
JSP e Servlets

JSP e Servlets	1
Instalação Passo-a-Passo do Tomcat 5 em Windows	5
Servlets e JSP	13
Servlets	14
Applats e Servlets	17
CGI X Servlets	19
A API Servlet	21
Exemplo de Servlet	24
Compilando o Servlet	26
Criando uma aplicação no Tomcat	28
Invocando diretamente pelo Navegador	30
Invocando em uma página HTML	32
Diferenças entre as requisições GET e POST	34
Concorrência	36
Obtendo Informações sobre a Requisição	39
Lidando com Formulários	42
Lidando com Cookies	45
Lidando com Sessões	50
Reencaminhando ou Redirecionando requisições	54
Java Server Pages (JSP)	56
PHP X JSP	59
ASP X JSP	60
Primeiro exemplo em JSP	62
Executando o arquivo JSP	64
Objetos implícitos	66
Tags JSP	67
Expressões	67
Scriptlets	67
Declarações	69
Comentários	70
Diretivas	70
Diretiva page	70

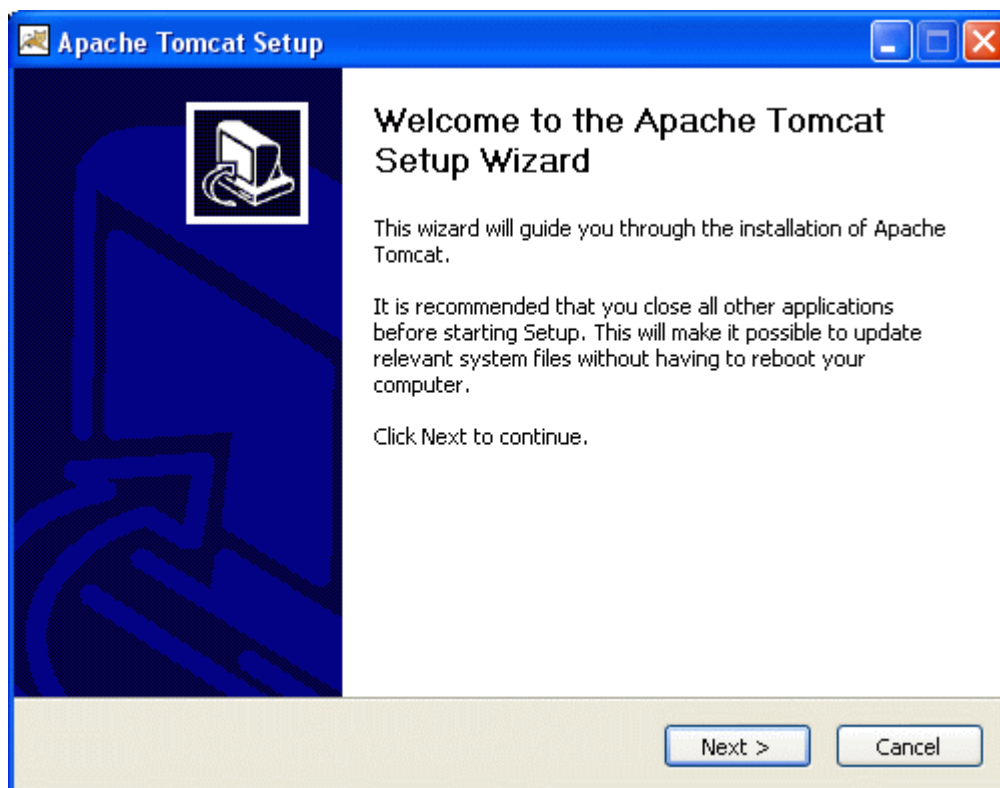
Diretiva include	72
Extraindo Valores de Formulários	74
Criando e Modificando Cookies	77
Lidando com sessões	80
O Uso de JavaBeans.....	83
Escopo	86
Implementação de um Carrinho de compras	86
Uma Arquitetura para comércio eletrônico.....	90
Tipos de aplicações na WEB	92
Arquitetura MVC para a Web	94
Agenda Web:	98
Um Exemplo de uma aplicação Web usando a arquitetura MVC	98
Instalação	118
Considerações sobre a solução	119
Tag Files.....	120
O que são Tag Files?	122
Preparação	122
Minha primeira Tag! (não é HelloWorld)	122
Tags com atributos	125
Empacotando suas tags para distribuição	130
JSTL.....	132
Instalação da JSTL	134
TLD	136
A Expression Language (EL)	138
Precedência de operadores:	139
Os operadores [] e .	139
Utilizando uma biblioteca de tags	145
Importando a bibliotecas de tags	145
TLD	145
Ações de Finalidades Gerais	148
<c:out>	148
<c:set>	150

<c:remove>	152
<c:catch>	152
Ações Condicionais	155
<c:if>	157
Condições Mutuamente Exclusivas	159
<c:choose>	159
<c:when>	160
<c:otherwise>	160
Exemplo de Cadastro Web com JSTL	169
Index.jsp – Tela de Cadastro	170
gravaCliente.jsp – Para gravação na tabela CLIENTES	172
JavaServer Faces – JSF	175
Implementando um exemplo com JSF	179
Index.jsf	179
Buscar.jsf	180
Inserir.jsf	181
Sucesso_busca.jsf	186
Falha_busca.jsf	187
Sucesso_insercao.jsf	188
Falha_insercao.jsf	188
Arquivos de Configuração	191

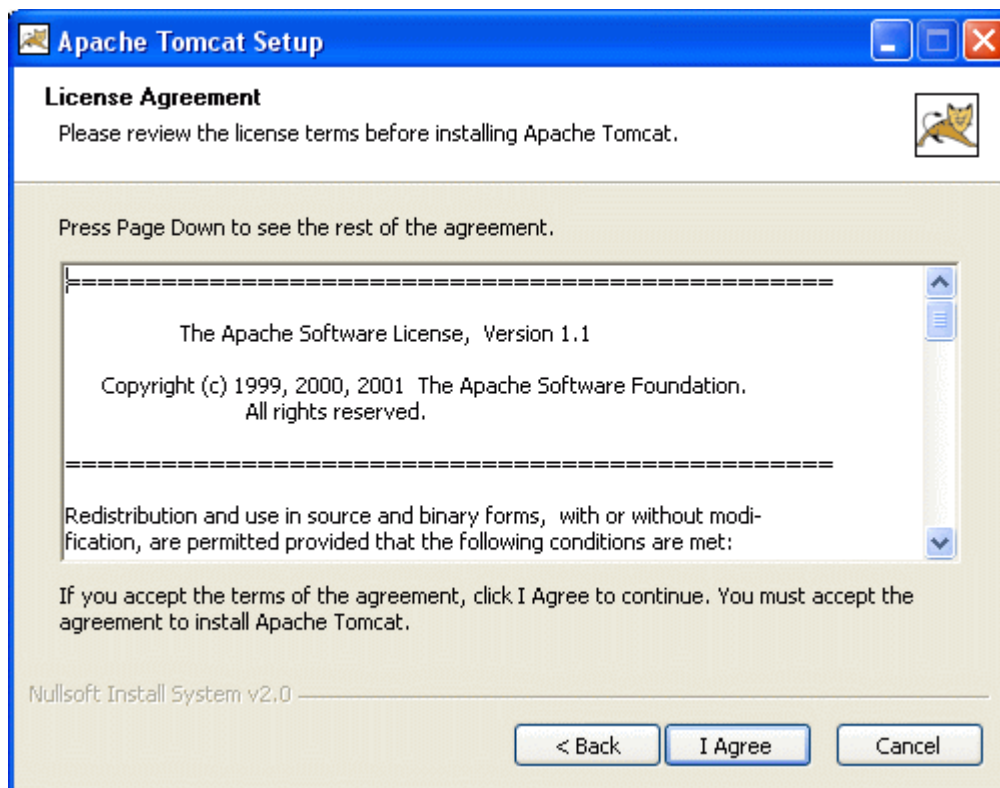
Instalação Passo-a-Passo do Tomcat em Windows

As etapas e ilustrações a seguir se baseiam na versão 5.0.19. Versões futuras podem, obviamente, apresentar alterações no processo de instalação.

1. Inicie o programa instalador. O assistente de instalação será iniciado. No diálogo de Boas-vindas, clique "Next".

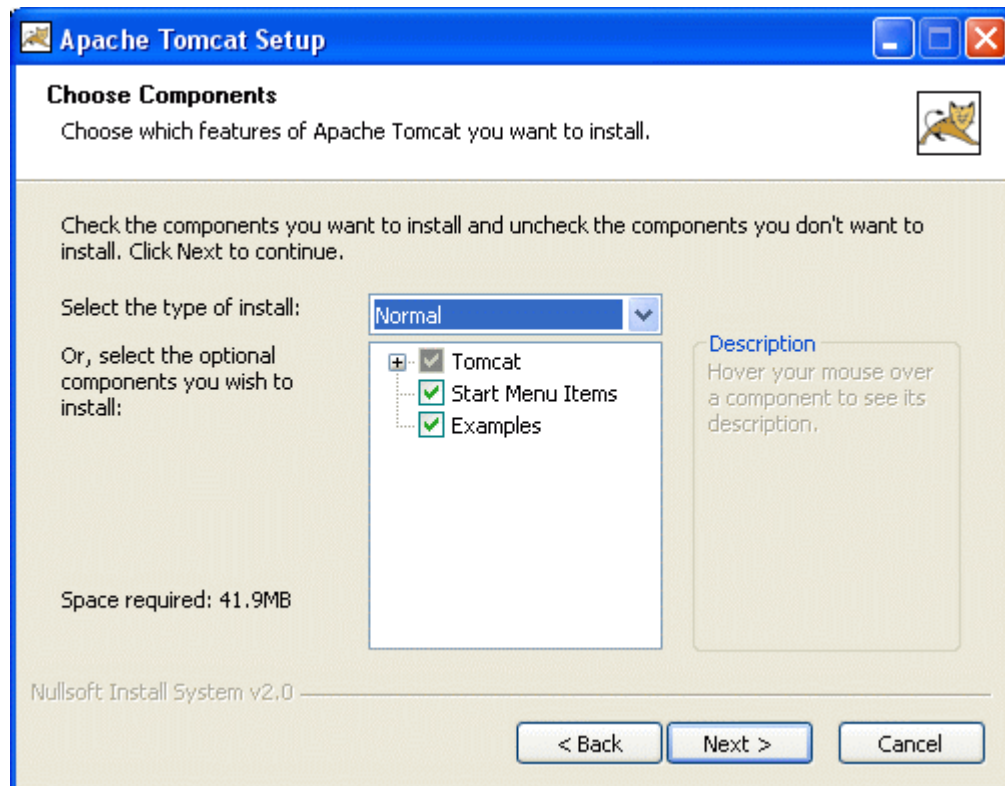


2. Estando de acordo com os termos da licença do software, clique "I Agree" para prosseguir. O Tomcat é software aberto, de re-distribuição e uso (comercial ou não) livres e gratuitos.



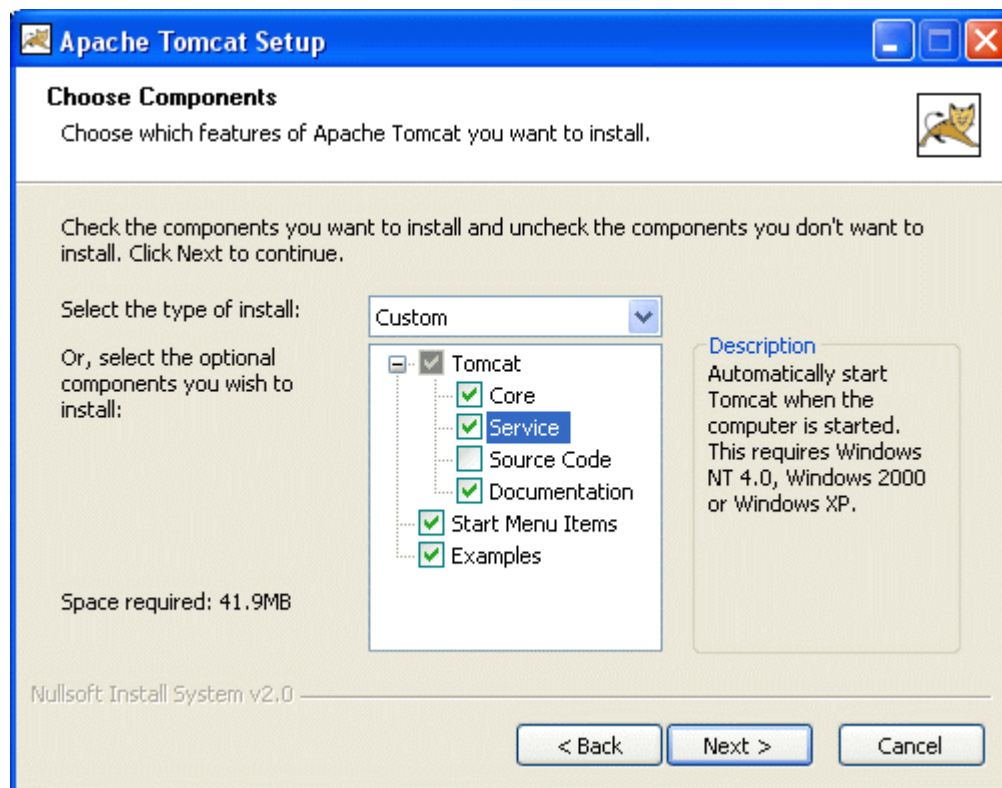
3. Na escolha de componentes do software, o padrão é o tipo de instalação "Normal", que inclui, além do núcleo essencial (core) do Tomcat Servlet container, toda a documentação, exemplos de Servlet e JSP e os ícones no Menu Iniciar. Esta seleção é adequada para o uso geral.

Na verdade, ficam de fora da instalação Normal apenas a ativação automática do Tomcat como serviço e o código-fonte do Tomcat. Este último só será de alguma utilidade se você pretender participar do projeto de desenvolvimento do Tomcat, ou ainda quiser conhecer a fundo os mecanismos de funcionamento e a implementação do Tomcat inspecionando seus fontes, e não apenas utilizá-lo.

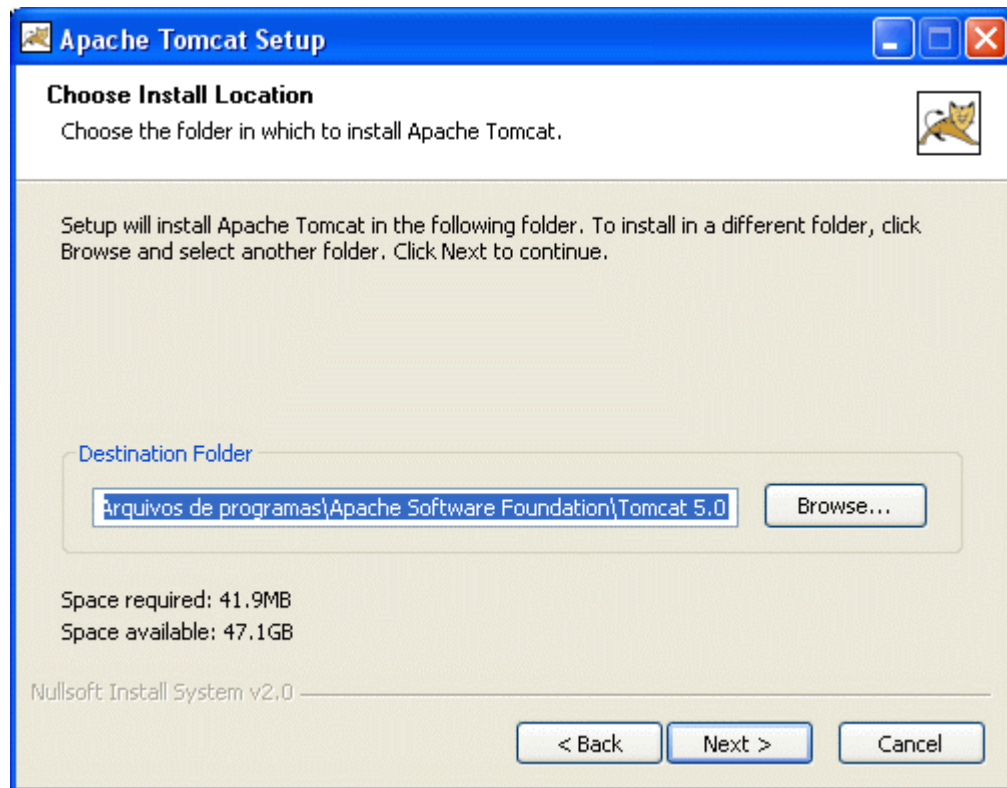


A instalação do Tomcat 5 sempre cria um serviço do Windows (NT/2000/XP/2003) para o Tomcat. Na opção Normal, porém, o serviço é instalado com ativação manual. Se você preferir que o serviço do Tomcat seja automaticamente ativado sempre que o Windows for iniciado, basta expandir o componente "Tomcat" e selecionar o componente "Service", conforme mostrado a seguir, e o instalador já configurará isto por você. O tipo de instalação mudará para "Custom" (personalizada). De qualquer forma, o tipo de inicialização do serviço do Tomcat pode ser facilmente alterado entre Manual ou Automático a qualquer tempo após a instalação, através da ferramenta administrativa de Serviços do Windows.

