preço do livro, conseguimos saber qual é a moeda olhando para o atributo `moeda`. Se quisermos somar o valor de vários livros, por exemplo, em um carrinho de compras, precisaremos considerar sempre o preço e a moeda.

Toda vez que olharmos para o preço do livro precisaremos olhar também para a moeda. E se, em algum lugar do código, esquecermos de fazer isso, podemos somar um preço em reais com um preço em euros, e chegar em um valor que não é nem uma coisa, nem outra!

Apesar de termos guardado a moeda na classe `Livro`, moeda não é um atributo do `Livro`, é um atributo do preço! Assim como não temos um atributo `mes` no livro e sim uma data de publicação que possui um mês, deveríamos ter um atributo `preco` no livro que, além de guardar o valor do preço, guardaria também a moeda. Para fazer isso, precisamos de uma classe que agrupa esses dois valores, a classe `Dinheiro`:

```
public class Dinheiro {
  private Moeda moeda;
```

```
  private BigDecimal montante;
}
```

E o preço da classe `Livro` passa a ser desse tipo:

```
public class Livro {

   //...
   //-private Moeda moeda;
   private Dinheiro preco;

}
```

Essa classe `Dinheiro` guarda dados, mas não é uma classe que salvaríamos no banco de dados como uma tabela. É uma classe que guarda um **valor**, nesse caso um valor em dinheiro, como por exemplo `R$ 29,90` . Dois objetos do tipo `Dinheiro` que têm a mesma `moeda` e o mesmo `montante` guardam exatamente o mesmo valor. Classes desse tipo são os chamados *Value Objects* (http://martinfowler.com/bliki/ValueObject.html), ou seja, classes que representam valores.

Ao criar a classe `Dinheiro` , podemos concentrar todas as operações monetárias dentro dela, como somar preços ou converter um preço de real para dólar. Também podemos fazer verificações adicionais, como não permitir somar preços de moedas diferentes.

Para popular o preço do livro no nosso sistema, agora precisaríamos de dois campos no formulário:

```
<li>Moeda: <br/>
  <input type="text" name="livro.preco.moeda"
      value="${livro.preco.moeda}"/></li>
<li>Montante: <br/>
  <input type="text" name="livro.preco.montante"
      value="${livro.preco.montante}"/></li>
```

Mas o livro não possui moeda e montante, ele possui um preço! O ideal é que o usuário possa digitar no campo de preço algo como "R$ 52,00", e o sistema conseguir salvar o preço correspondente.

Mas como fazer isso? Antes, vamos analisar os atributos da classe `Livro`.

```
public class Livro {

  private String isbn;
  private String titulo;
  private String descricao;
  private Calendar dataPublicacao;
  private Dinheiro preco;

}
```

Os primeiros atributos são `String`s e, para serem populados, basta pegar o parâmetro que vem da requisição, ou seja, o valor digitado no input. Neste caso, não importa o que foi digitado no input, os campos `String` serão populados com esse valor.

No entanto, os dois últimos valores são mais complexos: o `preco`, que é um `Dinheiro`, e a `dataPublicacao`, que é um `Calendar`. Quando queremos popular a data de publicação, digitamos um valor no campo correspondente do formulário, por exemplo, `17/06/2013`. Esse valor vai para o servidor como a `String "17/06/2013"`. Porém, para poder passar esse valor para a classe `Livro`, o VRaptor precisa converter `"17/06/2013"` em um `Calendar` que representa esse dia.

Precisamos que essa mesma lógica se aplique quando digitarmos `R$ 29,90` no campo "Preço" do formulário. Queremos que o VRaptor converta para o `Dinheiro` com moeda

`REAL` e montante `29.90`.

Isso é feito através de classes especializadas em converter os parâmetros da requisição em objetos que representam esses valores. Essas classes são `Converter`s, ou seja, implementam a interface `Converter` do VRaptor, e são responsáveis por converter uma `String` que veio da requisição em um tipo específico.

Mapeando a classe Dinheiro no Hibernate

Para mapear a classe `Dinheiro`, não vamos criar uma tabela separada, já que `Dinheiro` é um *Value Object*. Em vez disso, vamos mapeá-la dentro da tabela `Livro`, usando a anotação `@Embedded`:

```
@Entity
public class Livro {
  //...
  @Embedded
  public Dinheiro preco;
}
```

Assim, serão criadas duas colunas na tabela `Livro`, a `moeda` e o `montante`. Além disso, é necessário criar um construtor sem argumentos dentro da classe `Dinheiro`, mesmo que seja `protected`, para que o hibernate consiga instanciar essa classe a partir dos dados do banco.

O VRaptor possui um conjunto de converters já implementados, todos no pacote `br.com.caelum.vraptor.converter`. Eles são o bastante para

popular todos os tipos simples do Java: `String`, números, `boolean`, `enum` e datas.

Esses conversores padrão são suficientes para a maioria dos objetos que populamos na requisição, mas precisaremos criar um personalizado para a nossa classe `Dinheiro`. Para isso, devemos criar uma classe anotada `@Convert(Dinheiro.class)`, que implementa `Converter<Dinheiro>`.

```
import br.com.caelum.vraptor.Convert;
import br.com.caelum.vraptor.converter.Converter;

@Convert(Dinheiro.class)
public class DinheiroConverter
      implements Converter<Dinheiro> {

  @Override
  public Dinheiro convert(String value,
  Class<? extends Dinheiro> type) {
    return null;
  }

}
```

A interface `Converter` define um único método que deve pegar a String passada e converter no tipo desejado, no caso o `Dinheiro`. Para auxiliar a conversão, existe um segundo parâmetro, o `type`, que é a classe do tipo de destino, para ser usado caso estejamos convertendo para implementações de uma interface ou filhos de uma classe.

Para facilitar a implementação desse `Converter`, vamos usar TDD (*Test Driven Design*), criando a classe `DinheiroConverterTest` dentro de `src/main/test`. Para iniciar o ciclo do TDD, criaremos um teste que falha, usando **JUnit 4**:

```
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;

public class DinheiroConverterTest {
  @Test
  public void converteUmValorEmReais() {
    Converter<Dinheiro> converter = new DinheiroConverter();

    assertThat(converter.convert("R$ 1,00", null),
          is(new Dinheiro(Moeda.REAL, new BigDecimal("1.00"))));
  }
}
```

ONDE APRENDER TDD?

Recomendo fortemente que você aprenda TDD e adote a prática nos sistemas que desenvolver. Definitivamente, ele aumenta a qualidade do software escrito. Um excelente material é o livro escrito pelo Mauricio Aniche, disponível no site da Editora Casa do Código (http://www.casadocodigo.com.br).

Para testar, estamos usando a DSL (*Domain Specific Language*) do JUnit e os `Matchers` do Hamcrest. Nesse teste, estamos garantindo que a conversão do valor `"R$ 1,00"` seja o `Dinheiro` em real no valor de `1.00`.

Usando essa DSL, conseguimos ler essa frase anterior, em inglês, ou seja, temos um teste bastante legível. Ao rodar o teste (no Eclipse, `botão direito > Run As > JUnit Test`), vemos que ele falha:

```
java.lang.AssertionError:
Expected: is <br.com.casadocodigo.livraria.modelo
```

```
                                                .Dinheiro@692a3722>
    got: null
```

Essa saída fala que esperava um `Dinheiro` específico, mas veio `null`. O problema é que apenas usando essa mensagem não conseguimos saber qual é o valor em dinheiro que estamos esperando. Para corrigir isso, vamos implementar o `toString` da classe `Dinheiro`.

