Spring MVC I: Criando aplicações web
35%

AULA 08 Formatação de datas

ATIVIDADES
01 Vídeo 1 19min
02 Formatação de datas
03 Adicionando o campo data
04 Melhorando as configurações
05 Configurando pelo AppWebConfigurati...
06 Mantendo as informações inválidas

02 Formatação de datas

PRÓXIMA ATIVIDADE

Nos últimos capítulos, fizemos uma série de validações em nossos livros para garantir que os livros sejam cadastrados com pelo menos três informações que deixamos como obrigatórias, que são: O título, a quantidade de páginas e a descrição. Essas são as informações minimas para cadastrar um livro em nosso sistema.

Na Casa do Código quando ocorre o lançamento de um livro, a data do lançamento deste livro fica registrada no sistema. Vamos fazer com que essa data seja cadastrada em nosso sistema também. Vamos começar essas alterações pelo `form.jsp`

Nosso formulário agora precisa de um campo onde possamos digitar a data de lançamento do livro. Teremos então um novo `input` do tipo `text` com o atributo `name` com o valor `dataLancamento`. Já vamos deixar a validação dos erros com a tag `form:errors` pronta, com o `path` de valor `dataLancamento`. Seguindo o padrão dos outros campos do `form.jsp` teremos com nosso novo campo o seguinte código: **Observação:** Lembre-se de pôr o `label` deste campo.

```
<div>
    <label>Data de Lançamento</label>
    <input type="text" name="dataLancamento" />
    <form:errors path="dataLancamento" />
</div>
```

Vamos pôr esse campo logo após o campo de número de páginas. Nosso formulário no `form.jsp` fica assim:

```
<form:form action="${ s:mvcUrl('PC#gravar').build() }" method="post" co
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
        <form:errors path="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
        <form:errors path="descricao" />
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas" />
        <form:errors path="paginas" />
    </div>
    <div>
        <label>Data de Lançamento</label>
        <input type="text" name="dataLancamento" />
        <form:errors path="dataLancamento" />
    </div>
    <c:forEach items="${tipos}" var="tipoPreco" varStatus="status">
        <div>
            <label>${tipoPreco}</label> <input type="text"
                name="precos[${status.index}].valor" /> <input type="hi
                name="precos[${status.index}].tipo" value="${tipoPreco}
        </div>
    </c:forEach>
    <button type="submit">Cadastrar</button>
</form:form>
```

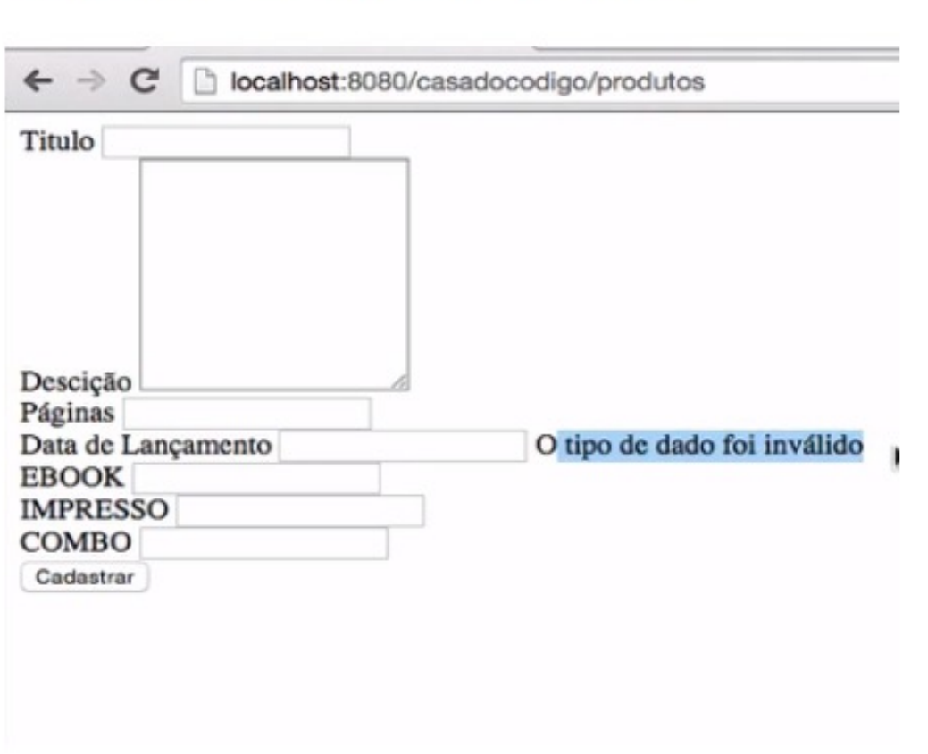
Precisamos agora, de um atributo que guarde datas em nosso `model` de produto. Existem duas principais classes para trabalhar com datas no `Java`. São elas a `Date` e a `Calendar`. Iremos usar a `Calendar` que é mais recente e mais simples de usar. Sendo assim, vamos criar um novo atributo na nossa classe `Produto` do tipo `Calendar`, sendo `private`.

