01

Configuração do ambiente e os primeiros passos com JSF

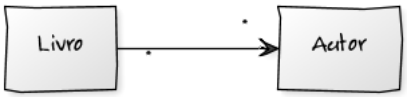
Transcrição

Nessa vídeo-aula começaremos a desenvolver uma aplicação web com JSF. Já temos um ambiente de desenvolvimento pré-configurado baseado no Java SE 7 (http://www.oracle.com/technetwork/java/javase/downloads/index.html) e utilizaremos como IDE o Eclipse IDE for Java EE Developers (http://www.eclipse.org/downloads/). Nosso *Servlet Container* será o Apache Tomcat (http://tomcat.apache.org/download-70.cgi) na versão 7 - todos disponíveis nos links indicados.

Além disso precisaremos o JAR da implementação referencial do JSF (Mojarra): aqui (http://s3.amazonaws.com/caelum-online-public/JSF/javax.faces-2.1.14.jar)

Entendendo o domínio da aplicação

O nosso projeto web facilitará o trabalho em uma Livraria, onde o usuário poderá cadastrar livros e autores. Trabalharemos com objetos do tipo **Livro** associados a **autores**. Trata-se de um relacionamento **muitos-para-muitos**. Antes de modelarmos as classes, vamos criar e configurar o projeto.



Atenção na importação

Ao importar o projeto o Eclipse automaticamente fará uma validação rígida dos arquivos. Nesse caso, o Eclipse acusa a assinatura de alguns métodos do projeto. Para desabilitar essa validação basta acessar:

Menu **Preferences** -> Item **Validation**:

Tire:

1) Facelet HTML Validator

2) JSF View Application Configuration Validator

3) JSF View Validator

Isto é necessário pois o Eclipse quer que todos os métodos associados ao atributo `action` do componente `h:commandButton` devolvam obrigatoriamente um `String` (que não é preciso).

Configurando o servidor web

A IDE Eclipse já está rodando, mas é preciso fechar a janela de "boas-vindas" para ver a perspectiva padrão. Durante o treinamento, utilizaremos a perspectiva `Java EE` , mas podemos trocar a qualquer momento essa perspectiva por meio do botão "+" no lado superior direito do Eclipse.

O primeiro passo é configurar o *Servlet Container* Tomcat dentro do Eclipse. Para tal, utilizaremos a aba (*view*) *Server*. Caso essa janela tenha sido fechada, podemos reabri-la através do atalho `Ctrl + 3` e digitando *server*.

Vamos então preparar o Tomcat definindo uma nova configuração do servidor. Na caixa de diálogo *New server*, escolheremos **Apache Tomcat v7.0 Server**, em seguida, apertando *next*, definimos o lugar onde o Tomcat foi instalado (ou descompactado). No nosso caso, como já temos uma distribuição do Tomcat baixada e extraída, vamos apontar para ela. Ao finalizar aparecerá o Tomcat na *view* ***Server***.

Criação do projeto JSF

O próximo passo é criar o projeto web. Vamos no menu File -> New -> Dynamic Web Project. Chamaremos o projeto de `jsf-livraria`. É preciso verificar se o Tomcat está escolhido como servidor, inclusive se o *module version* encontra-se na versão 3.0.

No combo box *Configuration*, escolhemos `Java Server Faces`. Confirmaremos as duas próximas telas até chegar na tela *JSF Capabilities*.

Aqui configuraremos de onde vem a biblioteca JSF. Para este projeto, copiaremos o JAR após a criação do projeto. Ou seja, desabilitamos o uso de qualquer biblioteca fornecida pelo Eclipse.

Um pouco mais abaixo há a configuração do servlet que representa o controlador. Usaremos praticamente a mesma configuração. Apenas o mapeamento será alterado. Queremos que qualquer requisição que termine com `*.xhtml` seja processada pelo `FacesServlet`.