O formato desejado, para facilitar a leitura, pode ser `"Dinheiro(R$ 1.00)"`. Para não precisarmos ficar fazendo um monte de `if`s, podemos modificar a `enum` de `Moeda` para incluir o símbolo de cada constante:

```
public enum Moeda {
  REAL("R$"), DOLAR("US$"),
  EURO("€"), LIBRA("£");

  private final String simbolo;

  private Moeda(String simbolo) {
    this.simbolo = simbolo;
  }

  public String getSimbolo() {
    return simbolo;
  }
}

public class Dinheiro {
  //...
  @Override
  public String toString() {
    return String.format("Dinheiro(%s %s)",
        moeda.getSimbolo(),
        montante
      );
  }
}
```

Agora, ao rodarmos o teste, veremos a seguinte mensagem:

```
java.lang.AssertionError:
Expected: is <Dinheiro(R$ 1.00)>
     got: null
```

Para fazer o teste passar rapidamente, vamos simplesmente retornar o `Dinheiro` desejado:

```
@Convert(Dinheiro.class)
public class DinheiroConverter
      implements Converter<Dinheiro> {

  @Override
  public Dinheiro convert(
      String value,
      Class<? extends Dinheiro> type,
      ResourceBundle bundle) {

    return new Dinheiro(Moeda.REAL, new BigDecimal("1.00"));
  }

}
```

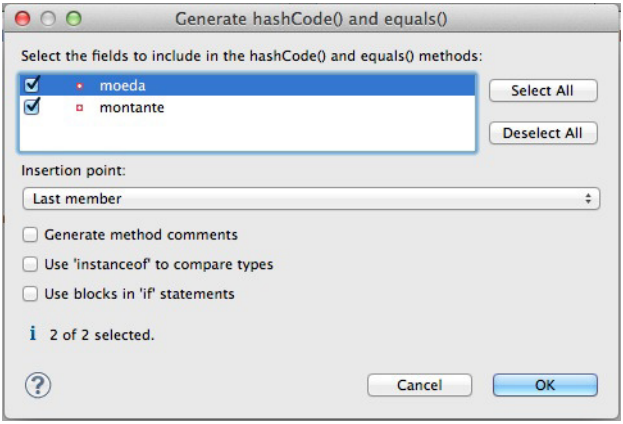
Agora, se rodarmos o teste, ele ainda falha, mas desta vez com essa mensagem:

```
java.lang.AssertionError:
Expected: is <Dinheiro(R$ 1.00)>
     got: <Dinheiro(R$ 1.00)>
```

Ora, esperamos um `Dinheiro` de `R$ 1.00` e apareceu um `Dinheiro` de `R$ 1.00`. O que deu errado?

Apesar de guardarem o mesmo valor, são dois objetos diferentes na memória, por isso o teste falha. Nossa classe `Dinheiro` é um *Value Object*, que é definido unicamente pelos seus atributos. Ou seja, um dinheiro é igual ao outro se tem a mesma moeda e o mesmo montante. E para dizer isso, precisamos implementar o `equals` e o `hashCode` da classe `Dinheiro`, de preferência gerando esses métodos com a ajuda da sua IDE. No

Eclipse, usamos o menu `Source > Generate hashCode() and equals()` e selecionamos os dois atributos.



Agora, ao rodarmos novamente o teste, ele passa! Completamos a segunda parte do ciclo do TDD, fazendo o teste passar.

Devemos passar para a próxima parte: a refatoração. Claro que retornar direto o `Dinheiro` de R$ 1.00 não é uma implementação razoável, então vamos modificá-la para converter valores em real. A primeira tentativa vai ser, se a string começar por `"R$"`, tentar criar um `BigDecimal` a partir dela.

```
public Dinheiro convert(String value, ...) {
  if (value.startsWith("R$")) {
    return new Dinheiro(Moeda.REAL,
      new BigDecimal(value.replace("R$ ", ""))
    );
  }
  return null;
}
```

Porém, ao rodarmos o teste, recebemos um erro:

```
java.lang.NumberFormatException
  at java.math.BigDecimal.<init>(BigDecimal.java:459)
  at java.math.BigDecimal.<init>(BigDecimal.java:728)
  at br.com.casadocodigo.livraria.conversor.DinheiroConverter
        .convert(DinheiroConverter.java:23)
```

Isso aconteceu porque o construtor do `BigDecimal` espera os números no formato `1.00`, e não `1,00`, como foi passado. Para resolver isso, vamos trocar toda vírgula por ponto:

```
@Convert(Dinheiro.class)
public class DinheiroConverter
      implements Converter<Dinheiro> {

  @Override
  public Dinheiro convert(
      String value,
      Class<? extends Dinheiro> type,
      ResourceBundle bundle) {

    if (value.startsWith("R$")) {
      return new Dinheiro(Moeda.REAL,
        new BigDecimal(
          value.replace("R$ ", "")
               .replace(',', '.')
        )
      );
    }
    return null;
  }
}
```

Ao rodarmos novamente o teste, vemos que ele passa e, se estivermos satisfeitos com a implementação, completamos a terceira parte do ciclo do TDD. Como é um ciclo, precisamos voltar para a primeira parte e escrever um teste que falha. Podemos escrever um teste para converter valores em outra moeda, por exemplo, em dólar.

```
@Test
public void converteUmValorEmDolares() {
```

```
  Converter<Dinheiro> converter = new DinheiroConverter();

  assertThat(converter.convert("US$ 49,95", null),
      is(new Dinheiro(Moeda.DOLAR, new BigDecimal("49.95"))));
}
```

Ao rodarmos o teste, ele falha com a mensagem:

```
java.lang.AssertionError:
Expected: is <Dinheiro(US$ 49.95)>
     got: null
```

A mensagem está clara o suficiente, então vamos modificar a implementação para fazer o teste passar. Uma implementação mais direta é copiar o `if` de reais e transformá-lo em dólar:

```
if (value.startsWith("R$")) {
  return new Dinheiro(Moeda.REAL,
    new BigDecimal(
        value.replace("R$ ", "")
             .replace(',', '.')
    )
  );
} else if (value.startsWith("US$")) {
  return new Dinheiro(Moeda.DOLAR,
    new BigDecimal(
        value.replace("US$ ", "")
             .replace(',', '.')
    )
  );
}
return null;
```

Ao rodarmos os testes, vemos que tanto o teste de reais quanto o de dólares passam. Podemos passar para a refatoração.

Se continuarmos com o código dessa maneira, teremos um `if` para cada moeda, com um código bem parecido dentro de cada `if`. As únicas coisas que mudam são a moeda e o símbolo, ambos acessíveis pelas constantes da enum `Moeda`. Vamos

modificar, então, para passar por todas as constantes da `enum` e testá-las:

```
for (Moeda moeda : Moeda.values()) {
  if (value.startsWith(moeda.getSimbolo())) {
    return new Dinheiro(moeda,
        new BigDecimal(
          value.replace(moeda.getSimbolo(), "")
               .replace(',', '.').trim()
        )
    );
  }
}

return null;
```

Para manter o comportamento anterior, adicionamos um `trim()` para remover os espaços em branco do começo da `String`. Ao rodar os testes, vemos que continuam passando, então completamos outro ciclo. Para começar outro ciclo, poderíamos testar outra moeda, mas o teste não falharia, já que estamos contemplando todas as moedas no converter.

Os casos felizes (*happy paths*) já foram implementados, agora precisamos pensar nos casos patológicos, por exemplo, quando se passa um valor que não é um dinheiro.

Do jeito que a implementação está, ela vai converter para `null` nesse caso. Mas se deixarmos assim, podemos mascarar erros. Quando a `String` que tentamos converter não é conversível para o tipo desejado, precisamos sinalizar com um erro.

A API de converters do VRaptor já possui um erro implementado para isso, o `ConversionException`. Essa exception recebe no construtor uma `Message`, que será

automaticamente adicionada como mensagem de validação. Podemos mostrá-la no formulário sem nenhum código adicional.

