Desenvolvimento para Servidores-II

Conversores e Validadores

Neste tópico abordaremos a conversão e validação de dados de entrada através de tags JSF

Prof. Ciro Cirne Trindade

Visão geral do processo de conversão e validação (1/3)

- Primeiro o usuário preenche os campos de um formulário
- Quando ele clica no botão de submissão, o browser manda os dados para o servidor usando uma requisição HTTP
- Estes valores são chamados de **valores de requisição** (*request value*)

Visão geral do processo de conversão e validação (2/3)

- Todos os valores de requisição são **strings**, enquanto o servidor lida com tipos arbitrários, como `int`, `Date`, etc.
- Um processo de **conversão** transforma strings nesses tipos
- Os valores convertidos não são transmitidos imediatamente para o *managed bean* responsável pela lógica do negócio

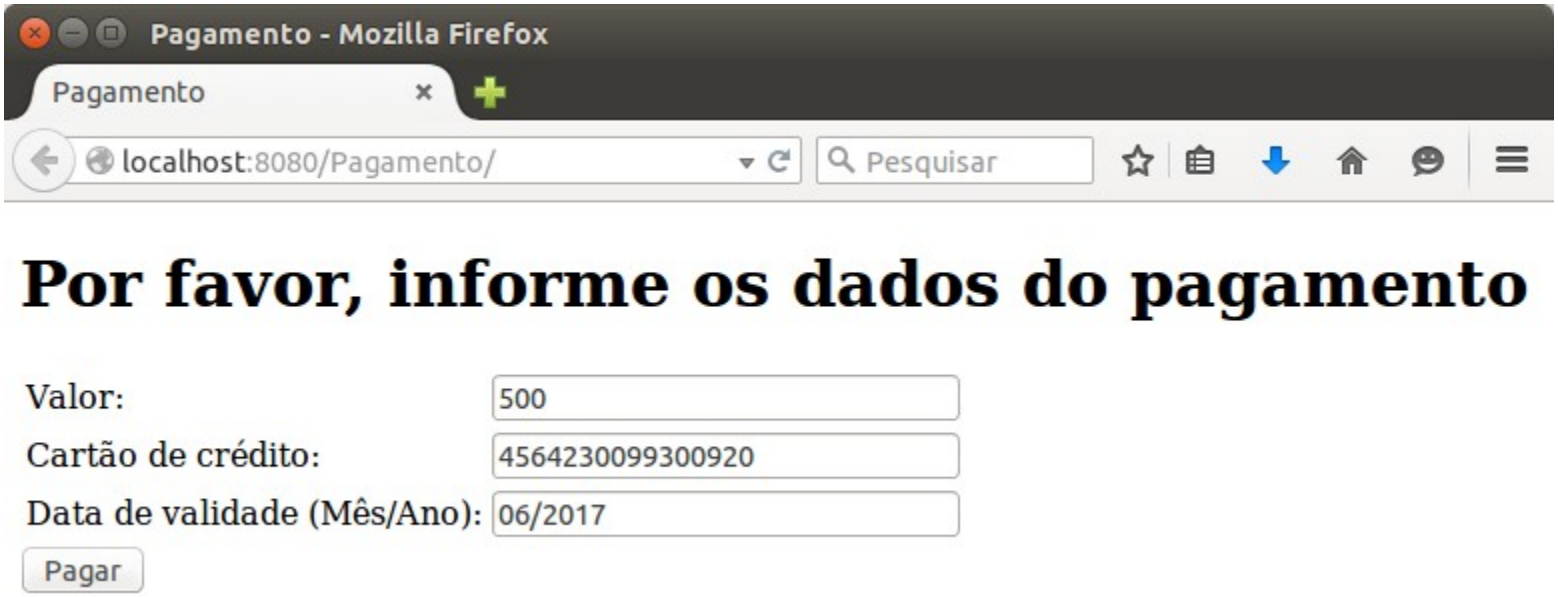
Visão geral do processo de conversão e validação (3/3)

- Os valores convertidos são armazenados em objetos como **valores locais**
- Após a conversão os valores locais são **validados**
  - O desenvolvedor pode especificar condições de validação
- Depois que todos os valores locais são validados, eles são armazenados no *managed bean*

Conversores de datas e números

- As tags `<f:convertNumber>` e `<f:convertDateTime>` podem ser usadas para definir como números e datas, respectivamente, devem ser convertidos de string para esses tipos
- Vamos construir uma aplicação para processar o pagamento de uma compra

Exemplo: informações de pagamento (1/5)



Managed bean **Pagamento** (1/2)

```
package bean;

import java.util.Date;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Pagamento {
    private Double valor;
    private String cartao;
    private Date validade;

    public Pagamento() { }
    public Double getValor() {
        return valor;
    }
    public void setValor(Double valor) {
        this.valor = valor;
    }
```

