UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA ESPECIAL DE TREINAMENTO
CIÊNCIAS DA COMPUTAÇÃO

Curso de JSP

Roberto Hartke Neto

Versão 1.0
Florianópolis, Outubro de 2002

Sumário

Capítulo 1

Introdução

1.1 O que é JSP?

JSP significa "Java Server Pages". Esta tecnologia é usada para servir conteúdo dinâmico para o usuário, usando lógica e dados no lado do servidor. JSP faz parte da plataforma J2EE (Java 2 Enterprise Edition) e juntamente com os Java Servlets e Java Beans pode ser usada para desenvolver aplicações web eficientes, escaláveis e seguras rapidamente.

1.1.1 Por que usar JSP se já existe PHP, ASP, etc?

Existem várias linguagens usadas para criar aplicações web. Entre elas ASP, PHP, ColdFusion e Perl. Por que usar JSP então?

- **JSP usa Java.**
  Java é uma das linguagens mais populares atualmente e é interpretada, portanto o código escrito em uma arquitetura pode ser portado para qualquer outra.

- **JSP é parte do pacote J2EE**
  J2EE é um dos modelos mais usados para contruir aplicações de grande porte, e é suportado por várias gigantes da computação como IBM, Oracle, Sun, etc.

- **Programação em rede é inerente a Java**
  O suporte inerente de Java para a área de redes faz dela uma ótima linguagem para a Internet.

- **JSP x ASP**
  Uma das diferenças que pode ser fundamental para a escolha entre estas duas tecnologias é que ASP é da Microsoft e só roda em ambiente Windows, e também todo software necessário é pago. JSP, feito pela Sun, roda em qualquer plataforma que tenha a máquina virtual de Java, e tem vários softwares gratuitos para disponibilizar a aplicação (Tomcat por exemplo).

1.2 Como JSPs funcionam

A finalidade de JSP é fornecer um método de desenvolvimento de servlets declarativo e centrado na apresentação. A especificação de JSP é definida como uma extensão da API de Servlets. Conseqüentemente, não é de se admirar que por trás dos panos, servlets e páginas JSP tem muito em comum.

Tipicamente, páginas JSP estão sujeitas a uma fase de tradução e outra de processamento da requisição. A fase de tradução é feita apenas uma vez, a menos que a página JSP mude, e no caso é traduzida novamente. Supondo que não houve nenhum erro de sintaxe na página, o resultado é uma página JSP que implementa a interface Servlet.

A fase de tradução é tipicamente realizada pela *engine JSP*, quando ela recebe uma requisição para a página JSP pela primeira vez. A especificação JSP 1.1 também permite que páginas JSP sejam pré-compiladas em arquivos *class*. Isto pode ser útil para evitar a demora para carregar a página JSP na primeira vez que ela é acessada. Várias informações da fase de tradução, como a localização de onde é armazenado o página JSP já compilada (portanto o servlet correspondente a esta página) são dependentes da implementação da *engine JSP*.

A classe que implementa uma página JSP estende a classe *HttpJspBase*, que implementa a interface Servlet. O método de serviço desta classe, *_jspService()*, essencialmente encapsula o conteúdo da página JSP. Ainda que o método *_jspService()* não pode ser sobrescrito, o desenvolvedor pode descrever eventos de inicialização e destruição fornecendo implementações dos métodos *jspInit()* e *jspDestroy* dentro da página JSP.

Uma vez que a classe é carregada no recipiente, o método *_jspService()* é responsável por responder às requisições do cliente.

1.3 Instalando e configurando o Tomcat

Primeiro você precisa ter a máquina virtual java (JDK 1.3 ou mais atual) instalada na sua máquina. Esta pode ser obtida gratuitamente no endereço *http://java.sun.com.*

1.3.1 Obtendo o Tomcat

Tomcat é um servidor de páginas JSP e Servlets. Desenvolvido pela fundação Apache, no projeto Jakarta, seu código é aberto e o programa é gratuito. Pode ser obtido em http://jakarta.apache.org.

1.3.2 Instalando o Tomcat

Instale ou descompacte o arquivo que você baixou em algum diretório. Depois você terá que criar duas variáveis de ambiente, *CATALINA_HOME* e *JAVA_HOME*, onde *CATALINA_HOME* é o diretório base do Tomcat e *JAVA_HOME* é o diretório base da plataforma Java.

Por exemplo, se o Tomcat foi instalado em c:\tomcat e a JVM (Java Virtual Machine) está em c:\java, estas variáveis devem ser:

- CATALINA_HOME = c:\tomcat
- JAVA_HOME = c:\java

Nos Windows 95/98 adicione os comandos **SET CATALINA_HOME = c:\tomcat** e **SET JAVA_HOME = c:\java** no arquivo **c:\autoexec.bat**.

No Windows 2000 vá em **Painel de Controle, Sistema, Avançado, Variáveis de Ambiente** e entre com as informações.

No caso do UNIX, use o comando **export** (ou **set**, dependendo da sua distribuição).

1.3.3 Executando o Tomcat

Vá no diretório CATALINA_HOME\bin (onde CATALINA_HOME é o diretório base do Tomcat), e execute o arquivo *startup.bat* para iniciar o Tomcat e *shutdown.bat* para encerrá-lo.

No caso dos UNIX, vá no mesmo diretório e digite *./tomcat4 start* para iniciar e *./tomcat4 stop* para finalizá-lo.
Para visualizar as páginas acesse *http://localhost:8080/* (localhost ou o endereço da máquina).

1.3.4 Disponibilizando a aplicação

Crie um diretório dentro do diretório *CATALINA_HOME/webapps*, ou coloque o arquivo da sua aplicação (.WAR) neste diretório. Para ver a aplicação acesse a URL *http://localhost:8080/(nome do diretório ou arquivo da aplicação)/* no navegador.

Exemplo:

Para acessar a aplicação localizada em *CATALINA_HOME/webapps/app*, acesse *http://localhost:8080/app*.

1.3.5 Estrutura de diretórios da aplicação

As aplicações JSP seguem um padrão. Devem ter um diretório *WEB-INF* (letras maiúsculas). Dentro desse, crie outro diretório chamado *classes* (letras minúsculas). Neste você deve colocar todos as classes que você usa na aplicação (arquivos .class), inclusive servlets e beans. Pacotes (arquivos .jar) podem ser colocados no diretório *WEB-INF/lib/*. Os arquivos JSP podem ser postos no diretório raiz ou em qualquer outro diretório, menos no WEB-INF e sub-diretórios!

Capítulo 2

Release 1

2.1 Java Beans

2.1.1 O que faz de um Bean um Bean?

Um Bean é simplesmente uma classe de Java que segue um conjunto de convenções simples de design e nomeação delineado pela especificação de JavaBeans. Os Beans não precisam estender uma determinada classe ou implementar uma determinada interface.

2.1.2 Convenções de um Bean

As convenções de JavaBean são o que nos permitem desenvolver Beans, porque elas permitem que o *container* Bean analise um arquivo de classe Java e interprete seus métodos como propriedades, designando a classe como um Bean de Java.

- Beans são simplesmente objetos Java. Mas como seguem um padrão fica mais fácil trabalhar com eles.
- Uma boa prática é colocar *Bean* no nome de uma classe que é um Bean, para que seja melhor identificado. Assim uma classe que representa uma pessoa ficaria *PessoaBean* ou *BeanPessoa*.

O construtor Bean

A primeira regra da criação do Bean JSP é que você tem que implementar um construtor que não tenha argumentos. Este construtor é usado por exemplo para instanciar um Bean através da tag <*jsp:useBean*> visto mais adiante. Se a classe não especificar um construtor sem argumentos, então um construtor sem argumentos e sem código será assumido.

```
public Bean() { }
```

Propriedades de um Bean

Coloque os atributos como privados, e faça métodos *get* e *set* para acessá-los, e este métodos então serão públicos.

```
private String nome;
public String getNome() { return nome;}
public void setNome(String novo) { nome = novo; }
```

Uma boa convenção de nome de propriedades, é começar com letra minúscula e colocar em maiúscula a primeira letra de cada palavra subseqüente. Assim como nos métodos de ajuste, a palavra *set* ou *get* começa em minúscula e a primeira letra da propriedade será maiúscula.

```
private String corCarro;
public String getCorCarro();"
```

Propriedades indexadas

Caso a propriedade seja um conjunto de valores (*array*), é uma boa prática criar métodos para acessar o conjunto inteiro de valores e para acessar uma posição específica.

```
private String[] telefone;

public String[] getTelefone() { return telefone; }

public String getTelefone(int index) { return telefone[index]; }
```

Propriedades booleanas

Para propriedades booleanas, você pode substituir a palavra *get* por *is*.

```
private boolean enabled;

public boolean isEnabled() { return enabled; }
```

2.2 JSP Tags

Numa olhada rápida, JSP parece com HTML (ou XML), ambos contém texto encapsulado entre tags, que são definidas entre os símbolos < e >. Mas enquanto as tags HTML são processadas pelo navegador do cliente para mostrar a página, as tags de JSP são usadas pelo servidor web para gerar conteúdo dinâmico.

A seguir estão os tipos de tags válidos em JSP:

2.2.1 Ações JSP

Executam diversas funções e estendem a capacidade de JSP. Usam sintaxe parecida com XML, e são usadas (entre outras coisas) para manipular Java Beans. Existem seis tipos de ações:

<jsp:forward>

Este elemento transfere o objeto *request* contendo informação da requisição do cliente de uma página JSP para outro arquivo. Este pode ser um arquivo HTML, outro arquivo JSP ou um servlet, desde que faça parte da mesma aplicação.

Sintaxe:

```
<jsp:forward page="(URL relativa | <%= expressão %>)" />

ou

<jsp:forward page="(URL relativa | <%= expressão %>)" >
    <jsp:param name="nome do parâmetro"
     value="(valor do parâmetro| <%= expressão %>)" />
</jsp:forward>
```

<jsp:getProperty>

Este elemento captura o valor da propriedade de um bean usando o método *get* da propriedade e mostra o valor na página JSP. É necessário criar ou localizar o bean com <jsp:useBean> antes de usar <jsp:getProperty>.

Sintaxe:

```
<jsp:getProperty name="nome do objeto (bean)"
 property="nome da propriedade" />
```

<jsp:include>

Este elemento permite incluir um arquivo estático ou dinâmico numa página JSP. Os resultados de incluir um ou outro são diferentes. Se o arquivo é estático, seu conteúdo é incluído quando a página é compilada num servlet. Se for dinâmico, funciona como uma requisição para o arquivo e manda o resultado de volta para a página. Quando estiver terminada a ação do *include* continua-se processando o restante da página.

