# Módulo 6

Programação WEB


# Lição 9

MVC Frameworks II – JavaServer Faces

***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, vimos o *framework Struts*, um sistema *open-source* (código aberto) para aplicações WEB que implementa a estrutura MVC-2. Nesta lição iremos analisar outro *framework* MVC: JavaServer Faces (JSF)

Ao final desta lição, o estudante será capaz de:

- Entender a estrutura MVC para a JSF
- Visão de outros componentes de *tags* JSF

# 2. Introdução ao Framework JSF

JSF é um *framework* para a construção de interfaces de usuário para aplicações da WEB. É construído sobre o conceito introduzido pelo *Struts* e traz consigo os benefícios de uma estrutura que separa claramente a lógica de negócio em um padrão baseado em componentes de interface do usuário parecido em muitos maneiras com a *Swing*.

Abaixo está a figura detalhando como o sistema funciona:

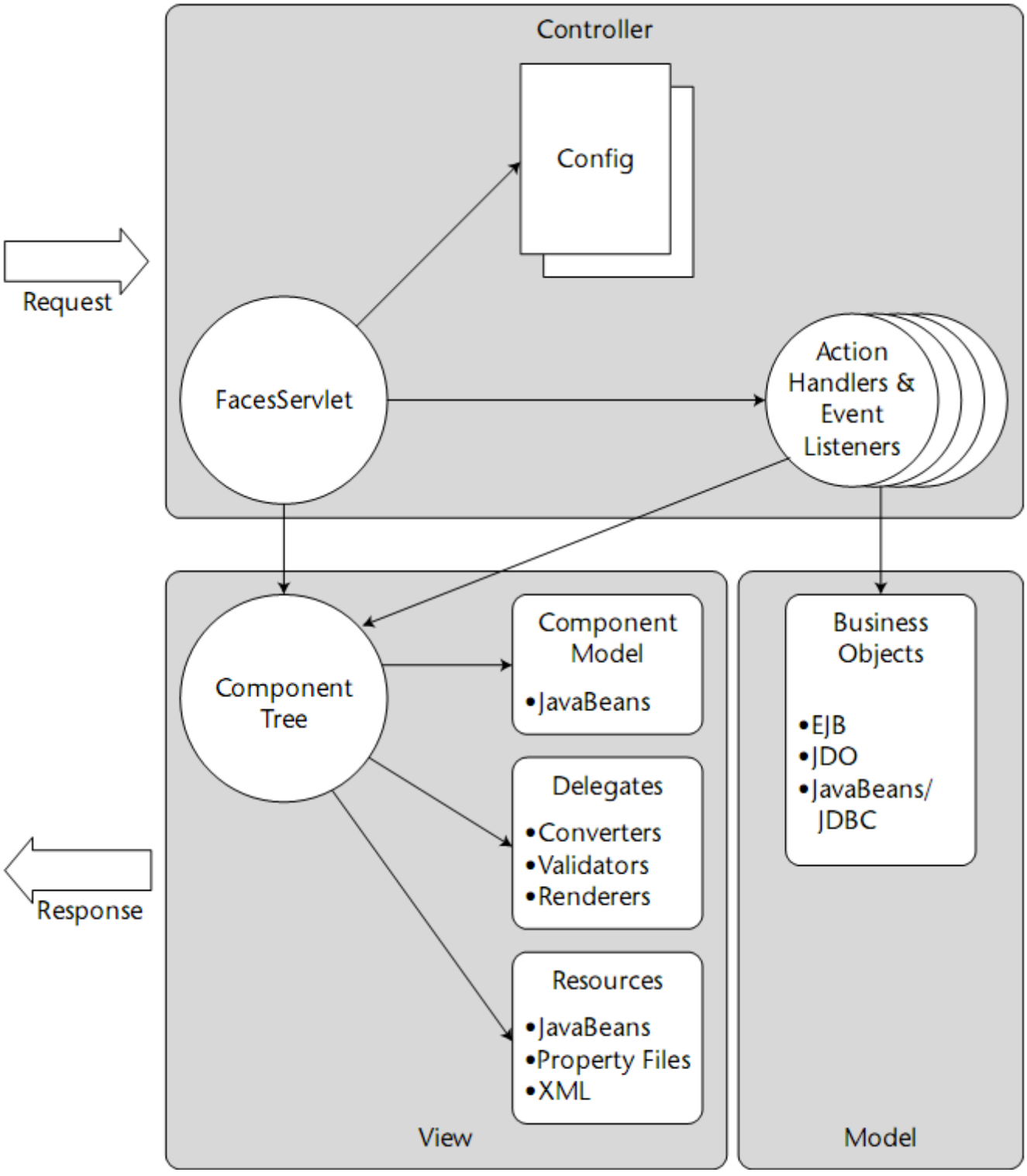


*Figura 1: Framework JavaServerFaces (imagem do livro Mastering JavaServerFaces, Wiley Publishing)*

Como se pode ver, JSF também tem uma clara separação entre os componentes para a camada *Model*, *View* e *Controller.* Como *Struts*, JSF tem uma *servlet front controller*, chamado *FacesServlet*, responsável por receber solicitações de clientes e então executar as ações apropriadas necessárias que são ditadas pelo sistema. Outra semelhança entre *Struts* e JSF é que ambos fazem uso das *Actions Handlers* separada do *servlet front controller* embora JSF reconheça essa pequena diferença quando comparada ao *Struts*.

JSF e *Struts* agem similarmente em relação à camada *View*. *Struts* fornece apenas um conjunto de biblioteca de *tags* que agregam funcionalidades HTML padrão. JSF, por outro lado, fornece seu próprio conjunto de componentes e *interfaces*, juntamente com uma biblioteca de *tags* para exibir estes componentes como *tags* e um componente de interpretação que traduz componentes gráficos em HTML.

Vamos examinar os diferentes aspectos de JSF.

## *2.1. Controller*

A camada *Controller* de JSF é feita por uma *Servlet Controller* denominada *FacesServlet*, um conjunto de arquivos de configuração XML e um conjunto de *Actions Handlers*.

### 2.1.1. FacesServlet

É responsável por aceitar solicitações de clientes e executar operações necessárias para produzir a resposta. Estas operações incluem preparar os componentes gráficos requeridos pela solicitação, atualizando o estado dos componentes, passar as *Actions Handlers* requeridas e traduzir os componentes gráficos que são parte da resposta.