Managed bean **Pagamento** (2/2)

```
    public String getCartao() {
        return cartao;
    }

    public void setCartao(String cartao) {
        this.cartao = cartao;
    }

    public Date getValidade() {
        return validade;
    }

    public void setValidade(Date validade) {
        this.validade = validade;
    }

}
```

index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Pagamento</title>
    </h:head>
    <h:body>
        <h1>Por favor, informe os dados do pagamento</h1>
        <h:form>
            <h:panelGrid columns="2">
                Valor:
                <h:inputText value="#{pagamento.valor}"/>
                Cartão de crédito:
                <h:inputText value="#{pagamento.cartao}"/>
                Data de validade (Mês/Ano):
                <h:inputText value="#{pagamento.validade}"/>

            </h:panelGrid>
            <h:commandButton value="Pagar" action="/pagamento"/>
        </h:form>
    </h:body>
</html>
```

Exemplo: informações de pagamento (2/5)

- Vamos associar um conversor ao campo para o valor e configurar o valor com pelo menos 2 casas decimais

```
<h:inputText value="#{pagamento.valor}">
      <f:convertNumber minFractionDigits="2"/>
</h:inputText>
```

Exemplo: informações de pagamento (3/5)

- No segundo campo (cartão de crédito) não vamos usar um conversor
- O terceiro campo (data de validade) iremos usar um conversor para data no formato MM/yyyy

```
<h:inputText value="#{pagamento.validade}">
        <f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

Exemplo: informações de pagamento (4/5)

- Para exibir o valor numa página `pagamento.xhtml` vamos usar um conversor para moeda (*currency*)

```
<h:outputText value="#{pagamento.valor}">
        <f:convertNumber type="currency"
                         locale="pt-br"/>
</h:outputText>
```

Exemplo: informações de pagamento (5/5)

- Para exibir a data de validade vamos usar o mesmo conversor para data no formato MM/yyyy

```
<h:outputText value="#{pagamento.validade}">
        <f:convertDateTime pattern="MM/yyyy"/>
</h:outputText>
```

Definindo a localização da aplicação (*locale*)

- A forma mais simples de definir a localização de sua aplicação JSF é através do faces-config.xml:

```
<faces-config>
  <application>
    <locale-config>
      <default-locale>pt-br</default-locale>
    </locale-config>
  </application>
</faces-config>
```

Exibindo mensagens de erro
(1/2)

- Quando um erro de conversão ocorre, o JSF exibe novamente a página com mensagens de erro no final da página
- Para exibir uma mensagem de erro próximo ao elemento que causou o erro, usamos a tag `<h:message>`
- A tag `<h:messages>` pode ser usada para exibir todas as mensagens de erro em um único local

Exibindo mensagens de erro
(2/2)

- Forneça um ID para o componente e referencie esse ID na tag `<h:message>`
- Também forneça o atributo `label` do componente para que ele seja exibido na mensagem de erro:

```
<h:inputText id="valor" label="valor"
                value="#{pagamento.valor}">
      <f:convertNumber minFractionDigits="2"/>
</h:inputText>
<h:message for="valor" style="color:red"/>
```

Validadores predefinidos

- O JSF possui alguns validadores predefinidos, tais como:
  - `f:validateLongRange`: verifica o valor de um componente numérico está num intervalo específico
  - `f:validateLength`: checa se o comprimento de um componente está num intervalo específico

Exemplos de validadores predefinidos

```
<h:inputText id="valor" label="Valor"
             value="#{pagamento.valor}">
    <f:convertNumber minFractionDigits="2"/>
    <f:validateLongRange minimum="10"
                         maximum="100000"/>
</h:inputText>
<h:message for="valor" styleClass="erro"/>
...
<h:inputText id="cartao"
             label="Cartão de Crédito"
             value="#{pagamento.cartaoCredito}">
     <f:validateLength minimum="16"/>
</h:inputText>
<h:message for="cartao" styleClass="erro"/>
```

CSS

- Para usar estilos CSS definidos em um arquivo, é necessário que este arquivo esteja em uma subpasta da pasta **`resources`**
- Para utilizar os estilos na página, usamos a tag `<h:outputStylesheet>`
- Exemplo:
  - `<h:outputStylesheet library="css" name="styles.css"/>`