Observação: Geralmente os atributos das classes são `private`, nestes casos lembre-se sempre de gerar os ***Getters and Setters** necessários. Use os atalhos do Eclipse, é mais rápido.

Com este novo atributo e seus ***Getters and Setters*** nossa classe `Produto` fica com o seguinte código:

```
@Entity
public class Produto {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    private String titulo;
    private String descricao;
    private int paginas;

    @ElementCollection
    private List<Preco> precos;

    private Calendar dataLancamento;


    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getPaginas() {
        return paginas;
    }
    public void setPaginas(int paginas) {
        this.paginas = paginas;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public List<Preco> getPrecos() {
        return precos;
    }
    public void setPrecos(List<Preco> precos) {
        this.precos = precos;
    }

    public Calendar getDataLancamento() {
        return dataLancamento;
    }
    public void setDataLancamento(Calendar dataLancamento) {
        this.dataLancamento = dataLancamento;
    }
    @Override
    public String toString() {
        return "Produto [titulo=" + titulo + ", descricao=" + descricao
    }

}
```

Pronto! Já podemos testar se o nosso sistema salva corretamente as datas. Abra o formulário em `localhost:8080/casadocodigo/produto/form` e tente cadastrar um novo livro com uma data qualquer. Teremos um erro, mas dessa vez um erro amigavel. Lembra que fizemos algumas validações? Veja o que acontece ao tentar cadastrar o livro com uma data, como por exemplo a data 10/05/2015.



O erro que recebemos faz todo o sentido. Note que apesar de estamos usando um data, no formulário só há textos. Ou seja, o **Spring** não está conseguindo converter nossa data que está em texto no formulário para um formato aceito pelo `Calendar`. O formato padrão é o internacional.

Podemos resolver esse problema, com uma anotação em nosso atributo `dataLancamento` chamada `@DateTimeFormat` passando o parametro `pattern` com o valor `dd/MM/yyyy`. Esse valor será usado para que o **Spring** consiga converter a data corretamente. O código do atributo em nossa classe `Produto` com a anotação fica da seguinte forma:

```
@Entity
public class Produto {
    [...]
    @DateTimeFormat(pattern="dd/MM/yyyy")
    private Calendar dataLancamento;
    [...]
}
```

Teste cadastrar um produto agora, com a data no padrão brasileiro, sendo este: `dia/mes/ano`. Apesar de funcionar, essa é uma forma trabalhosa de se fazer isso. Imagine que em todo lugar que precisemos usar data precisaremos lembrar dessa formatação. É fácil de esquecer certo? Sem contar que esse será o padrão da aplicação. Sendo um padrão, a melhor opção é configurarmos isso.

Em nossa classe `AppWebConfiguration`, que é nossa classe de configuração, faremos essa nova configuração para que não precisemos ficar repetindo essa configuração de data em todos os atributos das outras classes do sistema.

Criaremos então um novo método chamado `mvcConversionService` que retorna um objeto do tipo `FormattingConversionService`. Dentro deste novo método precisamos fazer duas coisas. A primeira delas é criar um objeto do tipo `DefaultFormattingConversionService` que será o responsável pelo serviço de conversão de formato. A segunda é criar um objeto do tipo `DateFormatterRegistrar` que fará o registro do formato de data usado para a conversão. Este segundo objeto espera receber outro objeto do tipo `DateFormatter` que será quem efetivamente guarda o padrão da data, que neste caso será `dd/MM/yyyy`, dia/mēs/ano.

O último passo será usar o registrador para registar o padrão de data no serviço de conversão. Lembre-se de usar a anotação `@Bean` para que o **Spring** consiga usar essa configuração. Encerramos o código desse método retornando o objeto `DefaultFormattingConversionService`. Nosso código final fica dessa forma:

```
@Bean
public FormattingConversionService mvcConversionService(){
    DefaultFormattingConversionService conversionService = new DefaultF
    DateFormatterRegistrar formatterRegistrar = new DateFormatterRegist
    formatterRegistrar.setFormatter(new DateFormatter("dd/MM/yyyy"));
    formatterRegistrar.registerFormatters(conversionService);

    return conversionService;
}
```

Nosso atributo agora, não precisa mais passar o formato da data. Ela agora só precisa da anotação simples, sem nenhum parametro. Dessa forma:

```
@Entity
public class Produto {
    [...]
    @DateTimeFormat
    private Calendar dataLancamento;
    [...]
}
```

Teste novamente, tudo deve funcionar normalmente.Nosso próximo passo é resolver um outro problema que é bem chatinho de se ver acontecendo. A esta altura do curso você já deve ter notado que quando enviamos os dados, com os campos preenchidos, quando validado o formulário nos exibe os erros, mas o forme aparece em branco. É trabalhoso preencher todos os campos novamente por causa de um campo que estava errado.