Vamos, então, começar outro ciclo do TDD, criando um teste que espera o erro de conversão:

```
@Test(expected=ConversionException.class)
public void lancaErroDeConversaoQuandoOValorEhInvalido() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  converter.convert("noventa pratas!", null);
}
```

Ao rodar o teste, recebemos a mensagem:

```
java.lang.AssertionError:
Expected exception: br.com.caelum.vraptor.converter
                                          .ConversionException
```

Para fazer o teste passar, precisamos lançar esse erro, caso não seja possível converter. Vamos usar a implementação `ConversionMessage`:

```
public Dinheiro convert(String value, ...) {
  for(Moeda moeda : Moeda.values()) {...}

  throw new ConversionException(
    new ConversionMessage("dinheiro.invalido", value));
}
```

Ao rodarmos os testes, vemos que eles passam. Para definir a mensagem real, precisamos colocar a chave `"dinheiro.invalido"` no `messages.properties`:

```
dinheiro.invalido = "{0}" não é um dinheiro válido
```

Agora sim, ao rodarmos os testes, vemos que todos eles passam. Outro caso em que vai falhar é quando a `String` começa com um símbolo, mas o número é inválido:

```
@Test(expected=ConversionException.class)
public void lancaErroDeConversaoQuandoOMontanteEhInvalido() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  converter.convert("R$ mil", null);
}
```

A mensagem de erro ao rodar o teste:

```
java.lang.Exception:
  Unexpected exception,
    expected<br.com.caelum.vraptor.converter.ConversionException>
    but was<java.lang.NumberFormatException>
...
Caused by: java.lang.NumberFormatException
...
```

Para corrigir, precisamos envolver a criação do `BigDecimal` em um `try..catch`, e lançar um `ConversionException` em vez de um `NumberFormatException`:

```
for (Moeda moeda : Moeda.values()) {
  if (value.startsWith(moeda.getSimbolo())) {
    try {
      return new Dinheiro(moeda,
          new BigDecimal(
            value.replace(moeda.getSimbolo(), "")
              .replace(',', '.').trim()
          )
      );
    } catch (NumberFormatException e) {
      throw new ConversionException(
          new ConversionMessage("dinheiro.invalido", value));
    }
  }
}
throw new ConversionException(
    new ConversionMessage("dinheiro.invalido", value));
```

Essa implementação faz os testes passarem, mas começou a ficar ruim. Vamos, portanto, refatorá-la, para completar o ciclo do TDD. Para isso, vou extrair a criação do `BigDecimal` para um método, levando junto o `try..catch`:

```
@Override
public Dinheiro convert(String value,
Class<? extends Dinheiro> type) {

  for (Moeda moeda : Moeda.values()) {
    if (value.startsWith(moeda.getSimbolo())) {
      return new Dinheiro(moeda, criaMontante(value, moeda));
    }
  }
  throw new ConversionException(
      new ConversionMessage("dinheiro.invalido", value));
}

private BigDecimal criaMontante(String value, Moeda moeda) {
  try {
    return new BigDecimal(
      value.replace(moeda.getSimbolo(), "")
        .replace(',', '.').trim()
    );
  } catch (NumberFormatException e) {
    throw new ConversionException(
        new ConversionMessage("dinheiro.invalido", value));
  }
}
```

Rodando os testes, eles continuam passando e poderíamos continuar refatorando, mas vamos parar por aqui. Um último caso que não cobrimos é quando o usuário não preenche o valor de dinheiro.

Nesse caso, a `String` vai vir vazia e, do jeito que está a implementação, lançará um `ConversionException` . Não é a resposta mais adequada — melhor seria retornar `null` , já que o campo não foi preenchido. Vamos adicionar um teste para isso:

```
@Test
public void converteStringVaziaEmNull() {
  Converter<Dinheiro> converter = new DinheiroConverter();

  assertThat(converter.convert("", null),
      is(nullValue()));
```

```
}
```

Esse teste falha com a mensagem:

```
br.com.caelum.vraptor.converter.ConversionException:
  "" não é um dinheiro válido
```

Para fazer o teste passar, vamos colocar essa condição de string vazia no começo do método `convert`:

```
public Dinheiro convert(String value, ...) {
  if (Strings.isNullOrEmpty(value)) { return null; }

  for (Moeda moeda : Moeda.values()) { ... }

  throw new ConversionException(...);
```

Os testes passam e esgotamos os casos que queremos suportar no converter. Com o apoio dos testes, agora conseguimos refatorar a classe `DinheiroConverter` sem medo. Fazer testes durante o desenvolvimento é essencial para que tenhamos segurança para evoluir o sistema sem quebrar comportamentos existentes.

Além disso, ao usarmos o ciclo do TDD, conseguimos criar classes com um design melhor, mais simples de usar. Mais ainda, ganhamos uma documentação viva sobre o que a classe faz: basta ler os casos de teste.

Para saber mais sobre TDD, existe o livro *Test Driven Design em Java*, por Maurício Aniche, pela Casa do Código, além das referências, em inglês, *TDD by Example* do Kent Beck, e *Growing Object Oriented Software Guided by Tests* de Nat Pryce.

10.2 TESTES DE UNIDADE EM PROJETOS QUE USAM VRAPTOR

Na seção anterior, vimos como criar testes usando TDD para a classe `DinheiroConverter`. Esse tipo de testes deveria ser feito para todas as classes do sistema, para garantir que cada unidade (classe, método, funcionalidade) funcione isoladamente. Como aplicar isso para as classes de um projeto que usa VRaptor?

Um dos grandes pilares do VRaptor é a injeção de dependências, que vimos em detalhes no capítulo *Organização do código com injeção de dependências*. Por causa disso, a maioria das classes do sistema não depende de nada do VRaptor: são componentes que dependem de outros componentes da aplicação. Se juntarmos isso à pratica de sempre depender de interfaces, temos classes independentes das implementações das suas dependências.

Com isso, ganhamos de graça uma melhor testabilidade das classes, principalmente se estivermos recebendo a injeção de dependências pelo construtor. Por exemplo, para testarmos a classe `AcervoNoAdmin` que criamos dentro do `livraria-site`, temos como dependência um `ClienteHTTP`:

```
public class AcervoNoAdmin implements Acervo {
  private ClienteHTTP http;

  @Inject
  public AcervoNoAdmin(ClienteHTTP http) {
    this.http = http;
  }

  public List<Livro> todosOsLivros() {...}
}
```

Essa classe faz uma requisição usando esse cliente HTTP e espera um resultado em XML, que é convertido para uma lista de livros. Para testá-la isoladamente, podemos passar uma

implementação falsa da interface `ClienteHTTP` que retorna um XML determinado:

```
@Test
public void converteUmaListaComApenasUmLivro() {
  ClienteHTTP http = new ClienteHTTP() {
    @Override
    public String get(String url) {
      return
        "<livros>" +
          "<livro>" +
            "<titulo>VRaptor 3</titulo>" +
            "<isbn>12345</isbn>" +
          "</livro>" +
        "</livros>";
    }
  };
  AcervoNoAdmin acervo = new AcervoNoAdmin(http);

  List<Livro> livros = acervo.todosOsLivros();

  assertThat(livros.size(), is(1));

  Livro livro = livros.get(0);
  assertThat(livro.getTitulo(), is("VRaptor 3"));
  assertThat(livro.getIsbn(),   is("12345"));
}
```

Para evitar a criação de classes para cada cenário de testes, podemos usar frameworks de **Mocks**, um dos nomes dados a essas implementações falsas usadas em testes. Um desses frameworks é o **Mockito** (http://mockito.org), que simplificaria o teste para:

```
import static org.mockito.Mockito.*;

//...

@Test
public void converteUmaListaComApenasUmLivro() {
  ClienteHTTP http = mock(ClienteHTTP.class);
  when(http.get(anyString())).thenReturn(
      "<livros>" +
```

```
        "<livro>" +
          "<titulo>VRaptor 3</titulo>" +
          "<isbn>12345</isbn>" +
        "</livro>" +
      "</livros>"
  );
  //...
}
```

Seguindo essa ideia, podemos fazer o teste da maioria dos componentes da aplicação, usando qualquer técnica de testes para objetos em geral. A exceção disso são os controllers, que geralmente dependem de `Result` e `Validator`. Essas interfaces são um pouco mais difíceis de mockar, pois têm uma interface fluente:

```
result.redirectTo(LoginController.class).formulario();
validator.onErrorRedirectTo(this).lista();
```

O mock de uma dessas chamadas, usando o mockito, seria algo como:

```
LoginController mockController = mock(LoginController.class);

when(result.redirectTo(LoginController.class)).
            thenReturn(mockController);

verify(mockController).formulario();
```

Se não fizermos pelo menos as duas primeiras linhas, a execução falha com uma `NullPointerException`. Se estivermos usando outro tipo de resultado, como XML ou JSON, a configuração dos mocks fica ainda mais difícil.

