Desenvolvimento Web com JSF

SUMÁRIO

1. Evolução da arquitetura MVC 3
1.1. Page Centric 3
1.2. Servlet-Centric 5
2. JSF 6
3. Conceitos chave de JSF 6
4. Ciclo de vida de requisição JSF 8
4.1 Fases do ciclo de vida de requisição JSF 8
5. Usando UI Components 11
6. Conversores personalizados 11
7. Conversores padrão 11
8. Validadores 12
9. Componentes customizados 13
10. Criando componentes JSF – Parte 1 14
11. Seu componenteTag.java 15
12. Definindo o componente 17
13. Frameworks e tecnologias Web 19
13.1. RichFaces 19
13.2.IceFaces 19
13.3 Adobe Flex 19
14. Apache Tiles 19
14.1. Características 20
14.2 .Definitions 20
Referências 23

1. Evolucao da arquitetura MVC

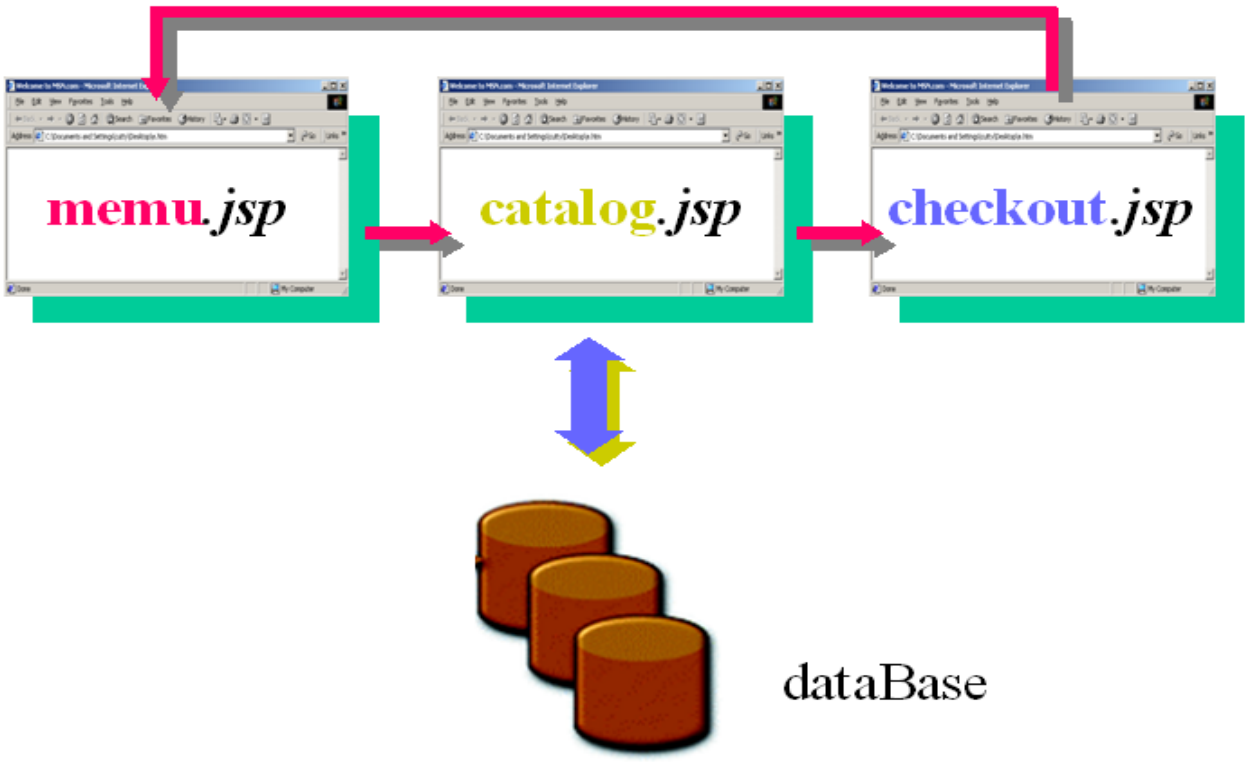
1.1. Page Centric

Quando nós falamos sobre frameworks de aplicações Web, nós estamos falando basicamente da evolução da arquitetura MVC (Model, View, and Controller).

No início o MVC não era usado. Depois começamos com a arquitetura MVC Modelo 1 e o MVC Modelo 2. O framework que popularizou os framework para desenvolvimento Web foi o Apache Struts. Ele é basedo na arquitetura MVC Modelo 2. Nós estamos na fase dos frameworks baseados em padrões, onde o JSF é o padrão estabelecido no JEE 5.0.

Vejamos a arquitetura MVC Modelo 1.

A arquitetura modelo 1 é uma arquitetura centrada em páginas (page-centric architecture).



Page centric - Aplicações baseadas em páginas

A literatura da tecnologia da camada Web na plataforma JEE freqüentemente usa os termos “Modelo 1” e “Modelo 2” sem explicação. Esta terminologia se refere aos primeiros esboços da especificação JSP, que descrevia dois padrões básicos de uso de páginas JSP. Apesar dos termos terem desaparecido do documento de especificação, eles continuam no vocabulário dos usuários.

Modelo 1 e Modelo 2 se referem simplesmente à ausência ou presença de um servlet controlador que dispacha as requisições da camada cliente e seleciona as visões.