Se você não tem familiaridade com a configuração de serviços do Windows, ou não estiver certo se deseja a inicialização automática do Tomcat como serviço, deixe a instalação como "Normal", inalterada. Para prosseguir, clique "Next".



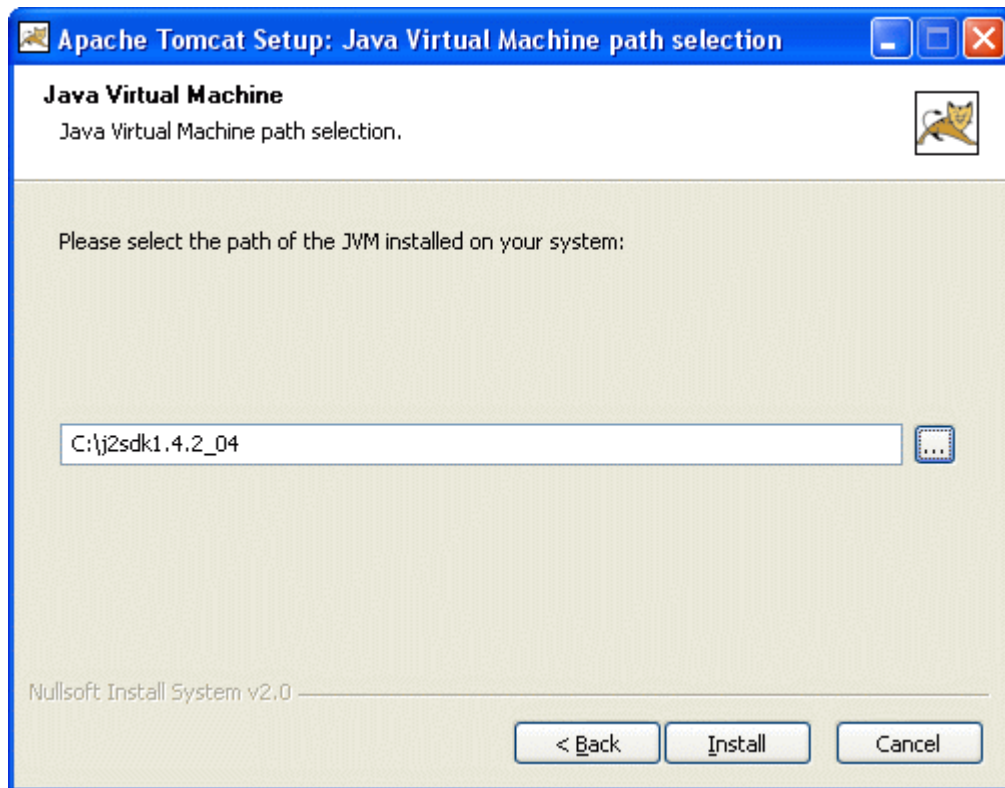
4. A seguir, será confirmado o local de instalação do software. Confirme a pasta principal onde o Tomcat será instalado e clique "Next" para prosseguir.



5. O diálogo de Configuração permite definir duas opções administrativas do Tomcat: o porto de rede pelo qual o Tomcat atenderá as requisições HTTP, funcionando como um servidor web com o propósito de testes e administração do Tomcat, e o usuário e senha para o acesso à Administração do Tomcat.

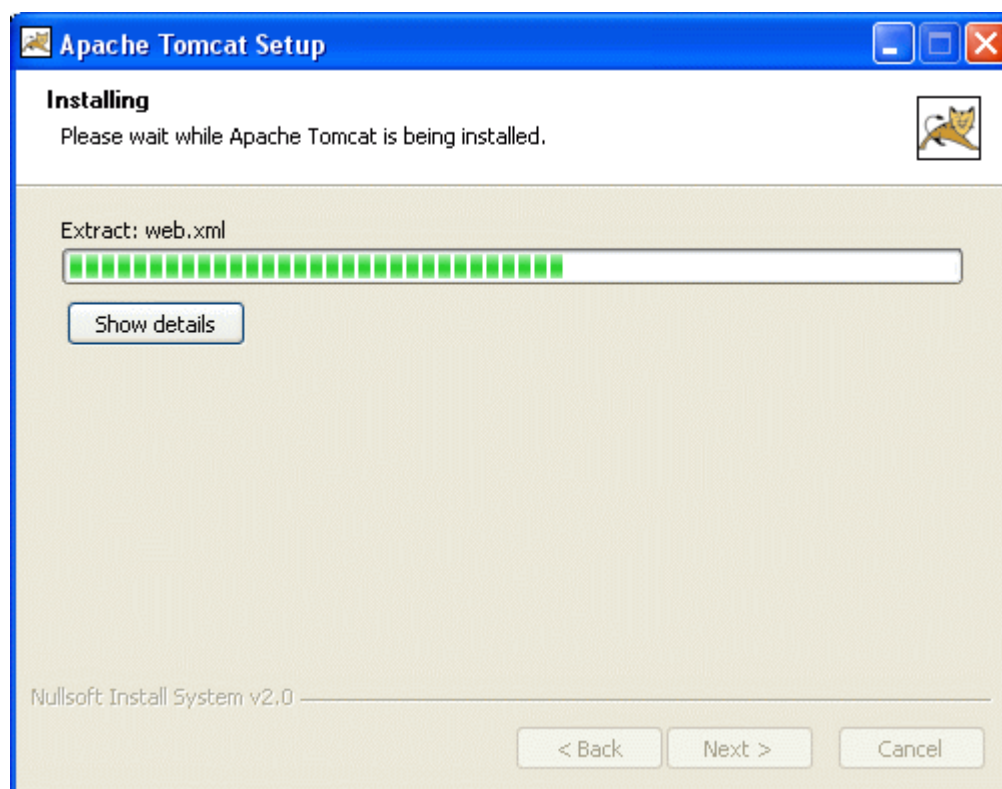
A menos que no computador já exista outro serviço simultâneo utilizando o porto padrão 8080 do Tomcat, é recomendável deixar o porto inalterado. Em especial, tenha muito cuidado se pretender modificar para o porto 80, que é o padrão para o protocolo HTTP e usado pelos servidores web como o Microsoft IIS e o Apache Web Server (httpd). Esta configuração pode ser alterada após a instalação, pelo arquivo de config do Tomcat.

A senha do usuário administrativo do Tomcat (nome padrão "admin") pode ser deixada em branco, o que é plausível se esta instalação do Tomcat for apenas em uma estação de trabalho de desenvolvimento. Em um servidor, ou se o computador for acessível em uma rede (LAN interna ou Internet) e o Tomcat for ficar ativo constantemente, é recomendável definir um nome de usuário e senha personalizados, por segurança.

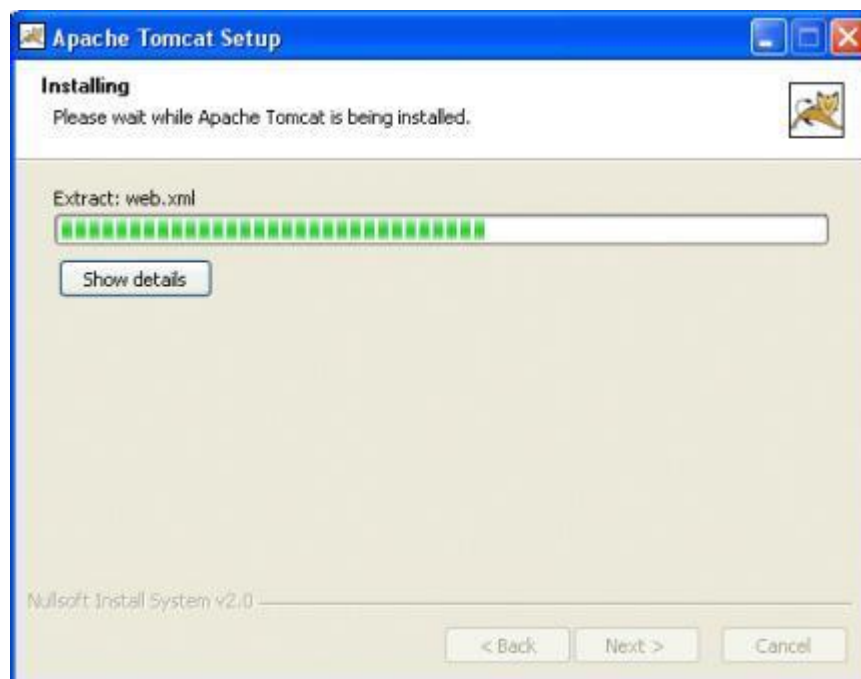


6. O instalador do Tomcat procura detectar uma versão de Java SDK (JDK) instalada, necessária para seu funcionamento. Neste diálogo "Java Virtual Machine", você deve verificar com atenção se a pasta pré-selecionada pelo instalador corresponde ao local de instalação da versão de JDK instalada que você quer que o Tomcat utilize.

Quando a variável de ambiente `JAVA_HOME` não está corretamente definida, o instalador do Tomcat 5 Windows costuma detectar erroneamente o JRE (Java Runtime, também instalado junto com o JDK), cuja localização padrão é similar a `C:\Arquivos de programas\Java\j2re1.4\`, ao invés de detectar o SDK completo, normalmente localizado em algo como `C:\j2sdk1.4\`. Se for este o caso, **altere** a localização para informar o caminho correto do Java SDK (use o botão "... " para navegar nas pastas), caso contrário o Tomcat executará, mas não será capaz de efetuar a compilação dinâmica de páginas JSP novas/alteradas!

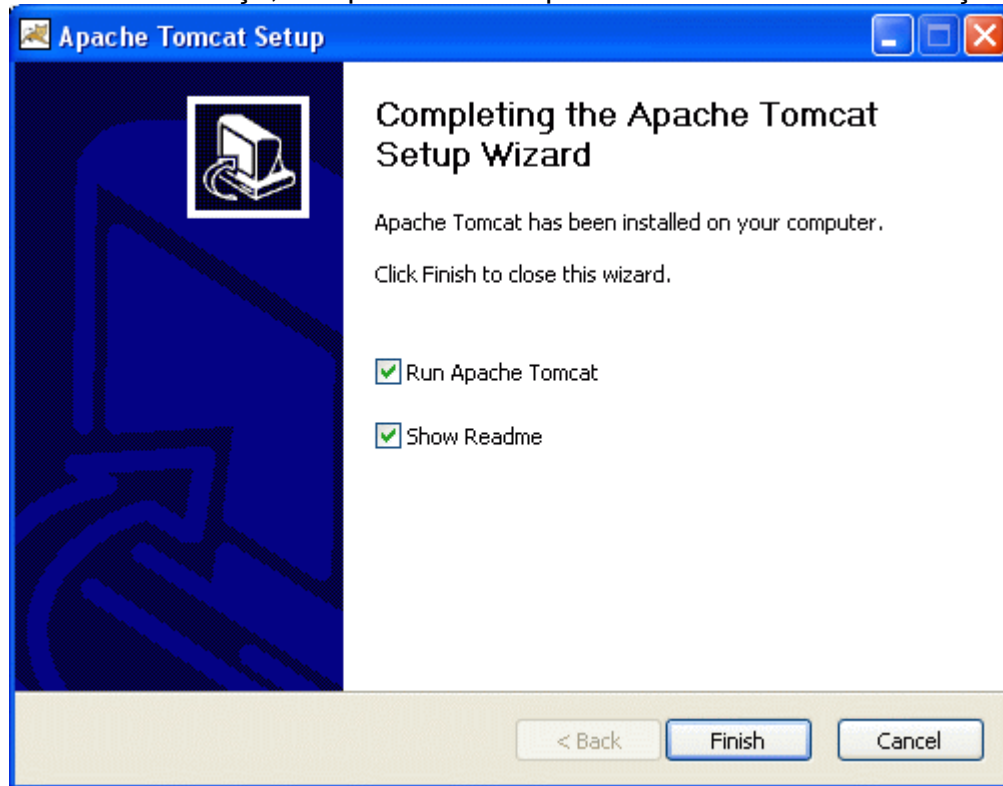


7. Completadas as informações solicitadas pelo assistente, clique "Install" e aguarde o término da instalação.



8. Ao término da instalação, o assistente dá a opções de executar o Tomcat e de visualizar o Leia-Me.

Note que a opção "Run" do instalador executa o tomcat como processo, e não como serviço, independente do que foi selecionado na instalação.



Servlets e JSP

Servlets e JSP são duas tecnologias desenvolvidas pela Sun para desenvolvimento de aplicações na Web a partir de componentes Java que executem no lado servidor. Essas duas tecnologias fazem parte da plataforma J2EE (Java 2 Platform Enterprise Edition) que fornece um conjunto de tecnologias para o desenvolvimento de soluções escaláveis e robustas para a Web. Nesta apostila abordaremos apenas as tecnologias Servlets e JSP, sendo o suficiente para o desenvolvimento de sites dinâmicos de razoável complexidade. Se a aplicação exigir uma grande robustez e escalabilidade o leitor deve considerar o uso em conjunto de outras tecnologias da plataforma J2EE.

Servlets

Servlets são classes Java que são instanciadas e executadas em associação com servidores Web, atendendo requisições realizadas por meio do protocolo HTTP. Ao serem acionados, os objetos Servlets podem enviar a resposta na forma de uma página HTML ou qualquer outro conteúdo MIME. Na verdade os Servlets podem trabalhar com vários tipos de servidores e não só servidores Web, uma vez que a API dos Servlets não assume nada a respeito do ambiente do servidor, sendo independentes de protocolos e plataformas. Em outras palavras Servlets é uma API para construção de componentes do lado servidor com o objetivo de fornecer um padrão para comunicação entre clientes e servidores. Os Servlets são tipicamente usados no desenvolvimento de *sites dinâmicos*. Sites dinâmicos são sites onde algumas de suas páginas são construídas no momento do atendimento de uma requisição HTTP. Assim é possível criar páginas com conteúdo variável, de acordo com o usuário, tempo, ou informações armazenadas em um banco de dados.

Servlets não possuem interface gráfica e suas instâncias são executadas dentro de um ambiente Java denominado de *Container*. O *container* gerencia as instâncias dos Servlets e provê os serviços de rede necessários para as requisições e respostas. O container atua em associação com servidores Web recebendo as requisições reencaminhada por eles. Tipicamente existe apenas uma instância de cada Servlet, no entanto, o *container* pode criar vários *threads* de modo a permitir que uma única instância Servlet atenda mais de uma requisição simultaneamente. A figura I-1 fornece uma visão do relacionamento destes componentes.

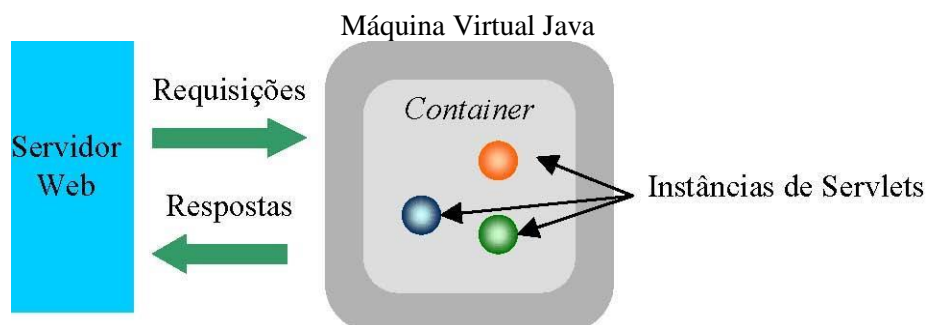


Figura I-1. Relacionamento entre Servlets, container e servidor Web

Servlets provêm uma solução interessante para o relacionamento cliente/servidor na Internet, tornando-se uma alternativa para a implantação de

sistemas para a Web. Antes de entrarmos em detalhes na construção de Servlets, compararemos esta solução com outras duas soluções possíveis para implantação de aplicações na Internet.