Subpasta de **`resources`** aonde está o arquivo CSS

styles.css

```
.erro {
    font-style: italic;
    color: red;
}
```

Definindo uma mensagem de erro de conversão personalizada

- Atributo **converterMessage** pode ser usado para definir uma mensagem de **erro de conversão** customizada
- Exemplo:
  - ```
    <h:inputText id="valor"
    label="valor"
    value="#{pagamento.valor}"
    converterMessage="O valor da conta
    de ser informado em Reais"
    required="true">
    ```
    ```

Definindo uma mensagem de erro de validação personalizada

- Atributo **`validatorMessage`** pode ser usado para definir uma mensagem de **erro de validação** customizada
- Exemplo:
  - ```
    <h:inputText id="cartao"
    label="Cartão de crédito"
    value="#{pagamento.cartao}"
    validatorMessage="O cartão de
    crédito tem 16 números"
    required="true">
    ```
    ```

Arquivo de mensagens (1/2)

- Para facilitar a internacionalização da aplicação, utiliza-se um arquivo de propriedades para definir mensagens que são exibidas na interface
- faces-config.xml

```
<application>
    <resource-bundle>
        <base-name>messages.mensagens</base-name>
        <var>msgs</var>
    </resource-bundle>
</application>
```

`messages` é o nome do pacote aonde o arquivo de mensagens reside

Define o nome que será usado para referenciar o arquivo de mensagens

Arquivo de mensagens (2/2)

- Um arquivo de propriedades é um arquivo do tipo texto que possui um conjunto de linhas com pares `propriedade=valor`
- Exemplo:
  - mensagens.properties

```
obrigatorio=Campo obrigatório
```

- Utilizando a mensagem

```
requiredMessage="#{msgs.obrigatorio}"
```

"Bypassando" a validação

- Erros de validação (e conversão) forçam o recarregamento da página
- Este comportamento pode não ser o desejado em certas ações, por exemplo, um botão "Cancelar" numa página que contém campos obrigatórios
- Para contornar a validação existe um atributo booleano chamado **`immediate`**
- Por exemplo:
  - `<h:commandButton value="Cancelar" action="/cancelamento" **immediate="true"**/>`

Classes de conversão (1/4)

- Um *conversor* é uma classe que faz a conversão entre strings e objetos
- Um conversor deve implementar a interface `Converter`, que possui 2 métodos:
  - `Object getAsObject(FacesContext context, UIComponent component, String value)`

    Usado para converter uma String em um objeto
  - `String getAsString(FacesContext context, UIComponent component, Object value)`

    Usado quando um objeto em uma String

Classes de Conversão (2/4)

- Para indicar um erro de conversão esses métodos devem gerar uma exceção do tipo `ConverterException`
- Exemplo:

```
if (conversão falhar) {
    FacesMessage message = ...;
    message.setSeverity(
            FacesMessage.SEVERITY_ERROR);
    throw new ConverterException(message);
}
```

Classes de Conversão (3/4)

- A partir do JSF 2.0 é possível registrar uma classe como um conversor através da anotação `@FacesConverter`
- Exemplo:

```
@FacesConverter("converter.Cartao")
public class CartaoConverter
                    implements Converter { … }
```

Identificador do conversor

Ou

```
@FacesConverter(forClass=Cartao.class)
public class CartaoConverter
                    implements Converter { … }
```

Conversor para uma classe

Classes de Conversão (4/4)

- Caso o conversor seja genérico, podemos usar a tag `<f:converter>` para especificá-lo
- Exemplo:

```
<h:inputText value="#{pagamento.cartao}">
   <f:converter converterId="converter.Cartao"/>
</h:inputText>
```

- Ou mais sucintamente usando o atributo `converter:`

```
<h:inputText value="#{pagamento.cartao}"
             converter="converter.Cartao">
```

Cartao.java

```
package model;

import java.io.Serializable;

public class Cartao implements Serializable {
    private String numero;

    public Cartao(String numero) {
        this.numero = numero;
    }

    @Override
    public String toString() {
        return numero;
    }

}
```

Mudar o tipo do atributo `cartao` na classe `Pagamento` para `Cartao`

CartaoConverter.java (1/3)

```
package converter;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;
import model.Cartao;

@FacesConverter(forClass=Cartao.class)
public class CartaoConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context, UIComponent
           component, String value) throws ConverterException {
        StringBuilder convertida =
                      new StringBuilder(value.toString());
        boolean invalido = false;
        char caractereInvalido = '\0';
```

CartaoConverter.java (2/3)

```
        // continuação do método getAsObject
        for (int i = 0; i < convertida.length() && !invalido; i++) {
            char ch = convertida.charAt(i);
            if (!Character.isDigit(ch)) {
                if (ch == ' ' || ch == '.') {
                    convertida.deleteCharAt(i);
                }
                else {
                    invalido = true;
                    caractereInvalido = ch;
                }
            }
        }
        if (invalido) {
            FacesMessage message = new FacesMessage(
       "Erro de conversão no número do cartão: caractere inválido ("
       + caractereInvalido + ")");
            message.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ConverterException(message);
        }
        return new Cartao(convertida.toString());
    }
```

