VRaptor 3 – Um framework Java MVC de desenvolvimento rápido e fácil

http://vraptor.caelum.com.br

Automatically generated by Caelum Objects Tubaina
15 de outubro de 2009

Índice

1 VRaptor3 - Guia de 1 minuto **1**

1.1 Começando um projeto 1

1.2 Um Controller simples 1

2 VRaptor3 - o guia inicial de 10 minutos **4**

2.1 Começando o projeto: uma loja virtual 4

2.2 Um cadastro de produtos 4

2.3 Criando o ProdutoDao: injeção de Dependências 5

2.4 Formulário de adição: redirecionamento 6

2.5 Validação 8

2.6 Usando o Hibernate para guardar os Produtos 9

2.7 Controlando transações: Interceptors 9

2.8 Carrinho de compras: Componentes na sessão 9

2.9 Um pouco de REST 10

2.10 Arquivo de mensagens 11

3 Resources-Rest **13**

3.1 O que são Resources? 13

3.2 Parâmetros dos métodos 13

3.3 Escopos 14

3.4 Http Methods 15

3.5 @Path 16

3.6 RoutesConfiguration 18

4 Componentes **20**

4.1 O que são componentes? . . . 20

4.2 Escopos . . . 20

4.3 ComponentFactory . . . 21

4.4 Injeção de dependências . . . 21

5 Conversores **23**

5.1 Padrão . . . 23

5.2 Tipos primitivos . . . 23

5.3 Wrappers de tipos primitivos . . . 23

5.4 Enum . . . 23

5.5 BigInteger e BigDecimal . . . 23

5.6 Calendar e Date . . . 23

5.7 LocalDate e LocalTime do joda-time . . . 24

5.8 Interface . . . 24

5.9 Registrando um novo conversor . . . 25

6 Interceptadores **26**

6.1 Para que interceptar . . . 26

6.2 Como interceptar . . . 26

6.3 Exemplo simples . . . 26

6.4 Exemplo com Hibernate . . . 27

6.5 Como garantir ordem: InterceptorSequence . . . 28

7 Validação **29**

7.1 Estilo clássico . . . 29

7.2 Estilo fluente . . . 29

7.3 Validação usando matchers do Hamcrest . . . 30

7.4 Hibernate validator . . . 30

7.5 Para onde redirecionar no caso de erro . . . 31

8 View e Ajax **32**

8.1 Custom PathResolver . . . 32

8.2 View . . . 32

8.3 Redirecionamento e forward . . . 33

8.4 Accepts e o parâmetro _format . . . 33
8.5 Ajax: construindo na view . . . 33

9 Injeção de dependências 35
9.1 ComponentFactory . . . 35
9.2 Providers . . . 37
9.3 Spring . . . 37
9.4 Pico Container . . . 37
9.5 Seu próprio provider . . . 37

10 Downloading 38
10.1 exemplo de 1 minuto: download . . . 38
10.2 Adicionando mais informações no download . . . 38
10.3 exemplo de 1 minuto: upload . . . 38
10.4 Mais sobre Upload . . . 39

11 Componentes Utilitários Opcionais 40
11.1 Registrando um componente opcional . . . 40
11.2 Hibernate Session e SessionFactory . . . 40
11.3 JPA EntityManager e EntityManagerFactory . . . 41

12 Configurações avancadas: sobrescrevendo as convenções e comportamento do VRaptor 42
12.1 Mudando a view renderizada por padrão . . . 42
12.2 Mudando a URI padrão . . . 42
12.3 Mudando o IoC provider . . . 43
12.4 Mudando ApplicationContext base do Spring . . . 43
12.5 Mudando o encoding da sua aplicação . . . 43

13 Spring, Joda Time, Hibernate e Google App Engine 45
13.1 Integração com Hibernate ou JPA . . . 45
13.2 Integração com Spring . . . 45
13.3 Conversores para Joda Time . . . 45
13.4 Rodando o VRaptor3 no Google App Engine . . . 45

14 Testando componentes e controllers 46
14.1 MockResult . . . 46
14.2 MockValidator . . . 46

15 ChangeLog **48**

15.1 3.0.1 (a ser lançado) . . . 48

15.2 3.0.0 . . . 48

15.3 3.0.0-rc-1 . . . 48

15.4 3.0.0-beta-5 . . . 49

15.5 3.0.0-beta-4 . . . 49

15.6 3.0.0-beta-3 . . . 50

16 Migrando do VRaptor2 para o VRaptor3 **51**

16.1 web.xml . . . 51

16.2 migrando de @org.vraptor.annotations.Component para @br.com.caelum.vraptor.Resource . . . 51

16.3 @In . . . 52

16.4 @Out e getters . . . 52

16.5 views.properties . . . 54

16.6 Validação . . . 54

16.7 Colocando objetos na sessão . . . 55

Version: 11.7.15

VRaptor3 - Guia de 1 minuto

O VRaptor 3 foca em simplicidade e, portanto, todas as funcionalidades que você verá têm como primeira meta resolver o problema do programador da maneira menos intrusiva possível em seu código.

Tanto para salvar, remover, buscar e atualizar ou ainda funcionalidades que costumam ser mais complexas como upload e download de arquivos, resultados em formatos diferentes (xml, json, xhtml etc), tudo isso é feito através de funcionalidades simples do VRaptor 3, que sempre procuram encapsular `HttpServletRequest`, `Response`, `Session` e toda a API do `javax.servlet`.

1.1- Começando um projeto

Você pode começar seu projeto a partir do `vraptor-blank-project`, que contem as dependências necessárias e a configuração no `web.xml`. Ele pode ser baixado em:

http://vraptor.caelum.com.br/download.jsp

1.2- Um Controller simples

Chamaremos de **Controller** as classes contendo a lógica de negócio do seu sistema. São as classes que alguns frameworks podem vir a chamar de actions ou services, apesar de não significarem exatamente a mesma coisa.

Com o VRaptor configurado no seu `web.xml`, basta criar os seus controllers para receber as requisições e começar a construir seu sistema.

Um controller simples seria:

```
/*
 * anotando seu controller com @Resource, seus métodos públicos ficarão disponíveis
 * para receber requisições web.
 */
@Resource
public class ClientsController {

    private ClientDao dao;

    /*
     * Você pode receber as dependências da sua classe no construtor, e o VRaptor vai
     * se encarregar de criar ou localizar essas dependências pra você e usá-las pra
     * criar o seu controller. Para que o VRaptor saiba como criar o ClientDao você
     * deve anotá-lo com @Component.
     */
    public ClientsController(ClientDao dao) {
        this.dao = dao;
    }
```

```
    /*
     * Todos os métodos públicos do seu controller estarão acessíveis via web.
     * Por exemplo, o método form pode ser acessado pela URI /clients/form e
     * vai redirecionar para a jsp /WEB-INF/jsp/clients/form.jsp
     */
    public void form() {
        // código que carrega dados para checkboxes, selects, etc
    }

    /*
     * Você pode receber parâmetros no seu método, e o VRaptor vai tentar popular os
     * campos dos parâmetro de acordo com a requisição. Se houver na requisição:
     * custom.name=Lucas
     * custom.address=R.Vergueiro
     * então teremos os campos name e address do Client custom estarão populados com
     * Lucas e R.Vergueiro via getters e setters
     * URI: /clients/add
     * view: /WEB-INF/jsp/clients/add.jsp
     */
    public void add(Client custom) {
        dao.save(custom);
    }

    /*
     * O retorno do seu método é exportado para a view. Nesse caso, como o retorno é
     * uma lista de clientes, a variável acessível no jsp será ${clientList}.
     * URI: /clients/list
     * view: /WEB-INF/jsp/clients/list.jsp
     */
    public List<Client> list() {
        return dao.listAll():
    }

    /*
     * Se o retorno for um tipo simples, o nome da variável exportada será o nome da
     * classe com a primeira letra minúscula. Nesse caso, como retornou um Client, a
     * variável na jsp será ${client}.
     * Devemos ter um parâmetro da requisição id=5 por exemplo, e o VRaptor vai
     * fazer a conversão do parâmetro em Long, e passar como parâmetro do método.
     * URI: /clients/view
     * view: /WEB-INF/jsp/clients/view.jsp
     */
    public Client view(Long id) {
        return dao.load(id);
    }
}
```

Repare como essa classe está completamente independente da API de `javax.servlet`. O código também está extremamente simples e fácil de ser testado como unidade. O VRaptor já faz várias associações para as URIs como default:

```
/client/form   invoca form()
/client/add    invoca add(client) populando o objeto client com os parâmetros da requisição
/clients/list  invoca list() e devolve ${clientList} ao JSP
/clients/view?id=3  invoca view(3l) e devolve ${client} ao JSP
```

Mais para a frente veremos como é fácil trocar a URI `/clients/view?id=3` para `/clients/view/3`, deixando a URI mais elegante.

O ClientDao será injetado pelo VRaptor, como também veremos adiante. Você já pode passar para o Guia inicial de 10 minutos.

VRaptor3 - o guia inicial de 10 minutos

2.1- Começando o projeto: uma loja virtual

Vamos começar baixando o *vraptor-blank-project*, do site http://vraptor.caelum.com.br/download.jsp. Esse blank-project já possui a configuração no web.xml e os jars no WEB-INF/lib necessários para começar a desenvolver no VRaptor. E você pode, ainda, importar esse projeto diretamente no Eclipse.