Applats e Servlets

Apesar de ser uma solução robusta existem problemas no uso de Applets para validação de dados e envio para o servidor. O programador precisa contar com o fato do usuário possuir um navegador com suporte a Java e na versão apropriada. Você não pode contar com isso na Internet, principalmente se você deseja estender a um grande número de usuário o acesso às suas páginas. Em se tratando de Servlets, no lado do cliente pode existir apenas páginas HTML, evitando restrições de acesso às páginas. Em resumo, o uso de Applets não é recomendado para ambientes com múltiplos navegadores ou quando a semântica da aplicação possa ser expressa por componentes HTML.

CGI X Servlets

Scripts CGI (*Common Gateway Interface*), acionam programas no servidor. O uso de CGI sobrecarrega o servidor uma vez cada requisição de serviço acarreta a execução de um programa executável (que pode ser escrito em com qualquer linguagem que suporte o padrão CGI) no servidor, além disso, todo o processamento é realizado pelo CGI no servidor. Se houver algum erro na entrada de dados o CGI tem que produzir uma página HTML explicando o problema. Já os Servlets são carregados apenas uma vez e como são executados de forma multi-thread podem atender mais de uma mesma solicitação por simultaneamente. Versões posteriores de CGI contornam este tipo de problema, mas permanecem outros como a falta de portabilidade e a insegurança na execução de código escrito em uma linguagem como C/C++

A API Servlet

A API Servlet é composta por um conjunto de interfaces e Classes. O componente mais básico da API é interface Servlet. Ela define o comportamento básico de um Servlet. A figura 1 mostra a interface Servlet.

```
public interface Servlet {
    public void init(ServletConfig config) throws ServletException;
    public ServletConfig getServletConfig();
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;
    public String getServletInfo();
    public void destroy();
}
```

Figura 1. Interface Servlet.

O método service() é responsável pelo tratamento de todas as requisições dos clientes. Já os métodos init() e destroy() são chamados quando o Servlet é carregado e descarregado do *container*, respectivamente. O método getServletConfig() retorna um objeto ServletConfig que contém os parâmetros de inicialização do Servlet. O método getServletInfo() retorna um String contendo informações sobre o Servlet, como versão e autor.

Tendo como base a interface Servlet o restante da API Servlet se organiza hierarquicamente como mostra a figura 2.

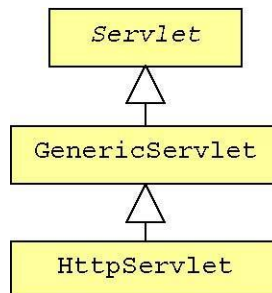


Figura 2. Hierarquia de classes da API Servlet.

A classe GenericServlet implementa um servidor genérico e geralmente não é usada. A classe HttpServlet é a mais utilizada e foi especialmente projetada para lidar com o protocolo HTTP. A figura 3 mostra a definição da classe interface HttpServlet.

```
HttpServlet
public abstract class HttpServlet
extends GenericServlet
```

```
implements java.io.Serializable
```

Figura 3. Definição da classe `HttpServlet`.

Note que a classe `HttpServlet` é uma classe abstrata. Para criar um `Servlet` que atenda requisições HTTP o programador deve criar uma classe derivada da `HttpServlet` e sobrescrever pelo menos um dos métodos abaixo:

<code>doGet</code>	Trata as requisições HTTP GET.
<code>doPost</code>	Trata as requisições HTTP POST.
<code>doPut</code>	Trata as requisições HTTP PUT.
<code>doDelete</code>	Trata as requisições HTTP DELETE.

Tabela 1. Métodos da classe `HttpServlet` que devem ser sobrescritos para tratar requisições HTTP.

Todos esses métodos são invocados pelo servidor por meio do método `service()`. O método `doGet()` trata as requisições GET. Este tipo de requisição pode ser enviada várias vezes, permitindo que seja colocada em um *bookmark*. O método `doPost()` trata as requisições POST que permitem que o cliente envie dados de tamanho ilimitado para o servidor Web uma única vez, sendo útil para enviar informações tais como o número do cartão de crédito. O método `doPut()` trata as requisições PUT. Este tipo de requisição permite que o cliente envie um arquivo para o servidor à semelhança de como é feito via FTP. O método `doDelete()` trata as requisições DELETE, permitindo que o cliente remova um documento ou uma página do servidor. O método `service()`, que recebe todas as requisições, em geral não é sobrescrito, sendo sua tarefa direcionar a requisição para o método adequado.

Exemplo de Servlet

Para entendermos o que é um Servlet nada melhor que um exemplo simples. O exemplo abaixo gera uma página HTML em resposta a uma requisição GET. A página HTML gerada contém simplesmente a frase *Ola mundo!!!*. Este é um Servlet bem simples que ilustra as funcionalidades básicas da classe.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;
public class Ola extends HttpServlet {
    public String getServletInfo() {
        return "Ola versão 0.1";
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException,
ServletException{
        res.setContentType("text/html");
        java.io.PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet</title>");
        out.println("</head>");
        out.println("<body>Ola mundo!!!");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

Exemplo . *Servlet Ola*.

O método `doGet()` recebe dois objetos: um da classe `HttpServletRequest` e outro da classe `HttpServletResponse`. O `HttpServletRequest` é responsável pela comunicação do cliente para o servidor e o `HttpServletResponse` é responsável pela comunicação do servidor para o cliente. Sendo o exemplo *Servlet Ola* apenas um exemplo simples ele ignora o que foi enviado pelo cliente, tratando apenas de enviar uma página HTML como resposta. Para isso é utilizado o objeto da classe `HttpServletResponse`. Primeiramente é usado o método `setContentType()` para definir o tipo do conteúdo a ser enviado ao cliente. Esse método deve ser usado apenas uma vez e antes de se obter um objeto do tipo `PrintWriter` ou `ServletOutputStream` para a resposta. Após isso é usado o método `getWriter()` para se obter um objeto do tipo `PrintWriter` que é usado para escrever a resposta. Neste caso os dados da resposta são baseados em caracteres. Se o programador desejar enviar a resposta em bytes deve usar o método `getOutputStream()` para obter um objeto `OutputStream`. A partir de então o programa passa usar o objeto `PrintWriter` para enviar a página HTML.

Compilando o Servlet

A API Servlet ainda não foi incorporado ao SDK, portanto, para compilar um Servlet é preciso adicionar a API Servlet ao pacote SDK. Existem várias formas de se fazer isso. A Sun fornece a especificação da API e diversos produtores de software executam a implementação. Atualmente, a especificação da API Servlet está na versão 2.3. Uma das implementações da API que pode ser baixada gratuitamente pela Internet é a fornecida pelo projeto Jakarta (<http://jakarta.apache.org>) denominada de Tomcat. A implementação da API Servlet feita pelo projeto Jakarta é a implementação de referência indicada pela Sun. Ou seja, é a implementação que os outros fabricantes devem seguir para garantir a conformidade com a especificação da API. No entanto, uma vez que o Tomcat é a implementação mais atualizada da API, é também a menos testada e, por consequência, pode não ser a mais estável e com melhor desempenho.

Antes de executar o Servlet é preciso compilá-lo. Para compilá-lo é preciso que as classes que implementam a API Servlet estejam no classpath. Para isso é preciso definir a variável de ambiente. No ambiente MS-Windows seria

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\servlet.jar
```

e no ambiente Unix seria

```
CLASSPATH=${CLASSPATH}:${TOMCAT_HOME}/lib/servlet.jar
```

Alternativamente, é possível indicar o classpath na própria linha de execução do compilador Java. Por exemplo, No ambiente MS-Windows ficaria na seguinte forma:

```
javac -classpath "%CLASSPATH%;c:\tomcat\lib\servlet.jar Ola.java
```


Criando uma aplicação no Tomcat

Agora é preciso definir onde deve ser colocado o arquivo compilado. Para isso é preciso criar uma aplicação no Tomcat ou usar uma das aplicações já existentes. Vamos aprender como criar uma aplicação no Tomcat. Para isso é preciso criar a seguinte estrutura de diretórios abaixo do diretório webapps do Tomcat:

```
webapps
|____ Nome aplicação
      |____ Web-inf
            |____ classes
            |____ lib
```

Diretório de Aplicações

Na verdade é possível definir outro diretório para colocar as aplicações do Tomcat. Para indicar outro diretório é preciso editar o arquivo server.xml e indicar o diretório por meio da diretiva home do tag ContextManager.

O diretório de uma aplicação é denominado de *contexto da aplicação*. É preciso também editar o arquivo server.xml do diretório conf, incluindo as linhas:

```
<Context path="/nome aplicação"
docBase="webapps/ nome aplicação" debug="0"
reloadable="true" >
</Context>
```

Finalmente, é preciso criar (ou copiar de outra aplicação) um arquivo web.xml no diretório Web-inf com o seguinte conteúdo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
</web-app>
```

Copie então o arquivo compilado Ola.class para o subdiretório /webapps/nome aplicação/Web-inf/classes do Tomcat.

Invocando diretamente pelo Navegador

Podemos executar um Servlet diretamente digitando a URL do Servlet no navegador. A URL em geral possui o seguinte formato:

```
http://máquina:porta/nome aplicação/servlet/nome servlet
```

A palavra servlet que aparece na URL não indica um subdiretório no servidor. Ela indica que esta é uma requisição para um Servlet. Por exemplo, suponha que o nome da aplicação criada no Tomcat seja teste. Então a URL para a invocação do Servlet do exemplo teria a seguinte forma:

```
http://localhost:8080/teste/servlet/Ola
```

A URL para a chamada do Servlet pode ser alterada de modo a ocultar qualquer referência à diretórios ou a tecnologias de implementação. No caso do Tomcat essa configuração é no arquivo web.xml do diretório Web-inf da aplicação. Por exemplo, para eliminar a palavra servlet da URL poderíamos inserir as seguintes linhas no arquivo web.xml entre os tags <web-app> e </web-app>:

```
<servlet>
  <servlet-name>
    Ola
  </servlet-name>
  <servlet-class>
    Ola
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>
    Ola
  </servlet-name>
  <url-pattern>
    /Ola
  </url-pattern>
</servlet-mapping>
```


Invocando em uma página HTML

No caso de uma página HTML basta colocar a URL na forma de link. Por exemplo,

```
<a href="http://localhost:8080/teste/servlet/Ola">Servlet Ola</a>
```

Neste caso o Servlet Ola será solicitado quando o link associado ao texto “Servlet Ola” for acionado.

Diferenças entre as requisições GET e POST

Os dois métodos mais comuns, definidos pelo protocolo HTTP, de se enviar uma requisições a um servidor Web são os métodos POST e GET.

Apesar de aparentemente cumprirem a mesma função, existem diferenças importantes entre estes dois métodos. O método GET tem por objetivo enviar uma requisição por um recurso. As informações necessárias para a obtenção do recurso (como informações digitadas em formulários HTML) são adicionadas à URL e, por consequência, não são permitidos caracteres inválidos na formação de URLs, como por espaços em branco e caracteres especiais. Já na requisição POST os dados são enviados no corpo da mensagem.

O método GET possui a vantagem de ser *idempotente*, ou seja, os servidores Web podem assumir que a requisição pode ser repetida, sendo possível adicionar à URL ao *bookmark*. Isto é muito útil quando o usuário deseja manter a URL resultante de uma pesquisa. Como desvantagem as informações passadas via GET não podem ser muito longas, um vez o número de caracteres permitidos é por volta de 2K.

Já as requisições POST a princípio podem ter tamanho ilimitado. No entanto, elas não são idempotente, o que as tornam ideais para formulários onde os usuários precisam digitar informações confidenciais, como número de cartão de crédito. Desta forma o usuário é obrigado a digitar a informação toda vez que for enviar a requisição, não sendo possível registrar a requisição em um *bookmark*.

Concorrência

Uma vez carregado o Servlet não é mais descarregado, a não ser que o servidor Web tenha sua execução interrompida. De modo geral, cada requisição que deve ser direcionada a determinada instância de Servlet é tratada por um *thread* sobre a instância de Servlet. Isto significa que se existirem duas requisições simultâneas que devem ser direcionadas para um mesmo objeto o *container* criará dois *threads* sobre o mesmo objeto Servlet para tratar as requisições. A figura 7 ilustra esta situação.

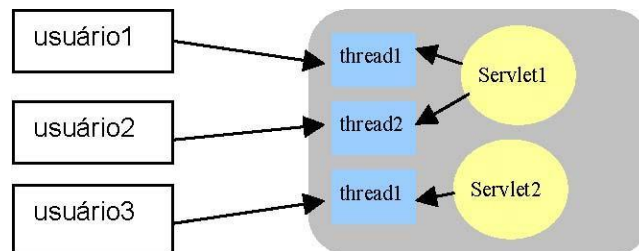


Figura 7. *Relacionamento entre as instâncias dos Servlets e os threads.*

Em consequência disto temos o benefícios de uma sobrecarga para servidor, uma vez que a criação de *threads* é menos onerosa do que a criação de processos, e uma aparente melhora no tempo de resposta.

Por outro lado, o fato dos Servlets operarem em modo multi-thread aumenta a complexidade das aplicações e cuidados especiais, como visto no capítulo sobre concorrência, devem tomados para evitar comportamentos erráticos. Por exemplo, suponha um Servlet que receba um conjunto de números inteiros e retorne uma página contendo a soma dos números. O exemplo abaixo mostra o código do Servlet. O leitor pode imaginar um código muito mais eficiente para computar a soma de números, mas o objetivo do código do exemplo é ilustrar o problema da concorrência em Servlets. O exemplo contém também um trecho de código para recebimento de valores de formulários, o que será discutido mais adiante.

```

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Soma extends HttpServlet {
    Vector v = new Vector(5);
    protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, java.io.IOException {
        v.clear();
        Enumeration e = req.getParameterNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = req.getParameter(name);
            if (value != null) v.add(value);
        }
    }
}
  
```

```
}
res.setContentType("text/html");
java.io.PrintWriter out = res.getWriter();
out.println("<html>");
out.println("<head><title>Servlet</title></head>");
out.println("<body>");
out.println("<h1> A soma e");
int soma =0;
for(int i =0; i< v.size() ; i++) {
    soma += Integer.parseInt((String)v.get(i));
}
out.println(soma);
out.println("<h1>");
out.println("</body>");
out.println("</html>");
out.close();
}
```

Exemplo- Servlet com problemas de concorrência.

Note que o Servlet utiliza uma variável de instância para referenciar o Vector que armazena os valores. Se não forem usadas primitivas de sincronização (como no código do exemplo) e duas requisições simultâneas chegarem ao Servlet o resultado pode ser inconsistente, uma vez que o Vector poderá conter parte dos valores de uma requisição e parte dos valores de outra requisição. Neste caso, para corrigir esse problema basta declarar a variável como local ao método doPost() ou usar primitivas de sincronização.

