02

Controlando o fluxo e resultado com Result

Fluxo padrão de uma Request no VRaptor

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

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

Modificando o fluxo da requisição com o Result

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

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

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

```
private final Result result;

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

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

```
@Post
public void adiciona(Produto produto) {
    EntityManager em = JPAUtil.criaEntityManager();
    em.getTransaction().begin();
    ProdutoDao dao = new ProdutoDao(em);
    dao.adiciona(produto);
    em.getTransaction().commit();
    result.forwardTo(this).lista();
}
```

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

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

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

Enviar informações para a View com o Result

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

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

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

```
@Post
public void adiciona(Produto produto) {
    EntityManager em = JPAUtil.criaEntityManager();
    em.getTransaction().begin();
    ProdutoDao dao = new ProdutoDao(em);
    dao.adiciona(produto);
    em.getTransaction().commit();
    result.include("mensagem", "Produto adicionado com sucesso!");
    result.forwardTo(this).lista();
}
```

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

```
<c:if test="${not empty mensagem}">
    <div class="alert alert-success">${mensagem}</div>
</c:if>
```

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

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

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

```
@Get
public List<Produto> lista() {
    EntityManager em = JPAUtil.criaEntityManager();
    ProdutoDao dao = new ProdutoDao(em);
    return dao.lista();
}
```

Podemos fazer o mesmo utilizando o método `include()` :

```
@Get
public void lista() {
    EntityManager em = JPAUtil.criaEntityManager();
    ProdutoDao dao = new ProdutoDao(em);
    result.include("produtoList", dao.lista());
}
```

Eu não preciso chamar essa variável de `produtoList` , claro.

Forward x Redirect

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

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

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

```
@Post
public void adiciona(Produto produto) {
    EntityManager em = JPAUtil.criaEntityManager();
    em.getTransaction().begin();
    ProdutoDao dao = new ProdutoDao(em);
    dao.adiciona(produto);
    em.getTransaction().commit();
    result.include("mensagem", "Produto adicionado com sucesso!");
    result.redirectTo(this).lista();
}
```

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

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

Outros formatos de resposta

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

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

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

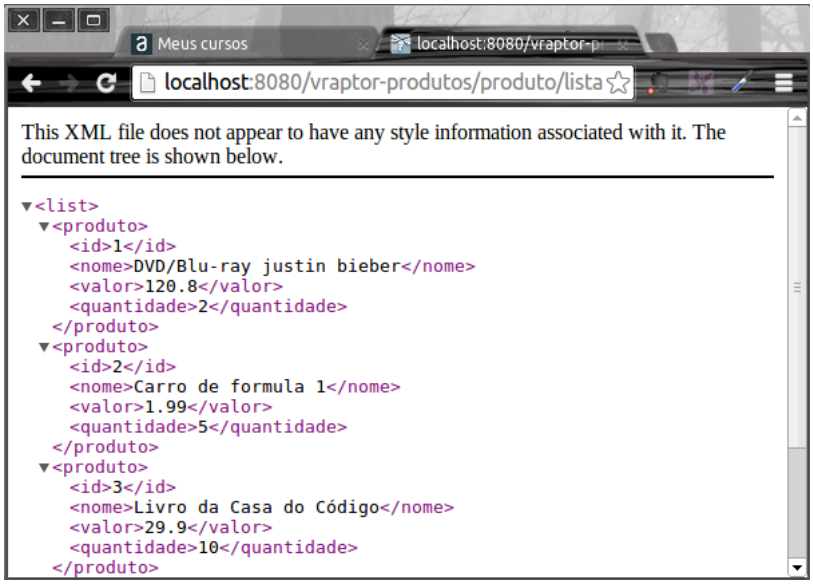
```
result.use(Results.xml()).from(dao.lista());
```

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

```
@Get
public void listaEmXml() {
    EntityManager em = JPAUtil.criaEntityManager();
    ProdutoDao dao = new ProdutoDao(em);
    result.use(Results.xml()).from(dao.lista()).serialize();
}
```

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

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

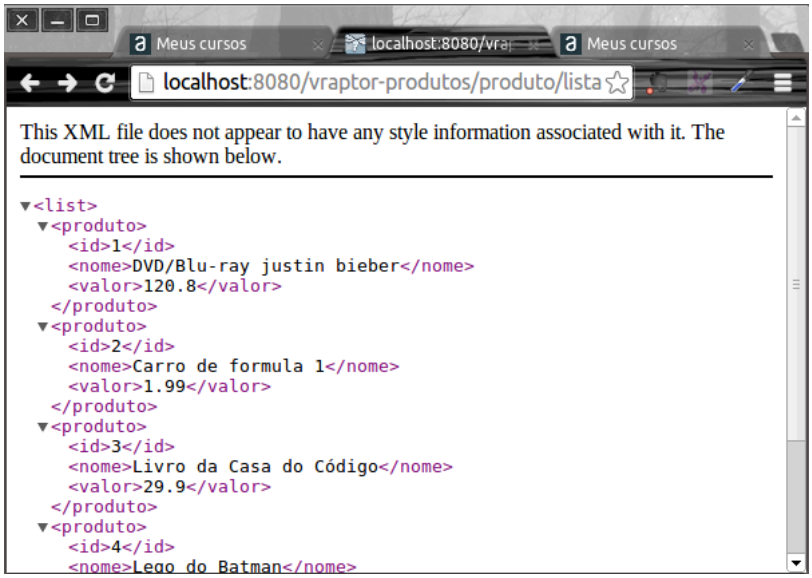


Opcionalmente também podemos adicionar ou excluir informações nesse XML que será gerado. Fazemos isso com os métodos `include` e `exclude`, por exemplo:

```
@Get
public void listaEmXml() {
    EntityManager em = JPAUtil.criaEntityManager();
```

```
        ProdutoDao dao = new ProdutoDao(em);
        result.use(Results.xml()).from(dao.lista())
                .exclude("quantidade").serialize();
    }
```

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



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

```
@Get
public void listaEmJson() {
    EntityManager em = JPAUtil.criaEntityManager();
    ProdutoDao dao = new ProdutoDao(em);
    result.use(Results.json()).from(dao.lista()).serialize();
}
```

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