Módulo 6

Programação WEB

Lição 10

Tópicos Avançados de 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, aprendemos sobre a *JavaServer Faces e* como este *framework* combina uma *servlet front controller*, *action handlers*, e um conjunto de componentes visuais para simplificar a tarefa de desenvolver uma aplicação sob a arquitetura de MVC (Model - View - Controller). Vimos como JSF pode criar um estilo, parecido com a biblioteca *Swing* com sua arquitetura de componentes baseada em eventos, para programação no ambiente WEB.

Nesta lição discutiremos o objeto *FacesContext*, que obtém todas as informações do contexto dentro da JSF. Veremos outros dois conjuntos de componentes fornecidos para uso neste *framework*: Componentes Validadores e para conversão de tipo. Aprenderemos como criar tais componentes, e como desenvolver nosso conjunto de *tags* personalizadas para que os componentes criados possam ser representados mais facilmente nas páginas JSP.

Ao final desta lição, o estudante será capaz de:

- Conhecer a classe *FacesContext*
- Entender os componentes validadores e para conversão de tipo
- Criar componentes e como desenvolver um conjunto de *tags* personalizadas

2. FacesContext

Na lição anterior, vimos como fazer uso do objeto *FacesContext* para acessar uma coleção de objetos correntes na sessão do usuário. Nesta lição entraremos em mais detalhes e veremos outros usos para este objeto.

Como item de revisão, uma instância do objeto de *FacesContext* pode ser obtida chamando o método *getCurrentInstance*() disponível na classe *FacesContext*.

2.1. FacesContext e a Árvore de Componentes

Para cada uma das visões fazemos uso dos elementos gráficos de JSF, existe um modelo disponível em estrutura de árvore de componente. A especificação JSF requer que todas as implementações armazenem uma estrutura de árvore de componente dentro do objeto *FacesContext*. Esta estrutura permite, aos desenvolvedores, o acesso a todos os componentes de *interface* do usuário e suas informações.

Existem diversas tarefas que podemos empregar este tipo de acesso a árvore de componentes. Através de programação, adicionamos componentes dinamicamente para a camada *View*, mudamos ou adicionamos componentes de conversão e validação ou ainda, podemos remover componentes.

Freqüentemente, usamos esta capacidade para um redirecionamento de tela dentro do *framework*. Isto pode ser feito modificando à árvore de componentes atual que é acessada pelo usuário pela árvore de componentes situada em outra página.

A seguir, temos uma amostra de um trecho que realiza esta tarefa (a linha relevante aparece em negrito).

```
...
String targetPage = "/newPage.jsp";
FacesContext context = FacesContext.getCurrentInstance();
context.getApplication().getViewHandler().createView(context, targetPage);
...
```

O método *createView* cria uma cópia da árvore de componentes que compõe o *targetPage* e o coloca como a árvore de componentes atual. Uma vez que o *framework* exibe o conteúdo novamente, exibirá o conteúdo da página.

2.2. FacesContext e o ExternalContext

O objeto *ExternalContext*, acessível pela classe *FacesContext*, permite o acesso ao ambiente atual do *framework*, em nosso caso (uma aplicação WEB), permite o acesso ao objeto de *HttpServletRequest* representando o requerimento atual, um objeto do tipo *Map* armazenado na sessão do usuário, como também ao objeto *ServletContext*, representando o contexto da aplicação WEB.

Para recuperar o objeto de *HttpServletRequest*:

```
FacesContext context = FacesContext.getCurrentInstance();
HttpServletRequest request = (HttpServletRequest)context.getExternalContext()
  .getRequest();
```

Infelizmente, não existe nenhum modo para acessar diretamente o objeto *HttpSession* associado à sessão. Entretanto, os objetos armazenados dentro desta podem ser acessados por um objeto do tipo *Map*, do seguinte modo:

```
Map sessionMap = context.getExternalContext().getSessionMap();
```

Para recuperar o *ServletContext* representado na aplicação, temos a seguinte instrução:

```
ServletContext servletContext =
  (ServletContext)context.getExternalContext().getContext();
```

Uma vez que temos acesso a estes objetos, podemos então fazer uso de outras técnicas de programação WEB.

2.3. Validadores

Na lição sobre o *framework Struts*, discutimos a validação e sua importância em qualquer aplicação baseada em dados. Qualquer resultado da aplicação será obtido através da informação e que os dados corretos estão sendo informados.