Obtendo Informações sobre a Requisição

O objeto `HttpServletRequest` passado para o Servlet contém várias informações importantes relacionadas com a requisição, como por exemplo o método empregado (POST ou GET), o protocolo utilizado, o endereço remoto, informações contidas no cabeçalho e muitas outras. O Servlet do exemplo abaixo retorna uma página contendo informações sobre a requisição e sobre o cabeçalho da requisição.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestInfo extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException,
        ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><head>");
        out.println("<title>Exemplo sobre Requisicao de Info </title>");
        out.println("</head><body>");
        out.println("<h3> Exemplo sobre Requisicao de Info </h3>");
        out.println("Metodo: " + req.getMethod()+"<br>");
        out.println("Request URI: " + req.getRequestURI()+"<br>");
        out.println("Protocolo: " + req.getProtocol()+"<br>");
        out.println("PathInfo: " + req.getPathInfo()+"<br>");
        out.println("Endereco remoto: " + req.getRemoteAddr()+"<br><br>");
        Enumeration e = req.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = req.getHeader(name);
            out.println(name + " = " + value+"<br>");
        }
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException,
        ServletException {
        doGet(req, res);
    }
}
```

Exemplo - Servlet que retorna as informações sobre a requisição.

Note que o método `doPost()` chama o método `doGet()`, de modo que o Servlet pode receber os dois tipos de requisição. A figura 8 mostra o resultado de uma execução do Servlet do exemplo anterior.

Exemplo sobre Requisicao de Info

```
Metodo: GET
Request URI: /servlet/RequestInfo
Protocolo: HTTP/1.0
PathInfo: null
Endereco remoto: 127.0.0.1

Connection = Keep-Alive
User-Agent = Mozilla/4.7 [en] (Win95; I)
Pragma = no-cache
Host = localhost:8080
Accept = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding = gzip
Accept-Language = en
Accept-Charset = iso-8859-1,*,utf-8
```

Figura 8. Saída da execução do Servlet que exibe as informações sobre a requisição.

Lidando com Formulários

Ser capaz de lidar com as informações contidas em formulários HTML é fundamental para qualquer tecnologia de desenvolvimento de aplicações para Web. É por meio de formulários que os usuários fornecem dados, preenchem pedidos de compra e (ainda mais importante) digitam o número do cartão de crédito. As informações digitadas no formulário chegam até o Servlet por meio do objeto `HttpServletRequest` e são recuperadas por meio do método `getParameter()` deste objeto. Todo item de formulário HTML possui um nome e esse nome é passado como argumento para o método `getParameter()` que retorna na forma de `String` o valor do item de formulário.

O Servlet do próximo exemplo exibe o valor de dois itens de formulários do tipo text. Um denominado nome e o outro denominado de sobrenome. Em seguida o Servlet cria um formulário contendo os mesmos itens de formulário. Note que um formulário é criado por meio do tag `<form>`. Como parâmetros opcionais deste tag temos método da requisição (`method`), é a URL para onde será submetida a requisição (`action`). No caso do exemplo, o método adotado é o POST e a requisição será submetida ao próprio Servlet Form.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Form extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException,
        ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<head><title>Trata formulario</title></head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h3>Trata formulario</h3>");
        String nome = req.getParameter("nome");
        String sobreNome = req.getParameter("sobrenome");
        if (nome != null || sobreNome != null) {
            out.println("Nome = " + nome + "<br>");
            out.println("Sobrenome = " + sobreNome);
        }
        out.println("<P>");
        out.print("<form action=\"Form\" method=POST>");
        out.println("Nome : <input type=text size=20 name=nome><br>");
        out.println("Sobrenome: <input type=text size=20 name=sobrenome><br>");
        out.println("<input type=submit>");
        out.println("</form>");
    }
}
```

```
        out.println("</body></html>");  
    }  
    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException,  
ServletException { doGet(req, res); }  
}
```


Lidando com Cookies

Um *cookie* nada mais é que um bloco de informação que é enviado do servidor para o navegador no cabeçalho página. A partir de então, dependendo do tempo de validade do *cookie*, o navegador reenvia essa informação para o servidor a cada nova requisição. Dependendo do caso o *cookie* é também armazenado no disco da máquina cliente e quando o site é novamente visitado o *cookie* enviado novamente para o servidor, fornecendo a informação desejada.

Os cookies foram a solução adotada pelos desenvolvedores do Netscape para implementar a identificação de clientes sobre um protocolo HTTP que não é orientado à conexão. Esta solução, apesar das controvérsias sobre a possibilidade de quebra de privacidade, passou ser amplamente adotada e hoje os cookies são parte integrante do padrão Internet, normalizados pela norma RFC 2109.

A necessidade da identificação do cliente de onde partiu a requisição e o monitoramento de sua interação com o site (denominada de *sessão*) é importante para o desenvolvimento de sistemas para a Web pelas seguintes razões:

- É necessário associar os itens selecionados para compra com o usuário que deseja adquiri-los. Na maioria das vezes a seleção dos itens e compra é feita por meio da navegação de várias páginas do site e a todo instante é necessário distinguir os usuários que estão realizando as requisições.
- É necessário acompanhar a interação do usuário com o site para observar seu comportamento e, a partir dessas informações, realizar adaptações no site para atrair um maior número de usuários ou realizar campanhas de marketing.
- É necessário saber que usuário está acessando o site para, de acordo com o seu perfil, fornecer uma visualização e um conjunto de funcionalidades adequadas às suas preferências.

Todas essas necessidades não podem ser atendidas com o uso básico do protocolo HTTP, uma vez que ele não é orientado à sessão ou conexão. Com os *cookies* é possível contornar essa deficiência, uma vez que as informações que são neles armazenadas podem ser usadas para identificar os clientes. Existem outras formas de contornar a deficiência do protocolo de HTTP, como a codificação de URL e o uso de campos escondidos nas páginas HTML, mas o uso de *cookies* é a técnica mais utilizada, por ser mais simples e padronizada. No entanto, o usuário pode impedir que o navegador aceite *cookies*, o que torna o ato de navegar pela Web muito desagradável. Neste caso, é necessário utilizar as outras técnicas para controle de sessão.

A API Servlet permite a manipulação explícita de *cookies*. Para controle de sessão o programador pode manipular diretamente os cookies, ou usar uma abstração de nível mais alto, implementada por meio do objeto `HttpSession`. Se o cliente não permitir o uso de cookies a API Servlet fornece métodos para a

codificação de URL. O exemplo abaixo mostra o uso de *cookies* para armazenar as informações digitadas em um formulário.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieTeste extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException,
ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");
        out.println("<head><title>Teste de Cookies</ title></head>");
        out.println("<body>");
        out.println("<h3>Teste de Cookies</h3>");
        Cookie[] cookies = req.getCookies();
        if (cookies.length > 0) {
            for (int i = 0; i < cookies.length; i++) {
                Cookie cookie = cookies[i];
                out.print("Cookie Nome: " + cookie.getName() + "<br>");
                out.println(" Cookie Valor: " + cookie.getValue() + "<br><br>");
            }
        }
        String cName = req.getParameter("cookiename");
        String cValor = req.getParameter("cookievalor");
        if (cName != null && cValor != null) {
            Cookie cookie = new Cookie(cName ,cValor);
            res.addCookie(cookie);
            out.println("<P>");
            out.println("<br>");
            out.print("Nome : "+cName + "<br>");
            out.print("Valor : "+cValor);
        }
        out.println("<P>");
        out.print("<form action=\"CookieTeste\" method=POST>");
        out.println("Nome : <input type=text length=20 name=cookiename><br>");
        out.println("Valor : <input type=text length=20 name=cookievalor><br>");
        out.println("<input type=submit></form>");
        out.println("</body>");
        out.println("</html>");
    }
    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException,
ServletException {
        doGet(req, res);
    }
}
```

Exemplo - Servlet para lidar com Cookies.

Para se criar um *cookie* é necessário criar um objeto *Cookie*, passando para o construtor um nome e um valor, sendo ambos instâncias de *String*. O *cookie* é enviado para o navegador por meio do método *addCookie()* do objeto

HttpServletResponse. Um vez que os *cookies* são enviados no cabeçalho da página, o método `addCookie()` deve ser chamado antes do envio de qualquer conteúdo para o navegador. Para recuperar os *cookies* enviados pelo navegador usa-se o método `getCookies()` do objeto `HttpServletRequest` que retorna um array de `Cookie`. Os métodos `getName()` e `getValue()` do objeto `Cookie` são utilizados para recuperar o nome o valor da informação associada ao *cookie*.

Os objetos da classe `Cookie` possuem vários métodos para controle do uso de *cookies*. É possível definir tempo de vida máximo do *cookie*, os domínios que devem receber o *cookie* (por default o domínio que deve receber o *cookie* é o que o criou), o diretório da página que deve receber o *cookie*, se o *cookie* deve ser enviado somente sob um protocolo seguro e etc. Por exemplo, para definir a idade máxima de um *cookie* devemos utilizar o método `setMaxAge()`, passando um inteiro como parâmetro. Se o inteiro for positivo indicará em segundos o tempo máximo de vida do *cookie*. Um valor negativo indica que o *cookie* deve apagado quando o navegador terminar. O valor zero indica que o *cookie* deve ser apagado imediatamente. O trecho de código abaixo mostra algumas alterações no comportamento default de um *cookie*.

```
...  
  
Cookie cookie = new Cookie(cName ,cValor);  
cookie.setDomain("*.ufrgs.br"); // todos os domínios como inf.ufrgs.br mas não *.inf.ufrgs.br  
cookie.setMaxAge (3600); // uma hora de tempo de vida  
  
...
```

Exemplo - Mudanças no comportamento default do cookie.

Lidando com Sessões

A manipulação direta de *cookies* para controle de sessão é um tanto baixo nível, uma vez que o usuário deve se preocupar com a identificação, tempo de vida e outros detalhes. Por isso a API Servlet fornece um objeto com controles de nível mais alto para monitorar a sessão, o `HttpSession`. O objeto `HttpSession` monitora a sessão utilizando *cookies* de forma transparente. No entanto, se o cliente não aceitar o uso de *cookies* é possível utilizar como alternativa a codificação de URL para adicionar o identificador da sessão. Essa opção, apesar de ser mais genérica, não é primeira opção devido a possibilidade de criação de gargalos pela necessidade da análise prévia de todas requisições que chegam ao servidor. O exemplo a seguir mostra o uso de um objeto `HttpSession` para armazenar as informações digitadas em um formulário.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionTeste extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException,
ServletException {    resp.setContentType("text/html");

        PrintWriter out = resp.getWriter();
        out.println("<html><head>");
        out.println("<title>Teste de Sessao</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h3>Teste de Sessao</h3>");
        HttpSession session = req.getSession(true);
        out.println("Identificador: " + session.getId());
        out.println("<br>");
        out.println("Data: ");
        out.println(new Date(session.getCreationTime()) + "<br>");
        out.println("Ultimo acesso: ");
        out.println(new Date(session.getLastAccessedTime()));
        String nomedado = req.getParameter("nomedado");
        String valordado = req.getParameter("valordado");
        if (nomedado != null && valordado != null)
        {
            session.setAttribute(nomedado, valordado);
```

```

    }
    out.println("<P>");
    out.println("Dados da Sessão:" + "<br>");
    Enumeration valueNames = session.getAttributeNames();
    while (valueNames.hasMoreElements())
    {
        String name = (String)valueNames.nextElement();
        String value = (String) session.getAttribute(name);
        out.println(name + " = " + value+"<br>");
    }
    out.println("<P>");
    out.print("<form action=\"SessionTeste\" method=POST>");
    out.println("Nome: <input type=text size=20 name=nomedado><br>");
    out.println("Valor: <input type=text size=20 name=valordado><br>");
    out.println("<input type=submit>");
    out.println("</form>");
    out.println("</body></html>");
}

public void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException,
ServletException {
    doGet(req, resp);
}
}

```

Exemplo - Servlet para lidar com Sessões.

Para controlar a sessão é necessário obter um objeto HttpSession por meio do método getSession() do objeto HttpServletRequest. Opcionalmente, o método getSession() recebe como argumento um valor booleano que indica se é para criar o objeto HttpSession se ele não existir (argumento true) ou se é para retorna null caso ele não exista (argumento false). Para se associar um objeto ou informação à sessão usa-se o método setAttribute() do objeto HttpSession, passando para o método um String e um objeto que será identificado pelo String. Note que o método aceita qualquer objeto e, portanto, qualquer objeto pode ser associado à sessão. Os objetos associados a uma sessão são recuperados com o uso método getAttribute() do objeto HttpSession, que recebe como argumento o nome associado ao objeto. Para se obter uma enumeração do nomes associados à sessão usa-se o método getAttributeNames() do objeto HttpSession.

A figura 9 mostra o resultado da execução.

Teste de Sessao

Identificador: session3
 Data: Sun May 28 15:19:15 GMT-03:00 2000
 Ultimo acesso: Sun May 28 15:19:43 GMT-03:00 2000

Dados da Sessao:
 Alcione = 4
 Alexandra = 6

Nome
 Valor

Enviar Consulta

Figura 9. Saída resultante da execução do Servlet que lida com Sessões.

Reencaminhando ou Redirecionando requisições

Existem algumas situações onde pode ser desejável transferir uma requisição para outra URL. Isto é feito com frequência em sistemas que combinam o uso de Servlets juntamente com JSP. No entanto, a transferência pode ser para qualquer recurso. Assim, podemos transferir uma requisição de um Servlet para uma página JSP, HTML ou um Servlet. Da mesma forma uma página JSP pode transferir uma requisição para uma página JSP, HTML ou um Servlet.

Existem dois tipos de transferência de requisição: o *redirecionamento* e o *reencaminhamento*. O redirecionamento é obtido usando o método `sendRedirect()` de uma instância `HttpServletResponse`, passando como argumento a URL de destino. O próximo exemplo mostra o código de um Servlet redirecionando para uma página HTML.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class Redireciona extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws
        ServletException, IOException {
        res.sendRedirect("/test/index.html");
    }
}
```

Exemplo – Redirecionamento de requisição.

Note pelo exemplo que é preciso passar o contexto do recurso (`/teste`). No caso de redirecionamento a requisição corrente é perdida e uma nova requisição é feita para a URL de destino. Por isso não se deve associar nenhum objeto à requisição, uma vez que o objeto `HttpServletRequest` corrente será perdido. O que ocorre na prática é que o servidor envia uma mensagem HTTP 302 de volta para o cliente informando que o recurso foi transferido para outra URL e o cliente envia uma nova requisição para a URL informada.

Já no caso de reencaminhamento a requisição é encaminhada diretamente para a nova URL mantendo todos os objetos associados e evitando uma nova ida ao cliente. Portanto, o uso de reencaminhamento é mais eficiente do que o uso de redirecionamento. O reencaminhamento é obtido usando o método `forward()` de uma instância `RequestDispatcher`, passando como argumento os objetos `HttpServletRequest` e `HttpServletResponse` para a URL de destino. Uma instância `RequestDispatcher` é obtida por meio do método `getRequestDispatcher()` de uma instância `ServletContext`, que é obtido, por sua vez, por meio do método `getServletContext()` do Servlet. O exemplo abaixo mostra o código de um Servlet reencaminhando a requisição para uma página JSP.

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;

public class Reencaminha extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        try {
            getServletContext().getRequestDispatcher("/index.html").
                forward(request,response);
        }
        catch (Exception e) {
            System.out.println("Servlet falhou: ");
            e.printStackTrace();
        }
    }
}
```

Exemplo – Reencaminhamento de requisição.

Note que não é necessário passar o contexto na URL, como é feito no redirecionamento, uma vez que a requisição é encaminhada no contexto corrente.

Java Server Pages (JSP)

Servlets é uma boa idéia, mas você se imaginou montando uma página complexa usando `println()`? Muitas vezes o desenvolvimento de um site é uma tarefa complexa que envolve vários profissionais. A tarefa de projeto do *layout* da página fica a cargo do *Web Designer*, incluindo a diagramação dos textos e imagens, aplicação de cores, tratamento das imagens, definição da estrutura da informação apresentada no site e dos links para navegação pela mesma. Já o Desenvolvedor Web é responsável pela criação das aplicações que vão executar em um site. O trabalho destes dois profissionais é somado na criação de um único produto, mas durante o desenvolvimento a interferência mútua deve ser a mínima possível. Ou seja, um profissional não deve precisar alterar o que é feito pelo outro profissional para cumprir sua tarefa. A tecnologia Servlet não nos permite atingir esse ideal. Por exemplo, suponha que um *Web Designer* terminou o desenvolvimento de uma página e a entregou para o Desenvolvedor Web codificar em um Servlet. Se após a codificação o Web Designer desejar realizar uma alteração na página será necessário que ele altere o código do Servlet (do qual ele nada entende) ou entregar uma nova página para o Desenvolvedor Web para que ele a codifique totalmente mais uma vez. Qualquer uma dessas alternativas são indesejáveis e foi devido a esse problema a Sun desenvolveu uma tecnologia baseada em Servlets chamada de JSP.