Sintaxe:

```
<jsp:include page="{URL relativa | <%= expressão %>}"
 flush="true" />

ou

<jsp:include page="{URL relativa | <%= expressão %>}"
 flush="true" >
   <jsp:param name="nome do parâmetro"
    value="{nome do parâmetro | <%= expressão %>}" />
</jsp:include>
```

<jsp:plugin>

Executa ou mostra um objeto (tipicamente um applet ou um bean) no navegador do cliente, usando o plug-in Java que está embutido no navegador ou instalado na máquina. Não será explicado o funcionamento deste elemento no curso.

<jsp:useBean>

Localiza ou instancia um componente. Primeiro tenta localizar uma instância do bean. Se não existe, instancia ele a partir da classe especificada. Para localizar ou instanciar o bean, são seguidos os seguintes passos, nesta ordem:

1. Tenta localizar o bean com o escopo e o nome especificados.
2. Define uma variável de referência ao objeto com o nome especificado.
3. Se o bean for encontrado, armazena uma referência ao objeto na variável. Se foi especificado o tipo, converte o bean para este tipo.
4. Se não encontrar, instancia-o pela classe especificada, armazenando uma referência ao objeto na nova variável.
5. Se o bean tiver sido instanciado (ao invés de localizado), e ele tem tags de corpo (entre <jsp:useBean> e </jsp:useBean>), executa estas tags.

Sintaxe:

```
<jsp:useBean
        id="nome da instancia"
        scope="page|request|session|application"
    {   class="package.class" |
        type="package.class" |
        class="package.class" type="package.class" |
        beanName="{package.class | <%= expressão %>}"
        type="package.class"
    }
    {   /> |
        > outros elementos (tags de corpo)
        </jsp:useBean>
    }
```

<jsp:setProperty>

O elemento <jsp:setProperty> ajusta o valor de uma ou mais propriedades em um *bean*, usando os métodos de ajuste (*set*) dele. É necessário declarar o *bean* com `<jsp:useBean>` antes de ajustar uma propriedade. Estas duas ações trabalham juntas, portanto o nome de instância usada nas duas deve ser igual.

Sintaxe:

```
<jsp:setProperty
        name="nome de instância do bean"
    {   property="*" |
        property="nome da propriedade" [ param="nome do parâmetro" ] |
        property="nome da propriedade" value="{string | <%= expressão %>}"
    }
/>
```

2.2.2 Diretivas

São instruções processadas quando a página JSP é compilada em um servlet. Diretivas são usadas para ajustar instruções no nível da página, inserir dados de arquivos externos, e especificar tag libraries. Diretivas são definidas entre <%@ e %>.

Existem três tipos de diretivas:

Include

Inclui um arquivo estático em uma página JSP.

Sintaxe:

```
<%@ include file="relativeURL" %>
```

Page

Define atributos que são aplicados a todo o arquivo JSP, e a todos os seus arquivos incluídos estaticamente.

Sintaxe:

```
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true|false" ]
    [ buffer="none|8kb|sizekb" ]
    [ autoFlush="true|false" ]
    [ isThreadSafe="true|false" ]
    [ info="text" ]
    [ errorPage="URL relativa" ]
    [ contentType="mimeType [ ;charset=characterSet ]" |
        "text/html ; charset=ISO-8859-1" ]
    [ isErrorPage="true|false" ]
%>
```

Taglib

Define uma *tag library* e seu prefixo a ser usado na página JSP.

Sintaxe:

```
<%@ taglib uri="localização do descritor da tag"
     prefix="prefixo da tag" %>
```

2.2.3 Declarações

São similares com as declarações de variáveis em Java, e definem variáveis para uso subseqüente em expressões ou scriptlets. São definidas entre <%! e %>.

Sintaxe:

```
<%! int x = 0; declaração;  ... %>
```

2.2.4 Expressões

Contém um comando válido da linguagem Java que é avaliado, convertido para um String, e inserido onde a expressão aparece no arquivo JSP. Não é usado ponte e vírgula para terminar a expressão, e só pode haver uma entre <%= e %>.

Sintaxe:

```
<%= pessoa.getNome() %>
```

2.2.5 Scriptlets

São blocos de código Java embutidos numa página JSP. O código do scriptlet é inserido literalmente no servlet gerado pela página. É definido entre <% e %>.

Sintaxe:

```
<% int x = 0;
    x = 4 * 9;
    String str = "PET";
    ...
%>
```

2.2.6 Comentários

São similares aos comentários HTML, mas são tirados da página quando o arquivo JSP é compilado em servlet. Isto significa que os comentários JSP não aparecem no código fonte da página visualizada pelo navegador do usuário. Comentários em HTML são feitos entre <!– e –>, enquanto comentários em JSP são entre <%– e –%>.

2.3 Objetos implícitos

Como uma característica conveniente, o container JSP deixa disponível objetos implícitos que podem ser usados nos scriptlets e expressões, sem que o autor tenha que criá-los. Estes objetos instanciam classes definidas na API (application program interface) de Servlets. São nove objetos:

Tabela 2.1: Objetos Implícitos

Objeto	Classe ou Interface	Descrição
page	`javax.servlet.jsp.HttpJspPage`	Instância de servlet da página
config	`javax.servlet.ServletConfig`	Dados de configuração do Servlet
request	`javax.servlet.http.HttpServletRequest`	Dados de solicitação incluindo parâmetros
response	`javax.servlet.http.HttpServletResponse`	Dados da resposta
out	`javax.servlet.jsp.JspWriter`	Fluxo de saída para o conteúdo da página
session	`javax.servlet.http.HttpSession`	Dados de sessão específicos de usuário
application	`javax.servlet.ServletContext`	Dados compartilhados por todas as páginas da aplicação
pageContext	`javax.servlet.jsp.PageContext`	Dados de contexto para execução da página
exception	`javax.lang.Throwable`	Erros não pegos ou exceção

Objeto page

O objeto page representa a própria página JSP ou, mais especificamente, uma instância da classe de servlet na qual a página foi traduzida.

Objeto config

O objeto config armazena dados de configuração de servlet — na forma de parâmetros de inicialização — para o servlet no qual uma página JSP é

compilada. Pelo fato das páginas JSP raramente serem escritas para interagir com parâmetros de inicialização, este objeto implícito raramente é usado na prática.

Objeto request

O objeto request representa a solicitação que acionou o processamento da página atual. Para solicitações de HTTP, este objeto fornece acesso a todas as informações associadas com uma solicitação, incluindo sua fonte, a URL solicitada e quaisquer cabeçalhos, cookies ou parâmetros associados com a solicitação. Dentre os usos mais comuns para o objeto request, encontra-se a procura por valores de parâmetros e cookies.

Objeto response

O objeto response representa a resposta que será enviada de volta para o usuário como resultado do processamento da página JSP.

Objeto out

Este objeto implícito representa o fluxo de saída para a página, cujo conteúdo será enviado para o navegador como o corpo de sua resposta.

Objeto session

Este objeto implícito de JSP representa a sessão atual de um usuário individual. Todas as solicitações feitas por um usuário, que são parte de uma única série de interações com o servidor da web, são consideradas parte de uma sessão. Desde que novas solicitações por aqueles usuários continuem a ser recebidas pelo servidor, a sessão persiste. Se, no entanto, um certo período de tempo passar sem que qualquer nova solicitação do usuário seja recebida, a sessão expira.

O objeto `session`, então armazena informações a respeito da sessão. Os dados específicos de aplicação são tipicamente adicionados à sessão através de atributos, usando os métodos da interface `javax.servlet.http.HttpSession`.

O objeto session não está disponível para todas as páginas JSP, seu uso é restrito às páginas que participam do gerenciamento da sessão. Isto é indicado através do atributo *session* da diretiva *page*. O padrão é que todas as páginas participem do gerenciamento de sessão. Se o atributo estiver definido para *false*, o objeto não estará disponível e seu uso resultará em um erro de compilação quando o recipiente JSP tentar traduzir a página para um servlet.

Objeto application

Este objeto implícito representa a aplicação à qual a página JSP pertence. Ela é uma instância da interface *javax.servlet.ServletContext*. As páginas JSP estão agrupadas em aplicações de acordo com suas URLs.

Este objeto permite acessar informações do *container*, interagir com o servidor e fornece suporte para *logs*.

Objeto pageContext

O objeto pageContext é uma instância da classe *javax.servlet.jsp.PageContext*, e fornece acesso programático a todos os outros objetos implícitos. Para os objetos implícitos que aceitam atributos, o objeto pageContext também fornece métodos para acessar aqueles atributos. Além disso, o objeto pageContext implementa métodos para transferir controle da página atual para uma outra página, temporariamente, para gerar output a ser incluído no output da página atual, ou permanentemente para transferir todo o controle.

Objeto exception

O objeto exception é uma instância da classe *java.lang.Throwable*. O objeto exception não está automaticamente disponível em todas as páginas JSP. Ao invés disso, este objeto está disponível apenas nas páginas que tenham sido designadas como páginas de erro, usando o atributo *isErrorPage* da diretiva *page*.

2.4 Construindo a aplicação

Iniciando a construção da aplicação, vamos criar um formulário de cadastro para a mídia, um *bean* que representa a mídia, e um arquivo JSP que processa o formulário. Criaremos também um formulário idêntico ao primeiro mas que mostra possíveis erros ocorridos no cadastro. E por último, uma página HTML simples que indica se o cadastro foi realizado ou não.

2.4.1 Construindo o formulário

Aqui iremos criar o formulário para o cadastro de fitas. Ele é puramente feito em HTML, não tem código jsp.

```
<html>
<head>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
<title>Cadastro de M&iacute;dia</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
```

Especificamos o arquivo que processará o formulário no atributo *action* da *tag form*:
<form action="ProcessarMidia.jsp"... >

```
<form action="ProcessarMidia.jsp" method=post name="midias">
<center>
<table  cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCDDEE" colspan=2>
<font size=5>Cadastro de M&iacute;dias</font>
```

```
<br>
<font size=1><sup>*</sup> Campos necess&aacute;rios</font>
</th>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>T&iacute;tulo<sup>*</sup></b>
<br>
```

O atributo *name* dos campos do formulário devem preferencialmente ter o mesmo nome que seu atributo correspondente no *bean* usado para armazenar as informações do formulário.

```
<input type="text" name="titulo" value="" size=30 maxlength=40></td>
<td valign=top>
<b>Ano<sup>*</sup></b>
<br>
<input type="text" name="ano" value="" size=8 maxlength=8>
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Descrição
<br>
<TEXTAREA NAME="descricao" ROWS="6" COLS="30">

</textarea>
</td>
<td valign=top>
<b>Tipo<sup>*</sup></b></b>
<br>
<input type="radio" name="tipo" value="VHS" checked> VHS   
<br>
<input type="radio" name="tipo" value="DVD"> DVD   
</td>
</tr>
<tr bgcolor="#F7F7F7">
<td align=center colspan=2>
<input type="submit" value="Confirma">
<input type="reset" value="Limpar">
</td>
</tr>


</table>
</center>
</form>
</body>
</html>
```

2.4.2 Criando um JavaBean

Vamos criar nosso *bean* que terá as informações da Mídia. Ele deve ter todos os atributos do formulário que construímos, e todos os métodos *set* e *get* para acessá-los. Além disso, teremos outro atributo, para controle

de eventuais erros no processamento do formulário (dados necessários não digitados por exemplo). Usaremos uma tabela de hash (*java.util.Hashtable*) que terá como chave o nome do atributo e como valor a mensagem de erro que desejarmos. Chamaremos a Hashtable de *erros*, e ela também terá seus métodos *set* e *get*.
Por fim, criaremos um método para verificar se todos os dados necessários foram digitados, validando ou não o formulário.