JSF também reconhece esta necessidade de validação e fornece um conjunto de validadores que podemos usar em nossa aplicação. Fornece também vários caminhos para unir o código de validação aos componentes de entrada de dados.

2.4. JSF Validadores Padrão

JSF fornece três validadores padrões:

- ***DoubleRangeValidator*** – verificar se o valor no campo de entrada pode ser convertido para um tipo *double* e seu valor está entre o mínimo e o máximo de determinados valores fornecidos. *Tag: ValidateDoubleRange*. *Atributos: minimum* e *maximum*.
- ***LengthValidator*** – verificar se o valor no campo de entrada pode ser convertido para uma String e que seu tamanho está entre o mínimo e o máximo de determinados valores fornecidos. *Tag: ValidateLength*. *Atributos: minimum* e *maximum*.
- ***LongRangeValidator*** - verifica se o valor no campo de entrada pode ser convertido para um tipo *long*, e seu valor está entre o mínimo e o máximo de determinados valores fornecidos. *Atributos: minimum* e *maximum*.

As *tags* JSP para o validadores podem ser encontradas dentro da biblioteca de *tags*. Para fazer uso de *tags* dentro da biblioteca, devemos adicionar a seguinte linha no topo da página:

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

2.5. Usando os validadores padrões

Fazer uso dos validadores padrões, basta inserir a *tag* de JSP do validador dentro do corpo do componente de entrada. Observe o seguinte exemplo.

```
 ...
 <h:outputLabel for="password">
  <h:outputText value="Password : "/>
 </h:outputLabel>
 <h:inputSecret id="password" value="#{loginPage.password}">
  <f:validateLength minimum="4"/>
 </h:inputSecret>
 <br/>
 <h:commandButton action="#{loginPage.performLogin}" value="Login"/>
</h:form>
...
```

Este é o código JSP para a página de *login* que implementamos na lição anterior usando JSF. Será modificado de tal forma que o campo senha aceita somente uma entrada com o tamanho mínimo de 4 caracteres. As modificações são:

- A *tag* que define o campo de senha <*h:inputSecret*> foi modificada de modo que a *tag* não é mais fechada propriamente na mesma linha que era declarada. Agora faz uso de uma outra *tag* </h:inputSecret> para fechar corretamente.
- Para que a *tag* que define a validação funcionar <*f:validateLength*>, foi inserida entre a abertura e fechamento da tag <h:inputSecret>.

Para aceitar uma entrada que restrinja pelo tamanho **máximo** em vez de um tamanho mínimo, devemos substituir o atributo *minimum* pelo atributo *maximum*, e especificar este tamanho.

Podemos também fazer uso de dois atributos simultaneamente, definindo um intervalo no qual uma entrada seja aceita. Este trabalho será o mesmo feito com as tags *<f:validateLongRange>* e *<f:validateDoubleRange>*, a única diferença serão os tipos de dados aceitos.

2.6. Validação customizada

Os validadores padrões são bastante limitados em relação quanto ao que podem realizar. Provavelmente necessitaremos criar determinados códigos de validação em nossa aplicação para verificar estes dados corretamente. Existem os seguintes caminhos para definirmos isso:

a) Estender a classe do componente visual que aceita a entrada de forma que anulamos seu método de validação

b) Criar um método de validação externa

c) Criar nossas implementações validadoras separadas, registrá-las no framework e então inserí-las no componente gráfico

O problema com a primeira opção é que é não portátil. A validação só pode funcionar quando usado um componente visual específico. Nesta lição, estaremos concentrando nas segunda e terceira opções.

2.7. Criando um método de validação externa

Como primeira opção discutida para criarmos a validação personalizada, veremos a criação de um método de validação externa. Este procedimento é semelhante ao criar um método *Aplicattion* que lidará com eventos de ação. Também precisamos criar um método para obter um conjunto de regras dentro de um *JavaBean* administrado pelo *framework* e depois ligar esse método ao componente gráfico apropriado.

2.7.1. Assinatura de método

O método criado, deve ser ajustado conforme as seguintes regras:

- O método deve ser declarado como ***public*** e com um tipo de retorno ***void***
- Não existe nenhuma restrição quanto ao nome do método
- Obter parâmetros na seguinte ordem: objeto *FacesContext*, Componente *UIInput* e um *Object* contendo o valor.
- Deve ser declarado uma proteção de retorno (*throws*) para *ValidatorException*.