Fornecido pelo sistema, JSF requer apenas configurações em um descritor das aplicações, antes que esteja pronto para ser utilizado.

A listagem a seguir, mostra um fragmento de como configurar o *FacesServlets* para a aplicação.

```
...
<servlet>
<servlet-name>FacesServlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
<servlet-name>FacesServlet</servlet-name>
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
...
```

### 2.1.2. Action Handlers

Foi mencionado anteriormente, que JSF faz uso de *Actions Handlers* independente da camada *controller* (*front controller servlet*), semelhante ao *Struts*. JSF executa essa função de modo diferente.

Em JSF, existem dois modos possíveis para se criar uma *Action Handler*. O primeiro vincula um método Java para servir como *Action Handler*. E o segundo, cria uma instância implementando a interface *ActionListener*.

### 2.1.3. Método Application

Método que é vinculado a um componente gráfico para servir como *Action Handler*, é chamado um método *Application*. Mais tarde, na camada *View*, veremos como esta ligação é realizada.

Existem algumas regras que devem ser observadas para se criar um método *Application*:

- O método deve ser declarado como público
- O método não deve ter parâmetros

- O tipo de retorno do método deve ser um objeto do tipo *String*

A seguir, temos um método que podemos usar para reconhecer um evento resultante na tentativa de identificação de um usuário.

```
...
public String performLogin() {
  // Se o nome não foi informado
  if (loginName == null || loginName.length() < 1) {
    return "failure";
  }

  // Criar um objeto de negócio para verificar a autorização
  User user = null;
  UserService service = new UserService();
  user = service.login(loginName, password);

  // Caso o usuário seja null retorna não autorizado
  if (user == null) {
    return "failure";
  }
  // Retorna uma instância de FacesContext
  FacesContext ctx = FacesContext.getCurrentInstance();

  // Resultado de uma seção para usar outros componentes
  Map sessionMap = ctx.getExternalContext().getSessionMap();
  sesionMap.put(ApplicationConstants.USER_OBJECT, user);

  // Login completo com sucesso
  return "success";
}
...
```

Uma das vantagens neste tipo de abordagem, é a diminuição do número de objetos que os desenvolvedores necessitam manter.