Para evitar isso, podemos usar as versões mockadas providenciadas pelo VRaptor, que já ignoram as chamadas do `result` e nos deixam inspecionar alguns tipos de dados. Esses mocks do VRaptor ficam no pacote

`br.com.caelum.vraptor.util.test` e os principais são:

- `MockResult` — Um `result` que ignora os redirecionamentos e o método `use`, enquanto possibilita a inspeção de todos os objetos incluídos:

```
MockResult result = new MockResult();

//dentro do controller:
result.include("mensagem", "Tudo deu certo!");

//dentro do teste:
assertThat(result.included("mensagem"),
           is("Tudo deu certo!"));
```

- `MockValidator` — Um validator que acumula as mensagens adicionadas e lança exceção no `validator.onErrorUse()`. Assim, podemos verificar que houve erros de validação:

```
MockValidator validator = new MockValidator();
LivrosController controller =
  new LivrosController(..., validator);

Livro livroSemTitulo = //...

try {
  controller.salva(livroSemTitulo);
  Assert.fail("Deveria ter dado erro de validação");
} catch (ValidationException e)
  assertThat(e.getErrors().size(), is(1));
  Message message = e.getErrors().get(0);
  assertThat(message.getCategory(), is("titulo"));
}
```

- `MockSerializationResult` — Igual ao `MockResult`, mas também permite inspecionar os dados de um objeto serializado:

```
MockSerializationResult result =
```

```
    new MockSerializationResult();

  //no controller
  result.use(Results.json()).from(livro).serialize();

  //no teste
  assertThat(result.serializedResult(), is(
    "{\"livro\": { \"titulo\": \"VRaptor 3\", ...  }}"
  ));
```

Desse modo, conseguimos também testar os controllers sem muitos problemas e manter uma boa cobertura de testes pela aplicação.

Capítulo 11

PRÓXIMOS PASSOS

Terminamos aqui o desenvolvimento da nossa livraria, que foi o projeto que usamos para acompanhar este livro e expor a maioria das funcionalidades do VRaptor. Mas não podemos parar por aqui, pois cada projeto que fazemos tem características diferentes, resolve problemas diferentes e exige coisas diferentes.

Para continuar a conhecer mais sobre o VRaptor, podemos seguir os seguintes passos:

- **Leia os apêndices deste livro**. Nos apêndices, temos uma visão geral dos plugins do VRaptor, que auxiliam o desenvolvimento de aplicações com ele, uma exposição de todos os containers de injeção de dependências suportados, exemplos de como usar AJAX e um pouco sobre seu funcionamento interno, com seus componentes e suas funções.

- **Leia a documentação do VRaptor**. Apesar de não estar completa, a documentação do VRaptor (http://vraptor.caelum.com.br) é uma boa referência para vermos suas funcionalidades. Além dela, existe o livro de receitas, com a solução para alguns problemas comuns do dia a dia.

- **Contribua com a documentação do VRaptor**. Colocar no papel (mesmo que seja digital) o que você conhece ajuda a consolidar o seu conhecimento e ajudar aos outros. Portanto, escreva um blog técnico com os problemas e soluções que você adotou, contribua com uma solução para o livro de receitas do VRaptor, ajude a melhorar a documentação existente, em https://github.com/caelum/vraptor4/tree/master/vraptor-site.

- **Participe da lista de discussões do VRaptor**. Cada vez que aparecer uma dúvida ou um problema durante o desenvolvimento das suas aplicações, ou mesmo se você tiver sugestões ou quiser chamar uma discussão, acompanhe e mande mensagens para o fórum da Casa do Código, em http://forum.casadocodigo.com.br/. Tentar resolver o problema de outras pessoas também ajuda bastante a conhecer novas formas de trabalhar com o VRaptor e aprender cada vez mais.

- **Crie e responda perguntas no GUJ**. O guj.com.br é o maior fórum de Java em português, então sempre que tiver uma dúvida que envolva não só o VRaptor, mas qualquer biblioteca que você estiver usando na aplicação, você pode usar o conhecimento das centenas de milhares de usuários do GUJ. Pesquise se já existe uma pergunta que responde a sua dúvida e, se não existir, crie uma nova pergunta. Não se esqueça de postar stacktraces, caso estejam acontecendo erros e, se possível, poste o pedaço do código onde você acha que esteja acontecendo o problema. Ganhe pontos e o respeito de outros desenvolvedores respondendo

perguntas!

- **Crie e responda tópicos no GUJ**. O guj.com.br possui uma seção de discussões, na qual está o fórum antigo do GUJ, que possui anos e anos de discussões já resolvidas e também novas que aparecem a cada dia. Da mesma forma que as perguntas, pesquise se já existe uma solução para o seu problema antes de postar a sua dúvida. Se o tópico é relacionado ao VRaptor, use o fórum de *Ferramentas e bibliotecas brasileiras*. Não se esqueça de retribuir o favor e responder a dúvida de outras pessoas!

- **Sugira novas funcionalidades e reporte bugs**. A sua opinião é muito importante para o desenvolvimento do VRaptor, então sempre que sentir falta de alguma funcionalidade ou encontrar um bug no VRaptor, mande uma mensagem na lista de discussões do desenvolvimento do próprio VRaptor, a caelum-vraptor-dev@googlegroups.com, para ver se a funcionalidade já existe, seja no VRaptor ou em algum de seus plugins, e se o bug possui alguma forma de ser contornado. Se preferir, pode abrir um ticket de nova funcionalidade ou de bug diretamente no repositório do VRaptor, em https://github.com/caelum/vraptor4/issues.

- **Contribua desenvolvendo para o VRaptor**. O código-fonte do VRaptor está no GitHub, que é, sem dúvida, a melhor ferramenta de compartilhamento de código que temos atualmente. Uma ótima forma de entender melhor o VRaptor é olhando o seu código-fonte, em https://github.com/caelum/vraptor4. O VRaptor possui

uma ótima suíte de testes, que exemplificam muito bem o uso de cada funcionalidade, além de garantir que elas estão funcionando. Com isso, além de aprender mais sobre ele, você verá também referência sobre formas de escrever testes em Java.

Quando se sentir confortável (não hesite em postar dúvidas sobre o código do VRaptor na lista de desenvolvimento caelum-vraptor-dev@googlegroups.com), escolha algum dos tickets em https://github.com/caelum/vraptor4/issues e implemente-o, abrindo um Fork no GitHub e mandando Pull Requests. Se você já tem uma ideia de funcionalidade ou já encontrou um bug, pode mandar Pull Requests mesmo que eles não tenham sido reportados ainda. Durante a revisão do seu Pull Request, você ainda pode ganhar dicas de design, de como escrever testes, e de como contribuir melhor para projetos open-source.

- **Escreva e divulgue plugins**. Conforme visto no apêndice de plugins, o VRaptor possui um mecanismo que facilita a criação e a distribuição deles. Se você criou um conjunto de componentes que pode ser reutilizado para outras aplicações, crie um plugin e divulgue-o no repositório: https://github.com/caelum/vraptor-contrib.

- **Continue lendo e estudando**. Mantenha-se atualizado e seja um desenvolvedor completo. Conhecer as técnicas para implementar um bom back-end é essencial, mas saber como fazer um front-end agradável para sua aplicação pode ser um grande diferencial e, claro, o tornar um desenvolvedor com muito mais conhecimento e domínio

do seu trabalho. A Editora Casa do Código tem diversos outros livros que cobrem tanto o back-end como o front-end. Não deixe de ver se é o caminho adequado para continuar seus estudos.

Muito obrigado por ler este livro. Espero que ele tenha ajudado-o a conhecer ou se aprofundar no VRaptor. Sua opinião é muito importante, tanto para este livro quanto para o próprio VRaptor, então se quiser nos dar seu feedback, use o fórum da Casa do Código (http://forum.casadocodigo.com.br/).

Muito obrigado também a você que contribuiu para o VRaptor, seja escrevendo código, documentação, respondendo dúvidas de outras pessoas, divulgando plugins, ou simplesmente usando e recomendando o VRaptor para outras pessoas.

