Módulo 6

Programação WEB

Lição 8

Tópicos avançados no Framework Struts

Versão 1.0 - Nov/2007

Autor
Daniel Villafuerte

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Na lição anterior, trabalhamos com o básico do *framework Struts*. Aprendemos como incluir o *framework Struts* em nossa aplicação configurando o *ActionServlet* para manipular as requisições. Também aprendemos como criar instâncias de classes *Action* que servem como manipuladores de ações para submissões de *forms* e outras requisições de usuário. Vimos como criar classes *ActionForm* que oferecem uma maneira fácil de transferir dados de *forms* para os *ActionHandlers* apropriados. Finalmente, vimos como usar as bibliotecas de *tag* incluídas a fim de auxiliar na junção dos *forms* HTML às páginas JSP dentro do *framework*.

Ao final desta lição, o estudante será capaz de:

- Usar os *DynaActionForms* para minimizar o número de classes que precisamos desenvolver
- Prover validação do lado servidor em nossa aplicação com o *framework Validator*
- Compartimentalizar a camada de apresentação com o *framework Tiles*

2. DynaActionForms

Em grandes aplicações, o número de classes que precisam ser criadas e mantidas pode se tornar extremamente alto. *Struts* suporta classes que contribuem, e muito, para esse elevado número, especialmente com respeito a seus *ActionForms*, os quais requerem uma sólida implementação para unir todos os *forms* na aplicação. Vários desenvolvedores se encontram-se em apuros por essa restrição já que *ActionForms* são, na maioria das vezes, *JavaBeans* com métodos *get* e *set* para cada um dos campos do *form* que ele precisa representar.

Struts traz uma solução no lançamento de sua versão 1.1 chamada *DynaActionForms*. *DynaActionForms* se comportam-se exatamente como *ActionForms*, nos quais uma instância pode ser obtida e seus métodos são chamados pelos manipuladores de ação que precisam de seus dados. A principal diferença é que cada *DynaActionForm* não é definido ou declarado como uma classe separada. Um *DynaActionForm* é simplesmente configurado dentro do arquivo *struts-config.xml*.

Abaixo um exemplo de como configurar e declarar um *DynaActionForms*. Fizemos uso do exemplo da lição anterior, criando aqui um *ActionForm* que irá manipular os dados requisitados para o *login* do usuário.

```
<?xml version="1.0"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.1//EN" "http://jakarta.apache.org/struts/dtds/struts-
config_1_1.dtd">
<struts-config>
 <form-beans>
  <form-bean
   name="loginForm"
   type="org.apache.struts.action.DynaActionForm">
   <form-property name="loginName" type="java.lang.String"/>
   <form-property name="password" type="java.lang.String"/>
  </form-bean>
 </form-beans>
 <action-mappings>
  <action name="loginForm"
   path="/login"
   scope="request"
   type="action.LoginAction">
   <forward name="success" path="/success.jsp"/>
   <forward name="failure" path="/failure.jsp"/>
  </action>
 </action-mappings>
</struts-config>
```

Como podemos ver, criar um *DynaActionForm* é bastante simples. Em alguns casos, declarar um *DynaActionForm* dentro do arquivo de configuração é mais simples e mais fácil do que escrever uma instância da classe *ActionForm*. Não precisamos mais listar todas as propriedades do *form* e criar os métodos *get* e *set* para cada uma delas. Com *DynaActionForms*, simplesmente declaramos o nome da propriedade e o tipo. É responsabilidade do *framework* prover uma instância de trabalho baseada nessa informação.

Fornecer essa informação de configuração é a único pré-requisito para se fazer uso dos *DynaActionForms*. Não há modificações que precisem ser feitas em nenhuma das instâncias *Action*. Eles ainda têm uma instância *ActionForm* válida que podem retornar dados.