Abaixo está o código fonte do BeanMidia, veja os comentários para maiores detalhes.

```
package beans;

import java.util.Hashtable;
/**
* Implementar a classe Serializable é requisito para ser um
* Enterprise Bean. Um objeto de uma classe que implementa
* esta interface pode ser escrito em disco ou enviado pela rede.
* Na aplicação do curso não fará diferença.
*/
public class BeanMidia implements java.io.Serializable {


    /**
    * Nomes dos atributos preferencialmente iguais aos usados
    * no formulário
    */
    private String titulo;
    private String ano;
    private String tipo;
    private String descricao;
    /* Este atributo serve para o controle de erros no formulário */
    private Hashtable erros;

    public BeanMidia() {
        /* Iniciamos os atributos com o String nulo */
        titulo = "";
        ano = "";
        descricao = "";
        tipo = "";
        erros = new Hashtable();
    }

    /**
    * Métodos para acessar os atributos.
    * getNome() para ver ser valor, e setNome() para ajustar seu valor
    */

    public String getTitulo() {
        return titulo;
    }

    public String getAno() {
        return ano;
    }

    public String getTipo() {
        return tipo;
```

```
}

public String getDescricao() {
    return descricao;
}

public void setTitulo(String valor) {
    titulo = valor;
}

public void setAno(String valor) {
    ano = valor;
}

public void setTipo(String valor) {
    tipo = valor;
}

public void setDescricao(String valor) {
    descricao = valor;
}


/**
* Verifica se todos os dados exigidos foram digitados,
* além de outras condições desejadas.
*/
public boolean ehValido() {
    boolean volta = true;

    if ((titulo == null) || titulo.equals(""))
    {
        erros.put("titulo", "Por favor, digite um título.");
        volta = false;
    }

    if ((ano == null) || ano.equals("") )
    {
        erros.put("ano", "Por favor, digite o ano da mídia .");
        volta = false;
    }
    if ((tipo == null) || tipo.equals("") )
    {
        erros.put("tipo", "Por favor, selecione o tipo da mídia .");
        volta = false;
    }

    return volta;
}

/**
*Usado para ver as mensagens de erro armazenados na tabela de Hash
*/
public String getErros (String s)
{
    String msg = (String) erros.get(s);
    return (msg == null) ? "" : msg;
}
```

```
    /**
    * Usado para colocar algum erro na tabela
    */
    public void setErros (String chave, String msg)
    {
        erros.put(chave,msg);
    }

}
```

2.4.3 Primeiro arquivo JSP

Agora vamos construir o arquivo que processa o formulário, este, até que enfim, feito em JSP.

O atributo *import* da diretiva *page* estende o conjunto de classes Java que podem ser referenciadas um uma página JSP sem ter que explicitamente especificar os nomes de pacote de classes.

```
<%@ page import="beans.BeanMidia" %>
```

A tag <jsp:useBean> diz à página que você quer disponibilizar um Bean para a página.

- O atributo *id* especifica um nome para o Bean, que será o nome usado para referenciar o Bean ao longo da página e de sua vida na aplicação.
- O atributo *class* especifica o nome de classe do próprio Bean.
- O atributo *scope* controla o escopo do Bean, ou seja, para quem e quanto tempo ele permanecerá disponível. Os valores podem ser *page, request, session e application.*

```
<jsp:useBean id="midia" class="beans.BeanMidia" scope="request">
```

Mostramos abaixo três maneiras de ajustar os atributos de um *bean.* A primeira diz-se o nome do atributo do *bean* no parâmetro *property* e então especifica-se o valor no parâmetro *value.*

Nas outras duas o atributo do *bean* tem o mesmo nome do valor colocado em *property.*

Pode-se também ajustar os valores dos atributos usando os métodos *set* do *bean*, usando o ambiente dos *scriptlets.*

```
<jsp:setProperty name="midia" property="titulo"
          value='<%=request.getParameter("titulo")%>' />
<jsp:setProperty name="midia" property="ano" />
<jsp:setProperty name="midia" property="descricao" />
<%
    midia.setTipo(request.getParameter("tipo"));
%>
</jsp:useBean>
```

Verifica-se se todos os atributos necessários foram digitados e então redireciona-se para a página de sucesso ou para o formulário que trata erros.

```
<%
if (midia.ehValido()) {
%>
<jsp:forward page="/sucesso.jsp"/>
<%
}  else {
%>
<jsp:forward page="RetryMidia.jsp" />
<%
}
%>
```

2.4.4 Formulário para tratar erros

Este arquivo é visualmente idêntico ao formulário anterior, mas ele é um arquivo JSP que mostra possíveis erros ocorridos durante o cadastro.

Declaramos o *bean* usado no arquivo nas primeiras linhas. É necessário que ele tenha o mesmo nome dos arquivos antecessores (no caso, ProcessarMidia.jsp).

```
<%@ page import="beans.BeanMidia" %>
<jsp:useBean id="midia" class="beans.BeanMidia" scope="request"/>

<html>
<head>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
<title>Cadastro de M&iacute;dia</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<form action="ProcessarMidia.jsp" method=post name="midias">
<center>
<table  cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCDDEE" colspan=2>
<font size=5>Cadastro de M&iacute;dias</font>
<br>
<font size=1><sup>*</sup> Campos necess&aacute;rios</font>
</th>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>T&iacute;tulo<sup>*</sup></b>
<br>
```

Colocamos em *value* o que tem no atributo *titulo* do *BeanMidia*, para que o formulário fique igual ao que foi digitado anteriormente. Isto poupa trabalho ao usuário que não precisa digitar tudo novamente.

```
<input type="text" name="titulo" value='<%= midia.getTitulo() %>'
      size=30 maxlength=40><br>
```

Se houve algum erro na digitação do título, o comando *midia.getErros("titulo")* retornará o *String* de erro que colocamos na tabela de hash associado a chave *"titulo"*, caso contrário não retornará nada e não aparecerá nada na página.

```
<font color=#FF0000><%=midia.getErros("titulo")%></font>
</td>
<td valign=top>
<b>Ano<sup>*</sup></b>
<br>
<input type="text" name="ano" value='<%= midia.getAno() %>'
        size=8 maxlength=8><br>
<font color=#FF0000><%=midia.getErros("ano")%></font>
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Descrição
<br>
<TEXTAREA NAME="descricao" ROWS="6" COLS="30">
<%= midia.getDescricao() %>
</textarea><br>
<font color=#FF0000><%=midia.getErros("descricao")%></font>
</td>
<td valign=top>
<b>Tipo<sup>*</sup></b></b>
<br>
```

O comando *midia.getTipo().equals(String)* verifica se o valor do atributo tipo é igual ao String. Se for, ele imprime *checked* no código fonte da página, fazendo com que o *checkbox* fique marcado. Isto serve, novamente, para deixar o formulário como o usuário digitou anteriormente.

```
<input type="radio" name="tipo" value="VHS"
<% if (midia.getTipo() != null && midia.getTipo().equals("VHS"))
    out.print("checked");
%>
> VHS   
<br>
<input type="radio" name="tipo" value="DVD"
<% if (midia.getTipo() != null && midia.getTipo().equals("DVD"))
    out.print("checked");
%>

> DVD   <br>
<font color=#FF0000><%=midia.getErros("tipo")%></font>
</td>
</tr>
<tr bgcolor="#F7F7F7">
<td align=center colspan=2>
<input type="submit" value="Confirma">
```

```
<input type="reset" value="Limpar">
</td>
</tr>


</table>
</center>
</form>
</body>
</html>
```

2.4.5 Página de sucesso

Vamos fazer uma página que nos indica se a operação foi realizada com sucesso. Sempre que houve êxito, redirecionaremos a aplicação para esta página.

```
<html>
<head>
<title>Sucesso</title>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<table align="center" border="0" cellspacing="2" cellpadding="2">
<tr>
<td bgcolor="#CCDDEE" align=center>
<h3>Sucesso</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
Opera&ccedil;&atilde;o realizada com sucesso.
</td>
</tr>
<td>
</table>

</body>
</html>
```

2.4.6 Disponibilizando a aplicação

Crie um diretório chamado *locadora* no diretório *webapps* do Tomcat. Dentro do diretório *locadora*, crie um diretório *cadastro* e coloque os arquivos *ProcessarMidia.jsp, CadastroMidia.html e RetryMidia.jsp.*

No diretório raiz da aplicação (*../locadora*) coloque o arquivo sucesso.jsp, e no diretório *WEB-INF/classes/beans* coloque os arquivos *BeanMidia.java e BeanMidia.class*

Agora, inicie o Tomcat, acesse no navegador a URL *http://localhost:8080/locadora* e teste a aplicação.

Capítulo 3

Release 2

3.1 Servlets

3.1.1 O que são servlets?

Servlets são programas simples feitos em Java os quais rodam em um *Servlet Container*. Um *Recipiente (Container) Servlet* é como um servidor Web que trata requisições do usuário e gera respostas. Recipiente Servlet é diferente de Servidor Web porque ele é feito somente para Servlets e não para outros arquivos (como .html etc). O Recipiente Servlet é responsável por manter o ciclo de vida do Servlet. Pode ser usado sozinho (*standalone*) ou conjugado com um servidor Web. Exemplo de Recipiente Servlet é o Tomcat, e de servidor Web o Apache.

Servlets são na verdade simples classes Java as quais necessitam implementar a interface *javax.servlet.Servlet*. Esta interface contém cinco métodos que precisam ser implementados. Na maioria das vezes você não precisa implementar esta interface. Por quê? Porque o pacote *javax.servlet* já provê duas classes que implementam esta interface i.e. *GenericServlet* e *HttpServlet*. Então tudo que você precisa fazer é estender uma dessas classes e sobrescrever o método que você precisa para seu Servlet. *GenericServlet* é uma classe muito simples que somente implementa a interface *javax.servlet.Servlet* e fornece apenas funcionalidades básicas. Enquanto *HttpServlet* é uma classe mais útil que fornece métodos para trabalhar com o protocolo HTTP. Assim se seu servlet usa o protocolo HTTP (o que ocorrerá na maioria dos casos) então ele deveria estender a classe *javax.servlet.http.HttpServlet* para construir um servlet e isto é o que faremos na nossa aplicação.

Servlets uma vez iniciados são mantidos na memória. Então toda requisição que chega, vai para o Servlet na memória e este gera uma resposta. Esta característica de "manter na memória" faz com que usar Java Servlets seja um método extremamente rápido e eficiente para construir aplicações Web.