Agradecimentos especiais aos irmãos Paulo e Guilherme Silveira, por criarem e incentivarem o VRaptor e por me chamarem para trabalhar na Caelum quando eu era apenas um estudante de Ciências da Computação. Ao Fabio Kung, por ser o grande evangelista do VRaptor 2. Ao Otávio Garcia, por ter me ajudado a coordenar o desenvolvimento do VRaptor 3 nos últimos anos.

Ao Rodrigo Turini, Alberto Souza, Otávio Garcia e Mario Amaral que tiraram o VRaptor 4 do papel e fizeram um trabalho incrível de migração para o CDI e criação de novas funcionalidades. À Ceci Fernandes, por aguentar minhas piadas ruins desde o tempo da faculdade. Ao Pedro Matiello, pelo mesmo motivo e por ajudar a revisar este livro. Ao Adriano Almeida, por me deixar escrever este livro. E à minha família e minha (futura) esposa Melissa Ágda, por todo o apoio. =)

CAPÍTULO 12

APÊNDICE A — MELHORANDO A USABILIDADE DA APLICAÇÃO COM AJAX

Foi-se o tempo em que os usuários apenas seguiam links e submetiam formulários na web. Hoje em dia, todo mundo espera que uma aplicação web tenha uma interface rica, com componentes inteligentes e interações suaves. Isso porque transferimos grande parte das aplicações que eram tipicamente desktop para a web e precisamos manter, e mesmo melhorar, a usabilidade delas.

Por um tempo, se supriu essa necessidade com pequenas aplicações em Flash, que eram bastante lentas e tinham músicas e animações totalmente desnecessárias, com o famoso "pular introdução". Agora que a maioria da internet não usa mais o famigerado Internet Explorer 6, temos condição de criar interfaces ricas usando apenas o que o browser nos dá: HTML, CSS e JavaScript — com a ajuda de algumas bibliotecas JavaScript, é claro.

Uma das principais bibliotecas JavaScript é o jQuery, com diversos plugins e componentes visuais que facilitam muito o trabalho de um desenvolvedor web. Além disso, existem bibliotecas para facilitar a escrita do CSS, como o LESS ou o SASS. Para escrever HTML em Java, temos o JSP com JSTL, ou podemos usar outras ferramentas como Freemarker ou Velocity.

O VRaptor é um framework MVC que estrutura a forma com que os Controllers vão trabalhar, facilita o desenvolvimento da camada de Modelo com componentes e injeção de dependências e, na parte da camada de Visualização, o VRaptor nos provê formas de se integrar facilmente com qualquer uma das bibliotecas. Mas ele deixa livre a escolha das bibliotecas de visualização, o que também significa que ele não vem com nenhuma delas por padrão, além do JSP e JSTL do servidor.

Neste apêndice, veremos algumas formas de interação com a camada de visualização, usando a biblioteca JavaScript jQuery. Mas não vamos muito a fundo, pois foge do escopo deste livro. Outros livros da Casa do Código ajudam nessa parte, como por exemplo, o *Dominando JavaScript com jQuery*, escrito pelo Plínio Balduíno.

É importante salientar que você, como desenvolvedor web, não precisa necessariamente ser um ás de HTML, CSS e JavaScript, mas é muito importante conhecer o que é possível fazer e conseguir trabalhar pelo menos com o básico dessas tecnologias para que, mesmo que exista um desenvolvedor front-end na sua equipe, você saiba como fazer a integração com a aplicação no servidor.

O navegador não roda JSPs!

Resolvi colocar esse *disclaimer* no livro, pois é uma confusão bastante comum de desenvolvedores Java que trabalham para a web, principalmente no começo da carreira. Quando editamos as páginas do sistema, usamos um arquivo JSP, que é o template padrão para criar páginas HTML em Java.

Mas o JSP é apenas uma ferramenta para facilitar a geração dinâmica de HTML do lado do servidor, com o uso de variáveis, `if`s, `for`s etc. No entanto, o JSP roda apenas **do lado do servidor**, ou seja, é executado usando as variáveis que passamos na requisição e gera um HTML, que é enviado para o navegador. Isto é, quando vemos a página no nosso browser, o que estamos vendo é o HTML gerado pelo JSP correspondente, e não mais o JSP!

Isso significa que, uma vez que o HTML foi para o navegador, não temos mais acesso a variáveis, nem a estruturas do JSP. Se quisermos executar alguma lógica que modifica os elementos da página, precisamos usar JavaScript, que é a linguagem de programação disponível no browser.

Para ficar claro: o JSP é executado do lado do servidor para produzir um HTML. Este HTML é enviado para o navegador que fez a requisição e a página gerada será mostrada. Para melhorar a usabilidade e interagir com os elementos da página sem sair dela, precisamos usar JavaScript, pois o JSP não está mais disponível.

12.1 EXECUTANTO UMA OPERAÇÃO PONTUAL: REMOÇÃO DE LIVROS

No nosso sistema, temos uma listagem de livros, na qual conseguimos ver algumas das informações cadastradas nesses

livros. Esta listagem é gerada pelo seguinte código JSP:

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
  <li>
    <img
      src="${linkTo[LivrosController].capa(livro.isbn) }"
      width="70" height="100"/>

    ${livro.titulo} - ${livro.descricao} -

    <a href="${linkTo[LivrosController].edita(livro.isbn) }">
      Modificar
    </a>
  </li>
</c:forEach>
</ul>
```

Ele produz a listagem de livros. Vamos modificar essa listagem para incluir um link para remoção do livro em questão. Esse link será incluído após o link de modificar o livro:

```
<c:forEach items="${livroList}" var="livro">
  <li>
    ...
    <a href="${linkTo[LivrosController].edita(livro.isbn)}">
      Modificar
    </a>
    -
    <a href="${linkTo[LivrosController].remove(livro.isbn)}">
      Remover
    </a>
  </li>
</c:forEach>
```

Para isso funcionar, precisamos criar esse novo método no controlador seguindo a interface REST, ou seja, usando o método HTTP `DELETE` e com a URL de um livro específico:

```
@Controller
public class LivrosController {
```

```
  //...

  @Delete("/livro/{isbn}")
  public void remove(String isbn) {
    Livro livro = estante.buscaPorIsbn(isbn);

    estante.retira(livro);
  }
}
```

Esse novo método cria alguns problemas. Primeiro, o método é `DELETE`, mas estamos usando um link que executa o método `GET`. Pelo menos, deveríamos estar usando o método `POST`, pois estamos modificando algo no servidor.

O segundo problema é que, mesmo que consigamos executar o método `remove`, o que deve acontecer, visualmente? Poderíamos levar para uma página dizendo que o produto foi removido com sucesso, ou poderíamos redirecionar de volta para a lista de livros, agora com o livro em questão removido. Neste último caso, estamos carregando a página inteira para apenas apagar um dos itens da lista.

Podemos melhorar essa usabilidade usando JavaScript: ao clicar em "Remover", fazemos a requisição de remoção do livro e, se der tudo certo, apagamos a linha do livro, sem sair da página! Vamos fazer isso com a ajuda do jQuery, que facilita bastante esse tipo de operações.

Como criamos o projeto usando o VRaptor Scaffold, o jQuery já está incluído. Mas, caso tivéssemos criado o projeto de outra maneira, precisaríamos baixá-lo (http://jquery.com/download) e incluí-lo na página.

Antes de poder executar o JavaScript, precisamos preparar a

nossa listagem para facilitar a interação com os elementos. Um dos jeitos de fazer isso é dando ids ou classes para os elementos que serão importantes para a nossa lógica JavaScript. Ids precisam ser únicos na página, ou seja, só podemos colocá-los em elementos que não se repetem, como é o caso da lista. Classes podem ser colocadas em vários elementos, portanto, vamos usá-las nos itens da lista e no link de remoção:

```
<ul id="livros">
<c:forEach items="${livroList}" var="livro">
  <li class="livro">
    ...
    <a href="${linkTo[LivrosController].edita(livro.isbn) }">
      Modificar
    </a>
    -
    <a class="remove"
        href="${linkTo[LivrosController].remove(livro.isbn) }">
      Remover
    </a>
  </li>
</c:forEach>
</ul>
```

Para adicionar o comportamento ao link de remoção, precisamos colocar código JavaScript, ou dentro de uma tag `<script>` dentro da página, ou em um arquivo `.js` separado que vamos incluir na página. Para ficar mais simples, vamos fazer a primeira opção:

```
<script>
  ...
</script>
```

Este código precisa rodar depois que o browser tiver lido o HTML da lista, então o colocamos dentro do bloco de inicialização do jQuery:

```
<script>
  $(function() {
    //...
  });
</script>
```

Agora precisamos adicionar um comportamento a todos os links de remoção. E esse comportamento será disparado assim que clicarmos no link. Para isso, vamos, dentro da listagem de livros (`#livros`), procurar todos os links de remoção (`.remove`) e executar um código ao clicar:

```
$(function() {
  $("#livros .remove").on("click", function() {

  });
});
```

O `#livros` indica o elemento com o id `livros` , e `.remove` indica todos os elementos que têm a classe `remove` . Ao fazermos `$("#livros .remove")` , estamos representando todos os elementos da classe `remove` que estão dentro do elemento `#livros` . O resto da expressão diz que, ao clicar (`on("click"`), vamos executar a função passada.

Apesar de ser um link, não queremos que o `Remover` saia da página, então precisamos indicar que não queremos esse comportamento padrão, recebendo um parâmetro na função que representa o evento do clique e chamando a função `preventDefault` :

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();

});
</script>
```

Vamos cuidar primeiro da parte visual da operação: queremos que a linha do livro suma após a remoção. Dentro da função que passamos para tratar o clique, o objeto `this` se refere ao próprio link. Tudo o que precisamos fazer é achar a linha do livro correspondente ao link que foi clicado.

Como o link está dentro do `li` do livro, podemos usar o método `closest`, que procura o primeiro pai do elemento que bate com o seletor passado. Para não depender do elemento HTML que usamos, vamos utilizar a classe `livro`:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();

  var livro = $(this).closest(".livro");
});
```

Com o elemento do livro em mãos, podemos escondê-lo usando o método `fadeOut`, que faz com que ele suma aos poucos:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();

  var livro = $(this).closest(".livro");

  livro.fadeOut();
});
```

Pronto, agora ao clicar no link, a linha do livro vai sumir. Mas apenas manipulamos o HTML do browser, nada foi enviado ao servidor. Precisamos executar a requisição que remove, de fato, o livro, mas sem sair da página.

A técnica que usamos para fazer isso é o que chamamos de AJAX, que originalmente significa *Asynchronous Javascript And XML*. Ou seja, uma requisição assíncrona que usa JavaScript e XML, mas pode usar outras coisas além de XML para trafegar os

dados, como por exemplo, JSON.

O código nativo para gerar essa requisição AJAX no browser é um pouco trabalhoso, mas podemos usar o jQuery para facilitar esse trabalho, pela função `$.ajax`. Passamos para essa função um objeto contendo os atributos da requisição, como a URL a ser chamada, o método da requisição e os parâmetros.

No nosso caso, a URL da requisição é a mesma do link que clicamos, e conseguimos acessá-la por meio do atributo `href`:

```
$("#livros .remove").on("click", function(event) {
  // ...

  $.ajax({
    url: $(this).attr("href")
  });
});
```

Por padrão, essa requisição usará o método `GET`, mas precisamos do método `DELETE`. O atributo para mudar o método da requisição é o `type` e poderíamos mudá-lo:

```
$("#livros .remove").on("click", function(event) {
  // ...

  $.ajax({
    url: $(this).attr("href"),
    type: 'DELETE'
  });
});
```

Embora isso seja possível, não é suportado por todos os browsers. Para garantir que vai funcionar, podemos trocar o método para `POST` e usar o parâmetro `_method: DELETE`, que podemos passar pelo atributo `data`:

```
$("#livros .remove").on("click", function(event) {
  // ...
```

```
  $.ajax({
    url: $(this).attr("href"),
    type: 'POST',
    data: { _method: "DELETE" }
  });
});
```

Dessa forma, executamos a requisição AJAX que removerá o livro do banco de dados. Como essa requisição é assíncrona, o código não fica esperando-a acabar. Além disso, a requisição pode terminar com sucesso, ou dar algum erro. Se quisermos tratar cada um desses casos, precisamos continuar a configuração do `$.ajax`.

Para isso, podemos passar funções que serão chamadas assim que a requisição terminar com sucesso, com erro, ou simplesmente terminar. Chamamos essa função de **callback**, pois será chamada assim que a resposta da requisição voltar:

```
$.ajax({
  url: $(this).attr("href"),
  type: 'POST',
  data: { _method: "DELETE" }
}).done(function(data, textStatus, jqXHR) {
  // executada em caso de sucesso
}).fail(function(jqXHR, textStatus, errorThrown) {
  // executada em caso de erro
}).always(function() {
  // executada assim que a requisição termina,
  // seja com sucesso ou com erro
});
```

No nosso caso, só queremos remover a linha do livro caso a requisição seja com sucesso. Em caso de erro, podemos apenas mandar uma mensagem de alerta para o usuário. O código completo ficaria:

```
$("#livros .remove").on("click", function(event) {
```

```
  event.preventDefault();

  var livro = $(this).closest(".livro");

  $.ajax({
    url: $(this).attr("href"),
    type: 'POST',
    data: { _method: "DELETE" }
  }).done(function(data, textStatus, jqXHR) {

    livro.fadeOut();

  }).fail(function(jqXHR, textStatus, errorThrown) {

    alert("O Livro não foi removido!");

  });
});
```

Dessa forma, completamos a remoção do livro usando AJAX, pelo menos na parte do browser. Na parte do servidor, tudo o que precisamos é gerar uma resposta com sucesso, caso o livro tenha sido removido. Não precisamos mostrar uma página nem redirecionar a requisição. Para isso, usamos o método `result.nothing()`.

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

  @Delete("/livro/{isbn}")
  public void remove(String isbn) {
    Livro livro = estante.buscaPorIsbn(isbn);

    estante.retira(livro);

    result.nothing();
  }
}
```

Repare que o método do Controller é um método normal, não

precisa de nenhuma configuração complicada para responder uma chamada AJAX. O máximo que fizemos foi dizer que não vamos retornar nenhum resultado, além do status de sucesso. Mas caso retornássemos algum resultado, ele estaria disponível na variável `data` da função passada para o `done` da chamada `ajax`.

Capítulo 13

APÊNDICE B — PLUGINS PARA O VRAPTOR

O VRaptor é um framework web MVC que se concentra bastante na parte do controller, ou seja, em tratar requisições e dispará-las para algum dos Controllers da sua aplicação. No entanto, o VRaptor roda dentro de um container de injeção de dependências, no qual você consegue criar diversos componentes que interagem entre si para conseguir executar toda a lógica da sua aplicação.

Essa arquitetura de componentes facilita bastante o reúso deles, então conseguimos usar as mesmas soluções de segurança, transações, controle de ambientes, relatórios entre diversas aplicações diferentes. E para facilitar o compartilhamento desses componentes, o VRaptor possui uma funcionalidade para criar e usar plugins, que são conjuntos de classes que resolvem algum problema e podem ser adicionados facilmente em qualquer aplicação. Uma vez criado o plugin, basta adicionar o seu `jar` ao classpath e todos os seus componentes podem ser usados.

Foi criado também um catálogo de todos os plugins do VRaptor, alguns que nasceram de projetos da própria Caelum e outros feitos pela comunidade de usuários. Ele está disponível no

GitHub, em https://github.com/caelum/vraptor-contrib, contendo referências aos respectivos projetos dos plugins. Alguns deles foram feitos para VRaptor 3 e outros para VRaptor 4, você pode obter mais informações dentro de cada plugin.

Neste apêndice, vamos ver alguns desses plugins, os oficiais e os mais utilizados. Contudo, se desejar, a partir do catálogo podemos ver a documentação de cada plugin para obter as suas informações de uso.

13.1 VRAPTOR JPA

É o plugin responsável por criar e gerenciar o `EntityManager` para que possamos executar as operações no banco de dados usando a JPA. O seu código-fonte pode ser encontrado em https://github.com/caelum/vraptor-jpa.

Para usá-lo, precisamos do seu `jar`, que está no repositório do Maven e podemos usá-lo declarando a sua dependência e alguma das implementações da JPA no `pom.xml`:

```
<dependency>
    <groupId>br.com.caelum.vraptor</groupId>
    <artifactId>vraptor-jpa</artifactId>
    <version>4.0.0</version>
</dependency>

<!-- Hibernate, OpenJPA ou alguma outra implementação da JPA -->
```

Ou baixe o `jar` direto do repositório do maven: http://repo1.maven.org/maven2/br/com/caelum/vraptor/vraptor-jpa/.

Com esse plugin, podemos simplesmente receber o

`EntityManager` no construtor dos componentes que se comunicarão com o banco de dados:

```
public class LivroDAO {

  private EntityManager manager;

  @Inject
  public LivroDAO(EntityManager manager) {
    this.manager = manager;
  }
  @Deprecated LivroDAO() {}

  public void salva(Livro livro) {
    this.manager.persist(livro);
  }
}
```

Este `EntityManager` será aberto ao começo da requisição e fechado ao final dela. O plugin também abre uma transação ao começo da requisição e automaticamente faz o `commit` se tudo der certo; ou o `rollback`, caso exista algum erro de validação ou alguma exception lançada.

Para usar este plugin, também é necessário haver um `persistence.xml` com uma `persistence-unit` chamada `default`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="default"
                    transaction-type="RESOURCE_LOCAL">
    <!-- Dados de conexão com o banco de dados -->
  </persistence-unit>
```

```
</persistence>
```

Como o `EntityManager` vive durante a requisição, só pode ser usado por componentes que são chamados dentro de uma requisição. Caso seja necessário usar um `EntityManager` em outras ocasiões, como agendamento de tarefas, receba um `EntityManagerFactory` no construtor e controle a abertura e o fechamento do `EntityManager` manualmente, de acordo com as necessidades do componente.

Além disso, podemos receber objetos já carregados do banco de dados usando a anotação `@Load` em um método do controller:

```
@Get("/livros/{livro.id}")
public void mostra(@Load Livro livro) {
  //livro já populado
}
```

Para isso, precisamos de um parâmetro da requisição que tenha o id da entidade, de preferência um parâmetro da URI, como no exemplo.

13.2 VRAPTOR HIBERNATE E VRAPTOR HIBERNATE 4

É o plugin responsável por criar e gerenciar a `Session` do Hibernate para acessarmos o banco de dados. Para usá-lo, adicione a seguinte dependência no `pom.xml` :

```
<dependency>
    <groupId>br.com.caelum.vraptor</groupId>
    <artifactId>vraptor-hibernate</artifactId>
    <version>4.0.0</version>
</dependency>
```

Ou baixe-o diretamente do maven:

http://central.maven.org/maven2/br/com/caelum/vraptor/vraptor-hibernate/.

O funcionamento desse plugin é bem parecido com o plugin da JPA: uma `Session` é aberta ao início da requisição e fechada ao final. Também é aberta uma transação ao início da requisição e feito o commit ou rollback dela ao final, assim como no plugin da JPA. Para usá-lo, basta receber a `Session` no construtor do componente desejado:

```
public class LivroDAO {

  private Session session;

  @Inject
  public LivroDAO(Session session) {
    this.session = session;
  }
  @Deprecated LivroDAO() {}

  public void salva(Livro livro) {
    this.session.save(livro);
  }

  @Get("/livros/{livro.id}")
  public Livro mostra(@Load Livro livro) {
     return livro;
  }
}
```

Também é necessário ter o `hibernate.cfg.xml` ou o `hibernate.properties` no classpath, para configurar as credenciais do banco e outros parâmetros do Hibernate.

13.3 VRAPTOR SIMPLE MAIL E VRAPTOR FREEMARKER

Uma tarefa bastante comum entre as aplicações web é enviar e-mails aos seus usuários, seja para avisar que uma conta foi criada, enviar dados de uma compra, promoções, notificações etc. Para que isso seja possível, precisamos de um servidor SMTP para o envio de e-mail, com credenciais válidas, e um endereço de e-mail para usarmos como origem (campo `From`).

O plugin **VRaptor Simple Mail** tem como objetivo facilitar a criação e o envio de e-mails na aplicação. O seu código-fonte e uma ótima documentação estão em https://github.com/caelum/vraptor-simplemail, e pode ser instalado usando a dependência:

```
<dependency>
  <groupId>br.com.caelum.vraptor</groupId>
  <artifactId>vraptor-simplemail</artifactId>
  <version>4.0.0</version>
</dependency>
```

Para completar a configuração do plugin, precisamos acrescentar ao arquivo de configuração do ambiente (por exemplo, o `development.properties`) as seguintes chaves:

```
vraptor.simplemail.main.server = localhost
vraptor.simplemail.main.port = 25
vraptor.simplemail.main.tls = false
vraptor.simplemail.main.from = no-reply@livraria.com

#se for necessário fazer autenticação:
vraptor.simplemail.main.username = meu-usuario
vraptor.simplemail.main.password = minha-senha
```

Agora, para enviar um e-mail, usamos o componente `Mailer` e usamos a API `javax.mail` para criá-lo:

```
@Controller
public class ComprasController {
  private Mailer mailer;
```

```
  @Inject
  public ComprasController(Mailer mailer) {
    this.mailer = mailer;
  }
  @Deprecated ComprasController() {}

  @Post
  public void finaliza(Compra compra) {
      //...

      Email email = new SimpleMail();
      email.setSubject("Compra efetuada com sucesso!");
      email.addTo(compra.getUsuario().getEmail());
      email.setMsg("Seu pedido de Compra no valor de " +
        compra.getValor() + " foi finalizado com sucesso!");
      mailer.send(email);
    }
}
```

É possível ainda enviar um e-mail de forma assíncrona usando o `AsyncMailer`, assim a requisição não fica parada esperando o e-mail ser enviado.

Porém, se quisermos enviar um e-mail um pouco mais elaborado, com HTML e imagens, não é muito prático criá-lo dentro de um código Java. Por esse motivo, o Simple Mail se integra com o plugin **VRaptor Freemarker**, no qual podemos usar templates do Freemarker para criar o corpo do e-mail. Para isso, recebemos o componente `TemplateMailer`:

```
@Controller
public class ComprasController {
  private Mailer mailer;
  private TemplateMailer templates;

  @Inject
  public ComprasController(Mailer mailer,
                           TemplateMailer templates) {
    this.mailer = mailer;
    this.templates = templates;
```

```
  }
  @Deprecated ComprasController() {}

  @Post
  public void finaliza(Compra compra) {
      //...

      Email email = this.templates
              .template("compraFinalizada.ftl")
              .with("compra", compra)
              .to(compra.getCliente().getNome(),
                  compra.getCliente().getEmail());

      mailer.send(email);
    }
}
```

Para mais informações sobre o VRaptor Freemarker: https://github.com/caelum/vraptor-freemarker.

13.4 AGENDAMENTO DE TAREFAS: VRAPTOR TASKS

Quando estamos desenvolvendo aplicações web, a maioria do trabalho é feita dentro de requisições HTTP. Mas algumas vezes precisamos executar tarefas na aplicação que não serão disparadas por um usuário interagindo com o sistema no navegador, e sim uma tarefa que precisa ser disparada pela própria aplicação.

Isso acontece quando precisamos enviar um e-mail para o usuário 2 horas depois da compra, ou quando precisamos consumir dados de um serviço a cada 5 minutos, por exemplo. Para executar esse tipo de tarefa agendada ou periódica, temos uma biblioteca bastante famosa em Java: o Quartz, com a qual podemos criar *tasks*, que podemos agendar para serem executadas.

Para podermos aproveitar todo o poder do Quartz junto com a injeção de dependências que o VRaptor nos dá, foi criado o plugin **VRaptor Tasks** pelo William Pivotto, que pode ser encontrado em https://github.com/wpivotto/vraptor-tasks. Esse plugin tem uma ótima documentação na página do GitHub, então vou apenas dar alguns exemplos do que é possível fazer com ele.