A seguir estão os tipos de dados Java suportados por *DynaActionForm*:

- java.lang.BigDecimal
- java.lang.BigInteger
- boolean e java.lang.Boolean
- char e java.lang.Character
- double e java.lang.Double
- float e java.lang.Float

- int e java.lang.Integer
- long e java.lang.Long
- short e java.lang.Short
- java.lang.String
- java.lang.Date
- java.lang.Time
- java.sql.TimeStamp

Usar *DynaActionForms* pode ser mais conveniente, mas não são sempre a melhor solução. Ainda há casos onde usar *ActionForms* é mais apropriado.

- *DynaActionForms* suportam apenas um conjunto limitado de tipos Java. Se nosso *form* precisa armazenar informação expressa como um tipo de dado diferente dos suportados, então os *ActionForms* ainda são o melhor caminho. Um exemplo disso seria se usássemos objetos Java desenvolvidos como propriedades do *form*.
- *DynaActionForms* não suportam o conceito de herança. Não há como criar uma definição base para um *form* que possa ser estendida posteriormente.

3. Validadores

Validação é uma atividade que deve ser executada em todos os casos de entrada de dados. Através da validação, podemos checar o formato e o conteúdo dos valores informados pelo usuário. Usuários, apesar de tudo, nem sempre informam a entrada correta: letras podem ser inseridas em um campo numérico e vice-versa; em um campo que requer 3 dígitos são inseridos apenas 2, e assim por diante. É responsabilidade da aplicação estar ciente disso e manipular tais erros de entrada a despeito além de qualquer erro resultante de processamento da lógica de negócio (por exemplo, a senha não está correta para determinado login).

Struts alivia a carga do desenvolvedor em realizar essa validação fornecendo um *famework* de validação chamado de *Validator Framework*. O uso desse *framework* trás alguns benefícios:

- **Várias regras de validação pré-definidas.** Há um conjunto comum de verificações que devem ser executadas em qualquer número de aplicações como a verificação que pode ser de formato, de tamanho, de existência entre outras. O *framework* fornece os componentes necessários para que os desenvolvedores não precisem criar código que irão manipular esses tipos comuns de validação. Geralmente, os componentes que o *framework* fornece são suficientes para a maioria das aplicações, embora validadores customizados também possam ser criados.
- **Elimina redundância no código de validação.** O *framework* separa os componentes que executam a validação dos *que necessitam* de validação. Ao invés de ter múltiplos componentes incorporando o mesmo código de validação, a funcionalidade pode ser mantida em um componente de validação externo e separado que pode ser reutilizado por toda a aplicação.
- **Um único ponto de manutenção.** Desenvolvedores não precisam mais percorrer toda a aplicação para checar as regras de validação que eles utilizam em seus vários componentes. Todas essas regras são declaradas em arquivos de configuração fornecidos pelo *framework*.

Há alguns passos necessários para incluir a funcionalidade do *Validator* em uma aplicação *Struts*:

- Configurar o *Plug-in* do *Validator*.
- Declarar os *forms* que requerem validação e o tipo de validação que eles necessitam.
- Criar as mensagens que serão exibidas no caso de falha na validação.
- Modificar o arquivo struts-config para permitir validação automática.

3.1. Configurando o Plug-In do Validator

Esse passo é necessário para tornar o *framework Struts* ciente do uso do *framework Validator*. Tudo que precisa ser feito é adicionar algumas poucas linhas ao nosso arquivo*struts-config.xml*. Abaixo está uma amostra:

```
...
<forward name="success" path="/success.jsp"/>
<forward name="failure" path="/failure.jsp"/>
</action>
</action-mappings>

<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
 <set-property
  property="pathnames"
  value="/WEB-INF/validator-rules.xml, /WEB-INF/validation.xml"/>
</plug-in>

</struts-config>
```

A propriedade *pathnames* informa ao *framework* onde ele pode achar os arquivos de configuração de que precisa. Há dois arquivos de configuração: *validator-rules.xml* e *validation.xml*.

3.2. validator-rules.xml

Esse arquivo de configuração define as classes que implementam o código de validação. O *framework* vem com um exemplo desse arquivo com classes pré-definidas de validação já configuradas.