Não será escrito nenhum Servlet neste curso.

3.2 Usando Java JDBC

JDBC (Java DataBase Connectivity) é a API Java para comunicação com bancos de dados. JDBC especifica a interface necessária para conectar a um banco de dados, executar comandos SQL e *queries*, e interpretar resultados. Esta interface é independente do banco de dados e da plataforma.

As classes JDBC estão no pacote *java.sql*, que precisa ser importado para a classe Java que faz uso de alguma classe desta API.

3.2.1 Acesso ao banco de dados

A API JDBC não pode comunicar-se diretamente com o banco de dados. O acesso é fornecido por um driver JDBC, que é uma classe específica para o banco de dados e implementa a interface definida pela API. Este driver é fornecido pelo vendedor do banco de dados ou por outro provedor de serviço.

3.2.2 Especificando o driver JDBC

No seu programa, você tem que especificar o driver JDBC que vai usar. Faz-se isso com uma simples, talvez estranha, operação:

`Class.forName(JDBC-driver-class-name)`

O método `Class.forName()` faz com que a JVM carregue o driver na memória. Você pode pensar que tem que instanciar a classe do driver e usar a referência a ele para acessar o banco de dados, mas JDBC não funciona desta maneira. Quando o driver é carregado na memória da JVM, ele se registra com a classe *java.sql.DriverManager*. Então você usa métodos estáticos da classe *DriverManager* para obter referência ao objeto *Connection*, que então fornecerá acesso ao banco de dados.

3.2.3 Estabelecendo a conexão

JDBC segue o paradigma de usar uma URL para conectar a um recurso. O exato formato da URL depende do driver usado. Mas normalmente a URL segue um formato parecido com este:

`jdbc:driver-id://host/database`

ou

`jdbc:driver-id:database-id`

Exemplo de como conectar a um banco de dados MySQL (usado no curso):

```
Connection connection;
try {
        Class.forName("org.gjt.mm.mysql.Driver");
        connection = DriverManager.getConnection(
        "jdbc:mysql://servidor/database", user , password );
    } catch (ClassNotFoundException ex) {
        System.out.println("Não foi possível encontrar
                            a classe do Driver do MySQL");
    } catch (SQLException ex) {
        System.out.println("Não foi possível conectar
                            ao servidor");
    } finally {
```

```
            try { if (connection != null) connection.close(); }
            catch (SQLException e) { }
        }
```

3.2.4 Encontrando dados em uma tabela

Antes de poder fazer consultas ao banco de dados com JDBC, você precisa criar um objeto *Statement*. Um objeto *Statement* pode ser reutilizado através de múltiplas requisições SQL. É criado pelo objeto *Connection* e é portanto dependente do driver que você está usando para conectar ao banco de dados. É muito simples, como você pode ver:

```
Statement statement = connection.createStatement();
```

3.2.5 Fazendo consultas com JDBC

Fazer uma consulta com JDBC é fácil uma vez que você tenha o objeto *Statement*: você simplesmente passa a consulta como parâmetro do método *executeQuery()*. A tabela de resultados é retornada pelo método como uma instância da classe *ResultSet* da API de JDBC. Abaixo está um exemplo de consulta:

```
String query = ‘‘SELECT * FROM TABELA’’;
ResultSet resultset = statement.executeQuery(query);
```

Estes métodos lançam exceções, portanto no seu programa deve estar dentro de um bloco *try/catch*.

O objeto *ResultSet* revela o conteúdo da tabela uma linha por vez e fornece métodos para acessar dados de cada coluna da tabela da linha corrente, indicada por um cursor interno. Quando o *ResultSet* é criado, o cursor de linha não está na primeira linha, mas sim antes dela. Chamando o método *next()*, o *ResultSet* avança o cursor em uma linha, retornando valor verdade se a chamada foi bem sucedida. Por exemplo, para passar por todas linhas do *ResultSet* faríamos o seguinte:

```
while (resultset.next()) {
    System.out.println ("Mais uma linha!");
}
```

Quando o cursor está posicionado em uma linha que você gostaria de examinar, você pode chamar vários métodos suportados pelo objeto *ResultSet* para restaurar dados das colunas e obter informações sobre os resultados.

3.3 Interface *RequestDispatcher*

Esta interface está presente no pacote *javax.servlet* e contém dois métodos:

- **forward(ServletRequest request, ServletResponse response)** transfere uma requisição para outro recurso no mesmo servidor. Este recurso pode ser um *servlet*, uma página JSP ou uma simples página HTML.
- **include(ServletRequest request, ServletResponse response)** atua no lado do servidor, inclui a resposta do recurso dado (Servlet, página JSP ou HTML) dentro da resposta de quem chamou o recurso.

Para obter uma referência a interface *RequestDispatcher* temos duas formas:

- *ServletContext.getRequestDispatcher(String resource)*
- *ServletRequest.getRequestDispatcher(String resource)*

Se o servlet é subclasse de *HttpServletRequest*, simplesmente chama-se o método *getRequestDispatcher(String resource)* para pegar uma referência ao objeto *RequestDispatcher*.

```
RequestDispatcher rd;
rd = request.getRequestDispatcher("caminho do recurso");
rd.forward(request, response);
```

Se não for subclasse, utilize a outra maneira.

```
rd = getServletContext().getRequestDispatcher("caminho do recurso");
rd.forward(request, response);
```

3.4 Construindo a aplicação

3.4.1 Escrevendo uma classe que conecta a um banco de dados

Na aplicação que estamos construindo neste curso, teremos que inserir e restaurar dados do bancos de dados diversas vezes. Para simplificar nossa vida, criaremos uma classe que conecta ao banco de dados, e tem métodos para acessá-lo. Assim para fazer a conexão bastará instanciar um objeto desta classe.

Já no construtor da classe faremos a conexão usando os métodos descritos na parte de JDBC desta apostila.

Para os outros métodos, leia os comentários do arquivo para maiores detalhes.

```
package conexao;

import java.sql.*;
```

```
public class Conexao {


    private Connection connection;
    private Statement statement;

    public Conexao (String servidor,String database,String user,
                    String password) throws SQLException {
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            connection = DriverManager.getConnection(
                "jdbc:mysql://"+servidor+"/"+database,user,password);
        } catch (ClassNotFoundException ex) {
            System.out.println("Não foi possível encontrar a classe do "+
            "Driver do MySQL");

        } catch (SQLException ex) {
            System.out.println("Não foi possível conectar ao servidor");

            throw ex;
        }
        try {
            statement = connection.createStatement();
        } catch (SQLException ex) {
            System.out.println("Não foi possível criar a statement");

            throw ex;
        }
    }

    public Conexao()
    {
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            connection = DriverManager.getConnection("jdbc:mysql://"+
            "/locadora", "", "");
        } catch (ClassNotFoundException ex) {
            System.out.println("Não foi possível encontrar a classe "+
            "do Driver do MySQL");

        } catch (SQLException ex) {
            System.out.println("Não foi possível conectar ao servidor");


        }
        try {
            statement = connection.createStatement();
        } catch (SQLException ex) {
            System.out.println("Não foi possível criar a statement");


        }
    }


    /**
    * Executa um update na base de dados
    * @param update String SQL a ser executado
    * @throws SQLException se não for possível executar
```

```
* o update (Erro de SQL).
*/
public synchronized void executeUpdate(String update)
    throws SQLException {
    try {
        statement.executeUpdate(update);
    } catch (SQLException ex) {
        System.out.println("Não foi possível executar o update");
        throw ex;
    }
}

/**
* Executa uma consulta na base de dados
* @param query String SQL a ser executado
* @return Um objeto do tipo ResultSet contendo o
* resultado da query
* @throws SQLException se não for possível executar a query
* (Erro de SQL).
*/
public synchronized ResultSet executeQuery(String query)
    throws SQLException {
    try {
        return statement.executeQuery(query);
    } catch (SQLException ex) {
        System.out.println("Não foi possível executar a query");
        throw ex;
    }
}

/**
* Fecha a conexao com a base de dados.
*/
public void fecharConexao()
{
    try {
        statement.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}


/**
* Retorna o maior número de um campo da tabela se ele
* for um inteiro. Assim, não teremos nenhum ID igual a outro.
*/
public int retornaIDMax(String tabela)
{
    try
    {
        String sql="SELECT max(ID) as contador FROM "+tabela;
        ResultSet rs = this.executeQuery(sql);
        rs.next();
        return rs.getInt("contador")+1;
    } catch (SQLException e)
    {
        System.out.println("Erro na seleção do ID Máximo");
        e.printStackTrace();
```

```
            return 0;
        }
    }


}
```

3.4.2 Criando tabelas

Vamos escrever uma classe que quando executada, conecta a um banco de dados e cria a tabela *Midias*, que vai ser usada no curso.

Basta conectar no banco de dados, criando uma instância da classe Conexao, e então executar um comando SQL usando o método *executeUpdate(String)*.

```
package conexao;

import java.sql.*;

public class CriaTabelas
{

   public static void main(String[] args) {

      Conexao con = new Conexao();
      ResultSet rs = null;

      try {


    con.executeUpdate("create table Midias "+
      "(ID int primary key not null, "+
      "Titulo varchar(50) not null, Ano char(4) not null, "+
      "Descricao text, Tipo enum ('VHS' , 'DVD') not null, "+
      "unique(Titulo))");

    con.executeUpdate("create table Pessoas "+
      "(ID int primary key not null, "+
      "Nome varchar(50) not null, Email varchar(50) not null, "+
      "Endereco text, Cidade varchar(30) not null, "+
      "Bairro varchar(40) not null, Telefone varchar(15), "+
      "RG varchar(20) not null, unique(Email))");
    } catch (SQLException e)    {
        System.out.println("Erro: nao foi possivel criar tabela "+
        Midias.\n "+ e.getMessage());
    }


    }

}
```

Para criar as tabelas, basta executar a classe CriaTabelas.