Para criar tarefas periódicas, que não dependem de componentes `@RequestScope`, devemos criar uma `Task` e anotá-la com `@Scheduled`, configurando a frequência.

```
@ApplicationScoped //ou @PrototypeScoped
@Scheduled(cron="0 */5 * * * ?")
public class Sincronizador implements Task {
  public Sincronizador(ClienteHTTP http) {...}

  public void execute() {
    // sincroniza com um serviço qualquer...
  }
}
```

Caso a tarefa precise de componentes de escopo de requisição, criamos um controller e anotamos os métodos que queremos agendar com `@Scheduled`:

```
@Controller
public EnviadorDeEmailController {
  @Inject
  public EnviadorDeEmailController(
    UsuarioRepository usuarios,
    Mailer mailer) {
    //...
  }

  @Scheduled(cron="0 0 * * * ?")
  public void avisaSobreCadastro() {
    List<Usuario> avisaveis =
      usuarios.queNaoPreencheramCadastro();
    // manda email pra todos
```

```
  }
}
```

Nesse caso, todo dia à meia-noite uma tarefa executará uma requisição que cairá nesse método, assim todos os componentes `@RequestScoped` estarão disponíveis. Ainda é possível agendar tarefas em tempo de execução, gerenciar e monitorar as tarefas já agendadas.

13.5 CONTROLE DE USUÁRIOS: VRAPTOR-SHIRO

Na seção *Controlando os métodos interceptados*, vimos como criar uma infraestrutura de controle de usuários no sistema, com um novo modelo `Usuario` e dois interceptors: o `AutenticacaoInterceptor` para verificar se o usuário está logado ou não; e o `AutorizacaoInterceptor` para ver se o usuário tem permissão de acessar o recurso desejado.

No entanto, esse controle de usuários pode ser bastante complicado, por exemplo, se quisermos criar papéis (Roles), ou definir uma hierarquia de usuários, segundo a qual cada um tem permissão de acessar um pedaço das funcionalidades do sistema.

Para facilitar esse controle, o **Rafael Dipold** criou um plugin chamado **VRaptor Shiro**, que pode ser encontrado em https://github.com/dipold/vraptor-shiro. Ele tem uma ótima documentação de suas funcionalidades em português. Para instalá-lo, acrescente essas linhas no `pom.xml` :

```
<dependency>
  <groupId>br.com.caelum.vraptor</groupId>
  <artifactId>vraptor-shiro</artifactId>
  <!-- cheque a versão mais atual -->
```

```
  <version>4.0.0-beta-1</version>
</dependency>
```

Ou ainda baixar o `jar` diretamente, em https://github.com/dipold/vraptor-shiro/releases.

Para usar esse plugin, precisamos ter uma forma de identificar o usuário logado. Podemos usar um usuário no banco de dados, como fizemos na seção *Controlando os métodos interceptados*, ou usar algo mais complexo para isso, como LDAP ou AD.

Tínhamos criado o `LoginController` e dois interceptors para fazer esse trabalho. Para adaptá-los ao plugin, precisamos que o `LoginController` implemente a interface `AuthorizationRestrictionListener`:

```
@Controller
public class LoginController
implements AuthorizationRestrictionListener {
  //...

  @Override
  public void
  onAuthorizationRestriction(AuthorizationException e) {
    result.include("error", e.toString());
    result.redirectTo(this).formulario();
  }
}
```

Em vez de usar o `UsuarioLogado` para controlar a sessão, vamos usar a classe `Subject` do plugin:

```
@Controller
public class LoginController ... {

  private Subject subject;
  @Inject
  public LoginController(...,
      Subject subject) {
    //...
```

```
    this.subject = subject;
  }

  //...

  @Post("/login")
  public void login(String login, String senha) {
    try {
      subject.login(new UsernamePasswordToken(login, senha));

      // ou a página inicial
      result.redirectTo(LivrosController.class).lista();
    }
    catch (UnknownAccountException e) {}
    catch (IncorrectCredentialsException e) {}
    catch (LockedAccountException e) {}
    catch (ExcessiveAttemptsException e) {}
    catch (AuthenticationException e) {}
  }

  @Get("/logout")
  public void logout() {
    subject.logout();
    result.redirectTo(this).formulario();
  }
}
```

Para falar para o plugin como a autenticação será feita, usamos uma classe que implementa `Permission`:

```
import br.com.caelum.vraptor.security.Permission;
import br.com.caelum.vraptor.security.User;

public class Autorizador implements Permission {

    @Inject RegistroDeUsuarios usuarios;

    @Override
    public User getUserByUsername(String username) {
      //a senha será verificada pelo apache Shiro
      Usuario usuario = usuarios.comLogin(username);

      return new User(usuario.getLogin(), usuario.getSenha());
    }
```

```
    @Override
    public Set<String> getRolesByUser(String username) {
      Usuario usuario = usuarios.comLogin(username);
      if (usuario.isAdmin()) {
        return Collections.singleton("admin");
      } else {
        return Collections.emptySet();
      }
    }

    @Override
    public Set<String> getPermissionsByRole(String role) {
        return role;
    }
}
```

Feito isso, não precisamos mais dos interceptors que havíamos criado, pois o VRaptor Shiro já faz a autenticação e a autorização. Agora, o que precisamos fazer é indicar para o Shiro quais métodos precisam ser autenticados.

Precisamos anotar a classe ou o método com `@Secured` para dizer que ele será autenticado. Se a autorização for apenas ter um usuário logado, podemos usar a anotação `@RequiresUser`. Se precisarmos controlar um nível maior — por exemplo, ver se o usuário é admin —, usamos a anotação `@RequiresRole`.

No nosso caso, queremos que o `LivrosController` seja autenticado totalmente, mas apenas alguns métodos precisam ser acessados só pelo admin. Então, a configuração seria:

```
@Secured
@RequiresUser
@Controller
public class LivrosController {

  @Get("/livros/formulario")
  public void formulario() { ... }
```

```
  @Get("/livros")
  public List<Livro> lista() { ... }

  @RequiresRole("admin")
  @Post("/livros")
  public void salva(Livro livro) { ... }

  //...
}
```

Para uma explicação mais detalhada sobre as funcionalidades do plugin, visite: https://github.com/dipold/vraptor-shiro/wiki.

13.6 CRIANDO O SEU PRÓPRIO PLUGIN

Muitas vezes desenvolvemos uma solução legal que precisa ser reutilizada entre diversos projetos da sua empresa, ou simplesmente queremos compartilhá-la com outros desenvolvedores. Para que o VRaptor considere essa solução como um plugin, precisamos agrupar todos os seus componentes em um projeto separado, para gerarmos um `jar`.

Para que esse `jar` seja registrado automaticamente no VRaptor, mais especificamente no CDI, ele precisa ter dentro dele um arquivo `META-INF/beans.xml` vazio. Vamos supor que queremos transformar o nosso serviço de arquivos que desenvolvemos no capítulo *Download e upload de arquivos* em um plugin.

Esse serviço é composto de duas classes básicas, o `Diretorio` e o `Arquivo`, e uma implementação que salva os arquivos no banco de dados, o `DiretorioNoBD`. Vamos extraí-las para um projeto à parte, usando um pacote mais genérico.

```
vraptor-arquivos/
  - src/main/java
    - br.com.casadocodigo.arquivos
      - Arquivo.java
      - Diretorio.java
      - DiretorioNoBD.java
  - src/main/resources
    - META-INF
      - beans.xml (vazio)
```

Agora basta gerar um `jar` com todos esses arquivos e temos um plugin! Se colocarmos o `vraptor-arquivos.jar` em qualquer projeto com o VRaptor, o `DiretorioNoBD` já será registrado como componente e poderemos receber um `Diretorio` no construtor de qualquer classe da aplicação.

Se o objetivo for compartilhar esse plugin:

1. Crie um projeto no GitHub, por exemplo em https://github.com/seu-usuario/seu-plugin.
2. Suba os arquivos do projeto para lá usando comandos de git em linha de comando, ou o próprio programa do GitHub.
3. Escreva um arquivo `README.md` explicando como usar o seu plugin.
4. Entre em https://github.com/caelum/vraptor-contrib e siga as instruções para incluir seu plugin na lista.
5. Divulgue o seu plugin no fórum da Casa do Código.
6. Seja reconhecido por todos. =)