Ao confirmar, será gerado o novo projeto, visível ao lado esquerdo, no `view project-explorer`. Vamos arrastar o projeto ao servidor Tomcat para simplificar o deploy da aplicação.

Ao abrir o projeto, veremos a estrutura básica com as classes Java dentro da pasta *src* e todos os arquivos Web dentro da pasta *WebContent*.

Dentro do *WebContent* encontramos uma pasta *WEB-INF* com dois arquivos de configuração. O primeiro, `web.xml`, é relacionado com a especificação servlet. Ele contém a declaração do `FacesServlet`, com aquela configuração que alteramos na criação do projeto.

O segundo XML é o arquivo de configuração relacionado com o mundo JSF. Como o JSF na versão dois encoraja o uso de anotações, este arquivo torna-se pouco usado, sendo muito mais importante na primeira versão do JSF.

Como vimos no `web.xml`, há a configuração do `FacesServlet`. Essa classe faz parte da implementação JSF e deve estar presente. É preciso copiar o JAR do Mojarra para o nosso projeto.

Já baixamos a biblioteca e como se trata de um projeto web, o JAR deve ser copiado para a pasta WEB-INF/lib e assim fará parte do *buildpath* do projeto.

Começando a implementar nossa livraria

Tudo pronto para desenvolver com JSF. No projeto, vamos começar com o cadastro de livros. O objetivo é criar um formulário com os componentes da especificação.

Vamos selecionar a pasta WebContent, clicar com o botão direito, *New HTML File*. O arquivo se chama **livro.xhtml**. Cuidado com a extensão que deve ser XHTML. Ao apertar next, podemos escolher um template. Usaremos *xhtml 1.0*

transitional.

No arquivo apagaremos tudo que está dentro das tags HTML, pois utilizaremos os componentes JSF. Para declará-los é preciso adicionar um XML namespace na abertura da tag HTML.

Para tal, digitamos `xmlns:h` , onde o "h" é o apelido do namespace para a uri "http://java.sun.com/jsf/html" (http://java.sun.com/jsf/html"). `Ctrl + Espaço` ajuda a auto-completar. Atenção para não confundir com o namespace JSTL.

```
<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">

</html>
```

A partir da definição no cabeçalho, podemos usar o apelido "h" para declarar os componentes JSF. O primeiro componente que usaremos é o `h:body` que define o corpo da página. Dentro do body vamos declarar o formulário através da tag `h:form` . Repare que aqui, diferente da tag form do HTML, o componente JSF não possui um atributo `action` .