3.4.3 Cadastrando os dados no banco de dados

Após verificar se o formulário foi digitado corretamente (no arquivo *ProcessarMidia.jsp*), vamos redirecionar a aplicação para um arquivo que vai cadastrar a mídia no banco de dados e redirecionar a aplicação para a página de sucesso. Modifique o arquivo *ProcessarMidia.jsp* dando um *forward* para o arquivo *CadMidia.jsp* em vez de *sucesso.jsp*.

```
<%@ page import="beans.BeanMidia, java.sql.*, conexao.*" %>

<%! String pagina = null; %>

<%

BeanMidia midia = (BeanMidia) request.getAttribute("midia");
Conexao con = null;
try
{
        con = new Conexao();


        con.executeUpdate("insert into Midias values ("+con.retornaIDMax("Midias")+
                ", '"+midia.getTitulo()+"' , '"+midia.getAno()+
                "' , '"+midia.getDescricao()+"' , '"+
                midia.getTipo()+"')");


        pagina = "/sucesso.jsp";


} catch (SQLException ex) {
        if (ex.getErrorCode() == 1062) {
                midia.setErros("titulo", "Título já existe.");
                pagina = "/cadastro/RetryMidia.jsp";
        }
        System.out.println(ex);
} finally {
        if (con != null)
                con.fecharConexao();
        con = null;
}



        if (pagina == null)
                pagina = "/erro.jsp";
%>

<jsp:forward page='<%= pagina %>' />
```

3.4.4 Página de erro

No arquivo CadMidia.jsp, há referência a página */erro.jsp*. Vamos criá-la agora. Veja o código:

O atributo *isErrorPage* da diretiva *page* é usado para marcar uma página JSP que serve como a página de erro para uma ou mais páginas. Como resultado disso, a página pode acessar o objeto implícito *exception*.
Já que a maioria das páginas JSP não serve como páginas de erro, o valor padrão para este atributo é false.

```
<%@ page isErrorPage="true" %>

<html>
<head>
<title>Erro!</title>
</head>
<body>

<table align="center" border="0" cellspacing="2"
            cellpadding="2" width="70%">
<tr>
<td bgcolor="#CCDDEE" align=center>
<h3>Erro</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
Algum erro inesperado aconteceu.<br>

</td>
</tr>
<tr>
<td bgcolor="#F0F0F0" align=left>

<li>Pode haver algum erro de conex&atilde;o, tente novamente
    mais tarde</li>
<li>Contate o administrador para depura&ccedil;&atilde;o
    do erro.</li>
</td>
</tr>

</table>

</body>
</html>
```

Capítulo 4

Release 3

4.1 *Tag Libraries*

4.1.1 O que é uma *Tag Library*?

Na tecnologia JSP, ações são elementos que podem criar e acessar objetos da linguagem de programação e afetar a saída de dados. A especificação de JSP define seis ações padrão. JSP permite desenvolver módulos reutilizáveis, chamados de ações personalizadas (*custom actions*). Uma *custom action* é invocada usando uma tag personalizada (*custom tag*) em uma página JSP. Uma *Tag Library* é uma coleção de *custom tags*.

Alguns exemplos de tarefas que podem ser feitas por *custom actions* incluem processamento de formulários, acesso a banco de dados e outros serviços. A vantagem de usar *Tag Libraries* em vez de outros métodos, como *JavaBeans* associados a *scriptlets*, é que as *Tags* tornam-se mais simples de utilizar (principalmente para programadores HTML que não conhecem Java) e são reutilizáveis.

Algumas características das *custom tags* são:

- Podem ser personalizadas através de atributos passados pela página que solicitou a ação.
- Tem acesso a todos os objetos implícitos.
- Podem modificar a resposta gerada pela página que solicitou a ação.
- Podem se comunicar com outras *tags*. Pode-se criar e inicializar *JavaBeans*, criar variáveis que referenciam o *bean* em uma *tag*, e então usar o *bean* em outra *tag*.
- Podem estar aninhadas dentro de outras, permitindo interações complexas dentro de uma página JSP.

4.1.2 O que é uma *Tag*?

Uma *Tag* é uma classe que implementa a interface *javax.servlet.jsp.tagext.Tag*. É usada para encapsular funcionalidades que podem ser usadas em uma página JSP.

Uma *Tag* pode ser de dois tipos (há uma terceira na versão 1.2 de JSP): *BodyTag* ou *Tag*. A diferença básica é que o corpo de uma *Tag* é avaliado apenas uma vez enquanto o corpo da *BodyTag* pode ser avaliado várias vezes.

Quando uma *Tag* é encontrada, são executados os seguintes métodos na seqüência:

1. **setPageContext()**: para ajustar o atributo PageContext.
2. **setParent()**: para ajustar alguma superclasse (null se nenhuma).
3. Ajustar os atributos, executando os métodos set de cada um.
4. **doStartTag()**: que inicia a tag.
5. **doEndTag()**: termina a tag.
6. **release()**: para liberar quaisquer recursos necessários.

Existe a classe TagSupport, que implementa a interface *Tag*, e pode ser estendida para criar-se uma *Tag*.

A interface *BodyTag* estende a interface *Tag* e nos fornece alguns métodos novos. São eles (na seqüência de execução):

1. **setPageContext()**: para ajustar o atributo PageContext.
2. **setParent()**: para ajustar alguma superclasse (null se nenhuma).
3. Ajustar os atributos, executando os métodos set de cada um.
4. **doStartTag()**: que inicia a tag.
5. **setBodyContent()**: ajusta a conteúdo do corpo da tag.
6. **doInitTag()** aqui colocam-se instruções que devem ser executadas antes de avaliar o corpo da tag.
7. **doAfterBody()** é executada após a avaliação do corpo. Seu valor de retorno é importante. Se for *SKIP_BODY* ele não avalia o corpo novamente, e se for EVAL_BODY_BUFFERED ou EVAL_BODY_AGAIN ele faz outra iteração.
8. **doEndTag()**: termina a tag.
9. **release()**: para liberar quaisquer recursos necessários.

Para criar uma *BodyTag*, basta que a classe implemente a interface *BodyTag*.

4.1.3 *Tag Handlers*?

Um tratador de tags (*tag handler*) é o objeto invocado por um recipiente JSP (*JSP Container*) para avaliar uma *custom tag* durante a execução da página JSP que referencia a *tag*. Métodos do *tag handler* são chamados pela classe que implementa a página JSP nos vários pontos que a *tag* é encontrada.

4.1.4 Descritor de *Tag Libraries*

Um *tag library descriptor (TLD)* é um documento XML que descreve a *tag library*. Um TLD contém informações sobre a biblioteca (*library*) como um todo e sobre cada *tag* contida na biblioteca. TLDs são usados pelo recipiente JSP para validar as *tags* e por ferramentas de desenvolvimento JSP. Os seguintes elementos são necessários para definir uma *Tag Library*:

```
<taglib>

<tlibversion>A versão da biblioteca </tlibversion>

<jspversion> A versão da especificação JSP usada </jspversion>

<shortname> Um nome para a tag library </shortname>

<uri> Uma URI que identifica unicamente a tag library </uri>

<info> Informações sobre a tag library </info>

<tag> ... </tag>
...
</taglib>
```

4.2 Construindo a aplicação

Vamos escrever uma tag que mostra todas as mídias contidas no banco de dados. Para isso, vamos fazer uma consulta e depois restaurar cada linha do resultado em uma linha de uma tabela HTML na página JSP. Para adicionar novos elementos dinamicamente e torná-los disponíveis para outras ações (análogo a definição de váriaveis de script) na mesma página, o Recipiente permite especificar quais elementos e quanto tempo eles estarão disponíveis. Estas informações são possíveis com a classe *javax.servlet.jsp.tagext.TagExtraInfo*.

4.2.1 Escrevendo uma *BodyTag*

Abaixo está a classe *TagMidia*, que implementa a interface *javax.servlet.jsp.tagext.Bodytag*.

```
package tags;

import conexao.Conexao;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;


public final class TagMidia implements BodyTag {
```

```
private PageContext pc = null;
private BodyContent body = null;
private StringBuffer sb = new StringBuffer();
private ResultSet rs = null;
private Conexao con = null;

public void setPageContext(PageContext p)
{
    pc = p;
}

public void setParent(Tag t) {}

public Tag getParent()
{
    return null;
}
```

No método *doStartTag()* faremos a consulta no banco de dados e armazenaremos o resultado no objeto *rs* e chamaremos o método *setVariaveis* para ajustar as variáveis para a primeira passada pelo corpo da tag. A partir daí, quem vai renovar os valores das variáveis é o método *doAfterBody()*, que vai ser executado tantas vezes quantas forem o número de linhas obtidos com a consulta SQL.

```
public int doStartTag() throws JspException
{
    try {
        con = new Conexao();

        rs = con.executeQuery("SELECT * FROM Midias order by Titulo");

        setVariaveis();
    } catch (SQLException e) {
        System.out.println(e);
    }
    if (pc.getAttribute("titulo") == null)
        return SKIP_BODY;


    return EVAL_BODY_BUFFERED;
}

public void setBodyContent(BodyContent b)
{
    body = b;
}

public void doInitBody() throws JspException {}

private boolean setVariaveis() throws JspTagException {
    try {
        if (rs.next()) {
```

Aqui vamos ajustar as variáveis que vamos definir na classe *TagExtraInfo*.

Armazena-se no objeto implícito page (*PageContext pc*, no caso desta classe) estas variáveis que serão usadas na página JSP.

```
                pc.setAttribute("titulo", rs.getString("Titulo"));
                pc.setAttribute("ano", rs.getString("Ano"));
                pc.setAttribute("descricao", rs.getString("Descricao"));
                pc.setAttribute("tipo", rs.getString("Tipo"));
                pc.setAttribute("ID", rs.getString("ID"));
                return true;
            } else {
                return false;
            }
        } catch (SQLException e) {
            System.out.println(e);
            return false;
        }
    }

    public int doAfterBody() throws JspException {
        try {
            sb.append(body.getString());
            body.clear();
        } catch (IOException e) {
            throw new JspTagException("Erro fatal: IOException!");
        }
        if(setVariaveis()) {
            return EVAL_BODY_AGAIN;
        }
        try {
            body.getEnclosingWriter().write(sb.toString());
        } catch (IOException e) {
            throw new JspTagException("Erro fatal: IOException!");
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        try {
            if(rs != null) {
                rs.close();
                rs = null;
            }
            if (con != null)
            {
                con.fecharConexao();
                con = null;
            }
        } catch (SQLException e) {}
        return EVAL_PAGE;
    }

    public void release() {
        pc = null;
        body = null;
        sb = null;
    }

}
```

4.2.2 *TagExtraInfo*

Precisamos definir as variáveis que colocamos na classe TagMidia (titulo, ano, descricao, tipo e ID).

```
package tags;

import javax.servlet.jsp.tagext.*;

public class TagTEIMidia extends TagExtraInfo {

    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[] {
```

O construtor da classe *VariableInfo* aceita quatro argumentos:

1. O nome do atributo.
2. A classe do atributo (repare que é um *String*, está entre aspas).
3. Se é uma nova variável (valor verdade) ou uma já existente na página (valor falso).
4. O escopo da variável, onde: NESTED vale da tag de início a tag final, AT_BEGIN vale da tag de início até o fim da página JSP e AT_END vale da tag final até o fim da página JSP.

```
        new VariableInfo("titulo", "java.lang.String", true,
            VariableInfo.NESTED),
        new VariableInfo("ano", "java.lang.String", true,
            VariableInfo.NESTED),
        new VariableInfo("tipo", "java.lang.String", true,
            VariableInfo.NESTED),
        new VariableInfo("ID", "java.lang.String", true,
            VariableInfo.NESTED),
        new VariableInfo("descricao", "java.lang.String", true,
            VariableInfo.NESTED)

        };
    }
}
```