A seguir, uma lista dos nomes lógicos dos validadores que vêm com o *framework*:

- **required** – propriedades marcadas como *required* devem ter ao menos um caractere.
- **mask** – propriedades sujeitas ao validador de máscara devem combinar com a expressão regular que submetemos usando o parâmetro ***mask***.
- **minlength** – se aplicado a uma propriedade, deve ter um tamanho igual ou maior que o valor do parâmetro ***min*** que passamos.
- **maxlength -** se aplicado a uma propriedade, deve ter um tamanho igual ou menor que o valor do parâmetro ***max*** que passamos.
- **range –** a propriedade deve estar entre os valores fornecidos através dos parâmetros ***min*** e ***max***.
- **byte, short, integer, long, float, double** – a propriedade deve poder ser convertida para o tipo de dado primitivo específico.
- **date –** valida se o valor da propriedade é uma data válida.
- **creditCard –** valida se o valor da propriedade pode ser um número de cartão de crédito válido.
- **email** – valida se o valor da propriedade pode ser um endereço de *email* válido.

Esses validadores pré-definidos são suficientes para a maioria dos propósitos de validação. Nos casos em que os requisitos de validação da aplicação não podem ser manipulados por esses validadores, implementações customizadas podem ser criadas, e suas definições adicionadas ao arquivo *validator-rules.xml*.

3.3. validation.xml

Esse arquivo de configuração declara ao *framework* quais *forms* requerem validação e quais regras de validação se deseja implementar. O *framework* apenas fornece a estrutura do arquivo. Desenvolvedores terão que configurar esse arquivo para tirar proveito da funcionalidade do *framework*.

3.3.1. Configurando o arquivo validation.xml

O arquivo *validation.xml* fornece uma estrutura que podemos usar para especificar nossa configuração de validação. A seguir são mostrados os elementos XML que o define:

<formset>

O elemento raiz desse arquivo de configuração.

<form>

Cada formulário da aplicação, que requer validação, deve ter pelo menos um elemento <*form*> correspondente. O atributo *name* pode mapear para um formulário a ser configurado no *struts-config.xml*. Esse elemento pode conter um ou mais elementos <*field*>.

<field>

Cada elemento *field* representa uma propriedade no *form* que requer validação. Esse elemento tem os seguintes atributos:

- **property** – o nome da propriedade no *form* que será validada.
- **depends** – a lista, separada por vírgulas, dos validadores que serão aplicados à propriedade. Os nomes dos validadores são definidos dentro do arquivo *validator-rules.xml.*

<arg0>

Define a chave para a mensagem de erro que será exibida no caso da propriedade não passar nas

regras de validação. A chave e a mensagem de erro que será mostrada devem ser encontradas em um pacote de recursos que o desenvolvedor deve criar.

<var>

Alguns validadores requerem que certos valores sejam passados para eles como argumentos para que possam funcionar corretamente. Esses argumentos podem ser fornecidos usando um ou mais elementos *var*, no qual cada elemento *var* corresponde a um argumento passado ao validador. Esse elemento define dois elementos filho:

- **<var-name>** - define o nome do argumento a ser fornecido.
- **<var-value>** - define o valor do argumento.

Um exemplo de tal arquivo de configuração é fornecido abaixo.

```
<form-validation>
  <formset>
    <form name="loginForm">
      <field property="loginName" depends="required">
        <arg0 key="error.loginname.required"/>
      </field>
      <field property="password" depends="required,minlength">
        <arg0 key="error.password.required"/>
        <var>
          <var-name>minlength</var-name>
          <var-value>4</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

O trecho do arquivo fornecido configura o *loginForm* usado em nossos exemplos anteriores. *loginForm* corresponde diretamente a um *ActionForm* que é definido dentro do arquivo de configuração do S*truts* sob o nome *loginForm*. Ele é configurado utilizando o *loginForm* com o atributo *name* do elemento <*form*>.

Depois do elemento <form>, descobrimos que há dois elementos <field> definidos. O primeiro elemento <field> configura a propriedade *loginName* do nosso *form*, o segundo configura a propriedade *password*. Podemos determinar quem é quem olhando o valor do atributo *property*.

3.4. Definindo o pacote de recursos

O elemento <arg0> define uma chave que necessita combinar com uma entrada em um pacote de recursos. Essa entrada é, então, usada para determinar a mensagem que será exibida ao usuário. O *framework Validator* faz uso do mesmo pacote de recursos que o framework *Struts*, o qual, por *default*, pode ser encontrado no diretório WEB-INF/classes sob o nome *ApplicationResources.properties*. Se tal pacote de recursos não existe em sua aplicação, crie um arquivo de texto com esse nome.

Entradas no pacote de recursos são simplesmente pares chave=valor. Para o exemplo de configuração abaixo, podemos ter o seguinte conteúdo:

```
error.loginname.required=Por favor informe seu login
error.password.required=Informada senha em branco ou com menos de 4 caracteres
```

O último passo é permitir ao *framework* de validação modificar nossas classes *ActionForm* e *DynaActionForm* para fazer uso das classes fornecidas.

Final da **struts-config.xml**:

```
  <form-beans>
    <form-bean
      name="loginForm"
      type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="loginName" type="java.lang.String"/>
```

```
      <form-property name="password" type="java.lang.String"/>
    </form-bean>
  </form-beans>

  <action-mappings>
    <action name="loginForm"
            path="/login"
            scope="request"
            type="action.LoginAction"
            validate="true"
            input="/loginStruts.jsp">
      <forward name="success" path="/success.jsp"/>
      <forward name="failure" path="/failure.jsp"/>
    </action>
    <action path="/Login" forward="/loginStruts.jsp"/>
  </action-mappings>

  <message-resources parameter="ApplicationResource"/>

  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
      property="pathnames"
      value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
  </plug-in>
```

Final da **LoginAction.java**:

```
package action;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class LoginAction extends Action {

  @Override
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    DynaActionForm dForm = (DynaActionForm) form;
    String loginName = (String) dForm.get("loginName");
    String password = (String) dForm.get("password");

    if (!(loginName.equals("JEDI") && password.equals("JEDI"))) {
      return mapping.findForward("failure");
    }
    HttpSession session = request.getSession();
    session.setAttribute("USER", loginName);
    return mapping.findForward("success");
  }
}
```

Final da **LoginStruts.jsp**:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>

<html:html>
  <head>
    <title>Login Page</title>
  </head>
  <body>
    <h1> Login Page </h1>
    <br/>
    <html:form action="/login">
      User Name:
```

```
        <html:text property="loginName"/>
        <html:errors property="loginName"/>
        <br/>
        Password:
        <html:password property="password"/>
        <html:errors property="password"/>
        <br/>
        <html:submit/>
      </html:form>
    </body>
  </html:html>
```

4. Tiles

Outro *framework* que trabalha especialmente bem com *Struts* é o *framework Tiles*. Usando *Tiles*, podemos facilmente definir *templates* e instâncias de telas os que podemos usar em nossa aplicação.

4.1. O que são templates?

De forma simples, uma página de *template* é um modelo projeto de desenho de página que pode ser reusado por qualquer outra página em sua aplicação. Fazer uso de *templates* fornece à sua aplicação um *look and feel* mais consistente.

Para melhor entender o conceito de *templates*, vamos dar uma olhada em algumas páginas de uma aplicação WEB:

Figura 1: Página exemplo de uma aplicação WEB

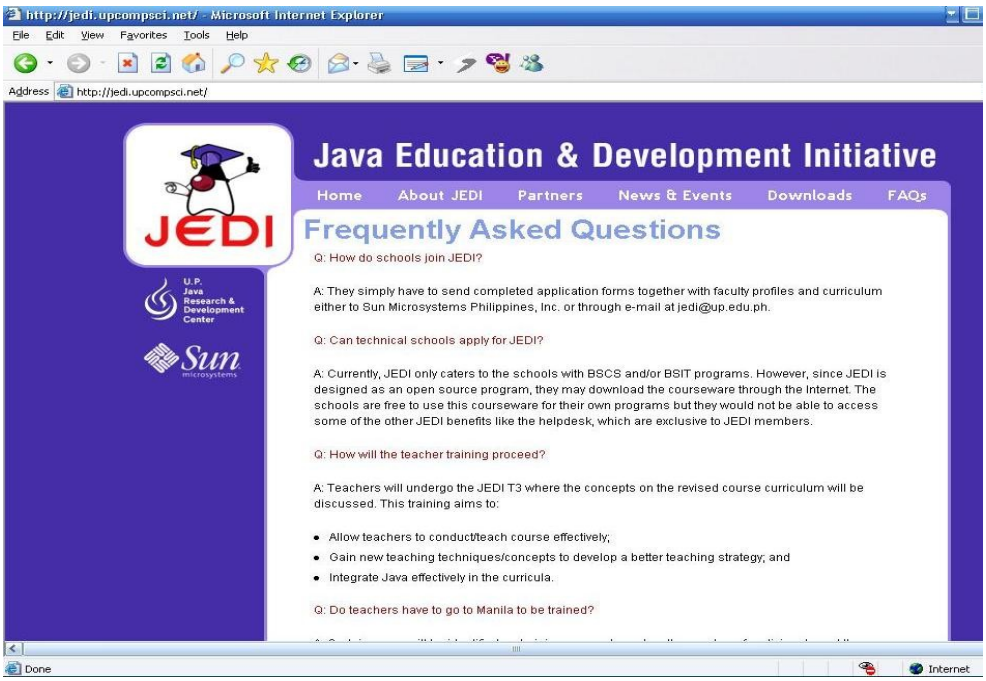

Figura 2: Página exemplo de uma aplicação WEB

Navegando nas páginas, podemos notar que há vários elementos de projeto comuns a todas elas. Todas compartilham o mesmo conjunto de *links* de navegação à esquerda, o mesmo conjunto de imagens na parte superior da página e o mesmo conjunto de imagens nos *links* na parte inferior da página. O que está diferente entra as páginas é apenas o conteúdo apresentado no meio desta.

Imagine como faríamos para implementar um conjunto de páginas similares. Se fôssemos implementar cada página tal qual como as vemos, isto é, exatamente uma página correspondendo para cada uma das telas que vemos, teríamos problemas com a sua manutenção no futuro. Isto ocorreria, pois ocorre uma grande quantidade de código duplicado – se mais tarde alguém decidisse que as imagens de cima não estão muito boas ou que o menu de navegação à esquerda não está suficientemente intuitivo, as mudanças teriam que ser reproduzidas manualmente por todas as páginas.

Como desenvolvedores que somos, sabemos o suficiente para distribuir em entidades separadas código que poderia ser aproveitado através da aplicação. Este conceito também pode ser aplicado nas páginas JSP/HTML: os *links* de navegação podem ser implementados em uma página, as imagens do cabeçalho em outra, assim como o rodapé. Estas páginas podem, então, ser adicionadas à página que implementaria o conteúdo do corpo do texto. Isto pode ser realizado sem o uso de qualquer *framework*, usando a ação *<jsp:include>* que discutimos na leitura básica de JSP.

Uma solução melhor seria ter o conteúdo do texto em um fragmento JSP separado, criar uma página JSP que defina a formatação com o *layout* padrão das páginas e simplesmente deixar espaços reservados para outros elementos que venham a ser incluídos na página. Um desenvolvedor que esteja implementando uma tela teria apenas que fazer uso desta página "modelo" e definiria uma página de conteúdo que seria inserida no devido espaço reservado.

Os benefícios desta solução são óbvios: quaisquer mudanças que teriam que ser feitas em um aspecto da camada de apresentação, digamos, a página de cabeçalho, seria aplicado em todas as páginas ao modificarmos uma única página. Se quiser alterar o *layout* do seu sítio, enquanto ainda mantém o seu conteúdo, isto poderia ser feito simplesmente alterando a página de modelo.

Aprenderemos como fazer esta modelagem usando o *framework Tiles*.