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h:form>
        </h:form>
    </h:body>
</html>
```

No formulário, usaremos o primeiro componente que captura uma entrada do usuário, o `h:inputText` . Além do `inputText` , vamos utilizar um botão para executar uma ação. A especificação define comandos para isto. Nesse caso, um `h:commandButton` . Com o atributo value definimos "Gravar", que aparecerá na tela.

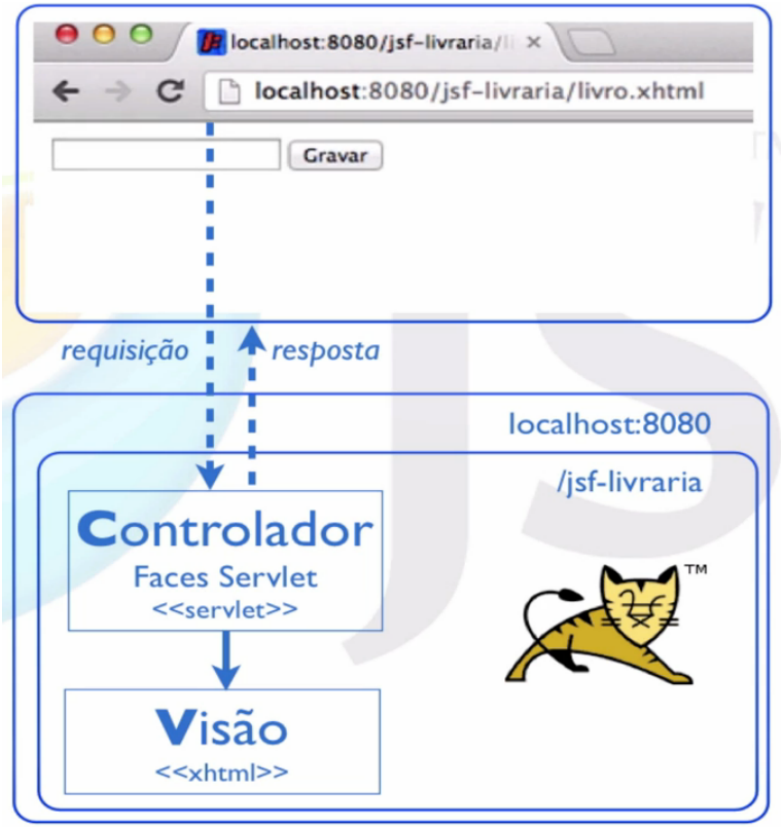
```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h:form>
            <h:inputText />
            <h:commandButton value="Gravar" />
        </h:form>
    </h:body>
</html>
```

Com o Tomcat rodando já podemos testar a página pelo navegador, accessando http://localhost:8080/jsf-livraria/livro.xhtml (http://localhost:8080/jsf-livraria/livro.xhtml). O formulário aparecerá.

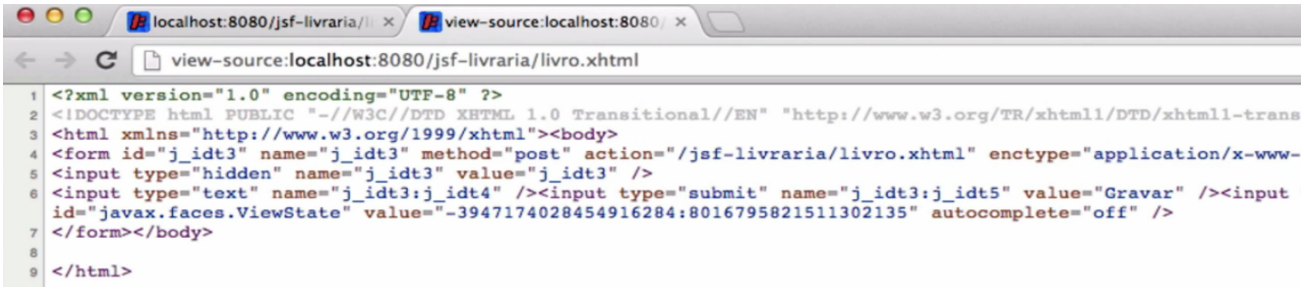
Controlador e Visão no JSF

Parece que chamamos diretamente a página `livro.xhtml`, mas isso não é verdade. Lembrando que no `web.xml` teremos o `FacesServlet` mapeado para receber qualquer requisição que termina com `xhtml`. Acontece que ao enviar a requisição, o Tomcat delega o fluxo para o servlet da nossa aplicação `jsf-livraria`. O servlet recebe a chamada e decide qual página chamará. É ele que está no controle do fluxo, e por isso também é chamado controlador.

O controlador lê o `xhtml` e instancia os componentes declarados. No final ele pede aos componentes a apresentação HTML e devolve o HTML como resposta para o navegador.



Podemos provar isso também no navegador. Ao visualizar o código fonte da página, aparece o HTML puro. Esse HTML é bem diferente da página `xhtml`.



Completando o formulário

Vamos continuar e completar o formulário. Primeiro utilizaremos o componente `h:outputLabel` para associar um label com o input. A ligação é feita pela id da componente inputText e o atributo `for` do outputLabel. Ao testar no navegador aparece o label na frente do input.

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h:form>
```

```
            <h:outputText value="Titulo: " for="titulo"/>
            <h:inputText id="titulo"/>
            <h:commandButton value="Gravar" />
        </h:form>
    </h:body>
</html>
```

Mesmo usando componentes JSF, podemos aproveitar as tags do mundo HTML. Vamos definir um cabeçalho na página usando a tag `h1`. Além disso, usaremos um fieldset para agrupar os elementos do formulário com um título indicado pela tag `legend`. Visualizando mais uma vez no navegador, podemos perceber que o formulário já está mais organizado.