CartaoConverter.java (3/3)

```
    @Override
    public String getAsString(FacesContext context, UIComponent
           component, Object value) throws ConverterException {
        // retorna o cartão no formato XXXX XXXX XXXX XXXX
        int[] limites = { 4, 8, 12 };
        int fim, inicio = 0;
        StringBuilder resultado = new StringBuilder();
        String valor = value.toString();
        for (int i = 0; i < limites.length &&
                        limites[i] < valor.length(); i++) {
            fim = limites[i];
            resultado.append(valor.substring(inicio, fim));
            resultado.append(" ");
            inicio = fim;
        }
        resultado.append(valor.substring(inicio));
        return resultado.toString();
    }

} // fim da classe CartaoConverter
```

Recuperando mensagens do arquivo de mensagens em uma classe (1/2)

- No método `getAsObject` da classe `CartaoConverter`, definimos fixo no código uma mensagem de erro

```
FacesMessage message = new FacesMessage(
    "Erro de conversão no número do cartão: "
    + "caractere inválido ("
    + caractereInvalido + ")" );
```

- Isso impede a internacionalização
- Vamos implementar um método que permita recuperar mensagens do arquivo de mensagens

Messages.java

```
package util;

import java.util.MissingResourceException;
import java.util.ResourceBundle;
import javax.faces.context.FacesContext;

public class Messages {

    public static String getString(String resourceBundleName,
                        String resourceBundleKey, Object ... params)
            throws MissingResourceException {
        FacesContext facesContext =
              FacesContext.getCurrentInstance();
        ResourceBundle bundle = facesContext.getApplication().
              getResourceBundle(facesContext, resourceBundleName);

        String msg = bundle.getString(resourceBundleKey);
        for (Object param : params) {
            msg += param;
        }
        return msg;
    }
}
```

Recuperando mensagens do arquivo de mensagens em uma classe (2/2)

- Agora é possível recuperar uma mensagem usando o método estático `Messages.getString`

```
String msg = Messages.getString("msgs",
                 "erroConversaoCartao",
                 " (" + caractereInvalido + ")");
FacesMessage message = new FacesMessage(msg);
```

Classes de validação (1/3)

- A implementação de classes de validação é semelhante ao processo de implementação de conversores
- A classe de validação deve implementar a interface `Validator`
- A interface `Validator` define apenas um método:
  - `void validate(FacesContext context, UIComponent component, Object value)`

Classes de Validação (2/3)

- Para sinalizar um erro de validação o método `validate` deve lançar uma exceção do tipo `ValidatorException`
- Por exemplo:

```
if (!valido(value.toString())) {
    FacesMessage msg =
         new FacesMessage(
            "O cartão deve possuir 16 dígitos");
    msg.setSeverity(
         FacesMessage.SEVERITY_ERROR);
    throw new ValidatorException(msg);

}
```

Classes de Validação (3/3)

- Assim como nas classes de conversão, é possível registrar uma classe de validação através de uma anotação, neste caso `@FacesValidator`
- Se for necessário especificar o validador, utiliza-se a tag `<f:validator validatorId="..."/>` ou o atributo `validator`

Validação através de método de um *bean* (1/2)

- É possível invocar um método de um *managed bean* para fazer a validação de um componente
- Para isso, utiliza-se o atributo `validator` do componente para referenciar o método do *bean*
- Por exemplo:

```
<h:inputText id="cartao"
             label="Cartão de Crédito"
             value="#{pagamento.cartaoCredito}"
             validator="#{pagamento.validaCartao}"/>
```

Validação através de método de um *bean* (2/2)

- A classe deve possuir um método com a mesma assinatura do método `validator` definido na interface `Validator`
- Por exemplo:

```
public class Pagamento {
    ...
    public void validaCartao(
                FacesContext context,
                UIComponent component,
                Object value)
            throws ValidatorException {
        ...
    }
}
```

Beans de validação

- O JSF 2.0 incorporou um framework de validação que permite especificar restrições de validação através da anotações no próprio *managed bean*
- As validações são associadas a atributos ou métodos *getters* da classe
- Por exemplo:

```
public class Pagamento {
    @Min(10) @Max(100000) private Double valor;
    @Size(min=15, max=16) private String cartao;
    @Future private Date validade;
    ...
```

Referências

- ORACLE Corporation. *The Java EE 7 Tutorial*. Disponível em: https://docs.oracle.com/javaee/7/JEETT.pdf, 2014.
- GEARY, David; HORSTMANN, Cay. *Core JavaServer Faces*. 3. ed., Prentice Hall, 2010.