01

Conversão e validação de dados

Transcrição

##Downloads Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (http://s3.amazonaws.com/caelum-online-public/JSF/livraria-capitulo5.zip). Só baixe este arquivo se **não tiver feito os exercícios dos capítulos anteriores**.

##Trabalhando com Datas

Abrindo a página **livro.xhtml** temos um `input` apontando para o atributo `dataLancamento` do objeto livro em **livroBean** utilizando a *expression language* `#{livroBean.livro.dataLancamento}` . Abrindo esta classe, vemos que o atributo é do tipo `String` , mas queremos mudá-lo para **`Calendar`** , pois a manipulação de datas em `String` nos causaria problemas.

Alterando o tipo de `String` para **`Calendar`** , queremos gravar no banco apenas a data, ignorando os segundos. Para isso, adicionamos a anotação JPA **`@Temporal`** que recebe como parâmetro a `enum` **`TemporalType.Date`** . Iniciaremos o atributo com a data atual através de `Calendar.getInstance()` .

Precisamos criar novos *getters* e *setters*, sendo assim, apagaremos os anteriores e criaremos os novos usando o próprio Eclipse. Pronto, já temos o nosso modelo preparado.

```
package br.com.caelum.livraria.modelo;

//outros imports omitidos

import java.util.Calendar;

import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Livro implements Serializable{

    //outros atributos omitidos

    @Temporal(TemporalType.DATE)
    private Calendar dataLancamento = Calendar.getInstance();

    //outros atributos omitidos

    public Calendar getDataLancamento() {
        return dataLancamento;
    }

    public void setDataLancamento(Calendar dataLancamento) {
        this.dataLancamento = dataLancamento;
    }

    //outros getters e setters omitidos
}
```

##Modificando o banco de dados

Como alteramos o tipo do atributo em nosso modelo, precisaremos recriar o nosso banco para refletir a alteração. O primeiro passo é alterar a classe **PopulaBanco**. Já temos um método privado `parseData` que realiza as conversões necessárias. Vamos agora modificar a linha de inserção de data no método `geraLivro(..)` para:

```
livro.setDataLancamento(parseData(data));
```

Vamos ao terminal nos conectar ao **mysql**. Apagaremos o banco com o comando `drop database livrariadb` , pois ele já não reflete a estrutura atual do nosso modelo. Com o banco apagado, criaremos novamente o banco **livrariadb** para que quando nossa aplicação entre no ar, o JPA crie novamente as tabelas baseado em nosso modelo. `mysql -u root drop database livrariadb; create database livrariadb;` Vamos agora executar a classe `PopulaBanco` .

##Exibindo mensagens de erro através do h:messages Repare que ao visualizar a página pela primeira vez temos a data de lançamento com um valor não esperado. Mesmo digitando alguns campos e salvando nada acontece. Voltando ao Eclipse, vemos em seu *console* que uma mensagem de aviso apareceu, mas não foi exibida para o usuário na página. Isto acontece porque houve um erro de validação com o campo data e não havia nenhum componente especializado para exibir esta mensagem automaticamente.

Utilizaremos o componente `<h:messages>` para exibir todas as mensagens geradas automaticamente pelo JSF. Outra alternativa seria utilizar `<h:message>` , mais seletivo, pois mostra apenas as mensagens de um componente específico.

```
<h:body>
    ##Novo Livro
    <h:form>
        <h:messages/>
        <fieldset>
```

Agora, ao tentarmos salvar novamente, a mensagem de erro é exibida na própria página. O JSF por padrão mostra a mensagem iniciando pelo `id` do formulário. Como não definimos nenhum `id` para ele, o JSF coloca automaticamente um para nós. Quando digitamos um valor válido, ainda recebemos uma mensagem de erro diferente, a respeito de um **null converter**.

Isto acontece, porque o JSF não pode prever qual formato de data desejamos utilizar. Por este motivo, a especificação já traz uma série de conversores, inclusive o de data.

##Aplicando os conversores do JSF Abrindo o formulário, adicionaremos para o `inputText` da data, o conversor `<f:convertDateTime pattern="dd//MM/yyyy" timeZone="America/Sao_Paulo">` . Nele temos um formato padrão de data através do atributo *pattern* e *timeZone*. O conversor só sabe lidar com objetos do tipo `java.util.Date` , por isso faremos o *bind* para `#{livroBean.livro.dataLancamento.time}` . Pelo padrão JavaBean, `time` corresponde ao método `getTime()` de `Calendar` , que retorna um `java.util.Date` . Este conversor também pode ser aplicado para componentes de saída, como o `<h:outputText>` .

```
<h:inputText id="dataLancamento" value="#{livroBean.livro.dataLancamento.time}">
    <f:convertDateTime pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo" />
</h:inputText>
```

E no componente `h:outputText` :

```
<h:outputText value="#{livro.dataLancamento.time}" >
    <f:convertDateTime pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo" />
</h:outputText>
```

Vamos visualizar a página **livro.xhtml** mais uma vez e inserir dados. Perceba que nenhum erro foi lançado e os dados foram gravados corretamente.

##Componentes de validação do JSF Não faz sentido gravarmos um livro sem título, sendo assim podemos modificar o comportamento do `inputText` para que isso não ocorra. Basta adicionarmos o atributo required="true". Com a alteração feita e testando o formulário no navegador, o JSF não permite a gravação das informações e ainda notifica o usuário através de uma mensagem padrão que o campo é necessário.