4.2.3 Descritor

Vamos agora criar o *TagLib Descriptor*. Crie o diretório *tld* dentro do diretório *WEB-INF* e nele crie o arquivo TagLib.tld com o seguinte conteúdo:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems,
    Inc.//DTD JSP Tag Library 1.1//EN"
```

```
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>taglocadora</shortname>
<info>Tags da Locadora</info>

<tag>
<name>mostremidias</name>
<tagclass>tags.TagMidia</tagclass>
<teiclass>tags.TagTEIMidia</teiclass>
<bodycontent>JSP</bodycontent>
<info>Tag que mostra o conteúdo do BD</info>
</tag>

</taglib>
```

Dentro de *<tag>*, *name* é nome que vai ser utilizado para chamar esta tag na página JSP, *tagclass* é a localização da classe (dentro do diretório *WEB-INF/classes*), e *teiclass* é a localização da classe *TagExtraInfo* (se for usada).

4.2.4 Página JSP

Vamos escrever uma página JSP que usa a *TagMidia* para listar todo conteúdo do banco de dados. Dê o nome de titulos.jsp e coloque na raiz da aplicação. Veja o código:

```
<%@ taglib uri="/WEB-INF/tld/TagLib.tld" prefix="todos" %>

<html>
<head>
<title>Todos os títulos</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">

<table align="center" border="0" width="90%"
            cellspacing="2" cellpadding="4">
<tr>
<th bgcolor="#CCDDEE" colspan=2>
Listagem das Mídias
</th>
</tr>
```

Aqui está o início da tag. Coloque o prefixo que você definiu, e o nome da tag que quer usar (este nome está no arquivo TagLib.tld).
Entre a tag de início e a tag final, pode-se usar livremente os atributos definidos na classe *TagExtraInfo*.

```
<todos:mostremidias>
<tr bgcolor="#E3E3E3">
<td align=center>
```

```
<b>Mídia:</b> <%= titulo %>
</td>
<td align=center>
<b>Ano:</b> <%= ano %>
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td align=center>
<%= descricao %>
</td>
<td align=center>
<%= tipo %>
</td>
</tr>
<tr>
<td colspan="2" bgcolor="#FFFFFF">
 
</td>
</tr>
</todos:mostremidias>
</table>

</body>
</html>
```

Capítulo 5

Release 4

5.1 Tags com parâmetros

Para criar uma tag com parâmetros, basta especificar os atributos na classe da Tag, e então escrever os métodos *set* e *get* para poder acessá-los. Também há modificações no descritor da tag.
Vamos construir a aplicação para melhor ilustrar esta situação.

5.2 Construindo a aplicação

Para mostrar a utilidade das tags com parâmetro, vamos fazer uma página de pesquisa de mídias, onde o usuário digita algum texto, procura-se no banco de dados este texto e mostra-se o resultado da consulta na tela.

5.2.1 Página de pesquisa

Vamos construir um arquivo chamado pesquisa.jsp, que procura por mídias no banco de dados:

```
<%@ taglib uri="/WEB-INF/tld/TagLib.tld" prefix="pesquisa" %>

<html>
<body>
<center>

<table align=center width="50%">
```

Repare que o arquivo que processa o formulário é o próprio arquivo pesquisa.jsp.

```
<form method="post" action="pesquisa.jsp">
<th bgcolor="#CCDDEE" colspan=3>
Digite a(s) palavra(s) chave(s) para a pesquisa
</th>
<tr bgcolor="#f7f7f7">
   <td align=center>Pesquisa</td>
   <td align=center>
    <input type="text" name="chave" size=30>
```

```
      
    <input type=submit value="Pesquisar >>">   
    <input type=reset value="Limpar">
   </td>
</tr>
</form>
</table>
```

Quando não for digitado nada no campo de texto ou for a primeira vez que a página é acessada, o comando *request.getParameter("chave")* retornará valor nulo.
Logo abaixo colocamos um comando de condição que só executa o que há dentro das chaves se o usuário digitou alguma chave de pesquisa.

```
<%
    String tem = (String) request.getParameter("chave");
    if (tem != null && !tem.equals(""))
    {
%>
<br><br>
<table align=center width="70%">
<th bgcolor="#CCDDEE" colspan=3 align=center>
Mídias encontradas
</th>
```

Aqui está a tag com parâmetro. Coloca-se o nome do atributo e a expressão ou valor associado a ele.

```
<pesquisa:mostremidias chave='<%= tem %>' >

<tr bgcolor="#E3E3E3">
<td align=center>
<b>Mídia:</b> <%= titulo %>
</td>
<td align=center>
<b>Ano:</b> <%= ano %>
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td align=center>
<%= descricao %>
</td>
<td align=center>
<%= tipo %>
</td> </tr> <tr> <td colspan="2" bgcolor="#FFFFFF">   </td>
</tr>

</pesquisa:mostremidias>
</table>
```

Este fechamento de chaves é relativo ao comando *if(tem != null ...)*.

```
<% } %>
```

```
</center>
</body>
</html>
```

5.2.2 Descritor das Tags

No descritor das tags (*/WEB-INF/tld/TagLib.tld*) há modificações a serem feitas para que a tag aceite parâmetros.
A parte que especifica a tag *mostremidias* ficará assim:

```
<tag>
<name>mostremidias</name>
<tagclass>tags.TagMidia</tagclass>
<teiclass>tags.TagTEIMidia</teiclass>
<bodycontent>JSP</bodycontent>
<info>Tag que mostra o conteúdo do BD</info>
<attribute>
      <name>chave</name>
    <required>false</required>
      <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

Onde *<name>* é o nome do atributo, *<required>* e se ele é obrigatório ou não, e *<rtexprvalue>* é se vão ser colocadas expressões (scriptlets ou expressões) dinâmicas no valor do atributo (*true*, e se o valor é estático *false*).

5.2.3 Modificando o arquivo TagMidia.java

Vamos modificar o arquivo TagMidia.java colocando um atributo chamado *chave* de tipo String e seus respectivos métodos *get* e *set*.
Também vamos modificar o método *doStartTag()* para que ele faça uma consulta diferente ao banco de dados caso o valor do atributo *chave* não seja nulo (ou seja, queremos fazer uma pesquisa no banco de dados).
Veja o novo código:

```
package tags;

import conexao.Conexao;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;


public final class TagMidia implements BodyTag {


    private PageContext pc = null;
```

```
private BodyContent body = null;
private StringBuffer sb = new StringBuffer();
private ResultSet rs = null;
private Conexao con = null;
private String chave = null;

public void setPageContext(PageContext p)
{
    pc = p;
}

public void setParent(Tag t) {}

public Tag getParent()
{
    return null;
}

public int doStartTag() throws JspException
{
    try {
        con = new Conexao();
```

Se foi passado algum parâmetro, *chave* vai ser diferente de nulo, e é feita uma consulta ao banco de dados, onde procura-se pela chave nos campos Titulo e Descricao.

```
if (chave != null) {
    rs = con.executeQuery("SELECT * FROM Midias where "+
   "Titulo like '%"+chave+"%' or Descricao like "+
   "'%"+chave+"%' order by Titulo");
} else  {
    rs = con.executeQuery("SELECT * FROM Midias order by Titulo");
}
    setVariaveis();
} catch (SQLException e) {
    System.out.println(e);
}
if (pc.getAttribute("titulo") == null)
    return SKIP_BODY;


    return EVAL_BODY_BUFFERED;
}

public void setBodyContent(BodyContent b)
{
    body = b;
}

public void doInitBody() throws JspException {}

private boolean setVariaveis() throws JspTagException {
    try {
        if (rs.next()) {
            pc.setAttribute("titulo", rs.getString("Titulo"));
            pc.setAttribute("ano", rs.getString("Ano"));
```

```
            pc.setAttribute("descricao", rs.getString("Descricao"));
            pc.setAttribute("tipo", rs.getString("Tipo"));
            pc.setAttribute("ID", rs.getString("ID"));
            return true;
        } else {
            return false;
        }
    } catch (SQLException e) {
        System.out.println(e);
        return false;
    }
}

public int doAfterBody() throws JspException {
    try {
        sb.append(body.getString());
        body.clear();
    } catch (IOException e) {
        throw new JspTagException("Erro fatal: IOException!");
    }
    if(setVariaveis()) {
        return EVAL_BODY_AGAIN;
    }
    try {
        body.getEnclosingWriter().write(sb.toString());
    } catch (IOException e) {
        throw new JspTagException("Erro fatal: IOException!");
    }
    return SKIP_BODY;
}

public int doEndTag() throws JspException {
    try {
        if(rs != null) {
            rs.close();
            rs = null;
        }
        if (con != null)
        {
            con.fecharConexao();
            con = null;
        }
    } catch (SQLException e) {}
    return EVAL_PAGE;
}

public void release() {
    pc = null;
    body = null;
    sb = null;
}

public void setChave(String nova) {
    chave = nova;
}

public String getChave() {
    return chave;
}
```

```
}
```

5.2.4 Passando parâmetros pela URL

Pode-se passar os valores dos formulários HTML pela URL. Tente acessar *http://localhost:8080/locadora/pesquisa.jsp?chave=string* e veja o que acontece.
Se houver mais de um parâmetro, separe-os usando ponto e vírgula(;).

Capítulo 6

Descritor da aplicação

O arquivo descritor da aplicação é chamado de web.xml e reside no diretório WEB-INF. Este arquivo serve como um depósito de informações; os dados que ele armazena sobre ele mesmo são, portanto, considerados metainformações, no sentido que são informações a respeito de informações. O arquivo web.xml é o arquivo de configuração central da aplicação.

Como sua extensão de arquivo sugere, a linguagem de marcação para os dados no arquivo web.xml é XML. O conteúdo básico do descritor é:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>


</web-app>
```

6.1 Construindo a aplicação

6.1.1 Web.xml

Vamos escrever o arquivo web.xml da nossa aplicação. Colocaremos apenas algumas informações, são elas:

- O tempo de validade da sessão
- Uma lista com nomes de arquivos que são abertos por padrão quando um diretório é acessado
- Ajustar as páginas de erro padrão, para que não apareçam as causas do erro para o usuário (se você não escreveu alguma página corretamente, já deve ter visto a página de erro que o Tomcat mostra).

Há uma ordem necessária das configurações usadas no arquivo web.xml. Trocando a ordem, a aplicação pode não funcionar. Abaixo está o código:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <session-config>
        <session-timeout>10</session-timeout>
    </session-config>


    <welcome-file-list>
        <welcome-file>opcoes.jsp</welcome-file>
            <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

    <error-page>
        <error-code>404</error-code>
        <location>/erro404.jsp</location>
    </error-page>



    <error-page>
        <exception-type>java.lang.Throwable</exception-type>
        <location>/erro.jsp</location>
    </error-page>
</web-app>
```

6.1.2 Página de erro

Vamos criar a página de erro que especificamos no arquivo web.xml.