4.2. Preparando Tiles

Antes que possamos nos beneficiar do que o *framework Tiles* pode oferecer, precisamos realizar alguns passos necessários.

1. Adicionar as seguintes linhas ao *struts-config.xml* imediatamente antes do elemento de fechamento *</struts-config>*:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
  <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
  <set-property property="moduleAware" value="true" />
</plug-in>
```

Isto informa ao *Struts* que queremos fazer uso do *Tiles* e o arquivo de configuração que será lido pode ser achado no diretório /WEB-INF, com o nome *tiles-defs.xml*.

2. Copiar os arquivos *struts-tiles.tld* e *tiles-config.dtd* do diretório *lib* do *Struts* para o diretório /WEB-INF da nossa aplicação. O primeiro arquivo contém informações das *tags* personalizadas que precisaremos usar para o *Tiles*, e o segundo define a estrutura do arquivo de configuração *tiles-defs.xml*.
3. Criar um arquivo de configuração em branco que será preenchido mais tarde.
4. Criar um arquivo padrão *xml* com o nome *tiles-defs.xml* e salvá-lo no diretório WEB-INF. Com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE tiles-definitions PUBLIC
```

```
"-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">

<tiles-definitions>

</tiles-definitions>
```

Após a realização desses passos, o *Tiles* está pronto para ser usado.

4.3. Criando um modelo de Layout

O primeiro passo na construção de um modelo é a identificação dos componentes que serão colocados nele. Por exemplo, a maioria das páginas WEB têm um cabeçalho, rodapé, barra de menu e corpo.

Se fôssemos desenhar um esboço rápido, tal *layout* se pareceria com o do diagrama a seguir.

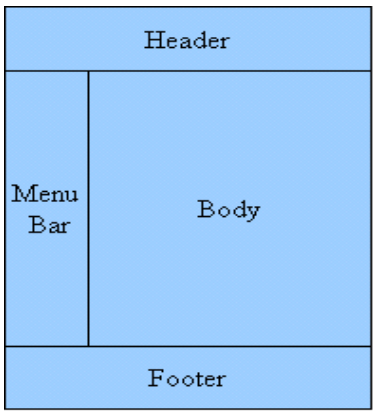

Figura 3: Layout básico

Para criar uma página de modelo implementando este *layout*, siga estes passos:

1. Crie uma nova página JSP (/layout/basicLayout.jsp).
2. Importe a biblioteca de *tags* do Tiles.
3. Crie o HTML implementando o *layout*, deixando em branco a implementação dos componentes reais.
4. Use a *tag* <tiles:insert> para agir como espaços reservados para os componentes do *layout*.

O HTML que implementa o *layout* pode ser simples como uma tabela, com os componentes modelados como células da tabela conforme mostrado no JSP abaixo:

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
<HTML>
 <HEAD>
  <TITLE><tiles:getAsString name="title"/></TITLE>
 </HEAD>
 <BODY>
  <TABLE border="0" width="100%" cellspacing="5">
   <TR>
    <TD colspan="2"><tiles:insert attribute="header"/></TD>
   </TR><TR>
    <TD width="140" valign="top"><tiles:insert attribute="menu"/></TD>
    <TD valign="top" align="left"><tiles:insert attribute="body"/></TD>
   </TR><TR>
    <TD colspan="2"><tiles:insert attribute="footer" /></TD>
   </TR>
  </TABLE>
 </BODY>
</HTML>
```

Há duas *tags* personalizadas no exemplo acima: <*tiles:insert*> e <*tiles:getAsString*>.

- ***<tiles:insert>*** - tem vários usos no *Tiles*. Neste caso, inserir um fragmento JSP que se refere ao nome fornecido no atributo ***attribute***. O fragmento corresponde ao nome será especificado na definição de tela que fará uso deste modelo.
- **<tiles:getAsString>** - esta *tag* recupera, como tipo *String*, um valor fornecido em uma definição de tela usando o nome no atributo *name*.

4.4. Criando definições de tela

Assim que tivermos um modelo, podemos fazer uso deste para definirmos completamente uma tela. A criação de definições de tela pode ser feito de duas maneiras no *framework* Tiles: as definições podem ser definidas dentro de páginas JSP, ou dentro de um arquivo XML reconhecido pelo *framework*.

4.4.1. Criando uma definição usando páginas JSP

A criação de uma definição dentro de uma página JSP faz uso de algumas *tags* personalizadas fornecidas pelo Tiles:

- ***<tiles:definition>*** - *tag* base usada para definir uma tela. O valor do atributo ***id*** é o nome pelo qual esta definição pode ser referenciada pelos outros componentes. O valor do atributo ***page*** é a localização do arquivo de modelo que será usado como base.
- **<tiles:put>** - esta *tag* é usada para associar um valor a um atributo específico dentro de um modelo. O atributo *name* especifica o nome da localização de destino dentro do modelo. O atributo *value* define o texto ou a localização do fragmento JSP a ser usado.

Considere o exemplo abaixo:

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
<tiles:definition id="welcomePage" page="/layout/basicLayout.jsp">
 <tiles:put name="title" value="Welcome!"/>
 <tiles:put name="header" value="/header.jsp"/>
 <tiles:put name="footer" value="/footer.jsp"/>
 <tiles:put name="menu" value="/menu.jsp"/>
 <tiles:put name="body" value="/welcome.jsp"/>
</tiles:definition>
```

Neste exemplo, criamos uma definição de tela para uma página de boas vindas. Ela faz uso do *layout* que definimos previamente e coloca título, cabeçalho, rodapé, menu e componentes de texto especificados em locais definidos no modelo.

Por padrão, esta definição é visível apenas dentro da página JSP contendo a declaração. Para mudar isto, podemos fornecer um valor ao atributo opcional *scope* da *tag* <*tiles:definition*>. Os valores possíveis são: *page, request, session e application*.

4.4.2. Criando uma definição usando um arquivo de configuração XML

A segunda maneira de se criar uma definição de tela é colocando uma entrada de configuração no arquivo de configuração *tiles-defs.xml* que criamos inicialmente. A sintaxe para se criar esta entrada é muito parecida com a <*tiles:definition*> usada previamente:

```
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">

<tiles-definitions>
 <definition name="welcomePage" page="/layout/basicLayout.jsp">
  <put name="title" value="Welcome!"/>
  <put name="header" value="/header.jsp"/>
  <put name="footer" value="/footer.jsp"/>
  <put name="menu" value="/menu.jsp"/>
  <put name="body" value="/welcome.jsp"/>
 </definition>
```

```
 <!-- ... mais definições ... -->
</tiles-definitions>
```

Qual é a diferença entre os dois métodos? Ambos os métodos carregam a definição como um *JavaBean* na memória. Entretanto, o método JSP apenas carrega a definição depois que o fragmento JSP que a contém tenha sido acessado e, por padrão, torna-o visível apenas na mesma página. Esta condição torna a reutilização da definição de tela pela aplicação um pouco mais problemática, já que deve-se tomar um cuidado extra para evitar a carga e a descarga da definição de, e para a memória, desnecessariamente.

O método XML, por outro lado, faz com que a definição de tela esteja disponível para toda a aplicação imediatamente após a inicialização. O *Tile* cuida da persistência em memória. A única coisa que os componentes precisam para fazer uso da definição é que seja fornecido o seu nome.

Outro benefício da utilização do método XML para a criação de definição de tela é que as próprias definições podem ser usadas como *ActionForwards* pelo *framework Struts*. Isto pode reduzir drasticamente o número de páginas JSP necessárias à sua aplicação.