Da configuração padrão, vamos apenas mudar o pacote base da configuração, no web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>br.com.nomedaempresa.nomedoprojeto</param-value>
</context-param>
```

Desse modo, toda a sua aplicação **deve** estar abaixo do pacote `br.com.nomedaempresa.nomedoprojeto`, para que o VRaptor consiga encontrar seus componentes e gerenciar suas dependências. No nosso caso será:

```
<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>br.com.caelum.lojavirtual</param-value>
</context-param>
```

Supondo que o contexto web da aplicação foi mudado para `/lojavirtual`, se você rodar o exemplo e acessar a URI http://localhost:8080/lojavirtual você deve ver um **It works!** na tela.

2.2- Um cadastro de produtos

Para começar o sistema, vamos começar com cadastro de produtos. Precisamos de uma classe que vai representar o produto, e vamos usá-la para guardar produtos no banco, usando o Hibernate:

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

    private String nome;
    private String descricao;
    private Double preco;
    //getter e setters
}
```

Precisamos também de uma classe que vai *controlar* o cadastro de produtos, tratando requisições web. Essa classe vai ser o `Controller` de produtos:

```
public class ProdutosController {
```

```
}
```

A classe ProdutosController vai expor URIs para serem acessadas via web, ou seja, vai expor recursos da sua aplicação. E para indicar isso, você precisa anotá-la com com `@Resource`:

```
@Resource
public class ProdutosController {
}
```

Ao colocar essa anotação na classe, todos os métodos públicos dela serão acessíveis via web. Por exemplo, se tivermos um método `lista` na classe:

```
@Resource
public class ProdutosController {
    public List<Produto> lista() {
        return new ArrayList<Produto>();
    }
}
```

o VRaptor automaticamente redireciona todas as requisições à URI `/produtos/lista` para esse método. Ou seja, a convenção para a criação de URIs é `/<nome_do_controller>/<nome_do_metodo>`.

Ao terminar a execução do método, o VRaptor vai fazer o `dispatch` da requisição para o jsp `/WEB-INF/jsp/produtos/lista.jsp`. Ou seja, a convenção para a view padrão é `/WEB-INF/jsp/<nome_do_controller>/<nome_do_metodo>.jsp`.

O método `lista` retorna uma lista de produtos, mas como acessá-la no jsp? No VRaptor, o retorno do método é exportado para a jsp através de atributos da requisição. No caso do método `lista`, vai existir um atributo chamado **produtoList** contendo a lista retornada:

lista.jsp

```
<ul>
<c:forEach items="${produtoList}" var="produto">
    <li> ${produto.nome} - ${produto.descricao} </li>
</c:forEach>
</ul>
```

A convenção para o nome dos atributos exportados é bastante intuitiva: se for uma collection, como o caso do método acima, o atributo será <tipoDaCollection>List, produtoList no caso; se for uma classe qualquer vai ser o nome da classe com a primeira letra minúscula. Se o método retorna `Produto`, o atributo exportado será `produto`.

2.3- Criando o ProdutoDao: injeção de Dependências

O VRaptor usa fortemente o conceito de Injeção de Dependências e Inversão de Controle. A idéia é que você não precisa criar ou buscar as dependências da sua classe, você simplesmente as recebe e o VRaptor se encarrega de criá-las pra você. Mais informações no capítulo de Injeção de Dependências.

Estamos retornando uma lista vazia no nosso método `lista`. Seria muito mais interessante retornar uma lista de verdade, por exemplo todas os produtos cadastrados no sistema. Para isso vamos criar um DAO de produtos, para fazer a listagem:

```
public class ProdutoDao {

    public List<Produto> listaTodos() {
        return new ArrayList<Produto>();
    }

}
```

E no nosso ProdutosController podemos usar o dao pra fazer a listagem de produtos:

```
@Resource
public class ProdutosController {

    private ProdutoDao dao;

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

}
```

Podemos instanciar o ProdutoDao direto do controller, mas é muito mais interessante aproveitar o gerenciamento de dependências que o VRaptor faz e receber o dao no construtor! E para que isso seja possível basta anotar o dao com @Component e o VRaptor vai se encarregar de criar o dao e injetá-lo na sua classe:

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

@Resource
public class ProdutosController {

    private ProdutoDao dao;

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

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

}
```

2.4- Formulário de adição: redirecionamento

Temos uma listagem de Produtos, mas ainda não temos como cadastrá-los. Vamos então criar um formulário de adição de produtos. Para não ter que acessar o jsp diretamente, vamos criar uma lógica vazia que só redireciona pro jsp:

```
@Resource
public class ProdutosController {
```

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

Podemos acessar o formulário pela URI `/produtos/form`, e o formulário estará em `/WEB-INF/jsp/produtos/form.jsp`:

```
<form action="<c:url value="/produtos/adiciona"/>">
    Nome:             <input type="text" name="produto.nome" /><br/>
    Descrição:    <input type="text" name="produto.descricao" /><br/>
    Preço:            <input type="text" name="produto.preco" /><br/>
    <input type="submit" value="Salvar" />
</form>
```

O formulário vai salvar o Produto pela URI `/produtos/adiciona`, então precisamos criar esse método no nosso controller:

```
@Resource
public class ProdutosController {
    //...
    public void adiciona() {
    }
}
```

Repare nos nomes dos nossos inputs: **produto.nome**, **produto.descricao** e **produto.preco**. Se recebermos um `Produto` no método `adiciona` com o nome **produto**, o VRaptor vai popular os seus campos **nome**, **descricao** e **preco**, usando os seus setters no Produto, com os valores digitados nos inputs. Inclusive o campo **preco**, vai ser convertido para Double antes de ser setado no produto. Veja mais sobre isso no capítulo de converters.

Então, usando os nomes corretamente nos inputs do form, basta criar seu método adiciona:

```
@Resource
public class ProdutosController {
    //...
    public void adiciona(Produto produto) {
        dao.adiciona(produto);
    }
}
```

Geralmente depois de adicionar algo no sistema queremos voltar para a sua listagem, ou para o formulário novamente. No nosso caso, queremos voltar pra listagem de produtos ao adicionar um produto novo. Para isso existe um componente do VRaptor: o `Result`. Ele é responsável por adicionar atributos na requisição, e por mudar a view a ser carregada. Se eu quiser uma instância de Result, basta recebê-lo no construtor:

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

E para redirecionar para a listagem basta usar o result:

```
result.use(Results.logic()).redirectTo(ProdutosController.class).lista();
```

Podemos ler esse código como: *Como resultado, use uma lógica, redirecionando para o método lista do ProdutosController*. A configuração de redirecionamento é 100% java, sem strings envolvidas! Fica explícito no seu código que o resultado da sua lógica não é o padrão, e qual resultado você está usando! Você não precisa ficar se preocupando com arquivos de configuração! Mais ainda, se eu quiser mudar o nome do método `lista`, eu não preciso ficar rodando o sistema inteiro procurando onde estão redirecionando pra esse método, basta usar o refactor do eclipse, por exemplo, e tudo continua funcionando!

Então nosso método `adiciona` ficaria:

```
public void adiciona(Produto produto) {
    dao.adiciona(produto);
    result.use(Results.logic()).redirectTo(ProdutosController.class).lista();
}
```

Mais informações sobre o Result no capítulo Views e Ajax.

2.5- Validação

Não faz muito sentido adicionar um produto sem nome no sistema, nem um produto com preço negativo. Antes de adicionar o produto, precisamos verificar se é um produto válido, com nome e preço positivo, e caso não seja válido voltamos para o formulário com mensagens de erro. Para fazermos isso, podemos usar um componente do VRaptor: o Validator. Você pode recebê-lo no construtor do seu Controller, e usá-lo da seguinte maneira:

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

    public void adiciona(Produto produto) {
        validator.checking(new Validations() {{
            that(!produto.getNome().isEmpty(), "produto.nome", "nome.vazio");
            that(produto.getPreco() > 0, "produto.preco", "preco.invalido");
        }});
        validator.onErrorUse(Results.page()).of(ProdutosController.class).form();

        dao.adiciona(produto);
        result.use(Results.logic()).redirectTo(ProdutosController.class).lista();
    }
}
```

Podemos ler as validações da seguinte maneira: *Valide que o nome do produto não é vazio e que o preço do produto é maior que zero. Se acontecer um erro, use como resultado a página do form do ProdutosController*. Ou seja, se por exemplo o nome do produto for vazio, vai ser adicionada a mensagem de erro para o campo "produto.nome", com a mensagem "nome.vazio" internacionalizada. Se acontecer algum erro, o sistema vai vol-

tar pra página do formulário, com os campos preenchidos e com mensagens de erro que podem ser acessadas da seguinte maneira:

```
<c:forEach var="error" items="${errors}">
    ${error.category}  ${error.message}<br />
</c:forEach>
```

Mais informações sobre o Validator no capítulo de Validações.

Com o que foi visto até agora você já consegue fazer 90% da sua aplicação! As próximas sessões desse tutorial mostram a solução para alguns problemas frequentes que estão nos outros 10% da sua aplicação.

2.6- Usando o Hibernate para guardar os Produtos

Agora vamos fazer uma implementação de verdade do ProdutoDao, usando o Hibernate para persistir os produtos. Para isso nosso ProdutoDao precisa de uma Session. Como o VRaptor usa injeção de dependências, basta receber uma Session no construtor!

```
@Component
public class ProdutoDao {

    private Session session;

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

    public void adiciona(Produto produto) {
        session.save(produto);
    }
    //...
}
```

Mas peraí, para o VRaptor precisa saber como criar essa Session, e eu não posso simplesmente colocar um @Component na Session pois é uma classe do Hibernate! Para isso existe a interface ComponentFactory, que você pode usar pra criar uma Session. Mais informações de como fazer ComponentFactories no capítulo de Componentes. Você pode ainda usar os ComponentFactories que já estão disponíveis para isso no VRaptor, como mostra o capítulo de Utils.

2.7- Controlando transações: Interceptors

Muitas vezes queremos interceptar todas as requisições (ou uma parte delas) e executar alguma lógica, como acontece com o controle de transações. Para isso existem os Interceptor's no VRaptor. Saiba mais sobre eles no capítulo de Interceptors. Existe um TransactionInterceptor já implementado no VRaptor, saiba como usá-lo no capítulo de Utils.