Crie o arquivo erro404.jsp (404 é o código de página não encontrada) no diretório raiz da aplicação. Veja o código:

```
<%@ page isErrorPage="true" %>

<html>
<head>
<title>Erro!</title>
</head>
<body>

<table align="center" border="0" cellspacing="2"
            cellpadding="2" width="70%">
<tr>
<td bgcolor="#CCDDEE" align=center>
<h3>Erro</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
Página inexistente.<br>
```

```
</td>
</tr>

</table>

</body>
</html>
```

Capítulo 7

Release 6

7.1 Construindo a aplicação

Agora é sua vez de construir a aplicação. Crie uma parte da aplicação que faz o cadastro de pessoas, nos mesmos moldes do cadastro de mídias.

É necessário para o cadastro obter as seguintes informações sobre a pessoa: nome, endereço, telefone, cidade e bairro onde mora e RG. A tabela do banco de dados tem o nome de Pessoas e foi criada (como você deve lembrar) no começo do curso.

Nas próximas páginas estão os códigos fonte para realizar tal processo. Mas antes de olhar lembre-se que só se aprende fazendo.

CadastroPessoa.html

```
<html>
<head>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
<title>Cadastro de Pessoa</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<form action="ProcessarPessoa.jsp" method=post name="pessoas">
<center>
<table  cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCDDEE" colspan=2>
<font size=5>Cadastro de Pessoas</font>
<br>
<font size=1><sup>*</sup> Campos necess&aacute;rios</font>
</th>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Nome<sup>*</sup></b>
<br>
<input type="text" name="nome" value="" size=30 maxlength=50></td>
<td valign=top>
<b>Telefone</b>
<br>
<input type="text" name="telefone" value="" maxlength=15>
</td>
</tr>
<tr bgcolor="#F7F7F7">
<td valign=top colspan=2>
<b>Email<sup>*</sup></b>
<br>
<input type="text" name="email" value="" size=30 maxlength=50></td>
</tr>

<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Endereço
<br>
<TEXTAREA NAME="endereco" ROWS="6" COLS="30">

</textarea>
</td>
<td valign=top>
<b>RG<sup>*</sup></b></b>
<br>
<input type="text" name="RG" value="" maxlength=20>
</td>
</tr>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Cidade<sup>*</sup></b></b>
<br>
<input type="text" name="cidade" value="" maxlength=30>
</td>
<td valign=top>
```

```
<b>Bairro<sup>*</sup></b></b>
<br>
<input type="text" name="bairro" value="" maxlength=40>
</td>

</tr>

<tr bgcolor="#F7F7F7">
<td align=center colspan=2>
<input type="submit" value="Confirma">
<input type="reset" value="Limpar">
</td>
</tr>


</table>
</center>
</form>
</body>
</html>
```

ProcessarPessoa.jsp

```
<%@ page import="beans.BeanPessoa" %>
<jsp:useBean id="pessoa" class="beans.BeanPessoa" scope="request">

<jsp:setProperty name="pessoa" property="*" />

</jsp:useBean>

<%
if (pessoa.ehValido()) {
%>
<jsp:forward page="CadPessoa.jsp" />
<%
}  else {
%>
<jsp:forward page="RetryPessoa.jsp" />
<%
}
%>
```

RetryPessoa.jsp

```
<%@ page import="beans.BeanPessoa" %>
<jsp:useBean id="pessoa" class="beans.BeanPessoa" scope="request"/>


<html>
<head>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
<title>Cadastro de Pessoa</title>
</head>
```

```
<body bgcolor="#FFFFFF" text="#000000">
<form action="ProcessarPessoa.jsp" method=post name="pessoas">
<center>
<table  cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCDDEE" colspan=2>
<font size=5>Cadastro de Pessoas</font>
<br>
<font size=1><sup>*</sup> Campos necess&aacute;rios</font>
</th>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Nome<sup>*</sup></b>
<br>
<input type="text" name="nome" value='<%= pessoa.getNome() %>' size=30 maxlength=50><br>
<font color=#FF0000><%=pessoa.getErros("nome")%></font>
</td>
<td valign=top>
<b>Telefone</b>
<br>
<input type="text" name="telefone" value='<%= pessoa.getTelefone() %>'  maxlength=15><br>
<font color=#FF0000><%=pessoa.getErros("telefone")%></font>

</td>
</tr>
<tr bgcolor="#F7F7F7">
<td valign=top colspan=2>
<b>Email<sup>*</sup></b>
<br>
<input type="text" name="email" value='<%= pessoa.getEmail() %>'  size=30 maxlength=50><><br>
<font color=#FF0000><%=pessoa.getErros("email")%></font>
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Endereço
<br>
<TEXTAREA NAME="endereco" ROWS="6" COLS="30">
<%= pessoa.getEndereco() %>
</textarea><br>
<font color=#FF0000><%=pessoa.getErros("endereco")%></font>

</td>
<td valign=top>
<b>RG<sup>*</sup></b></b>
<br>
<input type="text" name="RG" value='<%= pessoa.getRG() %>'  maxlength=20><br>
<font color=#FF0000><%=pessoa.getErros("RG")%></font>

</td>
</tr>
<tr bgcolor="#F7F7F7">
<td valign=top>
<b>Cidade<sup>*</sup></b></b>
<br>
<input type="text" name="cidade" value='<%= pessoa.getCidade() %>'  maxlength=30><br>
<font color=#FF0000><%=pessoa.getErros("cidade")%></font>

</td>
```

```
<td valign=top>
<b>Bairro<sup>*</sup></b></b>
<br>
<input type="text" name="bairro" value='<%= pessoa.getBairro() %>'  maxlength=40><br>
<font color=#FF0000><%=pessoa.getErros("bairro")%></font>

</td>

</tr>

<tr bgcolor="#F7F7F7">
<td align=center colspan=2>
<input type="submit" value="Confirma">
<input type="reset" value="Limpar">
</td>
</tr>


</table>
</center>
</form>
</body>
</html>
```

BeanPessoa.java

```
package beans;

import java.util.Hashtable;
/**
* Implementar a classe Serializable é requisito para ser um Enterprise Bean.
* Um objeto de uma classe que implementa esta interface pode ser escrito em disco ou enviado
* Na aplicação do curso não fará diferença.
*/
public class BeanPessoa implements java.io.Serializable {


        /**
        * Nomes dos atributos preferencialmente iguais aos usados no formulário
        */
        private String nome;
        private String endereco;
        private String cidade;
        private String telefone;
        private String bairro;
        private String RG;
        private String email;
        /* Este atributo serve para o controle de erros no formulário */
        private Hashtable erros;

        public BeanPessoa() {
                /* Iniciamos os atributos com o String nulo */
                nome = "";
                endereco = "";
                telefone = "";
                cidade = "";
                RG = "";
                bairro = "";
```

```
        email = "";
        erros = new Hashtable();
}

/**
* Métodos para acessar os atributos.
* getNome() para ver ser valor, e setNome() para ajustar seu valor
*/

public String getNome() {
        return nome;
}

public String getEndereco() {
        return endereco;
}

public String getCidade() {
        return cidade;
}

public String getTelefone() {
        return telefone;
}

public String getRG() {
        return RG;
}

public String getBairro() {
        return bairro;
}

public String getEmail() {
        return email;
}

public void setNome(String valor) {
        nome = valor;
}

public void setEndereco(String valor) {
        endereco = valor;
}

public void setCidade(String valor) {
        cidade = valor;
}

public void setTelefone(String valor) {
        telefone = valor;
}

public void setBairro(String valor) {
        bairro = valor;
}

public void setRG(String valor) {
        RG = valor;
```

```
}

public void setEmail(String valor) {
        email = valor;
}

/**
* Verifica se todos os dados exigidos foram digitados,
* além de outras condições desejadas.
*/
public boolean ehValido() {
        boolean volta = true;

        if ((nome == null) || nome.equals(""))
        {
                erros.put("nome", "Por favor, digite um nome.");
                volta = false;
        }

        if ((endereco == null) || endereco.equals("") )
        {
                erros.put("endereco", "Por favor, digite o endereco.");
                volta = false;
        }
        if ((cidade == null) || cidade.equals("") )
        {
                erros.put("cidade", "Por favor, selecione o cidade.");
                volta = false;
        }
        if ((bairro == null) || bairro.equals("") )
        {
                erros.put("bairro", "Por favor, digite o bairro.");
                volta = false;
        }
        if ((email == null) || email.equals("") || (email.indexOf("@") == -1))
        {
                erros.put("email", "Por favor, digite o email.");
                volta = false;
        }
        if ((RG == null) || RG.equals("") )
        {
                erros.put("RG", "Por favor, digite o RG.");
                volta = false;
        }

        return volta;
}

/**
*Usado para ver as mensagens de erro armazaenados na tabela de Hash
*/
public String getErros (String s)
{
        String msg = (String) erros.get(s);
    return (msg == null) ? "" : msg;
}

/**
* Usado para colocar algum erro na tabela
```

```
        */
        public void setErros (String chave, String msg)
        {
                erros.put(chave,msg);
        }

}
```

CadPessoa.jsp

```
<%@ page import="beans.BeanPessoa, java.sql.*, conexao.*" %>

<%! String pagina = null; %>

<%

BeanPessoa pessoa = (BeanPessoa) request.getAttribute("pessoa");
Conexao con = null;
try
{
        con = new Conexao();


        con.executeUpdate("insert into Pessoas values ("+
                con.retornaIDMax("Pessoas")+
                ", '"+pessoa.getNome()+"' , '"+pessoa.getEmail()+"' , '"+
                pessoa.getEndereco()+"' , '"+pessoa.getCidade()+"' , '"+
                pessoa.getBairro()+"' , '"+pessoa.getTelefone()+"' , '"+
                pessoa.getRG()+"')");


        pagina = "/sucesso.jsp";


} catch (SQLException ex) {
        if (ex.getErrorCode() == 1062) {
                pessoa.setErros("email", "E-mail já existe.");
                pagina = "/cadastro/RetryPessoa.jsp";
        }
        System.out.println(ex);
} finally {
        if (con != null)
                con.fecharConexao();
        con = null;
}



        if (pagina == null)
                pagina = "/erro.jsp";
%>

<jsp:forward page='<%= pagina %>' />
```

Capítulo 8

Release 7

8.1 Passando objetos para outras partes da aplicação

Para recuperar ou passar objetos de uma página JSP para outra, ou de uma página para um servlet (e vice-versa), uma das maneiras (e a mais comum) é usar os métodos *get/setAttribute(String chave, String objeto)* disponíveis para os objetos implícitos *pageContext, request, session e application.*

8.2 Construindo a aplicação

Vamos construir uma página de login, onde se o usuário é autorizado é criada uma sessão para ele.

Precisaremos de um formulário de acesso, uma página JSP para processá-lo, e uma página JSP para testar se a sessão é válida ou não.