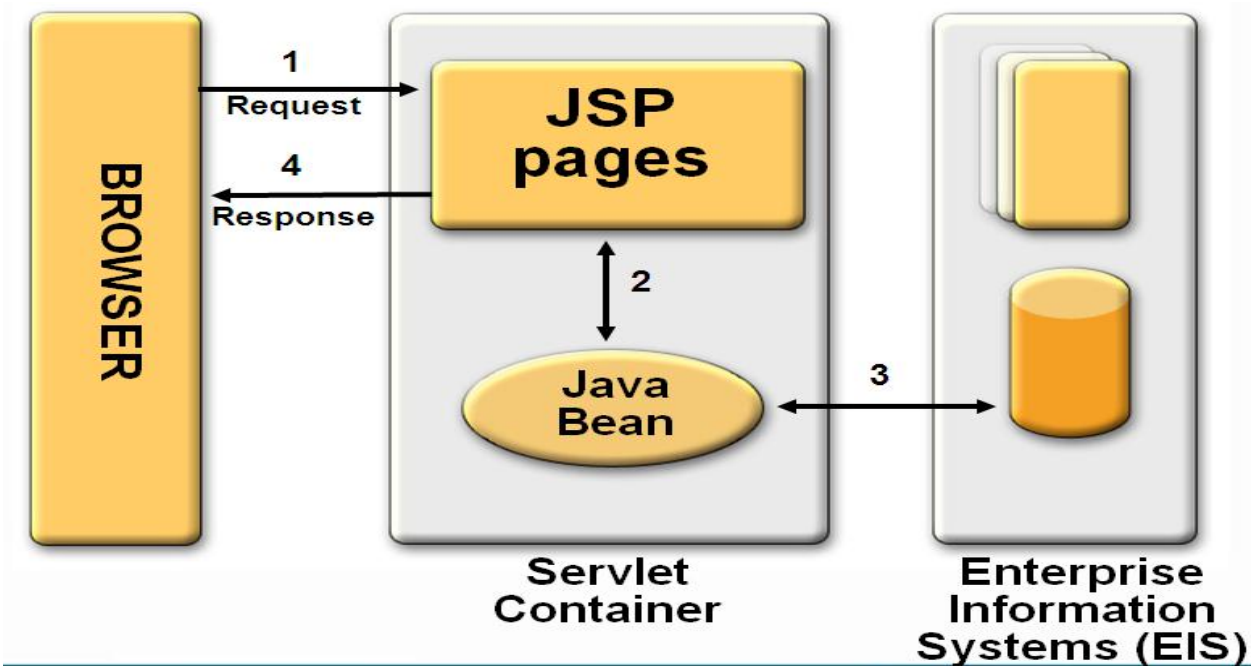
1 - Na arquitetura modelo 1 o browser Web acessa diretamente as páginas JSP na camada Web.

2 - As páginas JSP acessam os JavaBeans na camada Web.

3 - Os Javabeans representam a camada modelo. Eles são responsáveis por acessar os recursos da camada EIS.

4 - A próxima visão a ser mostrada (página JSP, servlet, página HTML ou outro recurso) é determinada pelo hyperlink selecionado no documento fonte ou pelos parâmetros da requisição.

Em uma arquitetura Modelo 1, a seleção de visão é descentralizada, porque a página corrente determina a próxima página a ser exibida. Cada página JSP ou servlet processa os seus próprios dados de entrada (parâmetros como GET ou POST). Isso é difícil de manter, por exemplo, se você tem de mudar a visão selecionada, você terá de alterar isso em diversas páginas JSP.



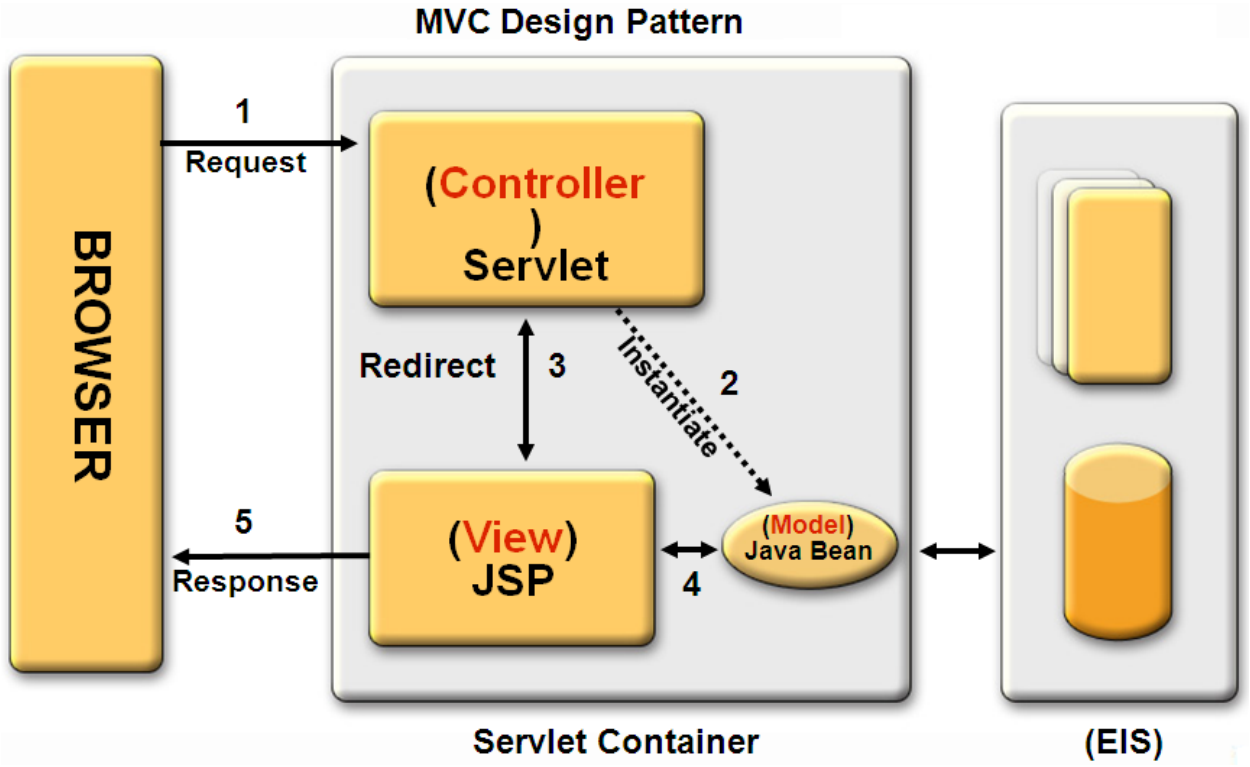
1.2. Servlet-Centric

A arquitetura Modelo 2 introduz um servlet controlador entre o browser Web e as páginas JSP.