2.8- Carrinho de compras: Componentes na sessão

Se quisermos criar um carrinho de compras no nosso sistema, precisamos de alguma forma manter os itens do carrinho na Sessão do usuário. Para fazer isso, podemos criar um componente que está no escopo de sessão, ou seja, ele vai ser único na sessão do usuário. Para isso basta criar um componente anotado com @SessionScoped:

```
@Component
@SessionScoped
public class CarrinhoDeCompras {
    private List<Produto> itens = new ArrayList<Produto>();

    public List<Produto> getTodosOsItens() {
        return itens;
    }

    public void adicionaItem(Produto item) {
        itens.add(item);
    }
}
```

Como esse carrinho de compras é um componente, podemos recebê-lo no construtor do controller que vai cuidar do carrinho de compras:

```
@Resource
public class CarrinhoController {

    private final CarrinhoDeCompras carrinho;

    public CarrinhoController(CarrinhoDeCompras carrinho) {
        this.carrinho = carrinho;
    }

    public void adiciona(Produto produto) {
        carrinho.adicionaItem(produto);
    }

    public List<Produto> listaItens() {
        return carrinho.getTodosOsItens();
    }
}
```

Além do escopo de sessão existe o escopo de Aplicação com a anotação @ApplicationScoped. Os componentes anotados com @ApplicationScoped serão criados apenas uma vez em toda a aplicação.

2.9- Um pouco de REST

Seguindo a idéia REST de que URIs devem identificar recursos na rede para então podermos fazer valer as diversas vantagens estruturais que o protocolo HTTP nos proporciona, note o quão simples fica mapear os diversos métodos HTTP para a mesma URI, e com isso invocar diferentes métodos, por exemplo queremos usar as seguintes URIs para o cadastro de produtos:

```
GET /produtos - lista todos os produtos
POST /produtos - adiciona um produto
GET /produtos/{id} - visualiza o produto com o id passado
PUT /produtos/{id} - atualiza as informações do produto com o id passado
DELETE /produtos/{id} - remove o produto com o id passado
```

Para criar esse comportamento REST no VRaptor podemos usar as anotações @Path - que muda qual é a uri que vai acessar o determinado método, e as anotações com os nomes dos métodos HTTP @Get, @Post,

@Delete, @Put, que indicam que o método anotado só pode ser acessado se a requisição estiver com o método HTTP indicado. Então uma versão REST do nosso ProdutosController seria:

```
public class ProdutosController {
    //...

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

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

    @Get
    @Path("/produtos/{produto.id}")
    public void visualiza(Produto produto) {...}

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

    @Delete
    @Path("/produtos/{produto.id}")
    public void remove(Produto produto) {...}

}
```

Note que podemos receber parâmetros nas URIs. Por exemplo se chamarmos a URI **GET /produtos/5**, o método visualiza será invocado, e o parâmetro produto vai ter o id populado com 5.

Mais informações sobre isso no capítulo de Resources-REST.

2.10- Arquivo de mensagens

Internacionalização (*i18n*) é um recurso poderoso, e que está presente em quase todos os frameworks Web hoje em dia. E não é diferente no VRaptor3. Com i18n podemos fazer com que nossa aplicação suporte várias línguas (francês, português, espanhol, inglês, etc) de uma maneira que não nos cause muito esforço, bastando apenas fazermos a tradução das mensagens da nossa aplicação.

Para isso é só criarmos um arquivo chamado *messages.properties* e disponibilizá-lo no classpath da nossa aplicação (`WEB-INF/classes`). O conteúdo desse arquivo são várias linhas compostas por um conjunto de chaves/valor, como por exemplo:

```
campo.nomeUsuario = Nome de Usuário
campo.senha = Senha
```

Até então está fácil, mas e se quisermos criar esses arquivos para várias línguas, como por exemplo, inglês? Simples, basta criarmos um outro arquivo properties chamado `messages_en.properties`. Repare no sufixo *_en* no nome do arquivo. Isso indica que quando o usuário acessar sua aplicação através de uma máquina configurada com locale em inglês as mensagens desse arquivo serão utilizadas. O conteúdo desse arquivo então ficaria:

```
campo.nomeUsuario = Username
```

```
campo.senha = Password
```

Repare que as chaves são mantidas, mudando apenas o valor para a língua escolhida.

Para usar essas mensagens em seus arquivos JSP, você pode utilizar a JSTL. Dessa forma, o código ficaria:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<html>
    <body>
        <fmt:message key="campo.usuario" /> <input name="usuario.nomeUsuario" />

        <br />

        <fmt:message key="campo.senha" /> <input type="password" name="usuario.senha" />

        <input type="submit" />
    </body>
</html>
```

Resources-Rest

3.1- O que são Resources?

Resources são o que poderíamos pensar como recursos a serem disponibilizados para acesso pelos nossos clientes.

No caso de uma aplicação Web baseada no VRaptor, um recurso deve ser anotado com a anotação `@Resource`. Assim que o programador insere tal anotação em seu código, todos os métodos públicos desse recurso se tornam acessíveis através de chamadas do tipo GET a URIs específicas.

O exemplo a seguir mostra um recurso chamado `ClienteController` que possui métodos para diversas funcionalidades ligadas a um cliente.

Simplesmente criar essa classe e os métodos faz com que as urls **/cliente/adiciona**, **/cliente/lista**, **/cliente/visualiza**, **/cliente/remove** e **/cliente/atualiza** sejam disponibilizadas, cada uma invocando o respectivo método em sua classe.

```
@Resource
public class ClienteController {

  public void adiciona(Cliente cliente) {

  }

  public List<Cliente> lista() {
    return ...
  }

  public Cliente visualiza(Cliente perfil) {
    return ...
  }

  public void remove(Cliente cliente) {

  ...
  }


  public void atualiza(Cliente cliente) {
    ...
  }

}
```

3.2- Parâmetros dos métodos

Você pode receber parâmetros nos métodos dos seus controllers, e se o objeto usar a convenção de java

beans (getters e setters para os atributos da classe), você pode usar pontos para navegar entre os atributos. Por exemplo, no método:

```
public void atualiza(Cliente cliente) {
   //...
}
```

você pode passar como parâmetro na requisição:

```
cliente.id=3
cliente.nome=Fulano
cliente.usuario.login=fulano
```

e os campos correspondentes, navegando via getters e setters a partir do cliente, serão setados.

Se um atributo do objeto ou parâmetro do método for uma lista (List<> ou array), você pode passar vários parâmetros usando colchetes e índices:

```
cliente.telefones[0]=(11) 5571-2751 #no caso de ser uma lista de String
cliente.dependentes[0].id=1 #no caso de ser qualquer objeto, você pode continuar a navegação
cliente.dependentes[3].id=1 #os índices não precisam ser sequenciais
cliente.dependentes[0].nome=Cicrano #se usar o mesmo índice, vai ser setado no mesmo objeto
clientes[1].id=23 #funciona se você receber uma lista de clientes no método
```

Reflection no nome dos parâmetros

Infelizmente, o Java não realiza reflection em cima de parâmetros, esses dados não ficam disponíveis em bytecode (a não ser se compilado em debug mode, porém é algo opcional). Isso faz com que a maioria dos frameworks que precisam desse tipo de informção criem uma anotação própria para isso, o que polui muito o código (exemplo no JAX-WS, onde é comum encontrar um método como o acima com a assinatura `void add(@WebParam(name="cliente”) Cliente cliente)`.
O VRaptor tira proveito do framework Paranamer (http://paranamer.codehaus.org), que consegue tirar essa informação através de pré compilação ou dos dados de debug, evitando criar mais uma anotação. Alguns dos desenvolvedores do VRaptor também participam no desenvolvimento do Paranamer.

3.3- Escopos

Por vezes você deseja compartilhar um componente entre todos os usuários, entre todas as requisições de um mesmo usuário ou a cada requisição de um usuário.

Para definir em que escopo o seu componente vive, basta utilizar as anotações `@ApplicationScoped`, `@SessionScoped` e `@RequestScoped`.

Caso nenhuma anotação seja utilizada, o VRaptor assume que seu componente ficará no escopo de request, isto é, você terá um novo componente a cada nova requisição.

3.4- Http Methods

O ideal é definir uma URI específica para diversos métodos HTTP diferentes, como GET, POST, PUT etc.

Para atingir esse objetivo, utilizamos as anotações `@Get`, `@Post`, `@Delete` etc juntamente com a anotação `@Path` que permite configurar uma URI diferente da URI padrão.

O exemplo a seguir altera os padrões de URI do `ClienteController` para utilizar duas URIs distintas, com diversos métodos HTTP:

```
@Resource
public class ClienteController {

  @Path("/cliente")
  @Post
  public void adiciona(Cliente cliente) {
  }

  @Path("/")
  public List<Cliente> lista() {
    return ...
  }

  @Get
  @Path("/cliente")
  public Cliente visualiza(Cliente cliente) {
    return ...
  }

  @Delete
  @Path("/cliente")
  public void remove(Cliente cliente) {
    ...
  }

  @Put
  @Path("/cliente")
  public void atualiza(Cliente cliente) {
    ...
  }

}
```

Como você pode notar, utilizamos os métodos HTTP + uma URI específica para identificar cada um dos métodos de minha classe Java.

No momento de criar os links e formulários HTML devemos tomar um cuidado **muito importante** pois os browsers só dão suporte aos métodos *POST* e *GET* através de formulários hoje em dia.

Por isso, você deverá criar as requisições para métodos do tipo *DELETE*, *PUT* etc através de JavaScript ou passando um parâmetro extra, chamado **_method**.

Esse parâmetro sobrescreverá o método HTTP real sendo invocado.

O exemplo a seguir mostra um link para o método visualiza de cliente:

```
<a href="/cliente?cliente.id=5">ver cliente 5</a>
```

Agora um exemplo de como invocar o método de adicionar um cliente:

```
<form action="/cliente" method="post">
    <input name="cliente.nome" />
    <input type="submit" />
</form>
```

E, note que para permitir a remoção através do método *DELETE*, temos que usar o parâmetro `_method`, uma vez que o browser não suporta ainda tais requisições:

```
<form action="/cliente" method="post">
    <input name="_method" value="DELETE" type="hidden" />
    <input name="cliente.id" value="5" type="hidden" />
    <input type="submit" value="remover cliente 5" />
</form>
```

3.5- @Path

A anotação `@Path` permite que você customize as URIs de acesso a seus métodos. O uso básico dessa anotação é feito através de uma URI fixa. O exemplo a seguir mostra a customização de uma URI para um método, que somente receberá requisições do tipo *POST* na URI /**cliente**:

```
@Resource
public class ClienteController {

  @Path("/cliente")
  @Post
  public void adiciona(Cliente cliente) {
  }

}
```

Path com injeção de variáveis

Em diversos casos desejamos que a *uri* que identifica meu recurso tenha como parte de seu valor, por exemplo, o identificador único de meu recurso.

Suponha o exemplo de um controle de clientes onde meu identificador único (chave primária) é um número, podemos então mapear as uris */cliente/{cliente.id}* para a visualização de cada cliente.

Isto é, se acessarmos a uri */cliente/2*, o método **visualiza** será invocado e o parâmetro *cliente.id* será setado para **2**. Caso a uri */cliente/1717* seja acessada, o mesmo método será invocado com o valor **1717**.

Dessa maneira criamos uris únicas para identificar recursos diferentes do nosso sistema. Veja o exemplo citado:

```
@Resource
public class ClienteController {

  @Get
  @Path("/cliente/{cliente.id}")
  public Cliente visualiza(Cliente cliente) {
    return ...
  }

}
```

Você pode ir além e setar diversos parâmetros através da uri:

```
@Resource
public class ClienteController {

  @Get
  @Path("/cliente/{cliente.id}/visualiza/{secao}")
  public Cliente visualiza(Cliente cliente, String secao) {
    return ...
  }

}
```

Paths com *

Você também pode utilizar o * como método de seleção para a sua uri. O exemplo a seguir ignora qualquer coisa após a palavra *foto/* :

```
@Resource
public class ClienteController {

  @Get
  @Path("/cliente/{cliente.id}/foto/*")
  public File foto(Cliente cliente) {
    return ...
  }

}
```

E agora o mesmo código, mas utilizado para baixar uma foto específica de um cliente:

```
@Resource
public class ClienteController {

  @Get
  @Path("/cliente/{cliente.id}/foto/{foto.id}")
  public File foto(Cliente cliente, Foto foto) {
    return ...
  }

}
```

Por vezes você deseja que o parâmetro a ser setado inclua também /s. Para isso você deve utilizar o padrão {...*}:

```
@Resource
public class ClienteController {

  @Get
  @Path("/cliente/{cliente.id}/download/{path*}")
  public File download(Cliente cliente, String path) {
    return ...
  }
```

```
}
```

Definindo prioridades para seus paths

É possível que algumas das nossas URIs possa ser tratada por mais de um método da classe:

```
@Resource
public class PostController {

  @Get
  @Path("/post/{post.autor}")
  public void mostra(Post post) { ... }

  @Get
  @Path("/post/atual")
  public void atual() { ... }
}
```

A uri `/post/atual` pode ser tratada tanto pelo método `mostra`, quanto pelo `atual`. Mas eu quero que quando seja exatamente `/post/atual` o método executado seja o atual. O que eu quero é que o VRaptor teste primeiro o path do método `atual`, para não correr o risco de cair no método `mostra`.

Para fazer isso, podemos definir prioridades para os `@Path`s, assim o VRaptor vai testar primeiro os paths com maior prioridade, ou seja, valor menor de prioridade:

```
@Resource
public class PostController {

  @Get
  @Path(priority = 2, value = "/post/{post.autor}")
  public void mostra(Post post) { ... }

  @Get
  @Path(priority = 1, value = "/post/atual")
  public void atual() { ... }
}
```

Assim, o path "/post/atual" vai ser testado antes do "/post/{post.autor}", e o VRaptor vai fazer o que a gente gostaria que ele fizesse.

Você não precisa definir prioridades se tivermos as uris: `/post/{post.id}` e `/post/atual`, pois na `/post/{post.id}` o vraptor só vai aceitar números.

3.6- RoutesConfiguration

Por fim, a maneira mais avançada de configurar rotas de acesso aos seus recursos é através de um **RoutesConfiguration**.

Esse componente deve ser configurado no escopo de aplicação e implementar o método config:

```
@Component
@ApplicationScoped
```

```
public class CustomRoutes implements RoutesConfiguration {

    public void config(Router router) {
    }

}
```

De posse de um **Router**, você pode definir rotas para acesso a métodos e, o melhor de tudo, é que a configuração é refactor-friendly, isto é, se você alterar o nome de um método, a configuração também altera, mas mantem a mesma *uri* :

```
@Component
@ApplicationScoped
public class CustomRoutes implements RoutesConfiguration {

    public void config(Router router) {
        new Rules(router) {
            public void routes() {
                routeFor("/").is(ClienteController.class).list();
                routeFor("/cliente/random").is(ClienteController.class).aleatorio();
            }
        };
    }

}
```

Você pode também colocar parâmetros na uri e eles vão ser populados diretamente nos parâmetros do método. Você pode ainda adicionar restrições para esses parâmetros:

```
// O método mostra recebe um Cliente que tem um id
routeFor("/cliente/{cliente.id}").is(ClienteController.class).mostra(null);
// Se eu quiser garantir que o parametro seja um número:
routeFor("/cliente/{cliente.id}").withParameter("cliente.id").matching("\\d+")
            .is(ClienteController.class).mostra(null);
```

Por fim, você pode escolher o nome da classe e o nome do método em tempo de execução, que permite criar rotas extremamente genéricas:

```
routeFor("/{webResource}/facaAlgo/{webMethod}").is(
    type("br.com.caelum.nomedoprojeto.{webResource}"),
    method("{webMethod}"));
```

Componentes

4.1- O que são componentes?

Componentes são instâncias de classes que seu projeto precisa para executar tarefas ou armazenar estados em diferentes situações.

Exemplos clássicos de uso de componentes seriam os Daos, enviadores de email etc.

A sugestão de boa prática indica *sempre* criar uma interface para seus componentes. Dessa maneira seu código também fica mais fácil de testar unitariamente.

O exemplo a seguir mostra um componente a ser gerenciado pelo vraptor:

```
@Component
public class ClienteDao {

  private final Session session;
  public ClienteDao(Session session) {
      this.session = session;
  }

  public void adiciona(Cliente cliente) {
        session.save(cliente);
  }

}
```

4.2- Escopos

Assim como os recursos, os componentes vivem em um escopo específico e seguem as mesmas regras, por padrão pertencendo ao escopo de requisicão, isto é, a cada nova requisição seu componente será novamente instanciado.

O exemplo a seguir mostra o fornecedor de conexoes com o banco baseado no hibernate. Esse fornecedor esta no escopo de aplicacação, portanto será instanciado somente uma vez por contexto:

```
@ApplicationScoped
@Component
public class HibernateControl {

  private final SessionFactory factory;
  public HibernateControl() {
      this.factory = new AnnotationConfiguration().configure().buildSessionFactory();
  }

  public Session getSession() {
      return factory.openSession();
```

```
  }

}
```

4.3- ComponentFactory

Muitas vezes você quer receber como dependência da sua classe alguma classe que não é do seu projeto, como por exemplo uma Session do hibernate ou um EntityManager da JPA.

Para poder fazer isto, basta criar um ComponentFactory:

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

    private final SessionFactory factory;
    private Session session;

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

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

    public Session getInstance() {
        return session;
    }

    @PreDestroy
    public void destroy() {
        this.session.close();
    }

}
```

Note que você pode adicionar os listeners @PostConstruct e @PreDestroy para controlar a criação e destruição dos recursos que você usa. Isso funciona para qualquer componente que você registrar no VRaptor.

4.4- Injeção de dependências

O VRaptor utiliza um de seus provedores de injeção de dependências para controlar o que é necessário para instanciar cada um de seus componentes e recursos.

Sendo assim, os dois exemplos anteriores permitem que qualquer um de seus recursos ou componentes receba um ClienteDao em seu construtor, por exemplo:

```
@Resource
public class ClienteController {
    private final ClienteDao dao;

    public ClienteController(ClienteDao dao) {
        this.dao = dao;
```

```
    }

    @Post
    public void adiciona(Cliente cliente) {
        this.dao.adiciona(cliente);
    }

}
```

Conversores

5.1- Padrão

Por padrão o VRaptor já registra diversos conversores para o seu uso no dia a dia.

5.2- Tipos primitivos

Todos os tipos primitivos (int, long etc) são suportados.

Caso o parametro da requisição seja nulo ou a string vazia, variáveis de tipo primitivo serão alterados para o valor padrão como se essa variável fosse uma variável membro, isto é:

- boolean - false
- short, int, long, double, float, byte - 0
- char - caracter de código 0

5.3- Wrappers de tipos primitivos

Todos os wrappers dos tipos primitivos (Integer, Long, Character, Boolean etc) são suportados.

5.4- Enum

Todas as enumerações são suportadas através do nome do elemento ou de seu ordinal. No exemplo a seguir, tanto o valor 1 como o valor DEBITO são traduzidos para Tipo.DEBITO:

```
public enum Tipo {
    CREDITO, DEBITO
}
```

5.5- BigInteger e BigDecimal

Ambos são suportados utilizando o padrão de localização da virtual machine que serve a sua aplicação. Para criar suporte a números decimais baseados no locale do usuário, basta olhar o funcionamento da classe LocaleBasedCalendarConverter.

5.6- Calendar e Date

LocaleBasedCalendarConverter e LocaleBasedDateConverter utilizam o locale do usuário, definido seguindo o padrão do jstl para entender a formatação que foi utilizada no parâmetro.

Por exemplo, se o locale é pt-br, o formato “18/09/1981” representa 18 de setembro de 1981 enquanto para o locale en, o formato “09/18/1981” representa a mesma data.