Java Server Pages (JSP) são páginas HTML que incluem código Java e outros tags especiais. Desta forma as partes estáticas da página não precisam ser geradas por `println()`. Elas são fixadas na própria página. A parte dinâmica é gerada pelo código JSP. Assim a parte estática da página pode ser projetada por um Web Designer que nada sabe de Java.

A primeira vez que uma página JSP é carregada pelo container JSP o código Java é compilado gerando um Servlet que é executado, gerando uma página HTML que é enviada para o navegador. As chamadas subsequentes são enviadas diretamente ao Servlet gerado na primeira requisição, não ocorrendo mais as etapas de geração e compilação do Servlet.

A figura 10 mostra um esquema das etapas de execução de uma página JSP na primeira vez que é requisitada. Na etapa (1) a requisição é enviada para um servidor Web que reencaminha a requisição (etapa 2) para o *container* Servlet/JSP. Na etapa (3) o container verifica que não existe nenhuma instância de Servlet

correspondente à página JSP. Neste caso, a página JSP é traduzida para código fonte de uma classe Servlet que será usada na resposta à requisição.

Na etapa (4) o código fonte do Servlet é compilado, e na etapa (5) é criada uma instância da classe. Finalmente, na etapa (6) é invocado o método `service()` da instância Servlet para gerar a resposta à requisição.

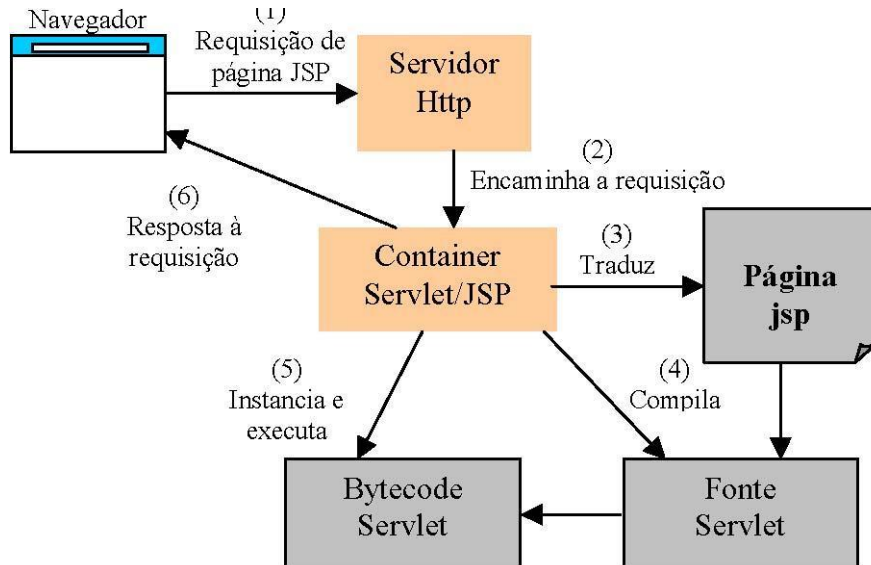


Figura 10. Etapas da primeira execução de uma página JSP.

A idéia de se usar *scripts* de linguagens de programação em páginas HTML que são processados no lado servidor para gerar conteúdo dinâmico não é restrita à linguagem Java. Existem várias soluções desse tipo fornecida por outros fabricantes. Abaixo segue uma comparação de duas das tecnologias mais populares com JSP.

PHP X JSP

PHP (Personal Home Pages) é uma linguagem *script* para ser executada no lado servidor criada em 1994 como um projeto pessoal de Rasmus Lerdorf. Atualmente encontra-se na versão 4. A sintaxe é fortemente baseada em C mas possui elementos de C++, Java e Perl. Possui suporte à programação OO por meio de classes e objetos. Possui também suporte extensivo à Banco de dados ODBC, MySql, Sybase, Oracle e outros. PHP é uma linguagem mais fácil no desenvolvimento de pequenas aplicações para Web em relação à JSP, uma vez que é uma linguagem mais simples e menos rígida do que JSP. No entanto, a medida que passamos para aplicações de maior porte, o uso de PHP não é indicado, uma vez que necessário o uso de linguagens com checagem mais rígidas e com maior suporte à escalabilidade, como é o caso de Java.

ASP X JSP

ASP (Active Server Pages) é a solução desenvolvida pela Microsoft® para atender as requisições feitas à servidores Web. Incorporada inicialmente apenas ao Internet Information Server (IIS), no entanto, atualmente já é suportada por outros servidores populares, como o Apache. O desenvolvimento de páginas que usam ASP envolve a produção de um *script* contendo HTML misturado com blocos de código de controle ASP. Este código de controle pode conter *scripts* em JavaScript ou VBScript. A primeira vantagem de JSP sobre ASP é que a parte dinâmica é escrita em Java e não Visual Basic ou outra linguagem proprietária da Microsoft, portanto JSP é mais poderoso e fácil de usar. Em segundo lugar JSP é mais portátil para outros sistemas operacionais e servidores WEB que não sejam Microsoft.

Primeiro exemplo em JSP

Para que o leitor possa ter uma idéia geral da tecnologia JSP apresentaremos agora a versão JSP do *Olá mundo*. O exemplo a seguir mostra o código da página.

```
<html> <head>
<title>Exemplo JSP</title>
</head>
<body>

<%
String x = "Ol Mundo!"; %> <%=x%>
</body> </html>
```

Exemplo - Versão JSP do Olá mundo.

Quem está habituado aos *tags* HTML notará que se trata basicamente de uma página HTML contendo código Java delimitado pelos símbolos “<%” e “%>”. Para facilitar a visualização destacamos os *scripts* Java com negrito. No primeiro trecho de *script* é declarada uma variável x com o valor “Olá mundo” (a sequência ´ é denota ‘á’ em HTML). No segundo trecho de *script* o conteúdo da variável x é extraído e colocado na página resultante da execução do Servlet correspondente. Em seguida mostraremos como executar o exemplo anterior.

Executando o arquivo JSP

Para executar o save-o com a extensão .jsp. Por exemplo ola.jsp. Se você estiver usando o servidor Tomcat, coloque-o arquivo no subdiretório /webapps/examples/jsp do Tomcat. Por exemplo examples/jsp/teste. Para invocar o arquivo JSP basta embutir a URL em uma página ou digitar diretamente a seguinte URL no navegador.

```
http://localhost:8080/examples/jsp/ola.jsp
```

Usamos o diretório /webapps/examples/jsp para testar rapidamente o exemplo. Para desenvolver uma aplicação é aconselhável criar um diretório apropriado como mostrado na seção que tratou de Servlets.

O Servlet criado a partir da página JSP é colocado em um diretório de trabalho. No caso do Tomcat o Servlet é colocado em subdiretório associado à aplicação subordinado ao diretório /work do Tomcat. O exemplo seguinte mostra os principais trechos do Servlet criado a partir da tradução do arquivo ola.jsp pelo tradutor do Tomcat. Note que o Servlet é subclasse de uma classe HttpJspBase e não da HttpServlet. Além disso, o método que executado em resposta à requisição é o método _jspService() e não o método service(). Note também que todas as partes estáticas da página JSP são colocadas como argumentos do método write() do objeto referenciado out.

```
public class _0002fjsp_0002fola_00032_0002ejspola_jsp_0 extends HttpJspBase {
....
public void _jspService(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException {
....
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    try {
        out.write("<html>\r\n <head>\r\n  <title>Exemplo JSP</title>\r\n </head>\r\n <body>\r\n");
        String x = "Ol&aacute; Mundo!";
        out.write("\r\n");
        out.print(x);
        out.write("\r\n </body>\r\n</html>\r\n");
    } catch (Exception ex) {
        ...
    }
}
```


Objetos implícitos

No exemplo pode-se ver a declaração de variáveis que referenciam a alguns objetos importantes. Estas variáveis estão disponíveis para o projetista da página JSP. As variáveis mais importantes são:

Classe	Variável
HttpServletRequest	request
HttpServletResponse	response
PageContext	pageContext
ServletContext	application
HttpSession	session
JspWriter	out

Os objetos referenciados pelas variáveis request e response já tiveram seu uso esclarecido na seção sobre Servlets. O objeto do tipo JspWriter tem a mesma função do PrintWriter do Servlet. Os outros objetos terão sua função esclarecida mais adiante.

Tags JSP

Os *tags* JSP possuem a seguinte forma geral:

`<% Código JSP %>`

O primeiro caractere % pode ser seguido de outros caracteres que determinam o significado preciso do código dentro do *tag*. Os *tags* JSP possuem correspondência com os *tags* XML. Existem cinco categorias de *tags* JSP:

- Expressões
- Scriptlets
- Declarações
- Diretivas
- Comentários

Em seguida comentaremos cada uma dessas categorias.

Expressões

`<%= expressões %>`

Expressões são avaliadas, convertidas para String e colocadas na página enviada. A avaliação é realizada em tempo de execução, quando a página é requisitada.

Exemplos:

`<%= new java.util.Date() %>`
`<%= request. getMethod() %>`

No primeiro exemplo será colocado na página a data corrente em milésimo de segundos e no segundo será colocado o método usado na requisição. Note que cada expressão contém apenas um comando Java. Note também que o comando Java não é terminado pelo caractere ‘;’.

Scriptlets

`<% código Java %>`

Quando é necessário mais de um comando Java ou o resultado da computação não é para ser colocado na página de resposta é preciso usar outra categoria de tags JSP: os *Scriptlets*. Os Scriptlets permitem inserir trechos de código em Java na página JSP. O exemplo seguinte mostra uma página JSP

contendo um *Scriptlet* que transforma a temperatura digitada em celcius para o equivalente em Fahrenheit.

```
<html>
<head><title>Conversao Celcius Fahrenheit </title></head>
<body>

<% String valor = request.getParameter("celcius");
    if (valor != null ){
        double f = Double.parseDouble(valor)*9/5 +32;
        out.println("<P>");
        out.println("<h2>Valor em Fahrenheit:" +f + "<h2><br>");
    }%>
<form action=conversao.jsp method=POST> Celcius: <input type=text size=20
name=celcius><br> <input type=submit>

</form>

</body> </html>
```

Exemplo - Página JSP que converte graus Celcius para Fahrenheit.

Note o uso das variáveis request e out sem a necessidade de declaração. Todo o código digitado é inserido no método `_jspService()`. A figura 11 mostra o resultado da requisição após a digitação do valor 30 na caixa de texto do formulário.

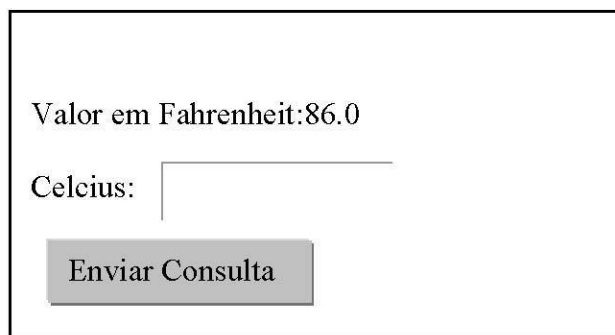


Figura 11. Resultado da conversão de 30 graus celcius.

O código dentro do *scriptlet* é inserido da mesma forma que é escrito e todo o texto HTML estático antes e após ou um *scriptlet* é convertido para comandos `print()`. Desta forma o *scriptlets* não precisa conter comandos para código estático e blocos de controle abertos afetam o código HTML envolvidos por *scriptlets*. O próximo exemplo mostra duas formas de se produzir o mesmo efeito. No código da esquerda os *Scriptlets* se intercalam com código HTML. O código HTML, quando da tradução da página JSP para Servlet é inserido como argumentos de métodos

println() gerando o código da direita. Ambas as formas podem ser usadas em páginas JSP e produzem o mesmo efeito.

<pre>Previs&atilde;o do Tempo <% if (Math.random() < 0.5) { %> Hoje vai fazer sol! <% } else { %> Hoje vai chover! <% } %></pre>	<pre>out.println("Previs&atilde;o do Tempo"); if (Math.random() < 0.5) { out.println(" Hoje vai fazer sol!"); } else { out.println(" Hoje vai chover!"); }</pre>
---	---

Exemplo - Dois códigos equivalentes.

Declarações

```
<%! Código Java %>
```

Uma declaração JSP permite definir variáveis ou métodos que são inseridos no corpo do Servlet. Como as declarações não geram saída, elas são normalmente usadas em combinação com expressões e scriptlets. O Exemplo abaixo mostra a declaração de uma variável que é usada para contar o número de vezes que a página corrente foi requisitada desde que foi carregada.

```
<%! Private int numAcesso = 0; %>
```

Acessos desde carregada:

```
<%= ++ numAcesso %>
```

Exemplo - Declaração de uma variável usando o tag de declaração.

As variáveis declaradas desta forma serão variáveis de instância. Já as variáveis declaradas em *Scriptlets* são variáveis locais ao método _jspService(). Por isso é possível contar o número de requisições. Se variável fosse declarada em um *Scriptlet* a variável seria local ao método _jspService() e, portanto, teria seu valor reinicializado a cada chamada.

Como já foi dito, os tags de declarações permitem a declaração de métodos. O Exemplo seguinte mostra a declaração de um método que converte celcius para Fahrenheit.

```
<%!
```

```
private double converte(double c)
{

return c*9/5 +32;
}
%>
```

Exemplo - Declaração de um método para a conversão de celcius para Fahrenheit.

Comentários

Existem dois tipos de comentários utilizados em páginas JSP. O primeiro exclui todo o bloco comentado da saída gerada pelo processamento da página. A forma geral deste tipo de comentário é a seguinte:

```
<%--comentário --%>
```

O segundo tipo de comentário é o utilizado em páginas HTML. Neste caso o comentário é enviado dentro da página de resposta. A forma geral deste tipo de comentário é a seguinte:

```
<!--comentário.-->
```

Diretivas

Diretivas são mensagens para JSP container. Elas não enviam nada para a página mas são importantes para definir atributos JSP e dependências com o JSP container. A forma geral da diretivas é a seguinte:

```
<%@ Diretiva atributo="valor" %>
```

ou

```
<%@ Diretiva atributo1 ="valor1 " atributo2 ="valor2 " .. atributoN =" valorN " %>
```

Em seguida comentaremos as principais diretivas.

Diretiva page

```
<%@ page atributo1 ="valor1 " ... atributoN =" valorN " %>
```

A diretiva page permite a definição dos seguintes atributos:

```
import
contentType
isThreadSafe
session
buffer
autoflush
info
```

Segue a descrição de

cada um desses atributos.

Atributo e Forma Geral	Descrição
import=" <i>package.class</i> " ou import=" <i>package.class1,.. ..,package.classN</i> "	Permite especificar os pacotes que devem ser importados para serem usados na página JSP. Exemplo: <%@ page import="java.util.*" %>
contentType="MIME-Type"	Especifica o tipo MIME da saída. O <i>default</i> é text/html. Exemplo: <%@ page contentType="text/plain" %>
	possui o mesmo efeito do scriptlet <% response.setContentType("text/plain") ; %>
isThreadSafe="true false"	Um valor true (<i>default</i>) indica um processamento normal do Servlet, onde múltiplas requisições são processadas simultaneamente. Um valor false indica que o processamento deve ser feito por instancias separadas do Servlet ou serialmente.
session="true false"	Um valor true (<i>default</i>) indica que a variável predefinida session (HttpSession) deve ser associada à sessão, se existir, caso contrário uma nova sessão deve ser criada e associada a ela. Um valor false indica que nenhuma sessão será usada.
buffer="sizekb none"	Especifica o tamanho do buffer para escrita usado pelo objeto JspWriter. O tamanho <i>default</i> não é menor que 8k..

autoflush="true false"	Um valor true (<i>default</i>) indica que o buffer deve ser esvaziado quando estiver cheio.
info="mensagem"	Define uma cadeia de caracteres que pode ser recuperada via <code>getServletInfo()</code> .
errorPage="url"	Especifica a página JSP que deve ser processada em caso de exceções não capturadas.
isErrorPage="true false"	Indica se a página corrente pode atuar como página de erro para outra página JSP. O default é false.
Language="java"	Possibilita definir a linguagem que está sendo usada. No momento a única possibilidade é Java.