O controlador centraliza a lógica para despacho das requisições para a próxima visão baseado na URL da requisição, parâmetros de entrada e o estado da aplicação. O controlador também trata a seleção de visões, o que desacopla as páginas JSP e os servlets um do outro.

Aplicações Modelo 2 são fáceis de manter e estender, porque as visões não são referenciadas diretamente. O servlet controlador Modelo 2 fornece um único ponto de controle para segurança e logging e geralmente encapsulam os dados de entrada de forma usável pelo modelo MVC.

Por estas razões, a arquitetura Modelo 2 é recomendada para a maioria das aplicações Web.



2. JSF

JavaServer Faces é um framework MVC para o desenvolvimento de aplicações Web, que permite o desenvolvimento de aplicações para a internet de forma visual, ou seja, arrastando e soltando os componentes na tela (JSP), definindo propriedades dos mesmos.

O JavaServer Faces teve sua expressão na versão 1.1 quando implementado pela comunidade utilizando a especificação 127 [1] do Java Community Process [2], evidenciando maturidade e segurança.

Hoje ele está na versão 1.2 da especificação 252 [3] do JCP. A fundação Apache vem realizando esforços na implementação da especificação através do projeto MyFaces. O reconhecimento do trabalho é visto por diversas empresas, tanto é que a Oracle doou os fontes do ADF Faces, conjunto de mais de 100 componentes JSF, para o projeto MyFaces que o denominará de Trinidad.

O JSF é atualmente considerado pela comunidade Java como a última palavra em termos de desenvolvimento de aplicações Web utilizando Java, resultado da experiência e maturidade adquiridas com o JSP/Servlet (Model1), Model2 (MVC) e Struts.

Características:

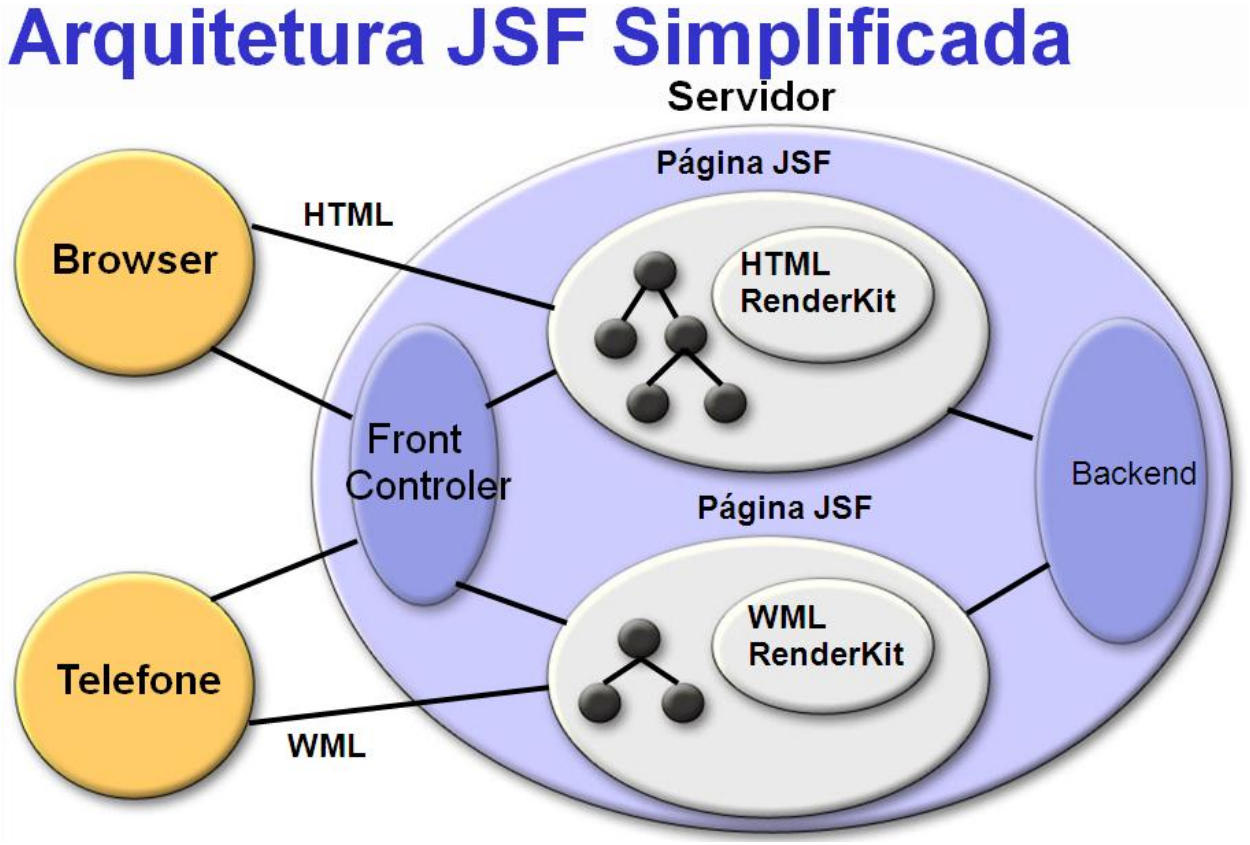
- Modelo extensível de componentes
- Modelo flexível de renderização
- Modelo de tratamento de eventos
- Framework de validação
- Suporte a navegação entre páginas
- Internacionalização
- Acessibilidade