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
<h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                <h:outputLabel value="Titulo:" for="titulo" />
                <h:inputText id="titulo" />
            </fieldset>
        </h:form>
    </h:body>
```

Falta criar os componentes para o ISBN, preço e data de lançamento do Livro. Copiaremos o `outputLabel` e `inputText` para facilitar o trabalho. Primeiro para ISBN, segundo o preço e terceiro a data de lançamento. Visualizaremos mais uma vez o resultado no navegador.

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                    <h:outputLabel value="Titulo:" for="titulo" />
                    <h:inputText id="titulo" />
                    <h:outputLabel value="ISBN:" for="isbn" />
                    <h:inputText id="isbn" />
                    <h:outputLabel value="Preço:" for="preco" />
                    <h:inputText id="preco" />
                    <h:outputLabel value="Data de Lançamento:" for="dataLancamento" />
                    <h:inputText id="dataLancamento" />
                    <h:commandButton value="Gravar" />
            </fieldset>
        </h:form>
    </h:body>
</html>
```

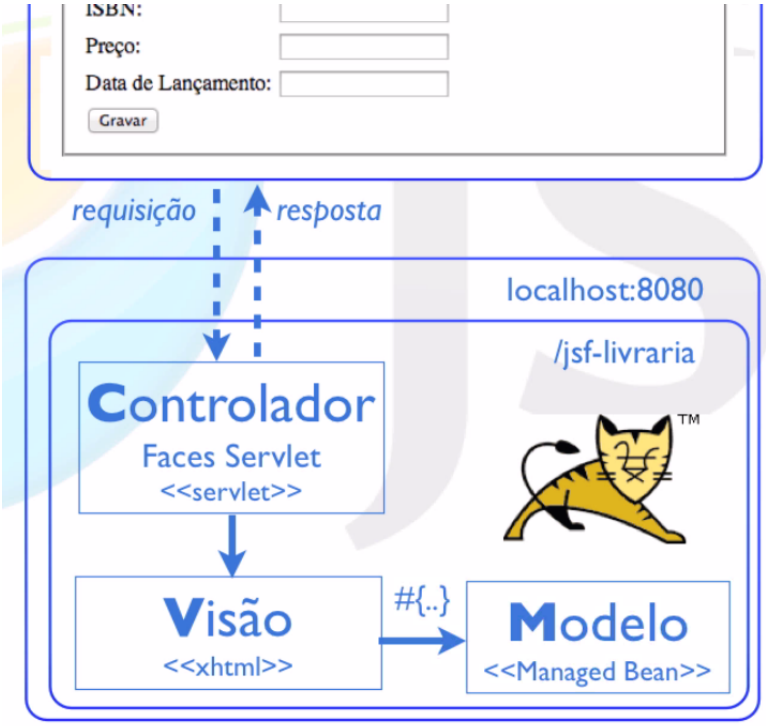
Todos os componentes foram renderizados para HTML. Um após o outro, ocupando todo espaço horizontal da tela. Para melhorar o layout, utilizaremos o componente `h:panelGrid`, que funciona como um simples *layoutmanager*, organizando os componentes verticais.