Este método pode estar em qualquer *JavaBean* reconhecido pelo sistema, embora possa ser encontrado no programa e ser utilizado como um *backing model* para a página. Mais a frente, iremos abordar sobre *backing model*.

O objeto do tipo *String* retornado pelo método informa ao *FacesServlet* o que será mostrado pelo usuário. Objetos *Strings* são nomes lógicos e algumas vezes chamados *outcomes*. Estes *outcomes* são verificados através das regras de navegação definidas no arquivo de configuração.

### 2.1.4. Trabalhando com o escopo *Session* na JSF

JSF tem um método diferente para acessar um contexto de sessão. Em exemplos anteriores vimos que devemos restaurar o objeto instanciado da classe *HttpSession* para manipular um contexto de sessão. Por exemplo, em *Struts*, registramos um objeto *HttpServletRequest* passado como parâmetro para executar um determinado método.

Visto que os métodos *Application* são definidos de modo que não levam em conta nenhum parâmetro, não existe nenhum arquivo válido com um objeto *HttpServletRequest* para restaurar um objeto *HttpSession*. JSF contorna esta limitação fornecendo o acesso ao contexto da sessão (ou outros contextos) com o uso de um objeto instanciado da classe *FacesContext*.

Em demonstrações anteriores salvamos um objeto no contexto de sessão. Em JSF utilizamos as seguintes instruções reproduzidas a seguir.

```
FacesContext ctx = FacesContext.getCurrentInstance();
...
Map sessionMap = ctx.getExternalContext().getSessionMap();
sessionMap.put(ApplicationConstants.USER_OBJECT, user);
```

Como podemos observar neste exemplo, obter um arquivo válido de um objeto *FacesContext* é simples. Chamamos um método estático da classe *FacesContext*. Um quadro representativo de

objetos posicionados em um contexto de sessão pode então ser retomado pelo usuário.

### 2.1.5. ActionListener

O outro modo de executar um *action handler* na JSF é criar uma implementação da interface *ActionListener*. Esta interface define um único método:

```
public void processAction(ActionEvent event)
```

A *ActionEvent* passada como um parâmetro no método fornece uma implementação de acesso para o componente que dispara o evento. É semelhante ao modo como os eventos funcionam em programação *Swing*.

Abaixo está um exemplo de implementação da *Action Listener* utilizado para as ações de entrada do usuário.

```
public class PerformedActionListener implements ActionListener {
  public void processAction(ActionEvent event) {
    // Retornar um componente que dispara um evento
    UICommand component = (UICommand)event.getComponent();

    // Retornar o nome de um botão ou um link
    String commandName = (String)component.getValue();

    // Criar um objeto de negócio
    LoggingService service = new LoggingService();

    // Operação de login
    service.logUserAction(commandName);
  }
}
```

Na maior parte do tempo, é mais apropriado utilizar métodos de aplicação para servir como *Action Handlers*. Primeiro, podem estar localizados no mesmo contexto que serve como *backing model* de um formulário, e assim ter acesso mais fácil aos dados fornecidos pelo usuário. Segundo, estar no *backing model* permite o desenvolvimento do grupo juntamente com os dados e os métodos capazes de retornar *outcomes* que informam a *FacesServlet* a próxima exibição da tela. *ActionListeners* podem trazer apenas o usuário para a página original após reconhecer o evento. No entanto, *ActionListeners* são a escolha mais apropriada caso se tenha uma determinada funcionalidade que será utilizada novamente através de múltiplos códigos de ação.

Veremos mais adiante a possibilidade em se ter um método de aplicação e um ou mais *ActionListeners* para atuar como *handlers* para uma *action* em particular. É então possível obter aspectos melhores de ambos os acessos e ter um método de aplicação para executar o gerenciamento específico de uma ação.

### 2.1.6. faces-config.xml

Serve como um arquivo de configuração primária para a camada *controller* da JSF. Ao contrário de uma aplicação em *Struts.* JSF **não contém** qualquer configuração de entrada para as *Actions Handlers*. **Não contém** configurações de entrada para as regras de navegação bem como para os JavaBeans que serão reconhecidos pelo sistema.