Tabela 2–Atributos da diretiva `page`.

Diretiva `include`

```
<%@ include file="relative url" %>
```

Permite incluir arquivos no momento em que a página JSP é traduzida em um Servlet.

Exemplo:

```
<%@ include file="/meuarq.html" %>
```


Extraindo Valores de Formulários

Uma página JSP, da mesma forma que um Servlet, pode usar o objeto referenciado pela variável request para obter os valores dos parâmetros de um formulário. O exemplo usado para converter graus Celcius em Fahrenheit fez uso deste recurso. O exemplo abaixo mostra outra página JSP com formulário. Note que o *scriptlet* é usado para obter o nome e os valores de todos os parâmetros contidos no formulário. Como o método `getParameterNames()` retorna uma referência a um objeto Enumeration é preciso importar o pacote `java.util`, por meio da diretiva `page`.

```
<%@ page import="java.util.*" %>
<html><body>
<H1>Formulário</H1>

<%

Enumeration campos = request.getParameterNames();

While(campos.hasMoreElements()) {
String campo = (String)campos.nextElement();
String valor = request.getParameter(campo); %>

<li><%= campo %> = <%= valor %></li>
<% } %>

<form method="POST" action="form.jsp">
Nome: <input type="text" size="20" name="nome" ><br>
Telefone: <input type="text" size="20" name="telefone"><br>
<INPUT TYPE=submit name=submit value="envie">

</form>
</body></html>
```

Exemplo – Página JSP com formulário.

A figura 12 mostra o resultado da requisição após a digitação dos valores Amariles e 33154535 nas caixas de texto do formulário.

Formulário

- telefone = 3315-4535
- nome = Amariles
- submit = envie

Nome:

Telefone:

Figura 12- Saída.

Criando e Modificando Cookies

Da mesma forma que em Servlets os *cookies* em JSP são tratados por meio da classe `Cookie`. Para recuperar os *cookies* enviados pelo navegador usa-se o método `getCookies()` do objeto `HttpServletRequest` que retorna um arranjo de `Cookie`. Os métodos `getName()` e `getValue()` do objeto `Cookie` são utilizados para recuperar o nome o valor da informação associada ao *cookie*. O *cookie* é enviado para o navegador por meio do método `addCookie()` do objeto `HttpServletResponse`. O exemplo seguinte mostra uma página JSP que exibe todos os *cookies* recebidos em uma requisição e adiciona mais um na resposta.

```
<html><body>
<H1>Session id: <%= session.getId() %></H1>

<%
Cookie[] cookies = request.getCookies();

For(int i = 0; i < cookies.length; i++) { %>
Cookie name: <%= cookies[i].getName() %> <br>
Value: <%= cookies[i].getValue() %><br>
antiga idade máxima em segundos:
<%= cookies[i].getMaxAge() %><br>
<% cookies[i].setMaxAge(5); %>

nova idade máxima em segundos:

<%= cookies[i].getMaxAge() %><br>
<% } %>
<%! Int count = 0; int dcount = 0; %>
<% response.addCookie(new Cookie(
"Cookie " + count++, "Valor " + dcount++)); %>

</body></html>
```

Exemplo – Página JSP que exibe os cookies recebidos.

A figura 13 mostra o resultado após três acessos seguidos à página JSP. Note que existe um *cookie* a mais com o nome `JSESSIONID` e valor igual à sessão. Este *cookie* é o usado pelo container para controlar a sessão.

Session id: 9ppfv0ls11

Cookie name: Cookie 0

value: Valor 0

antiga idade máxima em segundos: -1

nova idade máxima em segundos: 5

Cookie name: Cookie 1

value: Valor 1

antiga idade máxima em segundos: -1

nova idade máxima em segundos: 5

Cookie name: JSESSIONID

value: 9ppfv0ls11

antiga idade máxima em segundos: -1

nova idade máxima em segundos: 5

Figura 13- Saída do exemplo anterior após três acessos.

Lidando com sessões

O atributos de uma sessão são mantidos em um objeto HttpSession referenciado pela variável session. Pode-se armazenar valores em uma sessão por meio do método setAttribute() e recuperá-los por meio do método getAttribute(). O tempo de duração *default* de uma sessão inativa (sem o recebimento de requisições do usuário) é 30 minutos mas esse valor pode ser alterado por meio do método setMaxInactiveInterval(). O exemplo abaixo mostra duas páginas JSP. A primeira apresenta um formulário onde podem ser digitados dois valores recebe dois valores de digitados em um formulário e define o intervalo máximo de inatividade de uma sessão em 10 segundos. A segunda página recebe a submissão do formulário, insere os valores na sessão e apresenta os valores relacionados com a sessão assim como a identificação da sessão.

```
<%@ page import="java.util.*" %>
<html><body>
<H1>Formulário</H1>
<H1>Id da sess&atilde;o: <%= session.getId() %></H1>
<H3><li>Essa sess&atilde;o foi criada em
<%= session.getCreationTime() %></li></H3>

<H3><li>Antigo intervalo de inatividade =
<%= session.getMaxInactiveInterval() %></li>

<% session.setMaxInactiveInterval(10); %>
<li>Novo intervalo de inatividade=
<%= session.getMaxInactiveInterval() %></li>
</H3>
<%
Enumeration atribs = session.getAttributeNames();
while(atrib.hasMoreElements()) {

String atrib = (String)atrib.nextElement();
String valor = (String)session.getAttribute(atrib); %>
<li><%= atrib %> = <%= valor %></li>
<% } %>
<form method="POST" action="sessao2.jsp">
Nome: <input type="text" size="20" name="nome" ><br>
Telefone: <input type="text" size="20" name="telefone" >
<br>
<INPUT TYPE=submit name=submit value="envie">
</form>
</body></html>
```



```
<html><body>
<H1>Id da sessão: <%= session.getId() %></H1>

<%
String nome = request.getParameter("nome");
String telefone = request.getParameter("telefone");

if (nome !=null && nome.length()>0)
session.setAttribute("nome",nome);
if (telefone !=null &&telefone.length()>0)
session.setAttribute("telefone",telefone);
%>

<FORM TYPE=POST ACTION=sessao1.jsp>
<INPUT TYPE=submit name=submit Value="Retorna">
</FORM>
</body></html>
```

Exemplo – Exemplo do uso de sessão.

O exemplo mostra que a sessão é mantida mesmo quando o usuário muda de página. As figuras 14 e 15 mostram o resultado da requisição após a digitação dos valores Alcione e 333-3333 nas caixas de texto do formulário, à submissão para página sessao2.jsp e o retorno à página sessao1.jsp.

Formulário

Id da sessão: soo8utc4m1

Essa sessão foi criada em 1002202317590

Antigo intervalo de inatividade = 1800

Novo intervalo de inatividade= 10

telefone = 333-3333

nome = Alcione

Nome:

Telefone:

Figura 14- Tela da página sessao1.jsp.

Id da sessão: soo8utc4m1

Retorna

Figura 15. Tela da página *sessao2.jsp*.

O Uso de JavaBeans

A medida que o código Java dentro do HTML torna-se cada vez mais complexo o desenvolvedor pode-se perguntar: Java em HTML não é o problema invertido do HTML em Servlet? O resultado não será tão complexo quanto produzir uma página usando `println()`? Em outras palavras, estou novamente misturando conteúdo com forma?

Para solucionar esse problema a especificação de JSP permite o uso de JavaBeans para manipular a parte dinâmica em Java. JavaBeans já foram descritos detalhadamente em um capítulo anterior, mas podemos encarar um JavaBean como sendo apenas uma classe Java que obedece a uma certa padronização de nomeação de métodos, formando o que é denominado de *propriedade*. As propriedades de um bean são acessadas por meio de métodos que obedecem a convenção `getXxx` e `setXxx`, onde `Xxx` é o nome da propriedade. Por exemplo, `getItem()` é o método usado para retornar o valor da propriedade `item`. A sintaxe para o uso de um bean em uma página JSP é:

```
<jsp:useBean id="nome" class="package.class" />
```

Onde `nome` é o identificador da variável que conterà uma referência para uma instância do JavaBean. Você também pode modificar o atributo `scope` para estabelecer o escopo do bean além da página corrente.

```
<jsp:useBean id="nome" scope="session" class="package.class" />
```

Para modificar as propriedades de um JavaBean você pode usar o `jsp:setProperty` ou chamar um método explicitamente em um scriptlet. Para recuperar o valor de uma propriedade de um JavaBean você pode usar o `jsp:getProperty` ou chamar um método explicitamente em um scriptlet. Quando é dito que um bean tem uma propriedade `prop` do tipo `T` significa que o bean deve prover um método `getProp()` e um método do tipo `setProp(T)`. O exemplo abaixo mostra uma página JSP e um JavaBean. A página instancia o JavaBean, altera a propriedade mensagem e recupera o valor da propriedade, colocando-o na página.

Página `bean.jsp`

```
<HTML> <HEAD>
```

```
<TITLE>Uso de beans</TITLE>

</HEAD> <BODY> <CENTER>
<TABLE BORDER=5> <TR><TH CLASS="TITLE"> Uso de  JavaBeans </TABLE>
</CENTER> <P>

<jsp:useBean id="teste" class="curso.BeanSimples" />
<jsp:setProperty name="teste" property="mensagem" value="Ola mundo!" />

<H1> Mensagem: <I>
<jsp:getProperty name="teste" property="mensagem" /> </I></H1>
</BODY> </HTML>
```

Arquivo Curso/BeanSimples.java

```
package curso;

public class BeanSimples {
private String men = "Nenhuma mensagem";

public String getMensagem() {
return(men);
}

public void setMensagem(String men) {
this.men = men;
}

}
```

Exemplo – Exemplo do uso de JavaBean.

A figura 16 mostra o resultado da requisição dirigida à página bean.jsp.



Figura 16- Resultado da requisição à página bean.jsp.

Se no *tag* `setProperty` usarmos o valor “*” para o atributo `property` então todos os valores de elementos de formulários que possuírem nomes iguais à propriedades serão transferidos para as respectivas propriedades no momento do processamento da requisição. Por exemplo, seja uma página jsp contendo um formulário com uma caixa de texto com nome `mensagem`, como mostrado no exemplo abaixo. Note que, neste caso, a propriedade `mensagem` do JavaBean tem seu valor atualizado para o valor digitado na caixa de texto, sem a necessidade de uma chamada explícita no

tag `setProperty`. Os valores são automaticamente convertidos para o tipo correto no bean.

```
<HTML> <HEAD><TITLE>Uso de beans</TITLE> </HEAD>
<BODY> <CENTER>
<TABLE BORDER=5> <TR><TH CLASS="TITLE"> Uso de  JavaBeans </TABLE>
</CENTER> <P>

<jsp:useBean id="teste" class="curso.BeanSimples" /> <jsp:setProperty name="teste"
property="*" />
<H1> Mensagem: </>
<jsp:getProperty name="teste" property="mensagem" />
</></H1>
<form method="POST" action="bean2.jsp">   Texto: <input type="text" size="20"
name="mensagem" ><br>   <INPUT TYPE=submit name=submit value="envie">
</form>

</BODY> </HTML>
```

Exemplo – Exemplo de atualização automática da propriedade.

A figura 17 mostra o resultado da requisição dirigida à página `bean2.jsp` após a digitação do texto Olá!



The screenshot shows a web browser window. At the top, there is a title bar that says "Uso de JavaBeans". Below the title bar, the main content area displays "Mensagem: Ola!". Underneath this, there is a form with a label "Texto:" followed by a text input field. Below the input field is a button labeled "envie".

Figura 17- Resultado da requisição à página `bean2.jsp`.

Escopo

Existem quatro valores possíveis para o escopo de um objeto: page, request, session e application. O default é page. A tabela 3 descreve cada tipo de escopo.

Escopo	Descrição
page	Objetos declarados com nesse escopo são válidos até a resposta ser enviada ou a requisição ser encaminhada para outro programa no mesmo ambiente, ou seja, só podem ser referenciados nas páginas onde forem declarados. Objetos declarados com escopo page são referenciados pelo objeto pagecontext.
request	Objetos declarados com nesse escopo são válidos durante a requisição e são acessíveis mesmo quando a requisição é encaminhada para outro programa no mesmo ambiente. Objetos declarados com escopo request são referenciados pelo objeto request.
session	Objetos declarados com nesse escopo são válidos durante a sessão desde que a página seja definida para funcionar em uma sessão. Objetos declarados com escopo session são referenciados pelo objeto session.
application	Objetos declarados com nesse escopo são acessíveis por páginas no mesmo servidor de aplicação. Objetos declarados com escopo application são referenciados pelo objeto application.

Tabela 3 –Escopo dos objetos nas páginas JSP.

Implementação de um Carrinho de compras

O exemplo abaixo ilustra o uso de JSP para implementar um carrinho de compras virtual. O carrinho de compras virtual simula um carrinho de compras de supermercado, onde o cliente vai colocando os produtos selecionados para compra até se dirigir para o caixa para fazer o pagamento. No carrinho de

compras virtual os itens selecionados pelo usuário são armazenados em uma estrutura de dados até que o usuário efetue o pagamento. Esse tipo de exemplo exige que a página JSP funcione com o escopo session para manter o carrinho de compras durante a sessão. O exemplo compras.jsp mostra um exemplo simples de

implementação de carrinho de compras. O exemplo é composto por dois arquivos: um para a página JSP e um para o JavaBean que armazena os itens selecionados.

Página compras.jsp

```
<html>
<jsp:useBean id="carrinho" scope="session" class="compra.Carrinho" />
<jsp:setProperty name="carrinho" property="*" />
<body bgcolor="#FFFFFF">
<%
carrinho.processRequest(request);
String[] items = carrinho.getItems();
if (items.length>0) {
%>
<font size=+2 color="#3333FF">Voc&ecirc; comprou os seguintes itens:</font>
<ol>
<%
for (int i=0; i<items.length; i++) {
out.println("<li>" + items[i]);
}
}
%>
</ol>
<hr>
<form type=POST action= compras.jsp>
<br><font color="#3333FF" size=+2>Entre um item para adicionar ou remover:
</font><br>
<select NAME="item">
<option>Televis&atilde;o
<option>R&aacute;dio
<option>Computador
<option>V&iacute;deo Cassete
</select>
<p><input TYPE=submit name="submit" value="adicione">
<input TYPE=submit name="submit" value="remova"></form>
```

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

JavaBean compra/Carrinho.java

```
Package compra;
Import javax.servlet.http.*;
Import java.util.Vector;
Import java.util.Enumeration;
Public class Carrinho {
    Vector v = new Vector();
    String submit = null;
    String item = null;
    Private void addItem(String name) {
        v.addElement(name); }
    Private void removeItem(String name) {
        v.removeElement(name); }
    Public void setItem(String name) {
        item = name; }
    Public void setSubmit(String s) {
        submit = s; }
    Public String[] getItems() {
        String[] s = new String[v.size()];
        v.copyInto(s);
        return s;
    }
    private void reset() {
        submit = null;
        item = null;
    }
    public void processRequest(HttpServletRequest request)
    {
        if (submit == null) return;
```



```
if (submit.equals("adicione")) addItem(item);  
else if (submit.equals("remova")) removeItem(item);  
reset();  
}  
}
```

Exemplo – Implementação de um carrinho de compras Virtual.

O exemplo anterior implementa apenas o carrinho de compras, deixando de fora o pagamento dos itens, uma vez que esta etapa depende de cada sistema. Geralmente o que é feito é direcionar o usuário para outra página onde ele digitará o número do cartão de crédito que será transmitido por meio de uma conexão segura para o servidor. Existem outras formas de pagamento, como boleto bancário e dinheiro virtual. O próprio carrinho de compras geralmente é mais complexo, uma vez que os para compra devem ser obtidos dinamicamente de um banco de dados. A figura 18 mostra a tela resultante de algumas interações com o carrinho de compras.