3. Conceitos chave de JSF

- UIComponent
  - Classe base com comportamento padrão
- Subclasses de UIComponent
  - UICommand, UIForm, UIGraphic, UIInput, UIOutput, UIPanel, UISelectBoolean, UISelectMany, UISelectOne

- FacesEvent
  - Classe base para requisições e eventos de aplicação
- Validator
  - Classe base para validadores
- Converter
  - Conversão de String para Object
- FacesContext
  - Encapsula Servlet request, response e session
- Renderer
  - Converte componentes para uma linguagem de marcação específica
- RenderKit
  - Biblioteca de Renderers
  - RenderKit Basico para HTML é parte da especificação

Com esses conceitos e características, temos então um modelo arquitetural simplificado do JSF. A figura abaixo exibe com clareza essa arquitetura:



4. Ciclo de vida de requisição JSF

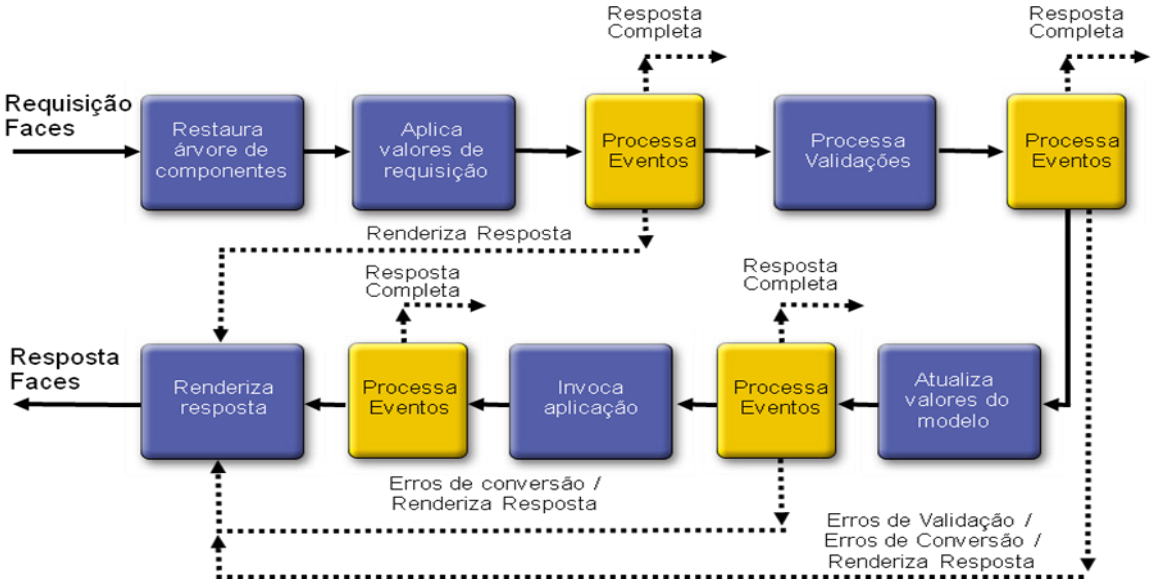
- Uma página JSF é representada por uma árvore de componentes UI, chamada de view.
- Quando um cliente faz uma requisição de uma página, o ciclo de vida inicia.
- Durante o ciclo de vida, a implementação JSF tem de construir a view enquanto salva o estado considerando a requisição enviada (Postback).
- Quando o cliente executa um envio de uma página, a implementação JSF executa os passos do ciclo de vida.

Atribuição, validação, conversão, ...

4.1. Fases do ciclo de vida de requisição JSF

1. Restaura arvore de componentes
2. Aplica valores de requisição
3. Processa validações
4. Atualiza valores do modelo
5. Invoca aplicação
6. Renderiza resposta

A imagem abaixo mostra com detalhes quando cada uma dessas fases entra em execução:



Fase 1: Restaura arvore de componentes

- Uma requisição chega através do controlador FacesServlet
- O controlador examina a requisição e extrai o view ID, o qual é determinado pelo nome da página JSP
- O controlador da framework JSF usa o view ID para carregar os componentes. A view contém todos os componentes GUI
  - Se a *view* ainda não existe, o controlador JSF cria
  - Se a *view* já existir, o controlador JSF usa a existente
- Quando uma requisição JSF é criada, clicando em um botão ou link por exemplo, a implementação JSF sempre executa a fase 1
  - Se for a primeira vez, uma view vazia é criada
- A implementação JSF executa
  - Cria a view da página JSF
  - Sobe os tratadores de eventos e validadores
- Salva a view no FacesContext

Fase 2: Aplica valores de requisição

- Os novos valores de cada componente da arvore são extraídos dos parâmetros da requisição
- Os valores são convertidos para o tipo correto do compoente
  - Para o componente userNo na página greeting.jsp, o tipo é convertido de String para Integer

    Erros de conversão são empilhados no FacesContext

Fase 3: Processa validações

- A implementação JSF processa todas as validações de entrada registradas nos componentes da arvore
  - Esta é uma validação de entrada (não é uma validação de lógica de negócio)

- No caso de erros de validação
  - Mensagens de erro são empilhadas no FacesContext
  - Ciclo de vida avança diretamente para a fase “6:Renderizar resposta”
- Exemplo
  - userNo tem de estar entre 1 e 10

Fase 4: Atualiza valores do modelo

- A implementação JSF percorre a arvore de componentes e atribui a propriedade correspondente do objeto do lado servidor ao valor local do componente.
  - Atualiza as propriedades do bean apontadas no atributo value do componente input.
  - Conversão do tipo do valor local para o tipo da propriedade do valor do modelo.
- Exemplo na greeting.jsp
  - A propriedade userNumber do UserNumberBean é armazenada no valor local do componente userNo.