A seguir temos uma amostra de tais arquivos de configuração para uma aplicação de implementação de um caso de uso para a entrada em uma aplicação.

```
<faces-config version="1.2"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
 <managed-bean>
  <description>
   This bean serves as the backing model for our login form
  </description>
  <managed-bean-name>loginPage</managed-bean-name>
```

```
    <managed-bean-class>sample.LoginPageBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
     <description>
      The property that will store the user's login name
     </description>
     <property-name>loginName</property-name>
     <null-value/>
    </managed-property>
    <managed-property>
     <description>
      The property that will store the user's password
     </description>
     <property-name>password</property-name>
     <null-value/>
    </managed-property>
   </managed-bean>

   <navigation-rule>
    <from-view-id>/login.jsf</from-view-id>
    <navigation-case>
     <description>
      Any failure result should bring the user to an error page
     </description>
     <from-outcome>failure</from-outcome>
     <to-view-id>/error.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
     <description>
      Successful login should bring user to welcome page
     </description>
     <from-outcome>success</from-outcome>
     <to-view-id>/welcome.jsp</to-view-id>
    </navigation-case>
   </navigation-rule>
  </faces-config>
```

Vejamos alguns dos elementos acima:

**<!DOCTYPE ...**

Usado para indicar que este arquivo serve como um arquivo de configuração JSF. Qualquer falha para incluir essa linha, ou um tipo irreconhecido, resultará em um erro.

**<faces-config>**

Serve como um elemento raiz para o arquivo XML. Todos os outros elementos deverão ser filhos desse.

**<managed-bean>**

Cada elemento *managed-based* serve para definir um *JavaBean* que será controlado e reconhecido pelo sistema. Possui os seguintes elementos filhos:

- **<description>** - Facilita a leitura e a compreensão dos arquivos de configuração
- **<managed-bean-name>** - Nome lógico pelo qual uma instância pode ser acessada ou usada dentro de um sistema. Deve ser único para cada classe
- **<managed-bean-class>** - Qualifica totalmente o nome do *JavaBean* gerenciado
- **<managed-bean-scope>** - Escopo no qual o componente deve ser armazenado. Pode ser *request*, *session*, application ou pode ser deixado em branco. Não ter um valor significa que o estado do arquivo será armazenado como *request*
- **<managed-property>** - Declara o valor a ser usado para iniciar propriedades no JAVABEAN. Tem os seguintes elementos filhos:
  - **<property-name>** - Nome do *JavaBean* a ser gerenciado.
  - **<property-class>** - Define o tipo, completamente qualificado, da propriedade (elemento opcional)

- **<null-value/>** - Define um valor da propriedade como nulo, ao invés de usar o elemento value como um valor *null-value*
- **<value>** - Define um valor específico para a propriedade para um valor

**<navigation-rule>**

Este elemento é usado para definir mapeamentos lógicos para um momento decisivo na sua aplicação. Podem também ser usados para definir regras específicas para uma página em particular ou definir as regras que podem ser usadas por qualquer página na aplicação. Possui os seguintes elementos filhos:

- **<from-view-id>** - Define a página para o qual esta regra se aplicará. Se não especificada essa regra se aplicará para toda a aplicação (opcional)
- **<navigation-case>** - Define o resultado para a regra de navegação. Tem os seguintes elementos filhos (como conseqüência):
  - **<from-outcome>** - Define o resultado que determina uma ação que tornará a navegação ativa
  - **<to-view-id>** - Define a próxima página que será exibida se a navegação se tornar ativa

Existem outros elementos disponíveis para o arquivo de configuração da JSF. Consulte sua documentação de implementação JSF para obter maiores detalhes.

### 2.1.7. Sumário para realizar uma camada *Controller*

Passos para a configuração:

- Configurar o *FaceServlet* para usar em sua aplicação.
- Para cada página WEB contendo os componentes JSF UI:
  - Criar uma configuração de entrada para o programa gerenciado que servirá como o modelo *backing model* da página.
  - Criar regras de navegação que definem onde a aplicação poderia ir para a próxima parte.