Também podemos definir a quantidade de colunas no grid. Para tal, usamos o atributo `columns` do componente `h:panelGrid` para criar duas colunas:

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                <h:panelGrid columns="2">
                    <!-- inputs omitido -->
                    <h:commandButton value="Gravar" />
                </h:panelGrid>
            </fieldset>
        </h:form>
    </h:body>
</html>
```

Ligando Managed Beans a componentes visuais

Conseguimos criar a tela, mas também queremos **executar uma ação para gravar o livro**. Ou seja, quando o usuário apertar o botão "Gravar" no navegador, este componente disparará a execução de um método no lado do servidor. Para isso vamos criar uma **classe, associando-a com o componente**.



Cada componente possui atributos especiais para criar uma ligação com uma classe. No caso do `commandButton`, usaremos o atributo `action`. Nele definiremos a classe e o metódo a executar, mas primeiro é preciso criar esta classe.

Clique com o botão direito na pasta `src`, em seguinda, *New -> Class*. Podemos utilizar qualquer nome da classe. Em nosso caso, vamos chamá-la de **LivroBean**, dentro do `package` *br.com.caelum.livraria.bean*:

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

public class LivroBean {
}
```

Na classe criaremos o método `gravar()`, que imprime apenas uma informação no console. Além disso, é preciso indicar que a classe será gerenciada pelo JSF. Isso é feito através da anotação **@ManagedBean**:

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

import javax.faces.bean.ManagedBean;

@ManagedBean
public class LivroBean {

    public void gravar() {
        System.out.println("Gravando livro ");
    }
}
```

Voltando ao formulário, podemos agora **ligar o comando ao método** `gravar()`. Essa ligação (ou binding) é feita com uma **linguagem de expressão (Expression Language)**. Uma expressão sempre começa com `#{` e termina com `}`, dentro dela usaremos o nome da classe, "ponto (.)" seguido do nome do método:

```
<h:commandButton value="Gravar" action="#{livroBean.gravar}"/>
```

Antes de testar no navegador é preciso reiniciar o Tomcat. Ao atualizar o formulário no navegador e clicar no botão, podemos observar a saída no console do Eclipse.

Capturando dados do formulário com a classe LivroBean

O próximo passo é capturar os dados do formulário com a classe LivroBean. Para isso definiremos, para cada componente de input, um atributo no `LivroBean`. Primeiro o `titulo` e `isbn`, ambos do tipo `String`, depois `preco` do tipo `double` e a `dataLancamento` novamente do tipo `String`. Mais para frente veremos como trabalhar com datas do tipo `Date` ou `Calendar`. Por enquanto vamos deixar a `data` como `String`.

Além disso, é preciso gerar os getters e setters para cada atributo. Para isso, pressione `Ctrl + 3`, digitando **GGAS**. Selecione *Generate Getters e Setters* e, na janela, todos os atributos para confirmar. Vamos também formatar a classe usando o atalho `Ctrl+Shift+F`.

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

import javax.faces.bean.ManagedBean;

@ManagedBean
public class LivroBean {

    private String titulo;
    private String isbn;
    private double preco;
    private String dataLancamento;

    public void gravar() {
        System.out.println("Gravando livro ");
    }

    //getters e setters
}
```

Voltando ao `livro.xhtml`, ligaremos cada input com o atributo ou propriedade da classe `LivroBean`. Nos componentes de input usaremos o atributo `value`, pois queremos receber o valor desse componente, mas novamente através de *Expression Language*. No primeiro input: `#{livroBean.titulo}`. Faremos a mesma coisa para os demais inputs `isbn`, depois o `preco` e no final, a `dataLancamento` - sempre usando *Expression Language*:

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">

    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                <h:panelGrid columns="2">
                    <h:outputLabel value="Titulo:" for="titulo" />
                    <h:inputText id="titulo" value="#{livroBean.titulo}"/>
                    <h:outputLabel value="ISBN:" for="isbn" />
                    <h:inputText id="isbn" value="#{livroBean.isbn}"/>
                    <h:outputLabel value="Preço:" for="preco" />
```

```
                    <h:inputText id="preco" value="#{livroBean.preco}"/>
                    <h:outputLabel value="Data de Lançamento:" for="dataLancamento" />
                    <h:inputText id="dataLancamento" value="#{livroBean.dataLancamento}"/>
                    <h:commandButton value="Gravar" action="#{livroBean.gravar}"/>
                </h:panelGrid>
            </fieldset>
        </h:form>
    </h:body>