Fase 5: Invoca aplicação

- Implementação JSF trata eventos no nível de aplicação, como envio de formulário ou link para outra página.
- A página greeting.jsp do exemplo guessNumber tem um evento no nível de aplicação associado com o componente UICommand.
- Uma implementação de ActionListener recupera o outcome, "success", do atributo action do componente.
- O listener passa o outcome para o NavigationHandler padrão.
- O NavigationHandler aplica o outcome à regra de navegação definida no arquivo de configuração para determinar qual a próxima página deve ser exibida.
- A implementação JSF então atribui esta nova página à resposta.

Fase 6: Renderiza resposta

- A implementação JSF chama os métodos de conversão dos componentes e renderiza os componentes da árvore salva no FacesContext.

- Cria marcação da árvore de componentes apropriada.
- Se ocorreram erros nas fases anteriores, a página original é renderizada com as mensagens de erro empilhadas.

- O estado da resposta é salvo, para que as requisições subseqüentes possam acessá-lo e para tornar disponível para a fase "1:Restaura árvore de componentes".

5. Usando UI Components

Conversores e seu papel

- Converter converte os dados entre as duas camandas
  - Modelo de apresentação da visão
  - Apresentação modelo para visualizar
- Tipos de Conversores
  - Conversores padrão fornecido com uma aplicação JSF

6. Conversores personalizados

A conversão automática de implementação JSF

- Aplicação JSF converte automaticamente dados de componentes entre as duas visões, quando a propriedade do bean associado com o componente é um dos tipos suportados

Exemplo:

- Se um componente UISelectBoolean está associado com uma propriedade de bean de tipo java.lang.Boolean, a implementação JSF irá converter automaticamente os dados de String para Boolean

7. Conversores Padrão

- BigDecimalConverter
- BigIntegerConverter
- BooleanConverter
- ByteConverter
- CharacterConverter

- DateTimeConverter (has its own tag)
- DoubleConverter
- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter (has its own tag)
- ShortConverter

Como usar?

Faça o componente que usa o conversor ligar a uma propriedade de bean de apoio do mesmo tipo. Veja o código abaixo:

```
01. <h:inputText value="#{livroMB.livro.quantidade}" required="true"
02. label="#{msg.label_quantidade}"  maxlength="2">
03. <f:convertNumber />
04. </h:inputText>
```

Código no lado do servidor:

```
01. Integer quantidade= 0;
02. public Integer getQuantidade(){ return quantidade;}
03. public void setQuantidade(Integer quantidade) {quantidade =
quantidade;}
```

8. Validadores

- Validação só pode ser executada em um componente UIInput ou componentes que herdem da classe UIInput
  - Componentes UI que aceitam valores do usuário final.

Como exibir mensagens de erro de validação?

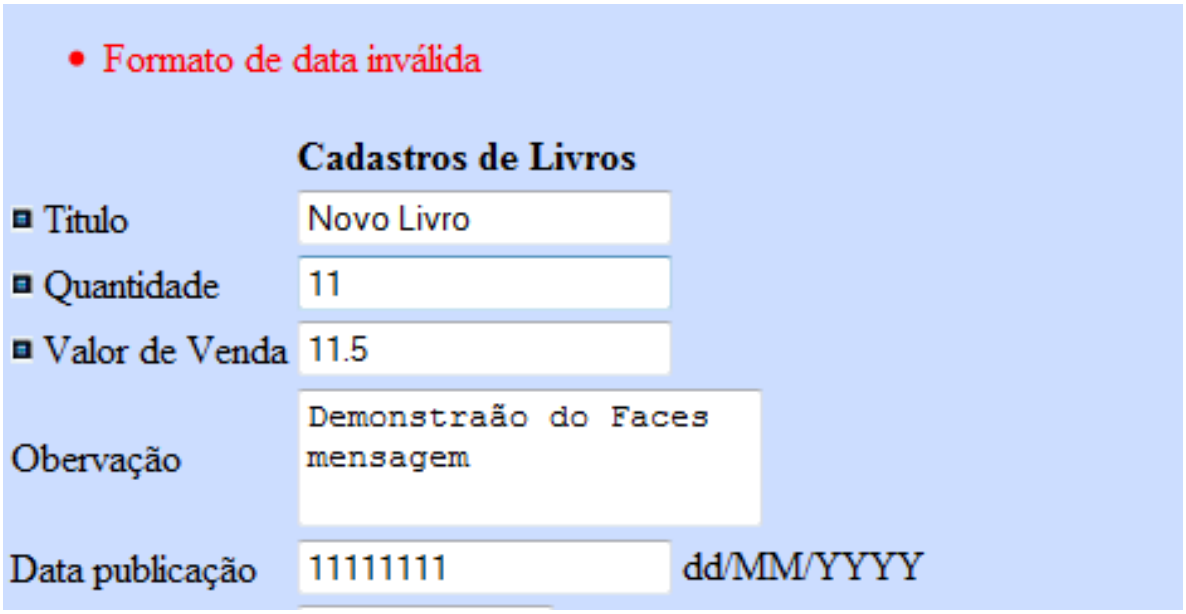
- O autor de páginas pode mandar mensagens de erro como resultado de falhas de validação padrão e personalizadas usando tag <h:messages>
  - A tag <h:messages> exibe a mensagem de erro de validação na posição onde ela for colocada na página.
  - O atributo for da tag tem de ser igual ao “id” do componente que terá o seu valor validado.

O atributo style especifica um estilo de exibição para a mensagem.

Exemplo: cadastroProduto.jsp:

```
01. <h:inputSecret id="novaSenha" value="#{usuarioMBean.usuario.senha}"
02.binding="#{usuarioMBean.senhaInput}" styleClass="inputTxtComum"style="w
03. idh:175px;" maxlength="10">
04. <f:validateLength minimum="6"/>
05. </h:inputSecret>

06. <h:messages for="novaSenha"
07. styleClass="#{(facesContext.maximumSeverity.ordinal eq 0 )? color :
08. red ' : ' color : green'}" showDetail="true" globalOnly="false"
09. showSummary="false" />
```



9. Componentes customizados

Para um componente qualquer, precisamos criar uma TLD com as propriedades do componente, um faces-config com os mapeamentos necessários, e finalmente as classes que definem as funcionalidades e o próprio componente. (Levando em consideração que certas classes que adicionam recursos como Javascript e imagens, já foram desenvolvidas pelo myfaces, bastando apenas talvez, extendê-las para aplicar funcionalidades específicas. Criamos então um projeto com uma pasta chamada META-INF. Nesta pasta incluímos o faces-config e a nossa TLD.