Para criar um método de validação personalizada para nosso campo de senha, vista no exemplo anterior, que aceite somente letras e números, a assinatura do método seria similar a seguinte instrução.

```
public void validatePassword(FacesContext ctxt, UIInput component, Object value)
       throws ValidatorException
```

2.7.2. Implementação do método

Na nossa implementação do método, podemos fazer uso de dados fornecidos pelos parâmetros. O objeto *FacesContext* fornece um acesso aos recursos externos, como os objetos de *request* e escopo da sessão, como também a outros objetos dentro do *framework* JSF. O componente visual *UIInput* é a instância do componente de entrada que requer a validação; tendo uma instância deste objeto, obtemos o acesso ao estado do componente. Finalmente, o Object é o valor deste dentro do componente que requer a validação.

O processo de validação é simples. Caso o *framework* não receba qualquer *ValidatorExceptions*, a entrada é aceita. Caso o *framework* receba um *ValidatorException* o processo é interrompido e a página contendo o componente de entrada é novamente reapresentada.

A seguir, temos uma amostra da implementação para este método.

```
public void validatePassword(FacesContext ctxt, UIInput component, Object value)
```

```
    throws ValidatorException {
    if (null == value)
      return;
    if (!(isPasswordValid(value.toString()))) {
      FacesMessage message = new FacesMessage("Input error",
        "Password is not valid");
      throw new ValidatorException(message);
    }
  }
  protected boolean isPasswordValid(String password) {
    ...
```

Uma nova classe no trecho acima é mostrada, a *FacesMessage*. Esta classe modela uma mensagem dentro da JSF. O primeiro parâmetro em seu construtor é um título, enquanto que o segundo são os detalhes da mensagem. Isto é utilizado para indicar a JSF que a mensagem será reconhecida como uma mensagem de erro.

Para ser realmente capaz de exibir uma mensagem de erro gerada, devemos adicionar uma *tag* <h:messages> ou <h:message> na página JSP conforme mostrado a seguir.

```
  ...
  <h:outputLabel for="password">
   <h:outputText value="Password : "/>
  </h:outputLabel>
  <h:inputSecret id="password" value="#{loginPage.password}">
   <f:validateLength minimum="4"/>
  </h:inputSecret>
   <h:message for="password" styleClass="error"/>
  <br/>
  <h:commandButton action="#{loginPage.performLogin}" value="Login"/>
 </h:form>
 ...
```

A *tag* <h:message> só exibe as mensagens para um componente específico, indicado pelo valor do atributo *for*. Também podemos especificar a classe de estilo CSS que será aplicada à mensagem. Uma *tag* <h:messages> exibirá todas as mensagens disponíveis para todos os componentes na página.

2.7.3. Criando uma implementação separada do validador

Em vez de criar nosso método dentro de um *JavaBean*, podemos criar uma classe separada que implementa a interface do validador. Esta interface define um único método, o método para validar, cuja assinatura é idêntica ao método de validação externo, isto é, ***public***, retorno ***void*** e os mesmos parâmetros: *FacesContext*, *UIInput* e *Object*.

Em termos de implementação, não é diferente de uma implementação de um método Validador separado por um método de validação externa. O que diferencia está no modo como são utilizados. Um **método de validação externa** é usado mais para a validação de um código específico por um componente em particular, enquanto uma **implementação separada do validador** é usada para conter o código de validação com um propósito comum e será extensivamente reusado por toda sua aplicação ou mesmo outras aplicações.

2.8. Registrando um componente validador

Para usar um componente validador personalizado, devemos primeiro registrá-lo para ser reconhecido pela JSF. Isto é realizado colocando uma entrada de configuração no arquivo *faces-config.xml*.

Vejamos o seguinte exemplo:

```
<validator>
 <description>A validator for checking the password field</description>
 <validator-id>AlternatingValidator</validator-id>
 <validator-class>jedi.sample.jsf.validator.AlternatingValidator</validator-
class>
```

```
</validator>
```

Como podemos observar, para configurar a entrada para um novo validador basta definir um identificador pelo qual será referenciado na JSF e o nome da classe deve ser totalmente qualificado para implementar a funcionalidade da validação.