</html>
```

Falta mostrar no método `gravar()` algum valor do livro que está sendo gravado para podermos verificar o recebimento dos valores. Vamos concatenar o título na saída.

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

import javax.faces.bean.ManagedBean;

@ManagedBean
public class LivroBean {

    private String titulo;
    //outros atributos

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

    //getters e setters
}
```

É uma boa prática reiniciar o servidor para ter certeza que as alterações foram publicadas. Também é boa prática chamar o formulário de novo (ou seja, enviando um novo request) para garantir que todos os componentes foram atualizados. Ao atualizar podemos ver uma pequena mudança no formulário: o campo *Preço* já vem preenchido. Vamos inserir os dados e apertar em *Gravar*.

O Eclipse mostra *Gravando livro* com o título no console. No nosso caso, "Fausto". Tudo funcionando. Recebemos os dados do formulário no `LivroBean` .

Refatoração do ManagedBean para popular o domínio

Ao observar a classe LivroBean podemos criticar vários atributos "soltos". Muito provavelmente precisaremos deles em outras classes a medida que a aplicação crescer, pois eles representam um livro. Faz todo sentido separar a responsabilidade de cadastrar o livro da responsabilidade de ser um livro. Por isso, selecionaremos os atributos e extrairemos uma nova classe que representa um Livro.

Ainda usando o Eclipse, no menu *Refactor*, selecione *Extract Class*. Chamaremos a classe de `Livro` e geraremos os getters e setters. No `LivroBean` , criaremos um único atributo chamado `livro` . O Eclipse gerou um nova classe `Livro` com os atributos e getters/setters.

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

```
public class Livro {
    private String titulo;
    private String isbn;
    private double preco;
    private String dataLancamento;

    //getters e setters
}
```

Na classe `LivroBean` podemos apagar todos os getters. Repare que sobrou apenas um atributo privado que necessita de um getter. Esse getter é utilizado pelos componentes para popular o livro:

```
@ManagedBean
public class LivroBean {

    private Livro livro = new Livro();

    public Livro getLivro() {
        return livro;
    }

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

Após reiniciar o Tomcat, vamos testar novamente no navegador. Ao chamar o formulário recebemos uma exceção. Há algum problema na página `livro.xhtml` , na linha 13, com a expressão `#{livroBean.titulo}` . Ao analisar percebemos que não existe o atributo `titulo` no `LivroBean` .

Esse atributo está na classe Livro, mas não atualizamos o **livro.xhtml**. Na página é preciso alterar as expressões de cada input e declarar o caminho correto. Para o `titulo` usaremos agora: `#{livroBean.livro.titulo}` .

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">

    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                <h:panelGrid columns="2">
                    <h:outputLabel value="Titulo:" for="titulo" />
                    <h:inputText id="titulo" value="#{livroBean.livro.titulo}"/>
                    <h:outputLabel value="ISBN:" for="isbn" />
                    <h:inputText id="isbn" value="#{livroBean.livro.isbn}"/>
                    <h:outputLabel value="Preço:" for="preco" />
                    <h:inputText id="preco" value="#{livroBean.livro.preco}"/>
                    <h:outputLabel value="Data de Lançamento:" for="dataLancamento" />
                    <h:inputText id="dataLancamento" value="#{livroBean.livro.dataLancamento}"/
                    <h:commandButton value="Gravar" action="#{livroBean.gravar}"/>
                </h:panelGrid>
            </fieldset>
        </h:form>
```

```
    </h:body>

</html>
```

Por fim, testaremos mais uma vez. Como alteramos apenas o formulário, não é preciso reiniciar o servidor. Vamos atualizar o formulário e inserir os dados. Ao gravar, nenhuma exceção foi lançada.