## *2.2. Model*

Em JSF é necessário que se tenha áreas que armazenarão o estado dos componentes visuais em cada uma das páginas. Estas áreas são chamadas de *backing model*. Estas áreas não são parte da camada *Model* observadas estritamente abaixo da perspectiva na estrutura MVC. MVC define a camada *Model* para ser o conjunto de áreas que implementam um centro lógico para os assuntos da aplicação. Entretanto, quando pensa apenas nos componentes visuais, faz sentido chamar estas áreas de *Model*, especialmente se compararmos com a implementação MVC de áreas dos componente gráficos de *Swing*. Relembrando, em *Swing*, a camada de tradução é a camada *View*, o estado dos componentes é a camada *Model* e a ação da parte gerenciadora de eventos é a camada *Controller*.

Embora sejam consideradas como parte da *Model*, deve-se tomar cuidado no desenvolvimento destas áreas, de tal modo que elas não influenciem na funcionalidade central de sua aplicação (o modelo real). É melhor manter em mente que estes componentes são feitos para se guardar os componentes gráficos e podem ser utilizados para definir as operações básicas que acessam os dados armazenados que podem servir como métodos de aplicação. Não são feitos para executar processo pesado de regras de negócio, ou qualquer processo que pode ser usado novamente em outras aplicações.

é fácil criar um *backing model* para a página contendo componentes gráficos JSF. Criar um *JavaBean* com propriedades correspondentes para cada componente na página. São muito semelhantes aos objetos *ActionForms* do *Struts*, com a exceção de não ser necessário estender a base fornecida pelo sistema.

A seguir temos um exemplo de um *backing model* para um formulário *login*:

```
public class LoginPageBean {
  private String loginName;
  private String password;

  public String getLoginName() {
    return loginName;
  }
  public void setLoginName(String loginName) {
    this.loginName = loginName;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
}
```

Esta camada *model*, pode, então ser acessada pelas páginas após ter sido configurada no arquivo *faces-config.xml*. A configuração de entrada para este arquivo é detalhada a seguir.

```
<managed-bean>
 <description>
  This bean serves as the backing model for our login form
 </description>
 <managed-bean-name>loginPage</managed-bean-name>
 <managed-bean-class>sample.LoginPageBean</managed-bean-class>
 <managed-bean-scope>request</managed-bean-scope>
 <managed-property>
  <description>
   The property that will store the user's login name
  </description>
  <property-name>loginName</property-name>
  <null-value/>
 </managed-property>
 <managed-property>
  <description>
   The property that will store the user's password
  </description>
  <property-name>password</property-name>
  <null-value/>
 </managed-property>
</managed-bean>
```

## *2.3. View*

A camada *View* é, indubitavelmente, onde JSF deixa sua marca. Nesta camada, não apenas temos uma biblioteca de *tags*, para utilizar com JSP, como também obtemos um conjunto de componentes e uma API padronizada para seu acesso e manipulação.

Discussões de componentes Java podem ser muito complexas, se tentarmos explicar tudo ao mesmo tempo. Ao invés disso, lidamos com os fundamentos básicos e incorporamos idéias mais complicadas à medida que avançamos.

### 2.3.1. Integração JSF-JSP

Podemos iniciar a *View* dos componentes JSF pelo usuário. Seguindo uma lista de páginas JSP contendo componentes JSF baseados no exemplo anteriormente visto, temos:

```
<%@taglib uri ="http://java.sun.com/jsf/core/" prefix="f" %>
<%@taglib uri ="http://java.sun.com/jsf/html" prefix="h" %>

<HTML>
 <TITLE>Login Page</TITLE>
 <BODY>
  Please login with your login name and password : <br/><br/>
```

```
    <f:view>
     <h:form id="simpleForm">
      <h:outputLabel for="loginName">
       <h:outputText value="Login Name:"/>
      </h:outputLabel>
      <h:inputText id="loginName" value="#{loginPage.loginName}"/><br/>
      <h:outputLabel for="password">
       <h:outputText value="Password : "/>
      </h:outputLabel>
      <h:inputSecret id="password" value="#{loginPage.password}"/><br/>
      <h:commandButton action="#{loginPage.performLogin}" value="Login"/>
     </h:form>
    </f:view>
   </BODY>
  </HTML>
```

Para exibirmos componentes JSF em nossas páginas JSP. Precisamos incluir duas bibliotecas de tags: *core* e *html*.