Começaremos com a página de acesso. Simples HTML. Veja o código:

```
<html>
<head>
<title>Página de Acesso</title>
</head>

<body>

<center>
<form action="VerificarSenha.jsp" method=POST name=login>
<table width=25%>
<tr>
<td colspan=2 align=center bgcolor="#CCDDEE">
Página de Acesso
</td>
</tr>
<tr>
<td bgcolor="#E0E0E0" width=50%>
Usuário:
</td>
<td bgcolor="#E0E0E0" width=50%>
<input type=text name=usuario value="">
```

```
</td>
</tr>
<tr>
<td bgcolor="#F3F3F3" width=50%>
Senha:
</td>
<td bgcolor="#F3F3F3" width=50%>
<input type=password name=senha>
</td>
</tr>
<tr>
<td bgcolor="#F0F0F0" align=center width=50%>
<input type=submit name=enviar value="Enviar">
</td>
<td bgcolor="#F0F0F0" align=center width=50%>
<input type=reset name=limpar value="Limpar">
</td>
</tr>
</table>
</form>

</center>
</body>
</html>
```

Agora o arquivo que valida o formulário. Dê o nome de *VerificarSenha.jsp*.

```
<%@ page import="java.sql.*, conexao.*" %>

<%

String senha = (String) request.getParameter("senha");
String usuario = (String) request.getParameter("usuario");


boolean login = false;

if (usuario != null && usuario.equals("pet"))
        if (senha != null && senha.equals("pet"))
                login = true;

if (login) {
        session.setAttribute("sessao", "autorizado");
}
%>

<jsp:forward page="/autorizado.jsp" />
```

Página de teste (*autorizado.jsp*. Note a diretiva *include* no início do código. Este arquivo (*sessao.jsp*) é que nos interessa.

```
<%@ include file ="/sessao.jsp" %>

<html>
<head>
<title>Autorizado!</title>
```

```
</head>
<body>

<table align="center" border="0" cellspacing="2"
        cellpadding="2" width="70%">
<tr>
<td bgcolor="#CCDDEE" align=center>
<h3>Autorizado</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
Autorizado!<br>
Você pode fazer o que quiser agora nesta página

</td>
</tr>

</table>

</body>
</html>
```

A página que verifica se a sessão é válida (**sessao.jsp**)é muito simples. Se o objeto implícito *session* tiver um atributo chamado de *sessao* e o valor dele for o *String autorizado*, este pedaço de código da página não faz nada, e então o código que está abaixo deste *scriptlet* será mostrado. Caso contrário, redireciona para a página de erro.

```
<%
String sess = (String) session.getAttribute("sessao");

if (!(sess != null && sess.equals("autorizado"))) {
%>
<jsp:forward page="/erro403.jsp" />
<%
}
%>
```

Colocamos os *Strings* acima no objeto *session* no código do arquivo *VerficarSenha.jsp* lembra? :)

Capítulo 9

Release 8

Neste último capítulo, vamos construir um carrinho virtual, desses que todo site de loja da intertet tem.

9.1 Carrinho de compras

Nosso carrinho vai ser um Bean, que vai ter como atributo principal um vetor (*java.util.Vector*) que armazenará as mídias que desejarmos. No caso, vamos armazenar apenas o título da mídia, mas com um pouco mais de código poderíamos armazenar um objeto *BeanMidia* ou qualquer outra coisa que fizesse sentido na aplicação.

Veja o código para *BeanCarrinho.java*:

```
package beans;

import java.util.Vector;
import java.util.Enumeration;

public class BeanCarrinho {

    Vector v;
    String comando = null;


    public BeanCarrinho() {
        v = new Vector();
    }

    private void adicionarItens(String[] array) {
        for (int i = 0; i < array.length; i++)
            v.addElement(array[i]);
    }

    private void removerItens(String[] array) {
        for (int i = 0; i < array.length; i++)
            v.removeElement(array[i]);
    }


    public void setComando(String s) {
        comando = s;
```

```
    }

    public String[] getItems() {
        String[] s = new String[v.size()];
        v.copyInto(s);
        return s;
    }

    public void processar(String[] itens) {

        if (itens != null && comando != null) {

            if (comando.equals("adicionar"))
                adicionarItens(itens);
            else if (comando.equals("remover"))
                removerItens(itens);

            reset();
        }
    }


    private void reset() {
        comando = null;

    }
}
```

9.2 Página JSP

Vamos mostrar na mesma página o carrinho e a pesquisa por palavras-chave. Para isso, basta modificar um pouco o arquivo *pesquisa.jsp*. Vamos adicionar um formulário com *checkboxes* para o usuário escolher as fitas que deseja botar no carrinho. E no início da página vamos colocar a parte do carrinho incluindo uma página que vamos criar em seguida.

Veja o código para a página *carrinho.jsp*:

```
<%@ taglib uri="/WEB-INF/tld/TagLib.tld" prefix="pesquisa" %>

<html>
<body>
<center>
```

Inclusão estática do arquivo *carrinho1.jsp*. Seria o mesmo que copiar seu código e colar aqui.

```
<%@ include file ="/carrinho1.jsp" %>

<table align=center width="50%">
<form method="post" action="carrinho.jsp">
<th bgcolor="#CCDDEE" colspan=3>
Digite a(s) palavra(s) chave(s) para a pesquisa
```

```
</th>
<tr bgcolor="#f7f7f7">
   <td align=center>Pesquisa</td>
   <td align=center>
    <input type="text" name="chave" size=30>
      
    <input type=submit value="Pesquisar >>">   
    <input type=reset value="Limpar">
   </td>
</tr>
</form>
</table>

<%
    String tem = (String) request.getParameter("chave");
    if (tem != null && !tem.equals(""))
    {
%>
<br><br>
<form action='carrinho.jsp' method=post>
<table align=center width="70%">
<th bgcolor="#CCDDEE" colspan=2 align=center>
Mídias encontradas
</th>
<th bgcolor="#CCDDEE" >Carrinho</th>
<pesquisa:mostremidias chave='<%= tem %>' >
<tr bgcolor="#E3E3E3">
<td align=center>
<b>Mídia:</b> 
<%= titulo %>
</td>
<td align=center>
<b>Ano:</b> <%= ano %>
</td>
<td>
<input TYPE=checkbox name='reserva' value='<%=titulo %>' >
</td>
</tr>

<tr bgcolor="#F7F7F7">
<td align=center>
<%= descricao %>
</td>
<td align=center>
<%= tipo %>
</td>
<td> </td>
</tr>
<tr>
<td colspan="3" bgcolor="#FFFFFF">
 
</td>

</tr>


</pesquisa:mostremidias>

<tr bgcolor="#E0E0E0">
```

```
<td align=center>
<input type="submit" value="Confirma">
</td>
<td align=center>
<input type="reset" value="Limpar">
</td>
<td> </td>
</tr>

</table>
```

Este atributo escondido, diz qual a ação desejada (adicionar ou remover).

```
<input TYPE=hidden name='comando' value='adicionar' >
</form>

<% } %>

</center>
</body>
</html>
```

Vamos ver a parte lógica do carrinho agora.

Usa-se um bean válido por toda a sessão. Com o método *processar(..)*, incluímos ou removemos mídias do BeanCarrinho.

Colocando o valor do atributo *property* da ação <jsp:setProperty> como "*", ele vai, para todos os parâmetros presentes na requisição (objeto *request*), tentar usar o método *setNomeDoParametro()* do Bean.

```
<jsp:useBean id="carrinho" scope="session" class="beans.BeanCarrinho" />

<jsp:setProperty name="carrinho" property="*" />
<%
    String[] temp = request.getParameterValues("reserva");
    if (temp != null) {
        carrinho.processar(temp);
    }
    String[] items = carrinho.getItems();
    if (items.length > 0) {
%>

<form action='carrinho.jsp' method=post>
<table>
<th bgcolor="#CCDDEE">Você tem no carrinho:</th>
<th bgcolor="#CCDDEE" >Remover</th>
<%

    for (int i=0; i<items.length; i++) {
%>
<tr>
<td bgcolor="#F0F0F0">
<%= items[i] %>
</td>
<td bgcolor="#E0E0E0">
<input type=checkbox name=reserva value='<%= items[i] %>'>
```

```
</td>
</tr>
<%
    }
%>
<tr bgcolor="#E0E0E0">
<td align=center>
<input type="submit" value="Confirma">
</td>
<td align=center>
<input type="reset" value="Limpar">
</td>
<td> </td>
</tr>


</table>
<input type=hidden name='comando' value='remover'>
</form>

<%
}
%>
```

Capítulo 10

Release 9

Apenas para deixar a aplicação com um interface melhor, vamos criar um arquivo *index.jsp* na raiz.

```
<html>
<head>
<title>P&aacute;gina inicial</title>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<table align="center" border="0" cellspacing="2" cellpadding="2">
<tr>
<td bgcolor="#CCDDEE" align=center colspan=5>
<h3>Escolha a op&ccedil;&atilde;o</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
<a href='cadastro/'>Cadastro</a>
</td>
<td>
<a href='login.jsp'>Login</a>
</td>
<td>
<a href='pesquisa.jsp'>Pesquisa</a>
</td>
<td>
<a href='carrinho.jsp'>Carrinho</a>
</td>
<td>
<a href='titulos.jsp'>Títulos disponíveis</a>
</td>


</tr>

</table>

</body>
</html>
```

E outro arquivo index.jsp no diretório */cadastro*.

```
<html>
<head>
<title>P&aacute;gina de cadastro</title>
<style>
p, td { font-family:Tahoma,Sans-Serif; font-size:11pt;
padding-left:15; }
</style>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<table align="center" border="0" cellspacing="2" cellpadding="2">
<tr>
<td bgcolor="#CCDDEE" align=center colspan=2>
<h3>Escolha a op&ccedil;&atilde;o</h3>
</td>
</tr>
<tr>
<td bgcolor="#F7F7F7" align=center>
<a href='CadastroMidia.html'>Mídias</a>
</td>
<td>
<a href='CadastroPessoa.html'>Pessoas</a>
</td>
</tr>
</table>

</body>
</html>
```

Referências Bibliográficas

[1] Duane K. Fields, Mark A. Kolb. *Desenvolvendo na Web com Java Server Pages*. Editora Ciência Moderna, Primeira Edição, 2000.

[2] Star Developer *Stardeveloper.com : Connecting to Databases using JDBC*. Star Developer, 2001. Disponível online em http://stardeveloper.com. Acessado em setembro de 2002.

[3] Star Developer *Stardeveloper.com : Java Server Pages and Servlets Articles*. Star Developer, 2001. Disponível online em http://stardeveloper.com. Acessado em setembro de 2002.

[4] Sun Microsystems *JavaServer Pages(TM) Technology*. Sun Microsystems, 2001. Disponível online em http://java.sun.com/products/jsp/. Acessado em setembro de 2002.

[5] Orion Application Server *OrionServer Taglibs Tutorial*. IronFlare AB, 2002. Disponível online em http://www.orionserver.com/. Acessado em setembro de 2002.