O ideal é que mesmo que um dos campos esteja com erro, os outros que não tiveram erro continuem com os dados que foram digitados antes de serem enviados. Para isso, devemos deixar com que o **Spring** gerencie todo o formulário. Fazemos isso usando as tags de formulário do **Spring**.

Da mesma forma que usamos `form:form` e `form:errors`, usaremos agora o `form:input` que cria um input do tipo `text`, mas em vez de usar o atributo `name`, usaremos o atributo `path` e não precisaremos usar o atributo `type`. Veja o antes e depois do campo de titulo por exemplo.Antes: `<input type="text" name="titulo" />` Depois: `<form:input path="titulo" />`

A nova tag parece ser bem mais simples certo? Para a descrição, que é um `textarea`, usaremos a tag `form:textarea` e para o tipo `hidden` usaremos a tag `form:hidden`. A mudança só requer quee troquemos o atributo `name` para `path` e remover o atributo `type`. Nosso formulário está assim atualmente:

```
<form:form action="${ s:mvcUrl('PC#gravar').build() }" method="post" co
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
        <form:errors path="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
        <form:errors path="descricao" />
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas" />
        <form:errors path="paginas" />
    </div>
    <div>
        <label>Data de Lançamento</label>
        <input type="text" name="dataLancamento" />
        <form:errors path="dataLancamento" />
    </div>
    <c:forEach items="${tipos}" var="tipoPreco" varStatus="status">
        <div>
            <label>${tipoPreco}</label> <input type="text"
                name="precos[${status.index}].valor" /> <input type="hi
                name="precos[${status.index}].tipo" value="${tipoPreco}
        </div>
    </c:forEach>
    <button type="submit">Cadastrar</button>
</form:form>
```

Com as mudanças, nosso formulário fica dessa forma:

```
<form:form action="${ s:mvcUrl('PC#gravar').build() }" method="post" co
    <div>
        <label>Título</label>
        <form:input path="titulo" />
        <form:errors path="titulo" />
    </div>
    <div>
        <label>descricao</label>
        <form:textarea rows="10" cols="20" path="descricao" />
        <form:errors path="descricao" />
    </div>
    <div>
        <label>Páginas</label>
        <form:input path="paginas" />
        <form:errors path="paginas" />
    </div>
    <div>
        <label>Data de Lançamento</label>
        <form:input path="dataLancamento" />
        <form:errors path="dataLancamento" />
    </div>
    <c:forEach items="${tipos}" var="tipoPreco" varStatus="status">
        <div>
            <label>${tipoPreco}</label>
            <form:input path="precos[${status.index}].valor" />
            <form:hidden path="precos[${status.index}].tipo" value="${t
        </div>
    </c:forEach>
    <button type="submit">Cadastrar</button>
</form:form>
```

Nosso formulário dessa forma ficou bem mais simples. Até o código ficou menor. Vamos agora atualizar nossa página de formulário no navegador para ver se tudo continua funcionando normalmente.

```
java.lang.IllegalStateException: Neither BindingResult nor plain target object for bean name 'produto' available as request attribute
	org.springframework.web.servlet.support.BindStatus.<init>(BindStatus.java:144)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getBindStatus(AbstractDataBoundFormElementTag.java:168)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getPropertyPath(AbstractDataBoundFormElementTag.java:188)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.getName(AbstractDataBoundFormElementTag.java:154)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.autogenerateId(AbstractDataBoundFormElementTag.java:141)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.resolveId(AbstractDataBoundFormElementTag.java:132)
	org.springframework.web.servlet.tags.form.AbstractDataBoundFormElementTag.writeDefaultAttributes(AbstractDataBoundFormElementTag.java:116)
	org.springframework.web.servlet.tags.form.AbstractHtmlElementTag.writeDefaultAttributes(AbstractHtmlElementTag.java:422)
```

Erro 500: Neither BindingResult nor plain target object for bean name 'produto' available as request attribute. Nosso formulário agora está com erro, ele não chega nem a ser exibido. A mensagem de erro diz que o objeto produto não está disponível como atributo da requisição. O **Spring** tenta usar um objeto da classe `Produto` para poder exibir o formulário. Isso acontece porque já que configuramos o formulário para guardar os dados mesmo quando acontecer erros de validação, dessa forma, ele precisa de um objeto para poder armazenar esses dados e para poder exibir o formulário, mesmo que vazio.