Podemos definir nossa própria mensagem no próprio `inputText` através do atributo requiredMessage="", em nosso caso, definiremos **"Título obrigatório"**. Visualizando novamente a página, vemos a nossa mensagem logo após salvarmos o autor.

```
<h:inputText id="titulo" value="#{livroBean.livro.titulo}"
        required="true" requiredMessage="Título obrigatório" />
```

Podemos ir além, definindo também um tamanho máximo para o campo título através de um validador padrão do JSF. Este validador é o **`<f:validateLength>`** . É através do atributo **`maximum`** que definimos o valor máximo de 40 caracteres.

Após salvar a página, já podemos testar o resultado sem precisarmos reiniciar o servidor. O JSF mais uma vez nos dá uma mensagem padrão de erro. Também podemos personalizar esta mensagem só que agora através do atributo **`validatorMessage=""`** . Visualizando mais uma vez vemos que nossa mensagem é exibida.

```
<h:inputText id="titulo" value="#{livroBean.livro.titulo}"
            required="true" requiredMessage="Título obrigatório"
            validatorMessage="Título não pode ser superior a 40">
        <f:validateLength maximum="40" />
</h:inputText>
```

##Criando validadores personalizados Como acabamos de ver, o JSF possui alguns *converters* e validadores padrões que podemos utilizar, mas nem sempre eles são suficientes. Por exemplo, queremos validar o `inputText` do ISBN, aceitando apenas números que comecem com o dígito 1 (um). O JSF tem uma solução para isso, que é a criação de **validadores personalizados**.

Nosso validador será um método no *managedBean* **livroBean**. O nome do método será **`comecaComDigitoUm`** . Ele recebe um **`FacesContext`** , um objeto que permite obter informações da *view* processada no momento. O segundo parâmetro é o **`UIComponent`** , o componente da *view* que está sendo validado. Por último, temos um **`objeto`** que é o *valor digitado pelo usuário*. Este método precisa lançar um `ValidatorException` , exceção que sinalizará para o JSF que algo saiu errado.

```
@ManagedBean
@ViewScoped
public class LivroBean implements Serializable {

    //trechos do código omitidos
```

```
    public void comecaComDigitoUm(FacesContext fc, UIComponent component, Object value) throws \

        //aqui faremos a validação
    }
}
```

Sua implementação consiste em testar se o valor digitado começa com um, caso contrário, será lançada a exceção. A exceção recebe em seu construtor a mensagem com o texto que queremos mostrar para o usuário.

```
public void comecaComDigitoUm(FacesContext fc, UIComponent component, Object value) throws Vali

    String valor = value.toString();
    if (!valor.startsWith("1")) {
        throw new ValidatorException(new FacesMessage("Deveria começar com 1"));
    }
}
```

Falta ainda associar o nosso validador personalizado ao `inputText` do ISBN. Isto é feito através do atributo **`validator`**, que apontará para **`#{livroBean.comecaComDigitoUm}`**.

```
<h:inputText id="isbn" value="#{livroBean.livro.isbn}" validator="#{livroBean.comecaComDigitoUm
```

Agora que terminamos a associação podemos testar o resultado. Vamos cadastrar os dados do livro, mas com um valor inválido de ISBN. Depois de associarmos um autor, clicamos em gravar. O JSF exibe a mensagem "Deveria começar com 1", indicando que nosso validador foi chamado. Se colocarmos um valor de ISBN válido, os dados são gravados.

##Criação de mensagens próprias

Aproveitando o nosso conhecimento sobre mensagens, vamos ver o que acontece quando gravamos um livro sem autor. Recebemos uma *exception*, lançada intencionalmente pelo método gravar de **livroBean**. Podemos melhorar isto, lançando um **`FacesMessage`**, algo muito mais elegante.

Voltando à classe **LivroBean**, vamos trocar o lançamento da *exception* por uma mensagem. Obtemos uma referência do contexto JSF no momento da chamada do método através de **`FacesContext.getCurrentInstance()`**. Agora, adicionamos neste contexto uma mensagem através do método `addMessage()`. Este método recebe dois parâmetros, o primeiro é o **`client ID`**, isto é, `id` definido no `XHTML` do componente. Neste caso, utilizaremos `autor`. O segundo parâmetro é um objeto do tipo **`FacesMessage`** que recebe em seu construtor a mensagem que queremos mostrar para o usuário.

```
@ManagedBean
@ViewScoped
public class LivroBean implements Serializable {

    // trechos do código omitidos

    public void gravar() {
        System.out.println("Gravando livro " + this.livro.getTitulo());

        if (livro.getAutores().isEmpty()) {
            FacesContext.getCurrentInstance().addMessage("autor",  new FacesMessage("Livro deve
```

```
            return;
        } else {
            new DAO<Livro>(Livro.class).adiciona(this.livro);
            this.livro = new Livro();
        }
    }
}
```

Vamos salvar as alterações e reiniciar o servidor. Visualizando a página mais uma vez, vamos tentar cadastrar um livro, mas sem adicionar um autor para ele. Quando gravamos, uma mensagem é exibida para o usuário.