META-INF/faces-config.xml:

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <!DOCTYPE faces-config PUBLIC
03. "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
04. "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
05. <faces-config xmlns="http://java.sun.com/JSF/Configuration">
06. <component>
07. <component-type>tipoDoComponente</component-type>
```

```
08. <component-class>pacote.ClasseDoComponente</component-class>
09. </component>

10. <converter>
11. <converter-for-class>[Ljava.lang.String;</converter-for-class>
12. <converter-

13. class>org.apache.myfaces.convert.StringArrayConverter</converter-class>
14. </converter>

15. <render-kit>
16. <render-kit-id>HTML_BASIC</render-kit-id>
17. <renderer>
18. <component-family>tipoDoComponente</component-family>
19. <renderer-type>familiaDoComponente</renderer-type>
20. <renderer-class>pacote.ClasseDoRenderizador</renderer-class>
21. </renderer>
22. </render-kit>
23. <managed-bean>
24. </managed-bean>
25. </faces-config>
```

10. Criando componentes JSF – Parte 1

Para um componente qualquer, precisamos criar uma TLD com as propriedades do componente, um faces-config com os mapeamentos necessários, e finalmente as classes que definem as funcionalidades e o próprio componente. (Levando em consideração que certas classes que adicionam recursos como Javascript e imagens, já foram desenvolvidas pelo myfaces, bastando apenas talvez, extendê-las para aplicar funcionalidades específicas. Criamos então um projeto com uma pasta chamada META-INF. Nesta pasta incluímos o faces-config e a nossa TLD.

META-INF/faces-config.xml:

```
01. <?xml version=”1.0″ encoding=”UTF-8"?>
02. <!DOCTYPE faces-config PUBLIC
03. “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN”
04. “http://java.sun.com/dtd/web-facesconfig_1_1.dtd”>
05. <faces-config xmlns=”http://java.sun.com/JSF/Configuration”>
06. <component>
07. <component-type>tipoDoComponente</component-type>
08. <component-class>pacote.ClasseDoComponente</component-class>
09. </component>

10. <converter>
11. <converter-for-class>[Ljava.lang.String;</converter-for-class>
12. <converter-

13. class>org.apache.myfaces.convert.StringArrayConverter</converter-class>
14. </converter>
```

```
15. <render-kit>
16. <render-kit-id>HTML_BASIC</render-kit-id>
17. <renderer>
18. <component-family>tipoDoComponente</component-family>
19. <renderer-type>familiaDoComponente</renderer-type>
20. <renderer-class>pacote.ClasseDoRenderizador</renderer-class>
21. </renderer>
22. </render-kit>
23. <managed-bean>
24. </managed-bean>
25. </faces-config>
```

Neste arquivo definimos qual a classe do nosso componente, qual o nosso renderizador, se houver um managed-bean para ele, especificamos (da mesma forma que o comum) e os eventuais conversores que poderemos utilizar(inseri apenas um para o exemplo).

META-INF/nomeDaTLD.tld :

```
01. <?xml version = '1.0' encoding = 'UTF-8'?>
02. <taglib xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03. xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
04. http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0"
05. xmlns="http://java.sun.com/xml/ns/j2ee">
06. <display-name>nomeDaTLD</display-name>
07. <tlib-version>1.0</tlib-version>
08. <short-name>nomeReduzido</short-name>
09. <uri>http://auri.algumacoisa</uri>
10. <tag>
11. <name>nomeDoComponente</name>
12. <tag-class>classeTagDoComponente</tag-class>
13. <body-content>JSP</body-content>
14. <attribute>
15. <name>value</name>
16. <required>true</required>
17. <rtexprvalue>false</rtexprvalue>
18. <type>java.lang.String</type>
19. </attribute>
20. </tag>
```

Especificamos alguns atributos (todos do tipo String), o nome do componente e a URI para acesso a nossa taglibrary. Feito esse processo, vamos criar as outras classes necessárias.

O ideal é criar 3 classes: Uma que representa a tag, uma que represente o componente em si e outra que realize a renderização do componente.

11. Seu ComponenteTag.java –

Esta classe deve possuir todos os atributos especificados no arquivo de TLD, pois através desta classe serão atribuitos os valores informados pelo usuario

(programador) ao componente. Como tudo é muito simples, desta vez, pouparei alguns comentários. Segue abaixo um exemplo:

```
01. public class SeuComponenteTag extends UIComponentTagBase{
02. private String _id;
03. private String _value;
04. private String _styleClass;

05. public void setId(String id) {
06. this._id = id;
07. }
08. public String getComponentType(){
09. return "tipoDoComponente"; // o mesmo que definimos no META-INF/faces-

10. config.xml
11. }

12. public String getRendererType(){
13. return "pacote.ClasseDoRenderizador"; // idem
14. }
15. public void release(){
16. super.release();
17. _id = null;
18. _value = null;
19. _styleClass = null;
20. }
21. protected void setProperties(UIComponent component){
22. super.setProperties(component);

23. FacesContext context = getFacesContext();

24. if (_id != null){
25. ((SeuComponente)component).setId(_id);
26. }

27. if (_value != null){
28. ValueBinding vb = context.getApplication().createValueBinding(_value);
29. component.setValueBinding("valor", vb);
30. }

31. if (_styleClass != null){
32. ((SeuComponente)component).setStyleClass(_styleClass);
33. }
34. }
```

Neste momento já temos um componente tomando forma, pois já definimos seus atributos e ele próprio nos arquivos xml da nossa biblioteca de componentes, já definimos como serão atribuídos os valores especificados pelo usuário ao componente.

Precisamos agora criar o componente em si e a classe que o renderiza.

12. Definindo o componente:

Basta criarmos uma classe que chamaremos HtmlSeuComponente. Nela redefiniremos os atributos definidos na classe SeuComponenteTag citados acima. Criaremos seus gets e sets e mais algumas funções necessárias.