A biblioteca *core* define as funcionalidades básicas, tal como, gerenciar os componentes JSF que sejam capazes salvar o estado. A biblioteca *html* define *tags* que ordenam o navegador como criar os componentes JSF dentro dos seus equivalentes em HTML. Uma vez que incluímos estas duas bibliotecas de *tags*, realizaremos o trabalho do mesmo modo. Olharemos para as *tag* utilizadas no exemplo anterior:

- **<f:view>** Definido na biblioteca *core*. Todas os *tags* dos componentes JSF devem ser inseridas dentro desta *tag*. Fornece um local para implementações JSF serem capazes de salvar o estado dos componentes
- **<h:form>** - Definido na biblioteca *HTML*. Cria um formulário em HTML
- **<outputLabel>** - Define um componente *label* que é associado a outro componente JSF. Este componente é associado a outro que recebe o valor do usuário, exibe como seu *label* o emissor na *tag <outputText>*
- **<h:outputText>** - Cria uma *tag* de texto dentro do valor atribuído dentro do seu equivalente HTML
- **<h:inputText>** - Cria um elemento HTML para receber informações do tipo texto
- **<h:inputSecret>** - Criar um elemento HTML do tipo de senha
- **<h:commandButton>** - Criar um botão HTML, por padrão ***SUBMIT***

A seguir veremos a página resultante da JSF anterior.


*Figura 2: HTML da página de login*

Esta página produz o seguinte código HTML:

```
<HTML>
 <TITLE>Login Page</TITLE>
 <BODY>
  Please login with your login name and password : <br/><br/>
  <form id="simpleForm" method="post" action="/JSFApplication/index.jsf"
```

```
    enctype="application/x-www-form-urlencoded">
    <label for="simpleForm:loginName">
     Login Name:
    </label>
    <input id="simpleForm:loginName" type="text"
     name="simpleForm:loginName" /><br/>
    <label for="simpleForm:password">
     Password :
    </label>
    <input id="simpleForm:password" type="password" name="simpleForm:password"
     value="" />
    <br/>
    <input type="submit" name="simpleForm:_id5" value="Login" />
    <input type="hidden" name="simpleForm" value="simpleForm" /></form>
   </form>
  </BODY>
 </HTML>
```

Podemos ver na página HTML mostrada, como a implementação JSF cria os complementos como definidos na biblioteca de *tags*.

Os elementos do formulário são definidos para utilizar o método *POST* quando na ação de submissão do formulário, apontando para uma *action point* na mesma página. Também, notaremos que os valores *id* dos elementos HTML devem ser informados com o nome do formulário. Isto assegura que o nome dos elementos do formulário são únicos dentro da aplicação, que é necessário para as operações JSF.

### 2.3.2. Value Binding

São os componentes gráficos que geram a exibição requerem um *backing model* seja capaz de armazenar dados que os usuários informarão. A implementação destes *backing models* foi discutida anteriormente. A única questão que resta é como conectar os componentes aos *backing models*.

```
...
<h:inputText id="loginName" value="#{loginPage.loginName}"/><br/>
<h:outputLabel for="password">
<h:outputText value="Password : "/>
</h:outputLabel>
<h:inputSecret id="password" value="#{loginPage.password}"/><br/>
...
```

A notação # serve como valor para o tipo atribuído e liga as propriedades ao nosso *JavaBean loginPage* com os nossos componentes visuais. Como nosso componente texto é unido ao *loginName* apropriado e o componente *InputSecret* é ligado a senha apropriada. Neste caso, *LOGINPAGE* refere-se a uma instancia de *loginPage* que armazenará os dados.

A página JSF é capaz de associar o identificador *loginPage* ao arquivo *loginPageBean* devido a nossa entada no *faces-config.xml*. A entrada relevante é apresentada abaixo.

Assim que a *LoginPageBean* é declarada para ser um arquivo gerenciado no sistema, um arquivo de *LoginPageBean* será criado e instalado na configuração armazenada (se uma delas já não existir) quando a página JSF é avaliada. Na *page submission*, os valores que o usuário anotou seriam automaticamente instalados dentro das propriedades do arquivo.