5.7- LocalDate e LocalTime do joda-time

Existem conversores para esses dois tipos no VRaptor e eles só serão carregados se você tiver o joda-time.jar no seu classpath

5.8- Interface

Todos os conversores devem implementar a interface Converter do vraptor. A classe concreta definirá o tipo que ela é capaz de converter, e será invocada com o valor do parâmetro do request, o tipo alvo e um bundle com as mensagens de internacionalização para que você possa retornar uma ConversionException no caso de algum erro de conversão.

```
public interface Converter<T> {
    T convert(String value, Class<? extends T> type, ResourceBundle bundle);
}
```

Além disso, seu conversor deve ser anotado dizendo agora para o VRaptor (e não mais para o compilador java) qual o tipo que seu conversor é capaz de converter, para isso utilize a anotação @Convert:

```
@Convert(Long.class)
public class LongConverter implements Converter<Long> {
    // ...
}
```

Por fim, lembre-se de dizer se seu conversor pode ser instanciado em um escopo de Application, Session ou Request, assim como recursos e componentes quaisquer do VRaptor. Por exemplo, um conversor que não requer nenhuma informação do usuário logado pode ser registrado no escopo de Application e instanciado uma única vez:

```
@Convert(Long.class)
@ApplicationScoped
public class LongConverter implements Converter<Long> {
    // ...
}
```

A seguir, a implementação de LongConverter mostra como tudo isso pode ser utilizado de maneira bem simples:

```
@Convert(Long.class)
@ApplicationScoped
public class LongConverter implements Converter<Long> {

    public Long convert(String value, Class<? extends Long> type, ResourceBundle bundle) {
        if (value == null || value.equals("")) {
            return null;
        }
        try {
            return Long.valueOf(value);
```

```
        } catch (NumberFormatException e) {
                    throw new ConversionError(MessageFormat
                        .format(bundle.getString("is_not_a_valid_integer"), value));
        }
    }

}
```

5.9- Registrando um novo conversor

Não é necessária nenhuma configuração além do @Convert e implementar a interface Converter para que o conversor seja registrado no Container do VRaptor.

Interceptadores

6.1- Para que interceptar

O uso de interceptadores é feito para executar alguma tarefa antes e/ou depois de uma lógica de negócios, sendo os usos mais comuns para validação de dados, controle de conexão e transação do banco, log e criptografia/compactação de dados.

6.2- Como interceptar

No VRaptor 3 utilizamos uma abordagem onde o interceptador define quem será interceptado, muito mais próxima a abordagens de interceptação que aparecem em sistemas baseados em AOP (programação orientada a aspectos) do que a abordagem da versão anterior do vraptor.

Portanto, para interceptar uma requisição basta implementar a interface **Interceptor** e anotar a classe com a anotação **@Intercepts**.

Assim como qualquer outro componente, você pode dizer em que escopo o interceptador, será armazenado através das anotações de escopo.

```
public interface Interceptor {

    void intercept(InterceptorStack stack, ResourceMethod method,
                   Object resourceInstance) throws InterceptionException;

    boolean accepts(ResourceMethod method);

}
```

6.3- Exemplo simples

A classe a seguir mostra um exemplo de como interceptar todas as requisições em um escopo de requisição e simplesmente mostrar na saída do console o que está sendo invocado.

Lembre-se que o interceptador é um componente como qualquer outro e pode receber em seu construtor quaisquer dependencias atraves de Injecao de Dependencias.

```
@Intercepts
@RequestScoped
public class Log implements Interceptor {

    private final HttpServletRequest request;

    public Log(HttpServletRequest request) {
        this.request = request;
    }
```

```
    /*
     * Este interceptor deve interceptar o method dado? Neste caso vai interceptar
     * todos os métodos.
     * method.getMethod() retorna o método java que está sendo executado
     * method.getResourceClass().getType() retorna a classe que está sendo executada
     */
    public boolean accepts(ResourceMethod method) {
        return true;
    }

    public void intercept(InterceptorStack stack, ResourceMethod method,
                          Object resourceInstance) throws InterceptionException {
        System.out.println("Interceptando " + request.getRequestURI());
        // código a ser executado antes da lógica

        stack.next(method, resourceInstance); // continua a execução

        // código a ser executádo depois da lógica
    }

}
```

6.4- Exemplo com Hibernate

Provavelmente, um dos usos mais comuns do Interceptor é para a implementação do famigerado pattern Open Session In View, que fornece uma conexão com o banco de dados sempre que há uma requisição para sua aplicação. E ao fim dessa requisição, a conexão é liberada. O grande ganho disso é evitar exceções como LazyInitializationException no momento da renderização dos jsps.

Abaixo, está um simples exemplo, que para todas as requisições abre uma transação com o banco de dados. E ao fim da execução da lógica e da exibição da página para o usuário, commita a transação e logo em seguida fecha a conexão com o banco.

```
@RequestScoped
@Intercepts
public class DatabaseInterceptor implements br.com.caelum.vraptor.Interceptor {

    private final Database controller;

    public DatabaseInterceptor(Database controller) {
        this.controller = controller;
    }

    public void intercept(InterceptorStack stack, ResourceMethod method, Object instance)
        throws InterceptionException {
        try {
            controller.beginTransaction();
            stack.next(method, instance);
            controller.commit();
        } finally {
            if (controller.hasTransaction()) {
                controller.rollback();
            }
            controller.close();
        }
```

```
    }

    public boolean accepts(ResourceMethod method) {
        return true;
    }

}
```

Dessa forma, no seu Recurso, bastaria o seguinte código para utilizar a conexão disponível:

```
@Resource
public class FuncionarioController {

    public FuncionarioController(Result result, Database controller) {
        this.result = result;
        this.controller = controller;
    }

    @Post
    @Path("/funcionario")
    public void adiciona(Funcionario funcionario) {
        controller.getFuncionarioDao().adiciona(funcionario);
        ...
    }
}
```

6.5- Como garantir ordem: InterceptorSequence

Se você precisa garantir a ordem em que os interceptors são executados, basta implementar a interface InterceptorSequence e passar a ordem em que você deseja executar os interceptors:

```
@Intercepts
public class MinhaSequencia implements InterceptorSequence {
    public Class<? extends Interceptor>[] getSequence() {
        return new Class[] { PrimeiroInterceptor.class, SegundoInterceptor.class };
    }
}
```

Você não deve anotar os interceptors retornados pela InterceptorSequence com @Intercepts.

Validação

O VRaptor3 suporta 2 estilos de validação. Clássico e fluente. A porta de entrada para ambos os estilos é a interface Validator. Para que seu recurso tenha acesso ao Validator, basta recebê-lo no construtor do seu recurso:

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

...

@Resource
class FuncionarioController {
    private Validator validator;

    public FuncionarioController(Validator validator) {
        this.validator = validator;
    }
}
```

7.1- Estilo clássico

A forma clássica é semelhante a forma como as validações eram feitas no VRaptor2. Dentro da sua lógica de negócios, basta fazer a verificação que deseja e caso haja um erro de validação, adicionar esse erro na lista de erros de validação. Por exemplo, para validar que o nome do funcionario deve ser Fulano, faça:

```
public void adiciona(Funcionario funcionario) {
    if(! funcionario.getNome().equals("Fulano")) {
        validator.add(new ValidationMessage("erro","nomeInvalido"));
    }
    validator.onErrorUse(page()).of(FuncionarioController.class).formulario();
    dao.adiciona(funcionario);
}
```

Ao chamar o validator.onErrorUse, se existirem erros de validação, o VRaptor para a execução e redireciona a página que você indicou. O redirecionamento funciona da mesma forma que o result.use(..).ed

7.2- Estilo fluente

No estilo fluente, a idéia é que o código para fazer a validação seja algo muito parecido com a linguagem natural. Por exemplo, caso queiramos obrigar que seja informado o nome do funcionario:

```
public adiciona(Funcionario funcionario) {
    validator.checking(new Validations(){{
        that(!funcionario.getNome().isEmpty(), "erro","nomeNaoInformado");
    }});
```

```
    validator.onErrorUse(page()).of(FuncionarioController.class).formulario();

    dao.adiciona(funcionario);
}
```

Você pode ler esse código como: “Validador, cheque as minhas validações. A primeira validação é que o nome do funcionário não pode ser vazio”. Bem mais próximo a linguagem natural.

Assim sendo, caso o nome do funcionario seja vazio, ele vai ser redirecionado novamente para a logica “formulario”, que exibe o formulario para que o usuário adicione o funcionário novamente. Além disso, ele devolve para o formulario a mensagem de erro que aconteceu na validação.

Muitas vezes algumas validações só precisam acontecer se uma outra deu certo, por exemplo, eu só vou checar a idade do usuário se o usuário não for null. O método that retorna um boolean dizendo se o que foi passado pra ele é válido ou não:

```
validator.checking(new Validations(){{
    if (that(usuario != null, "usuario", "usuario.nulo")) {
        that(usuario.getIdade() >= 18, "usuario.idade", "usuario.menor.de.idade");
    }
}});
```

Desse jeito a segunda validação só acontece se a primeira não falhou.

7.3- Validação usando matchers do Hamcrest

Você pode também usar matchers do Hamcrest para deixar a validação mais legível, e ganhar a vantagem da composição de matchers e da criação de novos matchers que o Hamcrest te oferece:

```
public admin(Funcionario funcionario) {
    validator.checking(new Validations(){{
        that(funcionario.getRoles(), hasItem("ADMIN"), "admin","funcionario.nao.eh.admin");
    }});
    validator.onErrorUse(page()).of(LoginController.class).login();
    dao.adiciona(funcionario);
}
```

7.4- Hibernate validator

O VRaptor também suporta integração com o HibernateValidator. No exemplo anterior para validar o objeto Funcionario usando o Hibernate Validator basta adicionar uma linha de código:

```
public adiciona(Funcionario funcionario) {
    //Validação do Funcionario com Hibernate Validator
    validator.add(Hibernate.validate(funcionario));

    validator.checking(new Validations(){{
        that(!funcionario.getNome().isEmpty(), "erro","nomeNaoInformado");
    }});

    dao.adiciona(funcionario);
}
```

7.5- Para onde redirecionar no caso de erro

Outro ponto importante que deve ser levado em consideração no momento de fazer validações é o redirecionamento quando ocorrer um erro. Como enviamos o usuário para outro recurso com o VRaptor3, caso haja erro na validação?

Simples, apenas diga no seu código que quando correr um erro, é para o usuário ser enviado para algum recurso. Como no exemplo:

```
public adiciona(Funcionario funcionario) {
    //Validação na forma fluente
    validator.checking(new Validations(){{
        that("erro","nomeNaoInformado", !funcionario.getNome().isEmpty());
    }});
    //Validação na forma clássica
    if(! funcionario.getNome().equals("Fulano")) {
        validator.add(new ValidationMessage("erro","nomeInvalido"));
    }
    validator.onErrorUse(page()).of(FuncionarioController.class).formulario();

    dao.adiciona(funcionario);
}
```

Note que se sua lógica adiciona algum erro de validação você **precisa** dizer pra onde o VRaptor deve ir. O validator.onErrorUse funciona do mesmo jeito que o result.use: você pode usar qualquer view da classe Results.

View e Ajax

8.1- Custom PathResolver

Por padrão, para renderizar suas views, o VRaptor segue a convenção:

```
public class ClientsController {
    public void list() {
        //...
    }
}
```

Este método acima renderizará a view `/WEB-INF/jsp/clients/list.jsp`.

No entanto, nem sempre queremos esse comportamento, e precisamos usar algum template engine, como por exemplo, Freemarker ou Velocity, e precisamos mudar essa convenção.

Um jeito fácil de mudar essa convenção é estendendo a classe `DefaultPathResolver`:

```
@Component
public class FreemarkerPathResolver extends DefaultPathResolver {
    protected String getPrefix() {
        return "/WEB-INF/freemarker/";
    }

    protected String getExtension() {
        return "ftl";
    }
}
```

Desse jeito, a lógica iria renderizar a view `/WEB-INF/freemarker/clients/list.ftl`. Se ainda assim isso não for o suficiente você pode implementar a interface `PathResolver` e fazer qualquer convenção que você queira, não esquecendo de anotar a classe com @Component.

8.2- View

Se você quiser mudar a view de alguma lógica específica você pode usar o objeto Result:

```
@Resource
public class ClientsController {

    private final Result result;

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

```
    public void list() {}

    public void save(Client client) {
        //...
        this.result.use(Results.logic()).redirectTo(ClientsController.class).list();
    }
}
```

Por padrão existem estes tipos de views implementadas:

- Results.logic(), que vai redirecionar para uma outra lógica qualquer do sistema
- Results.page(), que vai redirecionar diretamente para uma página, podendo ser um jsp, um html, ou qualquer uri relativa ao web application dir, ou ao contexto da aplicação.
- Results.http(), que manda informações do protocolo HTTP como status codes e headers.
- Results.referer(), que usa o header Referer para fazer redirects ou forwards.
- Results.nothing(), apenas retorna o código de sucesso (HTTP 200 OK).

8.3- Redirecionamento e forward

No VRaptor3, podemos tanto redirecionar ou fazermos um forward do usuário para uma outra lógica ou um jsp. A grande diferença entre fazer um redirecionamento (redirect) e um forward é que o redirecionamento acontece no lado do cliente, e o forward acontece no lado do servidor.

Um bom exemplo de uso do redirect, é o padrão 'redirect-after-post', por exemplo, quando você adiciona um cliente, e quer retornar para a listagem dos clientes, porém, não quer permitir que o usuário atualize a página (F5) e reenvie toda a requisição, acarretando em dados duplicados.

No caso do forward, um exemplo de uso é quando você possui uma validação e essa validação falhou, geralmente você quer que o usuário continue na mesma tela do formulário com os dados da requisição preenchidos.

8.4- Accepts e o parâmetro _format

Muitas vezes precisamos renderizar formatos diferentes para uma mesma lógica. Por exemplo queremos retornar um JSON, ao invés de um HTML. Para fazer isso, podemos definir o Header Accepts da requisição para que aceite o tipo desejado, ou colocar um parâmetro `_format` na requisição.

Se o formato for JSON, a view renderizada por padrão será: `/WEB-INF/jsp/{controller}/{logic}.json.jsp`, ou seja, em geral será renderizada a view: `/WEB-INF/jsp/{controller}/{logic}.{formato}.jsp`. Se o formato for HTML você não precisa colocá-lo no nome do arquivo.

O parâmetro `_format` tem prioridade sobre o header Accepts.

8.5- Ajax: construindo na view

Para devolver um JSON na sua view, basta que sua lógica disponibilize o objeto para a view, e dentro da view você forme o JSON como desejar. Como no exemplo, o seu `/WEB-INF/jsp/clients/load.json.jsp`:

```
{ nome: '${client.name}', id: '${client.id}' }
```

E na lógica:

```
@Resource
public class ClientsController {

    private final Result result;
    private final ClientDao dao;

    public ClientsController(Result result, ClientDao dao) {
        this.result = result;
        this.dao = dao;
    }

    public void load(Client client) {
        result.include("client", dao.load(client));
    }
}
```

Injeção de dependências

O VRaptor está fortemente baseado no conceito de injeção de dependências uma vez que chega até mesmo a utilizar dessa idéia para juntar seus componentes internos.

O conceito básico por trás de Dependency Injection (DI) é que você não deve buscar aquilo que deseja acessar mas tudo o que deseja acessar deve ser fornecido para você.

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

```
@Component

public class ClienteController {
    private final ClienteDao dao;

    public ClienteController(ClienteDao dao) {
        this.dao = dao;
    }

    @Post
    public void adiciona(Cliente cliente) {
        this.dao.adiciona(cliente);
    }

}
```

E anote também o componente ClienteDao como sendo controlado pelo vraptor:

```
@Component

public class ClienteDao {
}
```

A partir desse instante, o vraptor fornecerá uma instância de ClienteDao para seu ClienteController sempre que precisar instanciá-lo. Vale lembrar que o VRaptor honrará o escopo de cada componente. Por exemplo, se ClienteDao fosse de escopo Session (@SessionScoped), seria criada uma única instância desse componente por sessão. (note que é provavelmente errado usar um dao no escopo de session, isto é um mero exemplo).

9.1- ComponentFactory

Em diversos momentos queremos que nossos componentes recebam componentes de outras bibliotecas. Nesse caso não temos como alterar o código fonte da biblioteca para adicionar a anotação @Component (além de possíveis alterações requeridas na biblioteca).

O exemplo mais famoso envolve adquirir uma Session do Hibernate. Nesses casos precisamos criar um componente que possui um único papel: fornecer instâncias de Session para os componentes que precisam

dela.

O VRaptor possui uma interface chamada ComponentFactory que permite que suas classes possuam tal responsabilidade. Implementações dessa interface definem um único método. Veja o exemplo a seguir, que inicializa o Hibernate na construção e utiliza essa configuração para fornecer sessões para nosso projeto:

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

    private SessionFactory factory;

    @PostConstruct
    public void create() {
        factory = new AnnotationConfiguration().configure();
    }

    public SessionFactory getInstance() {
        return factory;
    }

    @PreDestroy
    public void destroy() {
        factory.close();
    }

}

@Component
@RequestScoped
public class SessionCreator implements ComponentFactory<Session> {

    private final SessionFactory factory;
    private Session session;

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

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

    public Session getInstance() {
        return session;
    }

    @PreDestroy
    public void destroy() {
        this.session.close();
    }


}
```

Essas implementações já estão disponíveis no código do VRaptor. Para usá-la veja o capítulo de utils.

9.2- Providers

Por trás dos panos, o VRaptor utiliza um provider de DI específico. Por padrão o vraptor vêm com suporte ao uso interno do Picocontainer ou do Spring IoC.

Cada implementação disponibiliza tudo o que você encontra na documentação do vraptor, mas acaba por fornecer também pontos de extensão diferentes, claro.

9.3- Spring

Ao utilizar o Spring, você ganha todas as características e componentes prontos do Spring para uso dentro do VRaptor, isto é, todos os componentes que funcionam com o Spring DI/Ioc, funcionam com o VRaptor. Nesse caso, todas as anotações.

Você não precisa fazer nada para usar o Spring, pois é o container padrão.

O VRaptor vai usar suas configurações do Spring, caso você já o tenha configurado no seu projeto (os listeners e o applicationContext.xml). Caso o VRaptor não tenha encontrado sua configuração, veja o capítulo de configurações avançadas.

9.4- Pico Container

Ao utilizar o Picocontainer por baixo do VRaptor, você poderá acessar o pico diretamente para fazer configurações avançadas que desejar.

Para utilizar o Picocontainer como provider de sua aplicação, basta colocar no seu arquivo web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.caelum.vraptor.ioc.pico.PicoProvider</param-value>
</context-param>
```

9.5- Seu próprio provider

Você também pode criar seu próprio Provider, seja para adicionar novas características avançadas a implementação do Picocontainer ou do Spring, ou ainda para se basear em outro contâiner de DI que seja de sua preferência.

Downloading

10.1- exemplo de 1 minuto: download

O exemplo a seguir mostra como disponibilizar o download para seu cliente.