```
01. public class HtmlSeuComponente extends UIComponentBase{
02. private String _id;
03. private String _value;
04. private String _styleClass;

05. public HtmlSeuComponente() {
06. super();
07. setRendererType("pacote.ClasseDoRenderizador");
08. }

09. public String getFamily(){
10. return "familiaDoComponente";
11. }

12. public Object saveState(FacesContext context){
13. Object values[] = new Object[4];
14. values[0] = super.saveState(context);
15. values[1] = _id;
16. values[2] = _value;
17. values[3] = _styleClass;

18. return values;
19. }
20. public void restoreState(FacesContext context, Object state){
21. Object values[] = (Object[])state;
22. super.restoreState(context, values[0]);
23. setId((String)values[1]);
24. setValue((String)values[2]);
25. setStyleClass((String)values[3]);
26. }

28. public void setId(String id) {
29. this._id = id;
30. }

31. public String getId(){
32. if (_id != null)
33. return _id;

34. ValueBinding vb = getValueBinding("id");
35. if (vb == null)
36. return null;
37. else
38. return (String)vb.getValue(getFacesContext());
39. }

40. }

41. public class HtmlSeuComponenteRenderer extends Renderer{
```

```
42. public void encodeBegin(javax.faces.context.FacesContext context,

43. UIComponent component) throws IOException{
44. super.encodeBegin(context, component);
45. }

46. public void encodeEnd(javax.faces.context.FacesContext context,

47. UIComponent component) throws IOException{
48. super.encodeEnd(context, component);
49. decode(context, component);
50. }

51. public void decode(FacesContext context, UIComponent component){
52. ResponseWriter writer = context.getResponseWriter();

53. }

54. protected Class getComponentClass() {
55. return HtmlSeuComponenteRenderer.class;
56. }
57.}
```

Como você pode notar, esta classe é extremamente simples, mas começa atingir níveis maiores de complexidade quando começamos a adicionar recursos na página onde o componente será renderizado, como por exemplo javascript e css. Ainda podemos ter requisições AJAX envolvidas.

Portanto, um componente JSF funciona da seguinte maneira:

Existe um xml que define o componente e suas propriedades (META-INF/nomeDaTLD.tld) e um xml que define as classes envolvidas no processo (META-INF/faces-config.xml). Quando instanciamos o componente numa página JSP/JSF, utilizamos a definição da TLD. O JSF se encarrega de definir os valores na classe SeuComponenteTag, que por sua vez repassa ao HtmlSeuComponente que faz o binding dos valores reais. Em seguida é chamado o HtmlSeuComponenteRenderer que renderiza o seu componente conforme a sua vontade.

Para finalizar vamos utilizar nosso componente. Para uma página JSF/JSP qualquer:

```
01. <%@ taglib uri="http://auri.algumacoisa" prefix="tb"%>

02. <tb:nomeDoComponente id="idDele"
03. value="#{AlgumaClasse.propriedade}"
04. binding="#{AlgumaClasse.objetoQueRepresentaOComponente}"
05. styleClass="nomeDaClasseCss"/>
```

Dessa forma temos um componente customizado.

13. Frameworks e tecnologias Web

13.1 - RichFaces

RichFaces é uma biblioteca de componentes para aplicações web que utilizam o framework JSF. Os componentes desta biblioteca possuem um incrível suporte AJAX, e ela, pode ser considerada uma extensão do Ajax4jsf com inúmeros componentes com Ajax "embutido" e com um suporte a Skins que podem deixar as interfaces da sua aplicação com um visual padronizado.

13.2 – IceFaces

O ICEFaces é um conju nto de comp o nente s o pen source, de senvolvido pela ICESoft. Tem por finalidade integrar as tecnologias JSF e Ajax de forma na tiva, ou seja, to dos os compo nentes do ICEfaces s ão compo nentes JSF q ue dão suporte ao Ajax.

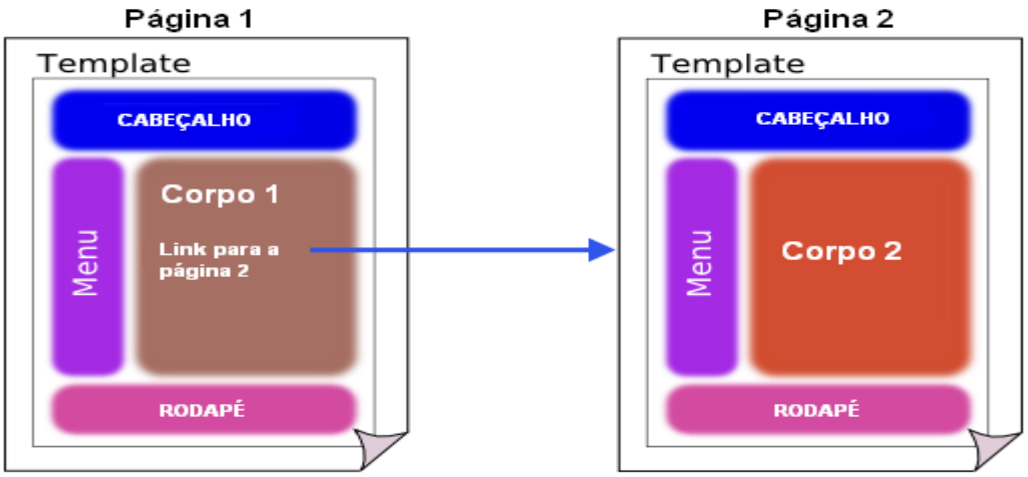
13.3 - AdobeFlex

O Adobe Flex (antes chamado de Macromedia Flex e depois rebatizado como Adobe Flex pela Adobe) é o nome de uma tecnologia lançada em Março de 2004 pela Macromedia, que suporta o desenvolvimento de aplicações ricas para a Internet, baseadas na plataforma do Macromedia Flash. A versão inicial possuía um SDK, um IDE e uma integração com o J2EE também conhecido como Flex Data Services. Desde que a Adobe adquiriu a Macromedia em 2005, as versões subsequentes do Flex começaram a requerer uma licença para o Flex Data Services, que era inicialmente um produto separado e que posteriormente foi rebatizado como LiveCycle Data Services.

Em abril de 2007, a Adobe anuncia planos de abrir o código do Flex 3 SDK. O Adobe Flash Player, aplicativo pelo qual são visualizados as aplicações Flex, e o Flex Builder, a IDE utilizada para desenvolver aplicações Flex, continuam proprietárias e comerciais.

14. Apache Tiles

O Apache Tiles é um framework para construção de páginas baseado em peças reutilizáveis (mais conhecido como framework de template). Ele é concebido sob o padrão Composite View e internamente, em seu core, o padrão Decorator. A figura abaixo nos dá idéia dessa reutilização:



Quando o corpo 1 requisita uma outra página, essa é incluída pelo Tiles dentro do template. Assim, as páginas contém somente seu conteúdo próprio evitando a reescrita de código.

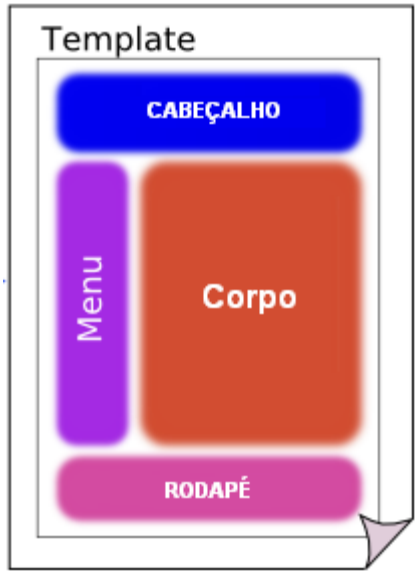
14.1 Características

- Suporte a wildcards, com as sub-features abaixo:
- nas definições de arquivos XML;
- em enumerações de listas nos arquivos do Tiles a serem carregados;
- Suporte a EL na definição de arquivos do Tiles.
- Habilidade para carregar arquivos de definição das versões 1.x do Tiles.
- Habilidade para sobrescrever templates quando definições são inseridas.
- Customização de atributos renderizáveis, como por exemplo para integrar com outros frameworks.
- Utilização de mais de um container do Tiles.

14.2. Definitions

Uma definição consiste em uma composição de páginas para o usuário final. Basicamente, um definition contém tags Tiles que definem um conteúdo padrão.

Veja a figura abaixo:



Essa figura pode ser dada como um template Tiles chamado por exemplo de classic.jsp. O código interno que a compões seria como o descrito abaixo:

```
01. <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
02. <table>
03.  <tr>
04.    <td colspan="2">
05.      <tiles:insertAttribute name="header" />
06.    </td>
07.  </tr>
08.  <tr>
09.    <td>
10.      <tiles:insertAttribute name="menu" />
11.    </td>
12.    <td>
13.      <tiles:insertAttribute name="body" />
14.    </td>
15.  </tr>
16.  <tr>
17.    <td colspan="2">
18.      <tiles:insertAttribute name="footer" />
19.    </td>
20.  </tr>
21. </table>
```

Aqui, é descrito como a definição ficaria no arquivo de configurações do Tiles:

```
01. <definition name="myapp.homepage" template="/layouts/classic.jsp">
02.  <put-attribute name="header" value="/tiles/banner.jsp" />
03.  <put-attribute name="menu" value="/tiles/common_menu.jsp" />
04.  <put-attribute name="body" value="/tiles/home_body.jsp" />
05.  <put-attribute name="footer" value="/tiles/credits.jsp" />
06. </definition>
```

De posse dessas definitions, podemos usá-las em nossas páginas de uma forma bem simples. Veja os código abaixo:

```
01. <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
02. <tiles:insertDefinition name="myapp.homepage" />
```

É importante perceber que o atributo name da tag insertDefinitions contém o mesmo nome dado no atributo name na tag definitions no arquivo de configuração do Tiles.

É possível também estender as definitions. Veja o exemplo abaixo:

```
01. <definition name="myapp.page.common" template="/layouts/classic.jsp">
02.  <put-attribute name="header" value="/tiles/banner.jsp" />
03.  <put-attribute name="menu" value="/tiles/common_menu.jsp" />
04.  <put-attribute name="footer" value="/tiles/credits.jsp" />
05. </definition>
```

E aqui temos um descendente de myapp.page.common:

```
01. <definition name="myapp.homepage" extends="myapp.page.common">
02.  <put-attribute name="title" value="Tiles tutorial homepage" />
03.  <put-attribute name="body" value="myapp.homepage.body" />
04. </definition>
```

Também é possível sobreescrever as definições. Veja o exemplo abaixo:

```
01. <definition name="myapp.homepage.alternate" extends="myapp.homepage"
02.    template="/layouts/alternate.jsp" />
```

e sua respectiva sobreescrita:

```
01. <definition name="myapp.homepage.customer" extends="myapp.homepage">
02.  <put-attribute name="menu" value="/common_menu_for_customers.jsp" />
03. </definition>
```

REFERÊNCIAS

1. JSF: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

2. Tiles: http://tiles.apache.org/framework/tutorial/index.html