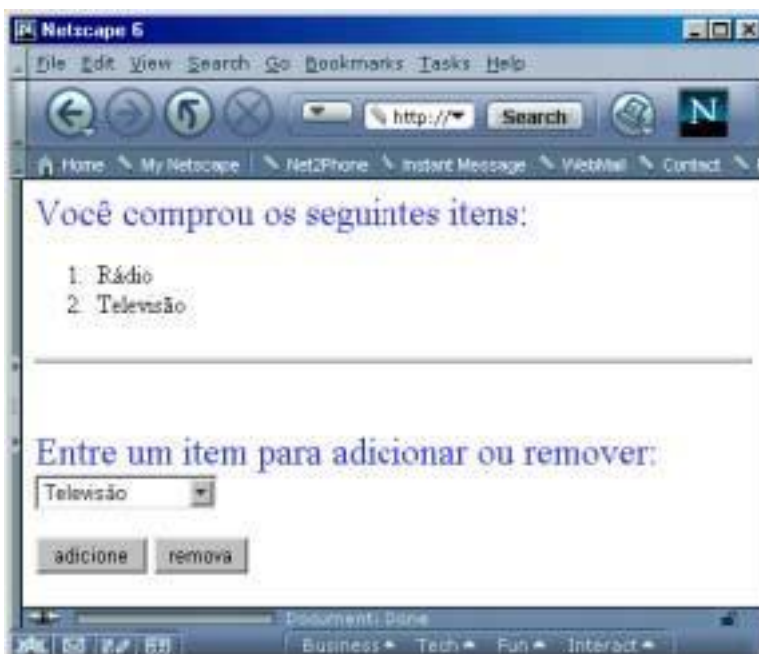


Figura 18- Carrinho de compras virtual.

Uma Arquitetura para comércio eletrônico

O projeto de uma solução para comércio eletrônico é uma tarefa complexa e deve atender diversos requisitos. Nesta seção mostraremos um modelo de arquitetura básico para comércio eletrônico que pode ser adaptado para soluções mais específicas. Este modelo implementa o padrão de projeto MVC, procurando, desta forma, isolar esses aspectos de um sistema de computação.

Tipos de aplicações na WEB

Podemos enquadrar as aplicações na Web em um dos seguintes tipos:

- **Business-to-consumer (B2C)** – entre empresa e consumidor. Exemplo: uma pessoa compra um livro na Internet.

- **Business-to-business (B2B)** – Troca de informações e serviços entre empresas. Exemplo: o sistema de estoque de uma empresa de automóveis detecta que um item de estoque precisa ser resposta e faz o pedido diretamente ao sistema de produção do fornecedor de autopeças. Neste tipo de aplicação a linguagem XML possui um papel muito importante, uma vez que existe a necessidade de uma padronização dos *tags* para comunicação de conteúdo.

- **User-to-data** – acesso à bases de informação. Exemplo: um usuário consulta uma base de informação.

- **User-to-user** – chat, e troca de informações entre usuários (napster).

O exemplo que mostraremos é tipicamente um caso de User-to-data, (agenda eletrônica na Web) mas possui a mesma estrutura de um B2C.

Arquitetura MVC para a Web

A figura 19 contém um diagrama de blocos que mostra a participação de Servlets, JSP e JavaBeans na arquitetura proposta. A idéia é isolar cada aspecto do modelo MVC com a tecnologia mais adequada. A página JSP é ótima para fazer o papel da visão, uma vez que possui facilidades para a inserção de componentes visuais e para a apresentação de informação. No entanto, é um pouco estranho usar uma página JSP para receber e tratar uma requisição. Esta tarefa, que se enquadra no aspecto de controle do modelo MVC é mais adequada a um Servlet, uma vez que neste momento componentes de apresentação são indesejáveis. Finalmente, é desejável que a modelagem do negócio fique isolada dos aspectos de interação. A proposta é que a modelagem do negócio fique contida em classes de JavaBeans. Em aplicações mais sofisticadas a modelagem do negócio deve ser implementada por classes de Enterprise JavaBeans (EJB), no entanto esta forma de implementação foge ao escopo da apostila. Cada componente participa da seguinte forma:

- Servlets – Atuam como controladores, recebendo as requisições dos usuários. Após a realização das análises necessária sobre a requisição, instancia o JavaBean e o armazena no escopo adequado (ou não caso o bean já tenha sido criado no escopo) e encaminha a requisição para a página JSP.

- JavaBeans – Atuam como o modelo da solução, independente da requisição e da forma de apresentação. Comunicam-se com a camada intermediária que encapsula a lógica do problema.

- JSP – Atuam na camada de apresentação utilizando os JavaBeans para obtenção dos dados a serem exibidos, isolando-se assim de como os dados são obtidos. O objetivo é minimizar a quantidade de código colocado na página.

- Camada Intermediária (Middleware) – Incorporam a lógica de acesso aos dados. Permitem isolar os outros módulos de problemas como estratégias de acesso aos dados e desempenho. O uso de EJB (Enterprise JavaBeans) é recomendado para a implementação do Middleware, uma vez que os EJBs possuem capacidades para gerência de transações e persistência. Isto implica na adoção de um servidor de aplicação habilitado para EJB.

A figura 19 mostra a interação entre os componenetes.

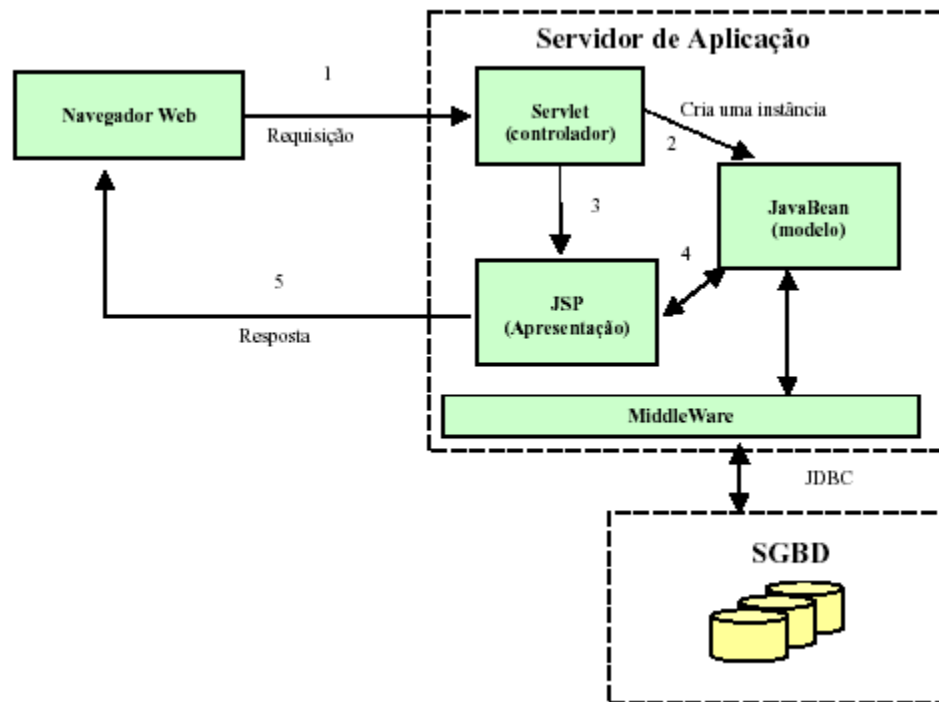


Figura 19. Arquitetura de uma aplicação para Comércio Eletrônico.

Essa arquitetura possui as seguintes vantagens:

1. Facilidade de manutenção: a distribuição lógica das funções entre os módulos do sistema isola o impacto das modificações.
2. Escalabilidade: Modificações necessária para acompanhar o aumento da demanda de serviços (database pooling, clustering, etc) ficam concentradas na camada intermediária.

A figura 20 mostra a arquitetura física de uma aplicação de comércio eletrônico.

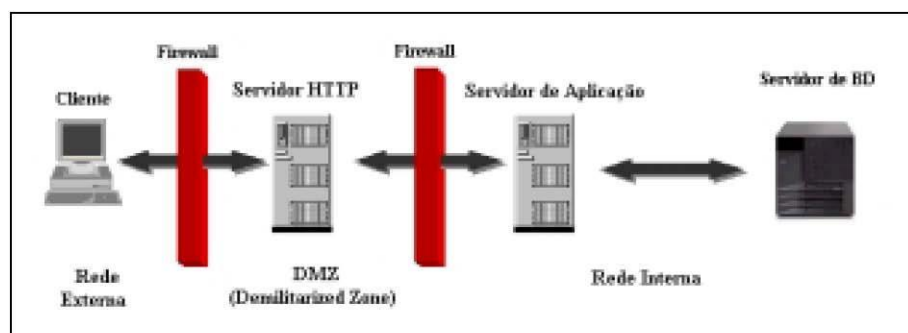


Figura 20. Arquitetura física de uma aplicação para Comércio Eletrônico.

Demilitarized Zone (DMZ) é onde os servidores HTTP são instalados. A DMZ é protegida da rede pública por um firewall, também chamado de firewall de protocolo. O firewall de protocolo deve ser configurado para permitir tráfego apenas através da porta 80. Um segundo firewall, também chamado de firewall de domínio separa a DMZ da rede interna. O firewall de domínio deve ser configurado para permitir comunicação apenas por meio das portas do servidor de aplicação.

Agenda Web:

Um Exemplo de uma aplicação Web usando a arquitetura MVC

O exemplo a seguir mostra o desenvolvimento da agenda eletrônica para o funcionamento na Web. A arquitetura adotada é uma implementação do modelo MVC. Apenas, para simplificar a solução, a camada intermediária foi simplificada e é implementada por um JavaBean que tem a função de gerenciar a conexão com o banco de dados. O banco de dados será composto por duas tabelas, uma para armazenar os usuários autorizados a usar a tabela e outra para armazenar os itens da agenda. A figura 21 mostra o esquema conceitual do banco de dados e a figura 22 mostra o comando para a criação das tabelas. Note que existe um relacionamento entre a tabela USUARIO e a tabela PESSOA, mostrando que os dados pessoais sobre o usuário ficam armazenados na agenda.

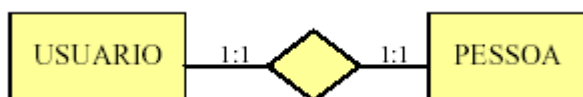


Figura 21. Esquema conceitual do banco de dados para a agenda.

As tabelas do BD devem ser criadas de acordo com o seguinte script:

```
CREATE TABLE PESSOA (ID INT PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL,  
TELEFONE VARCHAR(50),  
ENDERECO VARCHAR(80),  
EMAIL VARCHAR(50),  
HP VARCHAR(50),  
CELULAR VARCHAR(20),  
DESCRICAO VARCHAR(80));
```

```
CREATE TABLE USUARIO (ID INT PRIMARY KEY,  
LOGIN VARCHAR(20) NOT NULL,  
SENHA VARCHAR(20) NOT NULL,  
CONSTRAINT FK_USU FOREIGN KEY (ID)  
REFERENCES PESSOA(ID));
```

Figura 22. Script para criação das tabelas.

Para se usar a agenda é necessário que exista pelo menos um usuário cadastrado. Como no exemplo não vamos apresentar uma tela para cadastro de usuários será preciso cadastrá-los por meio comandos SQL. Os comandos da figura 23 mostram como cadastrar um usuário.

```
INSERT INTO PESSOA(ID,NOME,TELEFONE,ENDERECO,EMAIL)
VALUES(0,'Amariles Lopes','3315-4535,
        'Andradas,105','amarileslopes@ig.com.br');
INSERT INTO USUARIO(ID,LOGIN,SENHA) VALUES(0,'Amariles','senha');
```

Figura 23. Script para cadastra um usuário.

O sistema **e-agenda** é composta pelos seguintes arquivos:

Arquivo	Descrição
agenda.html	Página inicial do site, contendo o formulário para a entrada do login e senha para entrar no restante do site.
principal.jsp	Página JSP contendo o formulário para entrada de dados para inserção, remoção ou consulta de itens da agenda.
LoginBean.java	JavaBean responsável por verificar se o usuário está autorizado a acessar a agenda.
AgendaServlet.java	Servlet responsável pelo tratamento de requisições sobre alguma função da agenda (consulta, inserção e remoção)
AcaoBean.java	JavaBean responsável pela execução da ação solicitada pelo usuário.
ConnectionBean.java	JavaBean responsável pelo acesso ao DB e controle das conexões.

Tabela 4. Arquivos do sistema e-agenda.

O diagrama de colaboração abaixo mostra as interação entre os componentes do sistema.

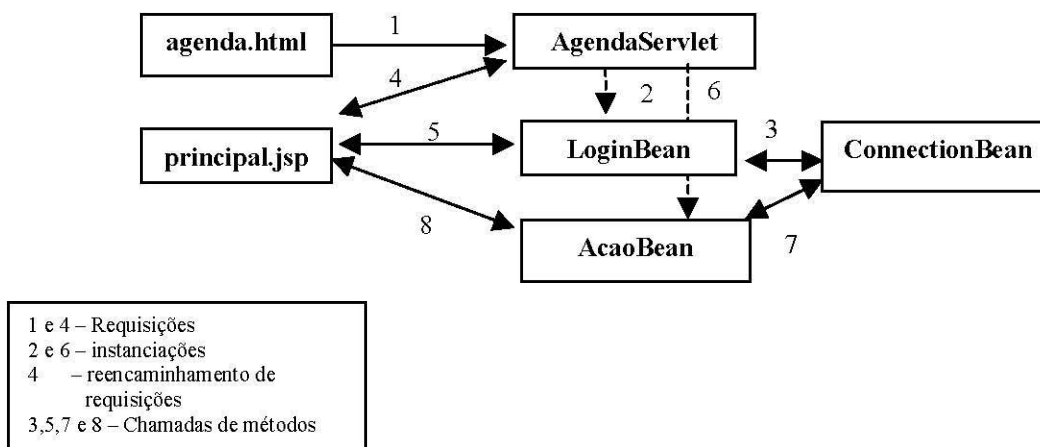


Figura 24. Interação entre os componentes do sistema.

Descreveremos agora cada componente da aplicação. O exemplo agenda.html mostra código HTML da página agenda.html. Esta é a página inicial da aplicação. Ela contém o formulário para a entrada do login e senha para entrar no restante do site.

```
1      <HTML>
2      <HEAD>
4      <TITLE>Agenda</TITLE>
5      </HEAD>
6      <BODY BGCOLOR="#FFFFFF">
7      <P align="center"><IMG src="tit.gif" width="350" height="100" border="0"></P>
8      <BR>
9
10     <CENTER>
11     <FORM method="POST" name="TesteSub" onsubmit="return TestaVal()"
12     action="/agenda/agenda"><BR>
13     Login:<INPUT size="20" type="text" name="login"><BR><BR>
14     Senha:<INPUT size="20" type="password" name="senha"><BR><BR><BR>
15     <INPUT type="submit" name="envia" value="Enviar">
16     <INPUT size="3" type="Hidden" name="corrente" value="0"><BR>
17     </FORM>
18     </CENTER>
19     </BODY>
20     </HTML>
```

Exemplo – agenda.html.

O formulário está definido nas linha 11 a 17. Na linha 12 o parâmetro action indica a URL que deve receber a requisição. A URL é virtual e sua associação com o Servlet AgendaServlet será definida no arquivo web.xml. Na linha 16 é definido um campo oculto (Hidden) como o nome de corrente e valor 0. Ele será usado pelo AgendaServlet reconhecer a página de onde saiu a requisição. As linha 19 a 31 definem uma função em JavaScript que será usada para verificar se o usuário digitou o nome e a senha antes de enviar a requisição ao usuário. O uso de JavaScript no lado cliente para criticar a entrada do usuário é muito comum pois diminui a sobrecarga do servidor.