2.8.1. Usando um componente validador

Para usar o componente validador registrado, usamos a tag <f:validator> e fornecemos o atributo *validatorId* como identificação do validador, conforme demonstrado a seguir:

```
 ...
 <h:outputLabel for="password">
  <h:outputText value="Password : "/>
 </h:outputLabel>
 <h:inputSecret id="password" value="#{loginPage.password}">
  <f:validator validatorId="AlternatingValidator"/>
 </h:inputSecret>
   <h:message for="password" styleClass="error"/> <br/>
 <h:commandButton action="#{loginPage.performLogin}" value="Login"/>
</h:form>
...
```

2.9. Adicionando atributos para nosso validador

Se observarmos o validador padrão *validateLength* fornecido pela JSF, podemos ver que este contém dois atributos pela qual a operação poderá ser modificada futuramente. De modo semelhante, podemos ter atributos para o validador de modo personalizado.

Para obtermos esta forma, adicionamos uma propriedade dentro de nossa classe para cada um dos atributos, isto sustenta e implementar os métodos padrões *getter* e *setter*. Em seguida, incluir uma entrada para cada atributo dentro da configuração de entrada do validador.

```
package jedi.sample.jsf.validator;

public class AlternatingValidator implements Validator {
  private Integer interval;

  public Integer getInterval() {
    return interval;
  }
  public void setInterval(Integer interval) {
    this.interval = interval;
  }
  public void validate(FacesContext ctxt, UIInput component, Object value)
    throws ValidatorException {
    if (null == value || interval == null)
      return;
    if (!(isPasswordValid(value.toString()))) {
      FacesMessage message = new FacesMessage("Input error",
        "Password is not valid");
      throw new ValidatorException(message);
    }
  }
  protected boolean isPasswordValid(String password) {
...

<validator>
 <description>A validator for checking the password field</description>
 <validator-id>AlternatingValidator</validator-id>
 <validator-class>jedi.sample.jsf.validator.AlternatingValidator</validator-
class>
 <attribute>
  <attribute-name>interval</attribute-name>
  <attribute-class>java.lang.Integer</attribute-class>
```

```
  </attribute>
 </validator>
```

Um valor que pode ser fornecido para este atributo e fazer uso da tag <f:attribute>:

```
  ...
  <h:outputLabel for="password">
   <h:outputText value="Password : "/>
  </h:outputLabel>
  <h:inputSecret id="password" value="#{loginPage.password}">
   <f:validator validatorId=”AlternatingValidator”/>
   <f:attribute name=”interval” value=”5”/>
  </h:inputSecret>
    <h:message for=”password” styleClass=”error”/> <br/>
  <h:commandButton action="#{loginPage.performLogin}" value="Login"/>
 </h:form>
 ...
```

3. Componentes de Conversão

Os componentes de conversão também são componentes importantes do conjunto fornecido pela JSF. Oferecem uma solução para um problema comum dos programadores WEB: como converter os valores informados pelo usuário de sua representação *String* nativa em um formato ou tipo apropriado usado internamente pelo servidor. São bidirecionais, tipos int podem ser usados para mudar a forma como dados internos são expostos para o usuário final. Os componentes de conversão podem fazer isto definindo um método *getAsString*() e *getAsObject*() que são chamados pela JSF na hora apropriada. O método *getAsString*() é chamado para fornecer um tipo *String* dos dados, enquanto que o método getAsObject() é utilizado para converter uma *String* para um um objeto determinado.

Os componentes de conversão são associados para introduzir um determinado tipo, fazendo uso do atributo de conversão embutido em alguns dos componentes de entrada fornecidos pela JSF. Detalharemos isto no exemplo a seguir, que modifica uma *tag inputText* de forma que permita naturalmente a conversão de um objeto tipo *Integer*:

```
<h:inputText converter="Integer" value="#{myFormBean.myIntProperty}"/>
```

Além do componente inputText, o atributo conveter está sustentado pela tag *inputSecret*, *inputHidden*, e o componentes *outputText*.