4.4.3. Usando as definições de tela

A criação da definição de tela não é suficiente para que ela seja exibida ao usuário. Para colocar uma definição em uso, podemos usar a *tag* <*tiles:insert*> e fornecer o nome da definição que deve ser mostrada:

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
<tiles:insert beanName="welcomePage" flush="true"/>
```

A página acima é tudo o que é necessário para exibir a tela de boas vindas ao usuário.

Um dos problemas desta abordagem é que aumenta o número de páginas JSP necessárias para mostrar telas diferentes ao usuário: com exceção do componente com o corpo do texto, cada tela requereria uma página separada que faria uso da definição de tela. Para aplicações que requerem 100 ou mais telas, o número de páginas que teriam que ser criadas é o dobro!

Uma melhor abordagem é fazer uso das definições como os alvos do *ActionForwards*. Incluindo o *Tiles* como um *plug-in* para o *Struts* (um dos passos preparatórios que realizamos). *Struts* torna ciente das definições de tela criadas usando o *Tiles*. O nome das telas pode ser usado no lugar de localizações reais nas *tags* <*forward*>. Considere o exemplo abaixo:

```
<action-mappings>
 <action name="loginForm"
  path="/login"
  scope="request"
  type="login.LoginAction"
  validation="true">
  <forward name="success" path="/welcomePage"/>
  <forward name="failure" path="/failure.jsp"/>
 </action>
</action-mappings>
```

Aqui modificamos o nosso exemplo anterior do *struts-config.xml* tal que a condição lógica de sucesso é mapeada para a nossa definição de *welcomePage*. Como podemos ver, esta abordagem é simples, fácil de usar, e diminui o número de páginas JSP que necessitariam ser criadas.

4.4.4. Estendendo definições

Uma das vantagens do *Tiles* sobre outros métodos de modelagem é permitir a extensão das definições de tela. A extensão de tela funciona de maneira parecida com a herança de classes em Java: a tela estendida herda todas as propriedades e atributos da definição pai. Isto nos permite criar uma definição base de tela que declara valores padrões para atributos que podem ser estendidas para definições especializadas em páginas específicas.

Tome o seguinte exemplo:

```
<definition name="basicDefinition" page="/layout/basicLayout.jsp">
```

```
<put name="header" value="/header.jsp"/>
<put name="footer" value="/footer.jsp"/>
<put name="menu" value="/menu.jsp"/>
</definition>

<definition name="welcomePage" extends="basicDefinition">
<put name="title" value="Welcome!"/>
<put name="body" value="/welcome.jsp"/>
</definition>

<definition name="otherPage" extends="basicDefinition">
<put name="title" value="My title"/>
<put name="body" value="/otherContent.jsp"/>
</definition>
```

Criando-se uma tela base que pode ser estendida, evitamos a repetição da definição de valores de atributos. Além disso, quando uma mudança precisa ser feita com relação a qual componente é colocado em um dos atributos base, esta mudança é, imediatamente e automaticamente, propagada para todas as telas simplesmente fazendo a alteração na definição da tela base.

5. Exercícios

O exercício deste capítulo será construído com base no exercício apresentado na lição anterior.

1. Usar o *framework Validator* para incluir um código de validação para todos os formulários.

2. Converter todas as *ActionForms* usadas para o equivalente em *DynaActionForm*.

3. Implementar as telas definidas usando o *framework Tiles*. O *layout* a ser usado é o *layout* básico definido nesta lição. Esta atividade consiste de várias tarefas:

   a) Separar o conteúdo criado previamente para cada página em páginas "corpo" distintas.

   b) Criar páginas de barras laterais contendo uma lista de *links* aplicáveis para cada subseção da aplicação. Por exemplo, se o usuário é um administrador e está na Página de Conteúdo Principal, deve ser capaz de ver *links* para a página de Gerenciamento de *download* de material, Gerenciamento de Usuário e Atividades do Usuário, assim como um *link* que permita ao usuário sair da sua seção da aplicação.

De modo geral, as barras laterais devem ser implementadas tal que:

- Apresentar um *link* para a página pai conforme definido na definição de fluxo de tela no exercício anterior.
- Apresentar *links* para o próximo conjunto de páginas que podem ser acessadas de acordo com o mesmo fluxo de telas.
- Apresentar um *link* que permita ao usuário sair da sua seção da aplicação.

   c) Criar definições *Tiles* para cada uma das telas da aplicação e usá-las no lugar de *ActionForwards*.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.