04

Criando nosso próprio Converter

Transcrição

Vamos fazer um teste, modificaremos a hora do *Sistema Operacional* para **23:30h** e pediremos para que a hora não seja automaticamente. Feito isto, entraremos no formulário novamente. A data exibida deve estar mostrando do próximo dia. Isso ocorre porque existe diferença de *time zone* (fuso horário). Quando não especificamos nenhum, o default é UTC-0, e podemos gerar problemas futuros.

Vamos corrigir nossa data, informando o *timeZone* do nosso converter para "America/Sao_Paulo". Você pode usar qualquer outro, dependendo da sua necessidade. Nosso converter ficará assim:

```
<!-- Mantenha os demais componentes intactos -->
<f:convertDateTime pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo"/>
```

Faça *Full Publish* novamente e perceba que ao entrar no formulário, a data está correta, devendo exibir o dia de hoje no campo.

Mas algo ainda não está legal. Imagine se em todos os campos de data tivermos que fazer essas configurações, dizer qual formato usar, timeZone, etc. Perderemos muito tempo apenas configurando a mesma coisa em vários lugares.

Para casos como esse, podemos criar nosso próprio `Converter`. Onde já informaremos tudo o que desejamos que seja configurado para um determinado tipo de dado. Mas antes, vamos voltar o *input* de `dataPublicacao` para usar `Calendar` e remover o `<f:convertDateTime />` que usamos antes.

```
<!-- Só o input simples -->
<h:inputText value="#{adminLivrosBean.livro.dataPublicacao}"
        id="dataPublicacao" />
```

Vamos criar uma nova classe chamada `CalendarConverter` no pacote `br.com.casadocodigo.loja.converters`. Nessa nova classe, precismaos informar ao JSF que ela será nosso conversor, utilizamos a anotação @FacesConverter para mapear a nossa classe, tornando-a de fato uma classe de conversão. Dentro da annotation, informamos qual o tipo de dado que ele converterá: `@FacesConverter(forClass=Calendar.class)`.

Além da annotation, precisamos implementar uma interface que possui os métodos base de conversão. Esta interface é a `javax.faces.convert.Converter`, que nos obriga implementar dois métodos, o `getAsObject` e o `getAsString` como podemos ver suas assinaturas abaixo:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
            UIComponent component, String dataTexto) {
        return null;
    }
```

```
    @Override
    public String getAsString(FacesContext context,
            UIComponent component, Object dataObject) {
        return null;
    }
}
```

Esses dois métodos funcionam assim. Quando a informação está na tela, ela é uma String, nesse ponto o **JSF** chama o método `getAsString()` do nosso `Converter`. Quando está no `ManagedBean` é um Objeto que queremos, desta forma o **JSF** chama o método `getAsObject()`.

Agora precisamos implementar de fato nosso converter, faremos uso do mesmo conversor que usamos na tela, só que agora reaproveitando o que ele já faz configurado em um único local. Usaremos o `DateTimeConverter`, e deixaremos o trabalho de conversão pesado com ele. Vamos declará-lo como atributo de nossa classe:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

        private DateTimeConverter converter = new DateTimeConverter();

        //Mais código abaixo...
}
```

Na assinatura dos métodos, perceba que a única diferença é o retorno de cada um e o último parâmetro. No `getAsObject()`, o ultimo parâmetro é a data em texto que queremos transformar para `Object`, e no `getAsString()`, o último parâmetro é a data `Object` que queremos transformar em texto `(String)`.

Vamos começar pelo método `getAsObject()`. Queremos recuperar a data como um objeto, então chamamos o `converter.getAsObject`, passando os mesmos parâmetros que recebemos:

```
Date data = (Date) converter.getAsObject(context, component, dataTexto);
```

Como nosso converter é para `Date`, vamos realizar a transformação desse `Date` em `Calendar`.

```
Calendar calendar = Calendar.getInstance();
calendar.setTime(data);
return calendar;
```

Quando estavamos trabalhando com a data no formulário, nós tinhamos o problema do `timeZone` e o problema do *patter* de formatação da data em texto, precisamos resolver estes problemas aqui também. Desta forma, precisamos que o converter saiba qual `timeZone` usar e qual *pattern* usar quando a data estiver no formato texto.

Tanto o método `getAsObject` como o `getAsString`, farão uso do mesmo `Converter`, o que significa que teríamos que setar os parâmetros de `timeZone` e *pattern* duas vezes. Mas podemos ser mais espertos e criar um construtor para nosso converter, onde já realizamos toda configuração necessária:

```
public CalendarConverter() {
    converter.setPattern("dd/MM/yyyy");
```

```
        converter.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
    }
```

Pronto, todas as configurações foram realizada em um único lugar, e poderemos usar nos dois métodos.

Ainda temos que implementar o método `getAsString` que faz justamente o inverso. Aqui queremos recuperar a data no formato texto, então chamamos o `converter.getAsString` , passando os mesmos parâmetros que recebemos no método, com um detalhe, o object é um `Calendar` , e o `DateTimeConverter` recebe um `Date` . Vamos transformar os dados, e como não tem muito segredo, segue o código abaixo:

```
public String getAsString(FacesContext context,
            UIComponent component, Object dataObject) {
    if (dataObject == null)
        return null;

    Calendar calendar = (Calendar) dataObject;
    return converter.getAsString(context, component, calendar.getTime());
}
```

O único ponto de destaque é a verificação que fazemos antes de tudo. Pois se passarmos um `null` para o converter, ele lançará uma *Exception* e não é o que desejamos.

Nosso converter está pronto. Para que você confira todo o código que temos, segue:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

    private DateTimeConverter converter = new DateTimeConverter();

    public CalendarConverter() {
        converter.setPattern("dd/MM/yyyy");
        converter.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
    }

    @Override
    public Object getAsObject(FacesContext context,
                UIComponent component, String dataTexto) {
        Date data = (Date) converter.getAsObject(context, component, dataTexto);
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(data);
        return calendar;
    }

    @Override
    public String getAsString(FacesContext context,
                UIComponent component, Object dataObject) {
        if (dataObject == null)
            return null;

        Calendar calendar = (Calendar) dataObject;
        return converter.getAsString(context, component, calendar.getTime());
    }
}
```

Pronto, agora sempre que fizermos uso do Calendar teremos nosso converter em ação.

Podemos remover o inicializador de nosso atributo `dataPublicacao` da classe `Livro`, não precisamos inicializar mais nosso atributo. Assim o campo vem vazio na tela, pronto para o usuário preencher, o que seria o esperado. Veja o trecho de código abaixo:

```
public class Livro {

    // Demais atributos acima

    @Temporal(TemporalType.DATE)
    private Calendar dataPublicacao;

    // Demais atributos e métodos abaixo

}
```

O que mais ganhamos com essa implementação do Converter? Ganhamos o uso de Calendar em qualquer entidade do sistema, sendo transformada para texto automaticamente pelo **JSF**, sem que tenhamos que nos preocupar com o formato e `timeZone`. Criamos um único objeto que já serve para todo o sistema.

Faça novamente um *Full Publish* teste o cadastro de Livro completo.