### 2.3.3. Registrando Action Handlers para os componentes da View

JSF introduz o conceito de programação baseada em eventos para o ambiente WEB. Alguns dos componentes gráficos fornecidos pela JSF, a partir de uma ação apropriada durante a entrada do usuário e gera eventos que podem ser processados pelas *Actions Handlers*.

Em nosso exemplo anterior, temos um componente JSF *<h:commandButton>*. Em qualquer momento que o usuário pressionar o botão, que este componente representa, um evento é gerado dentro do sistema e pode ser processado pela *Registered Handlers*.

Este conceito pode ser melhor entendido se comparado com outro modelo de programação

baseado em eventos: *Swing*. Em *Swing*, ao executar alguns processos, em qualquer momento, o usuário pressiona um botão e registra uma chamada, um ActionListener, que implementa uma determinada funcionalidade associada. De maneira semelhante será com JSF, para executar algumas funcionalidades relacionadas com o pressionamento em um botão, registramos *Actions Handlers* com o componente *commnandButton*.

O modo que fazemos este registro de um *handler* para um componente *commandButton* é mostrado a seguir.

```
...
<h:commandButton action="#{loginPage.performLogin}" value="Login"/>
...
```

Encontramos uma notação # semelhante a que usamos para conectar uma propriedade a um componente visual. Esta ação é semelhante a quando estamos conectando um método chamado *PerformLogin* encontrado em uma classe referenciada com o nome *LoginPage* ao botão. Nesse momento, ao invés de armazenar o valor de um componente, o método *bound* atua como um *Action Handler* para o pressionamento do botão.

### 2.3.4. Navegação de Página

JSF determina a próxima tela que será exibida para o usuário após a submissão do formulário utilizando um valor de retorno do método que serve como um *Action Handler* para o botão apresentado. Um valor do tipo *String* é visto nas regras de navegação definidas dentro do arquivo de configuração da JSF.

A entrada relevante na configuração de arquivo é mostrada a seguir.

```
<navigation-rule>
 <from-view-id>/login.jsf</from-view-id>
 <navigation-case>
  <description>
   Any failure result should bring the user to an error page
  </description>
  <from-outcome>failure</from-outcome>
  <to-view-id>/error.jsp</to-view-id>
 </navigation-case>
 <navigation-case>
  <description>
   Successful login should bring user to welcome page
  </description>
  <from-outcome>success</from-outcome>
  <to-view-id>/welcome.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Então, na página JSF de entrada (*login.jsf*), se o método *performLogin* retornar um resultado *failure*, o usuário será redirecionado para a página *error.jsp*.

Como alternativa para usar uma *Action Handler*, também é possível fornecer estaticamente uma regra de navegação a ser chamada:

```
...
<h:commandButton action="failure" value="Login"/>
...
```

Pressionar este botão redirecionará o usuário para a página *error.jsp* sem que seja necessário fornecer qualquer processo.

# 3. Outros componentes de tags JSF

Existem ainda muitas outras ações disponíveis dentro da JSF, a partir do uso de suas *tags*, que usamos em nossos exemplos anteriores para criar componentes HTML.

**<h:messages>**

Esta *tag* atua em um modo semelhante à *tag* <html:errors/> no arquitetura *Struts*. Apresenta mensagens da aplicação dentro do texto na página. Por exemplo, se um controle receptor informar um problema interno, possivelmente devido a um erro de validação, pode ser exibido usando esta *tag*.

**<h:dataTable>**

Esta *tag* exibe uma tabela em HTML (correspondente a *tag* <table>). O que torna esta *tag* útil é obter uma informação a partir de um banco de dados (ou através de uma coleção de objetos) mostrá-a automaticamente, criando os elementos necessários <tr> e <td>. Estas últimas são definidas de acordo com uma *tag* filha denominada <h:column>.

Inserido dentro de uma *tag form*. O modelo segue que cada linha corresponde a uma linha da tabela.

```
<h:dataTable id="myTable" value="#{personBean.personArray}" var="person"
 border="1" cellspacing="1" rowClasses="myTableRow">
 <h:column>
  <h:outputText value="#{person.firstName}"/>
 </h:column>
 <h:column>
  <h:outputText value=#{person.lastName}"/>
 </h:column>
</h:dataTable>
```

Este poderia ser o exemplo do HTML gerado (depende dos dados contidos em *myTable*).