Note novamente a simplicidade na implementação:

```
@Resource
public class PerfilController {

    public File foto(Perfil perfil) {
        return new File("/path/para/a/foto." + perfil.getId()+ ".jpg");
    }
}
```

10.2- Adicionando mais informações no download

Se você quiser adicionar mais informações ao download você pode retornar um FileDownload:

```
@Resource
public class PerfilController {

    // dao ...

    public Download foto(Perfil perfil) {
        File file = new File("/path/para/a/foto." + perfil.getId()+ ".jpg");
        String contentType = "image/jpg";
        String filename = perfil.getNome() + ".jpg";

        return new FileDownload(file, contentType, filename);
    }
}
```

10.3- exemplo de 1 minuto: upload

O primeiro exemplo será baseado na funcionalidade de upload multipart.

```
@Resource
public class PerfilController {

    private final PerfilDao dao;

    public PerfilController(PerfilDao dao) {
        this.dao = dao;
```

```
    }

    public void atualizaFoto(Perfil perfil, UploadedFile foto) {
        dao.atribui(foto.getFile(), perfil);
    }
}
```

10.4- Mais sobre Upload

O UploadedFile retorna o arquivo como um InputStream. Se você quiser copiar para um arquivo no disco facilmente, basta usar o `IOUtils` do `commons-io`, que já é dependência do VRaptor:

```
public void atualizaFoto(Perfil perfil, UploadedFile foto) {
    File fotoSalva = new File();
    IOUtils.copy(foto.getFile(), new PrintWriter(fotoSalva));
    dao.atribui(fotoSalva, perfil);
}
```

Componentes Utilitários Opcionais

11.1- Registrando um componente opcional

O VRaptor possui alguns componentes opcionais, que estão no pacote `br.com.caelum.vraptor.util`. Para registrá-los você pode fazer o seguinte:

- Crie uma classe filha do Provider que você está usando:

```
package br.com.nomedaempresa.nomedoprojeto;


public class CustomProvider extends SpringProvider {

}
```

- Registre essa classe como provider no web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.nomedaempresa.nomedoprojeto.CustomProvider</param-value>
</context-param>
```

- Sobrescreva o método registerCustomComponents e adicione os componentes opcionais:

```
package br.com.nomedaempresa.nomedoprojeto;


public class CustomProvider extends SpringProvider {

    @Override
    protected void registerCustomComponents(ComponentRegistry registry) {
        registry.register(ComponenteOpcional.class, ComponenteOpcional.class);
    }
}
```

11.2- Hibernate Session e SessionFactory

Se você precisa de Session's e SessionFactory nos seus componentes, você geralmente vai precisar de um ComponentFactory para criá-los. Se você usa entidades anotadas, e o hibernate.cfg.xml na raiz do WEB-INF/classes, você pode usar as ComponentFactory's para isso que já vêm com o VRaptor. O que você precisa fazer é:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(SessionCreator.class, SessionCreator.class);
    registry.register(SessionFactoryCreator.class, SessionFactoryCreator.class);
```

```
}
```

Inclusive você pode habilitar um interceptor que controla a transação do Hibernate:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(HibernateTransactionInterceptor.class,
                      HibernateTransactionInterceptor.class);
}
```

11.3- JPA EntityManager e EntityManagerFactory

Se você tiver um persistence.xml com o persistence-unit chamado “default”, você pode usar os ComponentFactories para criar EntityManager e EntityManagerFactory já disponíveis no vraptor:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(EntityManagerCreator.class, EntityManagerCreator.class);
    registry.register(EntityManagerFactoryCreator.class,
                      EntityManagerFactoryCreator.class);
}
```

Inclusive você pode habilitar um interceptor que controla a transação da JPA:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(JPATransactionInterceptor.class,
                      JPATransactionInterceptor.class);
}
```

Configurações avancadas: sobrescrevendo as convenções e comportamento do VRaptor

12.1- Mudando a view renderizada por padrão

Se você precisa mudar a view renderizada por padrão, ou mudar o local em que ela é procurada, basta criar a seguinte classe:

```
@Component
public class CustomPathResolver extends DefaultPathResolver {

    @Override
    protected String getPrefix() {
        return "/pasta/raiz/";
    }

    @Override
    protected String getExtension() {
        return "ftl"; // ou qualquer outra extensão
    }

    @Override
    protected String extractControllerFromName(String baseName) {
        return //sua convenção aqui
               //ex.: Ao invés de redirecionar UserController para 'user'
               //você quer redirecionar para 'userResource'
               //ex.2: Se você sobrescreveu a conveção para nome dos Controllers para XXXResource
               //e quer continuar redirecionando para 'user' e não para 'userResource'
    }

}
```

Se você precisa mudar mais ainda a convenção basta implementar a interface PathResolver.

12.2- Mudando a URI padrão

Por padrão, a URI para o método ClientesController.lista() é /clientes/lista, ou seja, nome_do_controller/nome_do_metodo. Para sobrescrever essa convenção, basta criar a classe:

```
@Component
@ApplicationScoped
public class MeuRoutesParser extends PathAnnotationRoutesParser {
    //delegate constructor
    protected String extractControllerNameFrom(Class<?> type) {
        return //sua convenção aqui
    }
```

```
    protected String defaultUriFor(String controllerName, String methodName) {
        return //sua convenção aqui
    }
}
```

Se você precisa mudar mais ainda a convenção basta implementar a interface RoutesParser.

12.3- Mudando o IoC provider

O IoC provider padrão é o spring. Para mudá-lo basta colocar no web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.classe.do.seu.provider.Preferido</param-value>
</context-param>
```

Entre os padrão existem: br.com.caelum.vraptor.ioc.spring.SpringProvider e br.com.caelum.vraptor.ioc.pico.PicoProvider. Você pode ainda estender alguma dessas duas classes e usar seu próprio provider.

12.4- Mudando ApplicationContext base do Spring

Caso o VRaptor não esteja usando o seu ApplicationContext como base, basta estender o SpringProvider e implementar o método getParentApplicationContext, passando o ApplicationContext da sua aplicação:

```
package br.com.nomedaempresa.nomedoprojeto;
public class CustomProvider extends SpringProvider {
    public ApplicationContext getParentApplicationContext(ServletContext context) {
        ApplicationContext applicationContext = //lógica pra criar o applicationContext
        return applicationContext;
    }
}
```

e mudar o provider no web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.nomedaempresa.nomedoprojeto.CustomProvider</param-value>
</context-param>
```

Por padrão o VRaptor tenta procurar o applicationContext via `WebApplicationContextUtils.getWebApplicationCon` ou carregando do applicationContext.xml que está no classpath.

12.5- Mudando o encoding da sua aplicação

Se você quiser que todas as requisições da sua aplicação sejam de um encoding determinado, para evitar problemas de acentuação por exemplo, você pode colocar o seguinte parâmetro no seu web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.encoding</param-name>
    <param-value>UTF-8</param-value>
</context-param>
```

Assim todas as suas páginas e dados passados para formulário usarão o encoding UTF-8, evitando problemas de acentuação.

Spring, Joda Time, Hibernate e Google App Engine

13.1- Integração com Hibernate ou JPA

Existem ComponentFactories implementadas para Session, SessionFactory, EntityManager e EntityManagerFactory. Você pode usá-las ou se basear nelas para criar sua própria ComponentFactory para essas classes.

Além disso existem interceptors implementados que controlam as transações tanto na JPA quanto com o Hibernate.

Para saber como fazer usar esses componentes veja o capítulo de utils.

13.2- Integração com Spring

O VRaptor roda dentro do Spring, e usa o ApplicationContext da sua aplicação como parent do ApplicationContext do VRaptor. Logo todas as funcionalidades e módulos do Spring funcionam com o VRaptor sem nenhuma configuração da parte do VRaptor!

13.3- Conversores para Joda Time

A api de datas do Java é bem ruim, e por esse motivo existe o projeto joda-time (http://joda-time.sourceforge.net/) que tem uma api bem mais agradável para trabalhar com datas. Se o jar do joda-time estiver no classpath, o VRaptor registra automaticamente os conversores para os tipos LocalDate e LocalTime, logo você pode recebê-los como parâmetro sem problemas.

13.4- Rodando o VRaptor3 no Google App Engine

Para rodar no Google App Engine(GAE), você precisa fazer algumas mudanças no VRaptor3 por causa das limitações que o GAE te traz. Existe um blank-project para rodar uma aplicação com VRaptor3 no GAE.

Testando componentes e controllers

Criar um teste unitário do seu controller VRaptor costuma ser muito fácil: dado o desacoplamento das suas classes com a api `javax.servlet` e os parâmetros serem populados através do request, seu teste será como o de uma classe qualquer, sem mistérios.

O VRaptor3 gerencia as dependências da sua classe, então você não precisa se preocupar com a criação do seus componentes e controllers, basta receber suas dependências no construtor que o VRaptor3 vai procurá-las e instanciar sua classe.

Na hora de testar suas classes você pode aproveitar que todas as dependências estão sendo recebidas no construtor, e passar implementações falsas (mocks) dessas dependências, para testar unitariamente sua classe.

Mas os componentes do VRaptor3 que vão ser mais presentes na sua aplicação - o `Result` e o `Validator` - possuem a interface fluente, o que dificulta a criação de implementações falsas (mocks). Por causa disso existem implementações falsas já disponíveis no VRaptor3: `MockResult` e `MockValidator`. Isso facilita em muito os seus testes que seriam mais complexos.

14.1- MockResult

O MockResult ignora os redirecionamentos que você fizer, e acumula os objetos incluídos, para você poder inspeciona-los e fazer as suas asserções.

Um exemplo de uso seria:

```
MockResult result = new MockResult();
ClienteController controller = new ClienteController(..., result);
controller.list(); // vai chamar result.include("clients", algumaCoisa);
List<Client> clients = result.included("clients"); // o cast é automático
Assert.assertNotNull(clients);
// mais asserts
```

Quaisquer chamadas do tipo result.use(...) vão ser ignoradas.