Para que o objeto do tipo `Produto` fique disponível em nosso formulário, só precisamos fazer uma pequena alteração em nosso `ProdutosController`. Em nosso método `form()` só precisamos colocar que queremos receber como parametro um objeto do tipo `Produto`. Dessa forma o **Spring** já deixará este objeto disponível na requisição. Nosso médoto `form()` que estava assim: `public ModelAndView form(){ ... }` agora fica assim: `public ModelAndView form(Produto produto){ ... }`.

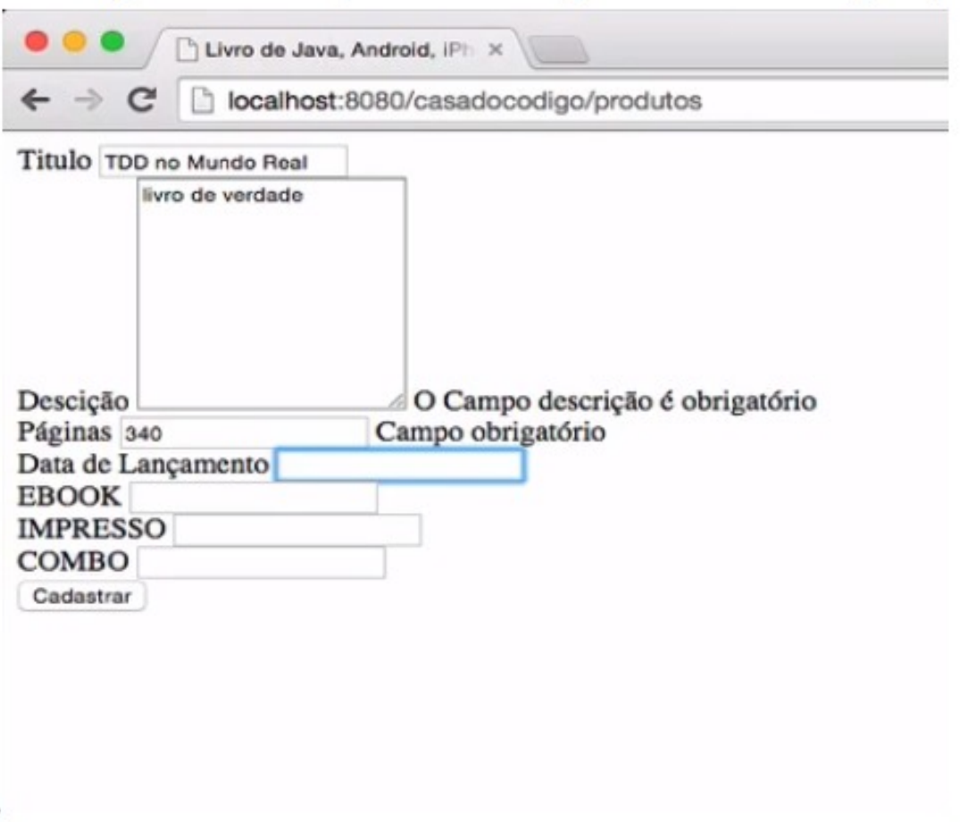
Lembre-se também de passar o objeto produto onde chamamos o médoto `form`. Lembra do método `gravar`? Ele caso não receba um produto válido, chama o método `form`. Devemos então passar para o método `form` o mesmo objeto recebido no método `gravar`. Nosso código atual do método `gravar` está assim:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(@Valid Produto produto, BindingResult result
    if(result.hasErrors()){
        return form();
    }
    produtoDao.gravar(produto);
    redirectAttributes.addFlashAttribute("message","Produto cadastrado
    return new ModelAndView("redirect:produtos");
}
```

E com a mudança fica assim:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(@Valid Produto produto, BindingResult result
    if(result.hasErrors()){
        return form(produto);
    }
    produtoDao.gravar(produto);
    redirectAttributes.addFlashAttribute("message","Produto cadastrado
    return new ModelAndView("redirect:produtos");
}
```

Atualize o formulário no navegador e teste novamente. Tente cadastrar livros com dados incorretos, tente com dados corretos também para verificar que tudo continua funcionando e agora nossas mensagens de erro aparecem sem que o formulário apareça todo em branco novamente.



Recapitulando

Neste capítulo fizemos com que nosso formulário de cadastro de produtos permaneça dos com dados mesmo que algum campo tenha sido enviado com informações inválidas. Deixamos o formulário com um código mais simples e fácil de entender.

Fizemos também uma série de configurações para conversão de datas, uma tarefa muito comum em aplicação de qualquer gênero. Agora é hora de praticar um pouco mais com os exercícios.

TIRAR DÚVIDA
PRÓXIMA ATIVIDADE