O objeto do *Integer* é a identificação atribuída a um dos elementos conversores padrões fornecidos pela JSF. Outros componentes de conversão padrões são listados a seguir:

Classe de Conversão	***ID de Conversão***
BigDecimalConverter	BigDecimal
BigIntegerConverter	BigInteger
IntegerConverter	Integer
ShortConverter	Short
ByteConverter	Byte
ShortConverter	Short
CharacterConverter	Character
FloatConverter	Float
DoubleConverter	Double
BooleanConverter	Boolean
DateTimeConverter	DateTime
NumberConverter	Number

Nesse conjunto de componentes de conversão, *DateTimeConverter* e *NumberConverter* merecem atenção especial. Os outros conjuntos de componentes de conversão não são configuráveis. Só podem ser usados como no exemplo acima. Os componentes *DateTimeConverter* e *NumberConverter*, no entanto, podem utilizar *tags* especiais e, através dessas *tags*, podemos manipular os atributos com os quais o desenvolvedor pode customizar seus comportamentos.

3.1. DateTimeConverter

O *DateTimeConverter* pode ser usado para converter dados de entrada em instâncias de java.util.Date. Ele prove atributos com os quais o desenvolvedor pode especificar o formato usado na conversão. É importante notar que o formato especificado é obrigatório SOMENTE quando o usuário fornece um dado de entrada. Isto é, se o usuário não fornecer um dado de entrada no formato especificado dos atributos, um erro de conversão irá ocorrer e o *framework* irá paralisar o processamento de dados.

Estes são os atributos disponíveis:

- **dateStyle** – um dos estilos de data definidos pelo java.text.DateFormat. Os valores possíveis são: *default*, *short*, *long*, *medium* ou *full*.

- **parseLocale** – a localidade a ser usada como referência durante a conversão.
- **timeStyle** – um dos estilos de data definido pelo java.text.DateFormat. Os valores possíveis são: *default*, *short*, *medium*, *long* ou *full*.
- **timeZone** – a time zone utilizada.
- **type** – uma String que define quando a saída será uma data, uma instância do tipo time, ou ambos. Os valores possíveis, são: *date*, *time* e *both*. O valor padrão é date.
- **pattern** – o padrão de formatação a ser usado na conversão. Se um valor para esse atributo for definido, o sistema irá ignorar qualquer valor para *dateStyle*, *timeStyle* e *type*.

Esses atributos são disponibilizados para o desenvolvedor através da *tag* <f:convertDateTime>.

Vamos fazer um pequeno exemplo de utilização do *DateTimeConverter*. Primeiro, crie uma aplicação WEB em branco pronta para JSF, como nas instruções da lição anterior. Então, vamos começar criando um *JavaBean* que irá guardar a data que iremos inserir:

```
import java.util.Date;

public class DateBean {
  private Date date;

  public Date getDate() {
    return date;
  }
  public void setDate(Date date) {
    this.date = date;
  }
}
```

Então, vamos adicionar suas configurações iniciais no arquivo *faces-config.xml*:

```
<managed-bean>
 <description>
  Modelo de suporte para o exemplo de conversão de data
 </description>
 <managed-bean-name>dateBean</managed-bean-name>
 <managed-bean-class>jedi.sample.DateBean</managed-bean-class>
 <managed-bean-scope>request</managed-bean-scope>
 <managed-property>
  <property-name>date</property-name>
  <null-value/>
 </managed-property>
</managed-bean>
```

Finalmente, criamos o arquivo JSP que irá gerar a *view*:

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

<f:view>
 <h:form id="testForm">
  <h:outputLabel for="dateField">Date : </h:outputLabel>
  <h:inputText id="dateField" value="#{dateBean.date}" required="true">
   <f:convertDateTime pattern="M/d/yy"/>
  </h:inputText> <br/>
  <h:message for="dateField"/> <br/>
  <h:commandButton id="button" value="Press me!!!"/>
 </h:form>
</f:view>
```

Alguns comentários sobre a página JSP de exemplo:

- Para nosso *DateTimeConverter*, especificamos um padrão de M/d/yy. Como o componente de conversão obriga esse padrão para o usuário, o usuário deve inserir uma data como 1/13/06, antes de poder continuar. Para outros possíveis padrões, procure na

documentação API por *java.text.SimpleDateFormat*.

- Adicionamos um atributo *required* com valor *true* para o campo de entrada. Desta forma asseguramos que o usuário não pode enviar dados de entrada em branco.
- Utilizamos uma *tag* <h:message> associada ao campo de entrada *Date*. Isto assegura que qualquer mensagem gerada pelo campo de entrada (como aqueles gerados por erros de conversão) serão mostrados.
- O atributo *action* do *commandButton* foi deliberadamente deixado em branco, por isso podemos nos focar apenas nas mensagens geradas pelo componente de conversão.

Carregando a aplicação web em um servidor compatível (no nosso caso uma instância do AppServer 8.1) e acessando a página que acabamos de criar, irá resultará na seguinte tela, mostrada abaixo:

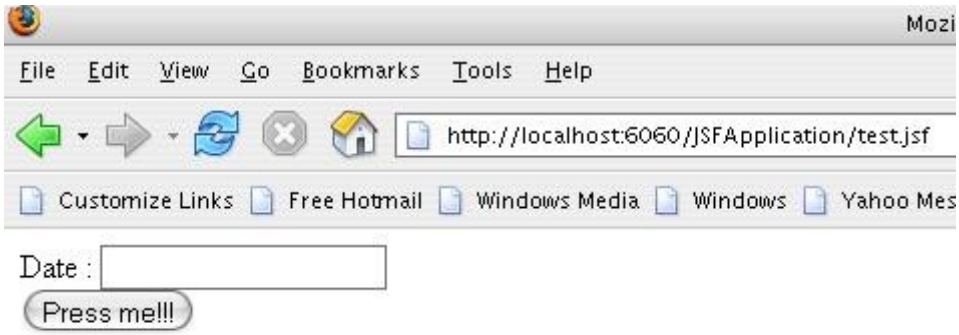

Figura 1: Título do JSP de exemplo

Testando o dado de entrada January 13, 2006 irá resultar no seguinte:

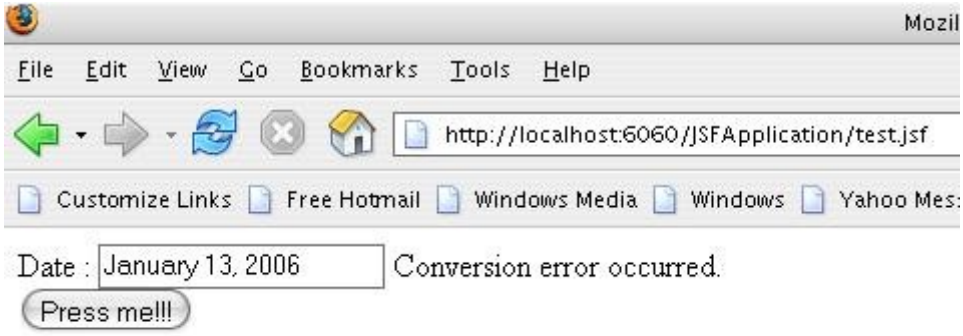

Figura 2: Resultado do JSP de exemplo utilizando um componente de conversão

Informando uma data no formato apropriado irá resultar em nenhuma mudança na nossa página. Isso indicará que o dado foi convertido com sucesso e armazenado em nosso *JavaBean*.

E se não utilizássemos um componente de conversão?

Para ilustrar o benefício do componente de conversão, modificaremos nossa página JSP para retirá-lo:

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

<f:view>
 <h:form id="testForm">
  <h:outputLabel for="dateField">Date : </h:outputLabel>
  <h:inputText id="dateField" value="#{dateBean.date}" required="true"/>
  <br/>
  <h:message for="dateField"/> <br/>
  <h:commandButton id="button" value="Press me!!!"/>
 </h:form>
</f:view>
```

Agora, entrando uma data com qualquer formato irá causar um erro:

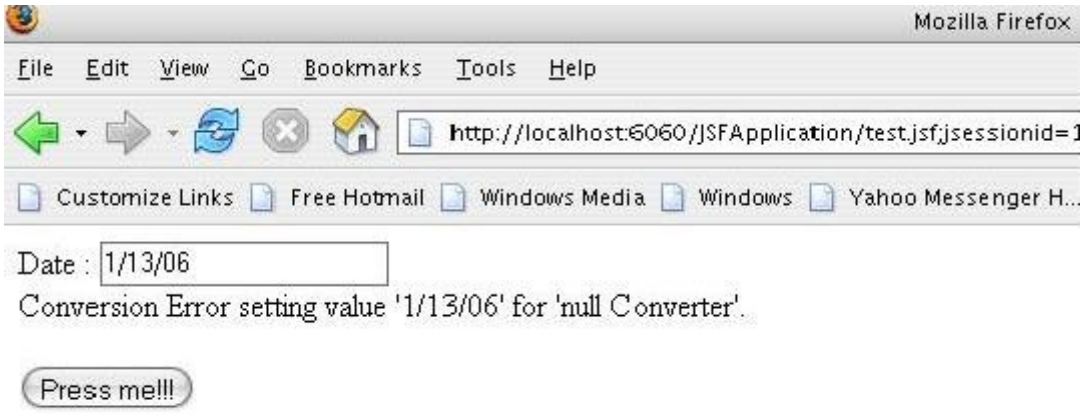

Figura 3: Resultado da JSP de exemplo sem o componente de conversão

Isso ocorre porque a propriedade associada ao campo de entrada é um objeto do tipo *Date*. O *framework* não pode mudar automaticamente a representação do objeto String do campo de entrada para um objeto tipo *Date*, então procura por um componente de conversão que irá tratar isso. Como nenhum componente de conversão é encontrado, é mostrado um erro null (nulo).

3.2. NumberConverter

Outro componente de conversão especial que iremos ver é o *NumberConverter*. Esse componente de conversão é usado para converter números ou obrigar formatações especiais. Os seguintes atributos são usados para podermos controlar o comportamento desse componente de conversão:

- **currencyCode** – código monetário tipo ISO 4217 a ser utilizado ao formatar.
- **currencySymbol** – o símbolo monetário a ser usado ao formatar.
- **groupingUsed** – indica se o valor utilizará separadores de grupo (por exemplo, ponto depois de cada 3 dígitos).
- **integerOnly** – indica que apenas a parte inteira de um número deve ser mostrada. Isso significa que a parte decimal será truncada antes do dado ser armazenado em uma propriedade escondida (*bound*).
- **locale** – a localidade usada para referência de formação.
- **maxIntegerDigits** – o número máximo de dígitos mostrados ou usados na parte inteira do número.
- **maxFractionDigits** – o número máximo de dígitos mostrados ou usados na parte decimal do número.
- **minFractionDigits** – o número mínimo de dígitos mostrados ou usados na parte decimal do número.
- **minIntegerDigits** – o número mínimo de dígitos mostrados ou usados na parte inteira do número.
- **pattern** – o padrão a ser utilizado quando formatamos ou mostramos o número. Para mais detalhes sobre os padrões permitidos, verifique no JavaDoc por *java.text.NumberFormat*.
- **type** – indica quando o número a ser convertido deve ser tratado como moeda, percentual ou número. Para mais detalhes, verifique no JavaDoc por *java.text.NumberFormat*.

É importante ressaltar que ao utilizar *NumberConverter*, que, como no *DateTimeConverter*, qualquer padrão ou símbolo indicado nos atributos são tratados como regras obrigados para o usuário. Se os dados de entrada do usuário não seguirem o padrão ou não apresentarem os símbolos solicitados, um erro de conversão irá ocorrer e o processamento dos dados será paralisado.

Um dos problemas com os componentes de conversão padrões fornecidos com JSF é que, eles apenas executam conversões padrão ou obrigarão padrões de entrada para o usuário. Por exemplo, o componente *NumberConverter*, se especificarmos um valor 'P' no atributo *currencySymbol*, qualquer usuário que não entrar 'P' junto com números encontrará um erro.

Felizmente, JSF fornece uma maneira fácil para desenvolvedores criarem seus componentes de conversão customizados a serem utilizados no *framework*.

3.3. Componentes de Conversão Customizados

Componentes de conversão customizados podem ser criados através da criação de uma classe que implementa a interface Converter. Essa interface define dois métodos:

- public Object getAsObject(FacesContext ctxt, UIComponent component, String input)
- public Object getAsString(FacesContext ctxt, UIComponent component, Object source)

Consideremos o seguinte cenário:

- Uma página com um formulário que requer dados de entrada do usuário com valores monetários. Gostaríamos de armazenar esses dados como objetos Double para facilitar o processamento mais tarde. No entanto, o campo de entrada deve ser flexível o suficiente para tratar apenas números (ex. 2000), ou incluir grupos de símbolos de grupos (2.000). Outra particularidade é como a página irá reverter o controle para si, os dados deverão ser representados contendo símbolos de agrupamentos.