14.2- MockValidator

O MockValidator vai acumular os erros gerados, e quando o validator.onErrorUse for chamado, vai lançar um ValidationError caso haja algum erro. Desse jeito você pode inspecionar os erros adicionados, ou simplesmente ver se deu algum erro:

```
@Test(expected=ValidationException.class)
public void testaQueVaiDarErroDeValidacao() {
    ClienteController controller = new ClienteController(..., new MockValidator());
    controller.adiciona(new Cliente());
```

```
}
```

ou

```
@Test
public void testaQueVaiDarErroDeValidacao() {
    ClienteController controller = new ClienteController(..., new MockValidator());
    try {
        controller.adiciona(new Cliente());
        Assert.fail();
    } catch (ValidationException e) {
        List<Message> errors = e.getErrors();
        //asserts nos erros
    }
}
```

ChangeLog

15.1- 3.0.1 (a ser lançado)

- paranamer atualizado para versão 1.5 (Atualize seu jar!)
- jars separados em opcional e obrigatório no vraptor-core
- dependências estão explicadas no vraptor-core/libs/mandatory/dependencies.txt e no vraptor-core/libs/optional/dependencies.txt
- possibilidade de setar o character encoding da aplicação no web.xml através do context-param br.com.caelum.vraptor.encoding
- nova view: Referer view: result.use(Results.referer()).redirect();

15.2- 3.0.0

- ValidationError foi renomeado para ValidationException
- result.use(Results.http()) para setar headers e status codes do protocolo HTTP
- Correção de bugs
- documentação
- novo site

15.3- 3.0.0-rc-1

- aplicação de exemplo: mydvds
- novo jeito de adicionar os componentes opcionais do VRaptor:

```
public class CustomProvider extends SpringProvider {


    @Override
    protected void registerCustomComponents(ComponentRegistry registry) {
        registry.registry(ComponenteOpcional.class, ComponenteOpcional.class);
    }
}
```

- Utils: HibernateTransactionInterceptor e JPATransactionInterceptor
- Um exemplo completo de aplicação na documentação.
- Docs em inglês

15.4- 3.0.0-beta-5

- Novo jeito de fazer validações:

```
public void visualiza(Cliente cliente) {

    validator.checking(new Validations() {{
        that(cliente.getId() != null, "id", "id.deve.ser.preenchido");
    }});
    validator.onErrorUse(page()).of(ClientesController.class).list();

    //continua o metodo
}
```

- UploadedFile.getFile() agora retorna InputStream.
- EntityManagerCreator e EntityManagerFactoryCreator
- bugfixes

15.5- 3.0.0-beta-4

- Novo result: result.use(page()).of(MeuController.class).minhaLogica() renderiza a view padrão (/WEB-INF/jsp/meu/minhaLogica.jsp) sem executar a minhaLogica.
- Classes Mocks para testes: MockResult e MockValidator, para facilitar testes unitários das lógicas. Eles ignoram a maioria das chamadas e guardam parâmetros incluídos no result e erros de validação.
- As URIs passadas para result.use(page()).forward(uri) e result.use(page()).redirect(uri) não podem ser URIs de lógicas, usem os forwards e redirects do result.use(logic())
- Os parâmetros passados para as URIs agora aceitam pattern-matching:
  - Automático: se temos a URI /clients/{client.id} e client.id é um Long, o parâmetro {client.id} só vai casar com números, ou seja, a URI /clients/42 casa, mas a uri /clients/random não casa. Isso funciona para todos os tipos numéricos, booleanos e enums, o vraptor vai restringir para os valores possíveis.
  - Manual: no CustomRoutes você vai poder fazer: routeFor(“/clients/{client.id}”).withParameter(“client.id”).matching(“\\d{1,4}”) .is(ClienteController.class).mostra(nul ou seja, pode restringir os valores para o determinado parâmetro via expressões regulares no método matching.
- Converters para LocalDate e LocalTime do joda-time já vêm por padrão.
- Quando o Spring é usado como IoC Provider, o VRaptor tenta buscar o spring da aplicação para usar como container pai. A busca é feita por padrão em um dos dois jeitos:
  - WebApplicationContextUtils.getWebApplicationContext(servletContext), para o caso em que você tem os listeners do Spring configurados.
  - applicationContext.xml dentro do classpath

  Se isso não for o suficiente você pode implementar a interface SpringLocator e disponbilizar o ApplicationContext do spring usado pela sua aplicação.

- Utils:
  - SessionCreator e SessionFactoryCreator para disponbilizar a Session e o SessionFactory do hibernate para os componentes registrados.
  - EncodingInterceptor, para mudar o encoding da sua aplicação.
- correção de vários bugs e melhorias na documentação.

15.6- 3.0.0-beta-3

- O Spring é o Provider de IoC padrão
- o applicationContext.xml no classpath é usado como configuração incial do spring, caso exista.
- a documentação http://vraptor.caelum.com.br/documentacao está mais completa e atualizada
- pequenos bugs e otimizações

Migrando do VRaptor2 para o VRaptor3

16.1- web.xml

Para migrar aos poucos, basta colocar no seu web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.caelum.vraptor.vraptor2.Provider</param-value>
</context-param>

<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>br.com.pacote.base.do.seu.projeto</param-value>
</context-param>

<filter>
    <filter-name>vraptor</filter-name>
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
</filter>

<filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Lembre-se de tirar a declaração antiga do VRaptorServlet do vraptor2, e o seu respectivo mapping.

16.2- migrando de @org.vraptor.annotations.Component para @br.com.caelum.vraptor.Resource

O correspondente ao @Component do VRaptor2 é o @Resource do VRaptor3. Portanto, para disponibilizar os métodos de uma classe como lógicas é só anotá-las com @Resource (removendo o @Component).

As convenções usadas são um pouco diferentes:

No VRaptor 2:

```
@Component
public class ClientsLogic {

    public void form() {

    }

}
```

No VRaptor 3:

```
@Resource
public class ClientsController {

   public void form() {

   }
}
```

O método form estará acessível pela uri: "/clients/form", e a view padrão será a `WEB-INF/jsp/clients/form.jsp`. Ou seja, o sufixo `Controller` é removido do nome da classe e não tem mais o sufixo .logic na uri. Não é colocado o resultado "ok" ou "invalid" no nome do jsp.

16.3- @In

O VRaptor3 gerencia as dependências para você, logo o que você usava como @In no vraptor2, basta receber pelo construtor:

No VRaptor 2:

```
@Component
public class ClientsLogic {
    @In
    private ClientDao dao;

    public void form() {

    }

}
```

No VRaptor 3:

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    public ClientsController(ClientDao dao) {
        this.dao = dao;
    }

    public void form() {

    }
}
```

E para que isso funcione você só precisa que o seu ClientDao esteja anotado com o @br.com.caelum.vraptor.ioc.Component do VRaptor3.

16.4- @Out e getters

No VRaptor2 você usava a anotação @Out ou um getter para disponibilizar um objeto para a view. No VRaptor3 basta retornar o objeto, se for um só, ou usar um objeto especial para expôr os objetos para a view. Este objeto é o Result:

No VRaptor 2:

```
@Component
public class ClientsLogic {
    private Collection<Client> list;

    public void list() {
        this.list = dao.list();
    }

    public Collection<Client> getClientList() {
        return this.list;
    }

    @Out
    private Client client;

    public void show(Long id) {
        this.client = dao.load(id);
    }

}
```

No VRaptor 3:

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    private final Result result;

    public ClientsController(ClientDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }

    public Collection<Client> list() {
        return dao.list(); // o nome será clientList
    }

    public void listaDiferente() {
        result.include("clients", dao.list());
    }

    public Client show(Long id) {
        return dao.load(id); // o nome será "client"
    }
}
```

Quando você usa o retorno do método, o vraptor usa o tipo do retorno para determinar qual vai ser o seu nome na view. No caso de uma classe normal, o nome do objeto será o nome da classe com a primeira letra minúscula. No caso de ser uma Collection, o nome será o nome da classe, com a primeira minuscula, seguido

da palavra List.

16.5- views.properties

No VRaptor3 não existe o arquivo views.properties, embora ele seja suportado no modo de compatibilidade com o vraptor2. Todos os redirecionamentos são feitos na própria lógica, usando o Result:

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    private final Result result;

    public ClientsController(ClientDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }

    public Collection<Client> list() {
        return dao.list();
    }

    public void save(Client client) {
        dao.save(client);

        result.use(Results.logic()).redirectTo(ClientsController.class).list();
    }
}
```

Se o redirecionamento for para uma lógica, você pode referenciá-la diretamente, e os parâmetros passados para o método serão usados para chamar a lógica.

Se for para uma jsp direto você pode usar:

```
result.use(Results.page()).forward("/WEB-INF/jsp/clients/save.ok.jsp");
```

16.6- Validação

Você não precisa criar um método validateNomeDaLogica para fazer a validação, basta receber no construtor um objeto do tipo br.com.caelum.vraptor.Validator, e usá-lo para sua validação, especificando qual é a lógica para usar quando a validação dá errado:

```
@Resource
public class ClientsController {

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

    public ClientsController(ClientDao dao, Result result, Validator validator) {
        this.dao = dao;
        this.result = result;
        this.validator = validator;
```

```
    }

    public void form() {

    }

    public void save(Client client) {
        if (client.getName() == null) {
            validator.add(new ValidationMessage("erro","nomeInvalido"));
        }
        validator.onErrorUse(Results.page()).of(ClientsController.class).form();
        dao.save(client);
    }
}
```

16.7- Colocando objetos na sessão

No VRaptor2 bastava colocar um @Out(ScopeType.SESSION) para que o objeto fosse colocado na sessão. Isso não funciona no VRaptor3, pois você perde totalmente o controle sobre as variáveis que estão anotadas desse jeito.

Para colocar objetos na sessão no VRaptor3 você deve fazer uma das duas coisas:

- O objeto vai ser acessível apenas por lógicas e componentes da aplicação, não pelos jsps:

```
@Component

@SessionScoped
public class MeuObjetoNaSessao {
    private MeuObjeto meuObjeto;
    //getter e setter para meuObjeto
}
```

  E nas classes onde você precisa do MeuObjeto basta receber no construtor o MeuObjetoNaSessao e usar o getter e setter pra manipular o MeuObjeto.

- O objeto vai ser acessível nos jsps também:

```
@Component

@SessionScoped
public class MeuObjetoNaSessao {
    private HttpSession session;
    public MeuObjetoNaSessao(HttpSession session) {
        this.session = session;
    }
    public void setMeuObjeto(MeuObjeto objeto) {
        this.session.setAttribute("objeto", objeto);
    }
    public MeuObjeto getMeuObjeto() {
        return this.session.getAttribute("objeto");
    }
}
```

  E nas classes basta receber o MeuObjetoNaSessao no construtor e usar o getter e o setter.