```
<table border="1" cellspacing="1">
 <tr class="myTableRow">
  <td>Ewan</td>
  <td>Coba</td>
 </tr>
 <tr class="myTableRow">
  <td>Devon</td>
  <td>Shire</td>
 </tr>
 <tr class="myTableRow">
  <td>California</td>
  <td>San Diego</td>
 </tr>
</table>
```

Estes são alguns atributos da *tag <h:dataTable>*:

- **value -** Define a coleção ou banco de dados a ser pesquisado. Isto poderia ser também uma propriedade dentro de um programa gerenciado por um método que retorna o conteúdo necessário.
- **var -** Define o nome da variável que irá expor o conteúdo da coleção ou banco de dados sendo pesquisado. Esta variável é visível apenas dentro do caractere <h:dataTable>. Uma vez fora do corpo deste caractere, esta variável não pode mais ser acessada.
- **border, cellspacing –** Atributos padrão de tabelas HTML. O valor estabelecido nestes atributos será transmitido à *tag table* gerada.
- **rowClasses –** Uma lista, separada por vírgulas, de tipos para o estilo CSS (*Cascading Style Sheets*) que podem ser alternadamente aplicados para cada linha da tabela.

**<f:verbatim>**

A t*ag* <h:dataTable> permite apenas *tags* JSF dentro dos seus corpos. Isso significa que as *tags*

padrões ou ainda *tags* JSTL não funcionam dentro desta. Isso pode ser obtido com uso da *tag* *<f:verbatim>*.

# 4. Exercícios

Uma empresa necessita manter os dados de entrada e saída de seus funcionários. Para realizar isto, precisa de uma aplicação WEB para registrar as movimentações de seus funcionários. Entretanto, é necessário que isto seja feito de modo transparente e também que o registro será visível para todos.

Eventualmente, foi sugerida a seguinte idéia de pagina:

A página principal da aplicação consiste de uma área com um campo para entrada de dados, um botão no canto superior esquerdo e uma tabela de registros que ocupará o resto da pagina.

**Exemplo do 1º caso:**

O funcionário A chega ao trabalho. Insere sua matrícula no campo e pressiona o botão para confirmar sua ação. Uma nova linha será adicionada ao topo da tabela, contendo seu nome, a palavra chave IN, assim como a hora de entrada.

**Exemplo do 2º caso:**

O funcionário B chega ao trabalho e tenta submeter sua matrícula da mesma maneira que o funcionário A. Entretanto, acidentalmente comete um erro na entrada dos dados. A aplicação o leva a uma outra página aonde será informado que a matrícula não é válida. Será apresentado um *link* para retornar a página principal e corrigir o erro.

**Exemplo do 3º caso:**

O funcionário A está saindo do trabalho para almoçar. Então informa o número de sua matrícula e pressiona o botão para confirmar. A aplicação reconhece que sua entrada anterior foi o IN. O aplicativo então insere uma nova linha na tabela, com seu nome, a hora de saída e a palavra chave OUT.

Criar a aplicação a partir da sinopse descrita, o *JavaBean* padrão de funcionário está descrito a seguir.

```
public class LoginPageBean {
  private String loginName;
  private String password;

  public String getLoginName() {
    return loginName;
  }
  public void setLoginName(String loginName) {
    this.loginName = loginName;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
}
```

Esse modelo pode ser acessado por outras páginas após ter suas propriedades configuradas no arquivo *faces-config.xml*. A configuração de entrada para esse *JavaBean* é mostrada a seguir.

```
<managed-bean>
 <description>
  This bean serves as the backing model for our login form
 </description>
 <managed-bean-name>loginPage</managed-bean-name>
 <managed-bean-class>sample.LoginPageBean</managed-bean-class>
 <managed-bean-scope>request</managed-bean-scope>
 <managed-property>
  <description>
   The property that will store the user's login name
  </description>
```

```
    <property-name>loginName</property-name>
    <null-value/>
   </managed-property>
   <managed-property>
    <description>
     The property that will store the user's password
    </description>
    <property-name>password</property-name>
    <null-value/>
   </managed-property>
  </managed-bean>
```

# 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.