Alguém poderá pensar que, como essa operação envolve números e símbolos de grupos, poderíamos simplesmente utilizar o *NumberConverter* fornecido com o *framework*. No entanto, o *NumberConverter* obriga qualquer padrão que fornecemos para ele, então ao solicitar o tratamento de valores monetários com símbolos de grupos, ele poderia apenas tratar valores monetários com símbolos de grupos. Dados de entradas numéricos resultariam em erros de conversão. Também não é possível ter múltiplos componente de conversão para um simples componente de entrada de dados. Então não podemos simplesmente utilizar duas instâncias diferentes de *NumberConverter*. É com esse cenário em mente que os exemplos para criação de componentes de conversão customizados foram criados.

3.3.1. Método *getAsObject*

Esse método é chamado pelo *framework* para o componente de conversão associado quando precisa converter os dados de entrada do usuário de sua representação textual em outro tipo de objeto. O conceito na implementação desse método é simples: atua na operação de processamento do argumento String fornecido e retorna um *Object* que irá representá-lo no servidor.

Considerando o cenário descrito, nossa tarefa é converter uma entrada em tipo *String* em um objeto do tipo *Double*, não importando se o número é apenas numérico ou com símbolos de grupos.

A implementação para esse método é mostrada a seguir:

```
public Object getAsObject(FacesContext facesContext, UIComponent uIComponent,
  String str) {
  Double doubleObject = null;
  try {
    doubleObject = Double.valueOf(str);
  } catch (NumberFormatException nfe) {
    // Se a exceção ocorrer
    // uma causa provável é a existência de símbolos de agrupamento

    // Recuperar uma instância do objeto NumberFormat
    // e a faz reconhecer símbolos agrupados
    NumberFormat formatter = NumberFormat.getNumberInstance();
    formatter.setGroupingUsed(true);

    // Utilizar o objeto NumberFormat para recuperar o valor numérico do
    // dado de entrada
    try {
      Number number = formatter.parse(str);
      if (number.doubleValue() <= Long.MAX_VALUE) {
        Long value = (Long) number;
```

```
          doubleObject = new Double(value.doubleValue());
        } else {
          doubleObject = (Double)number;
        }
      } catch (ParseException pe) {
        // Se o código atingir esse ponto, nenhum dos formatos suportados
        // foram seguidos.
        // Criar uma mensagem de erro e a mostra para o usuário através de
        // um objeto exception
        FacesMessage message = new FacesMessage("Formato não Suportado",
          "Este campo suporta apenas números (5000), " +
          "ou números com vírgula (5,000)");
        throw new ConverterException(message);
      }
    }
    return doubleObject;
  }
```

3.3.2. Método getAsString

Esse método faz conversões bidirecionais. Ele dita a representação em String do dado interno relevante. O conceito por trás da implementação desse objeto é mais simples do que a do método *getAsObject* acima. Ela mostra o valor armazenado no objeto tipo *Double* como um número com símbolos de grupos. Essa implementação é mais simples porque sabemos com clareza o formato exato do nosso dado de entrada.

Aqui está a implementação:

```
public String getAsString(FacesContext context, UIComponent component,
  Object source) {
  Double doubleValue = (Double)source;
  NumberFormat formatter = NumberFormat.getNumberInstance();
  formatter.setGroupingUsed(true);
  return formatter.format(doubleValue);
}
```

3.4. Usando um componente de conversão customizado

Depois de termos criado o componente de conversão customizado, teremos que configurar a JSF para que, então, reconheça sua existência. Isso é feito, adicionando-se uma entrada de configuração no arquivo f*aces-config.xml*, que define uma estrutura rígida para seus elementos. Entradas de configuração para componentes de conversão aparecem depois de todas as entradas de validadores, mas antes das entradas dos *JavaBeans* gerenciados.

Abaixo temos a entrada de configuração para o componente de conversão que acabamos de criar.

```
<converter>
 <description>Customizado para aceitar dados monetários</description>
 <converter-id>myConverter</converter-id>
 <converter-class>jedi.sample.MyCustomConverter</converter-class>
</converter>
```

Depois de termos criado uma entrada de configuração para o componente de conversão, a utilizaremos para nosso elemento de entrada especificando o id do componente de conversão em uma *tag* <f:converter>, da seguinte forma:

```
<h:inputText id="amount" value="#{numberBean.amount}">
 <f:converter converterId="myConverter"/>
</h:inputText>
```

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.