O exemplo a seguir mostra código da página principal.jsp. Esta página contém o formulário para entrada de dados para inserção, remoção ou consulta de itens da agenda.

```

1      <HTML><HEAD>
2      <TITLE>Tela da Agenda </TITLE>
3      </HEAD><BODY bgcolor="#FFFFFF">
4      <% @ page session="true" import="agenda.*" %>
5
6      <%
7      agenda.LoginBean lb = (agenda.LoginBean) session.getAttribute("loginbean");
8      agenda.AcaoBean ab = (agenda.AcaoBean) request.getAttribute("acaobean");
9      if (lb != null && lb.getStatus())
10     { %>
11     <H2>Sess&atilde;o do <%= lb.getNome() %></H2>
12     <%
13     if (ab!=null) out.println(ab.toString());
14     %>
15
16     <P><BR></P>
17     <FORM method="POST" name="formprin" onsubmit="return TestaVal()"
18     action="/agenda/agenda">
19     Nome: <INPUT size="50" type="text" name="nome"><BR>
20     Telefone: <INPUT size="20" type="text" name="telefone"><BR>
21     Endere&ccedil;o: <INPUT size="50" type="text" name="endereco"><BR>
22     Email: <INPUT size="50" type="text" name="email"><BR><BR>
23     P&aacute;gina: <INPUT size="50" type="text" name="pagina"><BR>
24     Celular: <INPUT size="20" type="text" name="celular"><BR>
25     Descri&ccedil;&atilde;o: <INPUT size="20" type="text" name="descricao">
26     <BR><CENTER>
27     <INPUT type="submit" name="acao" value="Consulta">
28     <INPUT type="submit" name="acao" value="Insere">
29     <INPUT type="submit" name="acao" value="Apaga"></CENTER>
30     <INPUT size="3" type="Hidden" name="corrente" value="1">
31     </FORM>

```

```

32
33     <SCRIPT language="JavaScript"> <!--
34     function TestaVal()
35     {
36     if (document.formprin.nome.value == "" &&
37     document.formprin.descricao.value== "")
38     {
39     alert ("Campo Nome ou Descricao devem ser Preenchidos!")
40     return false
41     }
42     else
43     {
44     return true
45     }
46     }
47     //--></SCRIPT>
48
49     <%
50     }
51     else
52     {
53     session.invalidate();
54     %>
55     <H1>Usuário não autorizado</H1>
56     <%
57     }
58     %>
59     </BODY></HTML>

```

Exemplo – principal.jsp.

Na linha 4 a diretiva page define que o servidor deve acompanhar a sessão do usuário e importa o pacote agenda.

Na linha 7 um objeto da classe agenda.LoginBean é recuperado da sessão por meio do método `getAttribute()`. Para recuperar o objeto é preciso passar para o método o nome que está associado ao objeto na sessão.

De forma semelhante, na linha 8 um objeto da classe agenda.AcaoBean é recuperado da requisição por meio do método `getAttribute()`. Este objeto é recuperado da requisição porque cada requisição possui uma ação diferente associada.

Na linha 9 é verificado se objeto agenda.LoginBean foi recuperado e se o retorno do método `getStatus()` é `true`. Se o objeto agenda.LoginBean não foi recuperado significa que existe uma tentativa de acesso direto à página principal.jsp sem passar primeiro pela página agenda.html ou que a sessão se esgotou. Se o método `getStatus()` retornar `false` significa que o usuário não está autorizado a acessar essa página. Nestes casos é processado o código associado ao comando `else` da linha 51 que apaga a sessão por meio do método `invalidate()` do objeto `HttpSession` (linha 53) e mostra a mensagem “Usuário não autorizado” (linha 55). Caso o objeto indique que o usuário está autorizado os comandos internos ao `if` são executados.

Na linha 11 é mostrada uma mensagem com o nome do usuário obtido por meio do método `getNome()` do objeto agenda.LoginBean . Na linha 13 é mostrado o resultado da ação anterior por meio do método `toString()` do objeto agenda.AcaoBean. A ação pode ter sido de consulta, inserção de um novo item na agenda e remoção de um item na agenda.

No primeiro caso é mostrado uma lista dos itens que satisfizeram a consulta.

No segundo e terceiro casos é exibida uma mensagem indicado se a operação foi bem sucedida.

As linhas 17 a 31 definem o código do formulário de entrada.

Nas linhas 17 e 18 são definidos os atributos do formulário. O atributo `method` indica a requisição será enviada por meio do método `POST`. O atributo `name` define o nome do formulário como sendo `formprin`. O atributo `onsubmit` define que a função `javaSript TestaVal()` deve ser executada quando o formulário for submetido.

Finalmente, o atributo `action` define a URL para onde a requisição deve ser enviada. Neste caso a URL é `agenda/agenda` que está mapeada para o Servlet `AgendaServlet`. O mapeamento é feito no arquivo `web.xml` do diretório `web-inf` do contexto `agenda`, como mostrado na figura 24.

As linhas 19 a 25 definem os campos de texto para entrada dos valores.

As linhas 27 a 29 definem os botões de submit. Todos possuem o mesmo nome, de forma que o Servlet precisa apenas examinar o valor do parâmetro `acao` para determinar qual ação foi solicitada

Na linha 30 é definido um campo oculto (`Hidden`) como o nome de corrente e valor 0. Ele será usado pelo `AgendaServlet` reconhecer a página de onde saiu a requisição.

As linha 33 a 47 definem uma função em JavaScript que será usada para verificar se o usuário entrou com valores nos campos de texto nome ou decricao.

No mínimo um desses campos deve ser preenchido para que uma consulta possa ser realizada.

O exemplo `ConnectionBean.java` mostra código do JavaBean usado para intermediar a conexão com o banco de dados. O JavaBean `ConnectionBean` tem a responsabilidade de abrir uma conexão com o banco de dados, retornar uma referência desta conexão quando solicitado e registrar se a conexão esta livre ou ocupada. Neste exemplo estamos trabalhando com apenas uma conexão com o banco de dados porque a versão gerenciador de banco de dados utilizado (`PointBase™`), por ser um versão limitada, permite apenas uma conexão aberta.

Se o SGBD permitir varias conexões simultâneas pode ser necessário um maior controle sobre as conexões, mantendo-as em uma estrutura de dados denominada de *pool de conexões*.

Na linha 12 podemos observar que o construtor da classe foi declarado com o modificador de acesso `private`. Isto significa que não é possível invocar o construtor por meio de um objeto de outra classe. Isto é feito para que se possa ter um controle sobre a criação de instâncias da classe.

No nosso caso permitiremos apenas que uma instância da classe seja criada, de modo que todas as referências apontem para esse objeto. Esta técnica de programação, onde se permite uma única instância de uma classe é denominada de padrão de projeto *Singleton*. O objetivo de utilizarmos este padrão é porque desejamos que apenas um objeto controle a conexão com o banco de dados.

Se o construtor não pode ser chamado internamente como uma instância da classe é criada e sua referência é passada para outros objetos? Esta é a tarefa do método estático `getInstance()` (linhas 14 a 19). Este método verifica se já existe a instância e retorna a referência. Caso a instância não exista ela é criada antes de se retornar a referência.

O método `init()` (linhas 21 a 27) é chamado pelo construtor para estabelecer a conexão com o SGBD. Ele carrega o driver JDBC do tipo 4 para `PointBase` e obtém uma conexão com o SGBD.

O método `devolveConnection()` (linhas 29 a 34) é chamado quando se deseja devolver a conexão. Finalmente, o método `getConnection()` (linhas 36 a 46) é chamado quando se deseja obter a conexão.


```
1    package agenda;
2
3    import java.sql.*;
4    import java.lang.*;
5    import java.util.*;
6
7    public class ConnectionBean {
8        private Connection con=null;
9        private static int clients=0;
10       static private ConnectionBean instance=null;
11
12       private ConnectionBean() { init(); }
13
14       static synchronized public ConnectionBean getInstance() {
15           if (instance == null) {
16               instance = new ConnectionBean();
17           }
18           return instance;
19       }
20
21       private void init() {
22           try {
23               Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
24               con=
25               DriverManager.getConnection("jdbc:pointbase:agenda","PUBLIC","public");
26           } catch(Exception e){System.out.println(e.getMessage());};
27       }
28
29       public synchronized void devolveConnection(Connection con) {
30           if (this.con==con) {
31               clients--;
32               notify();
33           }
```

```

34     }
35
36     public synchronized Connection getConnection() {
37         if(clients>0) {
38             try { wait(5000); }
41             catch (InterruptedException e) {};
42             if(clients>0) return null;
43         }
44         clients ++;
45         return con;
46     }
47
48     } //fim da classe

```

Exemplo – *ConnectionBean.java*.

O exemplo abaixo mostra código do JavaBean usado para verificar se o usuário está autorizado a usar a agenda. O JavaBean LoginBean recebe o nome e a senha do usuário, obtém a conexão com o SGBD e verifica se o usuário está autorizado, registrando o resultado da consulta na variável status (linha 10). Tudo isso é feito no construtor da classe (linhas 12 a 35).

Note que na construção do comando SQL (linhas 17 a 20) é inserido uma junção entre as tabelas PESSOA e USUARIO de modo a ser possível recuperar os dados relacionados armazenados em ambas as tabelas. Os métodos getLogin(), getNome() e getStatus() (linhas 36 a 38) são responsáveis pelo retorno do login, nome e status da consulta respectivamente.

```

1     package agenda;
2
3     import java.sql.*;
4     import java.lang.*;
5     import java.util.*;
6
7     public class LoginBean {
8         protected String nome = null;
9         protected String login= null;
10        protected boolean status= false;
11        public LoginBean(String login, String senha)
12        {

```

```

13     this.login = login;
14     Connection con=null;
15     Statement stmt =null;
16     String consulta = "SELECT NOME FROM PESSOA, USUARIO "+
17     "WHERE USUARIO.ID = PESSOA.ID AND "+
18     "USUARIO.SENHA='"+senha+"' AND "+
19     "USUARIO.LOGIN='"+login+"'";
20     try {
21         con=ConnectionBean.getInstance().getConnection();
22         stmt = con.createStatement();
23         ResultSet rs =stmt.executeQuery(consulta);
24         if(rs.next()) {
25             status = true;
26             nome = rs.getString("NOME");
27
68     }
28     } catch(Exception e){System.out.println(e.getMessage());}
29     finally {
30         ConnectionBean.getInstance().devolveConnection(con);
31         try{stmt.close();}catch(Exception ee){};
32     }
33 }
34 public String getLogin(){return login;}
35 public String getNome(){return nome;}
36 public boolean getStatus(){return status;}
37 }
38 }
39 } catch(Exception e){System.out.println(e.getMessage());}

```

Exemplo – LoginBean.java.

O exemplo AgendaServlet.java mostra código do Servlet que implementa a camada de controle do modelo MVC. O Servlet AgendaServlet recebe as requisições e, de acordo com os parâmetros, instância os JavaBeans apropriados e reencaminha as requisições para as páginas corretas.

Tanto o método doGet() (linhas 9 a 12) quanto o método doPost() (linhas 13 a 17) invocam o método performTask() (linhas 19 a 61) que realiza o tratamento da requisição. Na linhas 24 a 26 do método performTask() é obtido o valor do parâmetro corrente que determina a página que originou a requisição. Se o valor for nulo é assumido o valor default zero.

Na linha 30 é executado um comando switch sobre esse valor, de modo a desviar para bloco de comandos adequado. O bloco que vai da linha 32 até a linha 43 trata a requisição originada em uma página com a identificação 0 (página agenda.html).

Nas linhas 32 e 33 são recuperados o valor de login e senha digitados pelo usuário. Se algum desses valores for nulo então a requisição deve ser reencaminhada para a página de login (agenda.html) novamente (linha 35).

Caso contrário é instanciado um objeto LoginBean, inserido na sessão corrente e definida a página principal.jsp como a página para o reencaminhamento da requisição (linhas 38 a 41).

Já o bloco que vai da linha 44 até a linha 54 trata a requisição originada em uma página com a identificação 1 (página principal.jsp).

Na linha 44 é recuperado o objeto HttpSession corrente. O argumento false é utilizado para impedir a criação de um novo objeto HttpSession caso não exista um corrente. Se o valor do objeto for null, então a requisição deve ser reencaminhada para a página de login (linha 47). Caso contrário é instanciado um objeto AcaoBean, inserido na requisição corrente e definida a página principal.jsp como a página para o reencaminhamento da requisição (linhas 50 a 52).

Na linha 56 a requisição é reencaminhada para a página definida (página agenda.html ou principal.jsp).

```
1    package agenda;
2
3    import javax.servlet.*;
4    import javax.servlet.http.*;
5    import agenda.*;
6
7    public class AgendaServlet extends HttpServlet
8    {
9        public void doGet(HttpServletRequest request, HttpServletResponse response)
10       {
11           performTask(request,response);
```

```
12     }
13     public void doPost(HttpServletRequest request,
14         HttpServletResponse response)
15     {
16         performTask(request,response);
17     }
18
19     public void performTask(HttpServletRequest request,
20         HttpServletResponse response)
21     {
22         String url;
23         HttpSession sessao;
24         String corrente = request.getParameter("corrente");
25         int icorr=0;
26         if (corrente != null) icorr = Integer.parseInt(corrente);
27
28         try
29         {
30             switch(icorr)
31             {
32                 case 0: String login = request.getParameter("login");
33                     String senha = request.getParameter("senha");
34                     if (login == null||senha == null)
35                         url= "/agenda.html";
36                     else
37                     {
38                         sessao = request.getSession(true);
39                         sessao.setAttribute("loginbean",
40                             new agenda.LoginBean(login,senha));
41                         url= "/principal.jsp";
42                     };
43                     break;
44                 case 1:
```

```
45     sessao = request.getSession(false);
46     if (sessao == null)
47         url= "/agenda.html";
48     else
49     {
50         request.setAttribute("acaoBean",
51             new agenda.AcaoBean(request));
52         url= "/principal.jsp";
53     };
54     break;
55 }
56 getServletContext().getRequestDispatcher(url).forward(request,response);
57 }catch (Exception e) {
58     System.out.println("AgendaServlet falhou: ");
59     e.printStackTrace();
60 }
61 }
62 }
```

Exemplo – *AgendaServlet.java*.

O exemplo *AcaoBean* mostra código do *JavaBean* usado para realizar a manutenção da agenda. O *JavaBean AcaoBean* é responsável pela consulta, remoção e inserção de novos itens na agenda. Um objeto *StringBuffer* referenciado pela variável *retorno* é utilizado pelo *JavaBean* para montar o resultado da execução. O construtor (linhas 16 a 27) verifica o tipo de requisição e invoca o método apropriado.

O método *consulta()* (linhas 29 a 77) é responsável pela realização de consultas. As consultas podem ser realizadas sobre o campo nome ou descrição e os casamentos podem ser parciais, uma vez que é usado o operador *LIKE*. A consulta SQL é montada nas linhas 40 a 47. Na linha 50 é obtida uma conexão com SGBD por meio do objeto *ConnectionBean*. Na linha 57 o comando SQL é executado e as linhas 59 a 72 montam o resultado da consulta.

O método *insere()* (linhas 79 a 148) é responsável por inserir um item na agenda. Na linha 95 é obtida uma conexão com SGBD por meio do objeto *ConnectionBean*. Para inserir um novo item é preciso obter o número do último identificador usado, incrementar o identificador e inserir na base o item com o identificador incrementado. Esta operação requer que não seja acrescentado nenhum identificador entre a operação de leitura do último identificador e a inserção

de um novo item. Ou seja, é necessário que essas operações sejam tratadas como uma única transação e o isolamento entre as transações sejam do nível *Repeatable Read*. A definição do início da transação é feita no comando da linha 102. A mudança do nível de isolamento é feita pelos comandos codificados nas linha 103 a 109.

Na linha 112 é invocado o método `obtemUltimo()` (linhas 150 a 171) para obter o último identificador utilizado. As linhas 114 a 128 montam o comando SQL para a execução. O comando SQL é executado na linha 131. O fim da transação é definido pelo comando da linha 132. Ao fim da transação, de forma a não prejudicar a concorrência, o nível de isolamento deve retornar para um valor mais baixo. Isto é feito pelos comandos das linhas 133 a 137.

O método `apaga()` (linhas 173 a 201) é responsável por remover um item da agenda. As linhas 175 a 180 contém o código para verificar se o usuário digitou o nome associado ao item que deve ser removido.

A linha 181 montam o comando SQL para a execução. Na linha 184 é obtida uma conexão com SGBD por meio do objeto `ConnectionBean`. O comando SQL é executado na linha 191.

```

1      package agenda;
2
3      import java.lang.*;
4      import java.util.*;
5      import java.sql.*;
6
7      public class AcaoBean
8      {
9          private Connection con=null;
10         private StringBuffer retorno = null;
11         private Statement stmt=null;
12         private String [] legenda= {"C&oacute;digo","Nome","Telefone",
13             "Endere&cedil;o", "email","hp",
14             "celular","Descri&cedil;&atilde;o"};
15
16         public AcaoBean(javax.servlet.http.HttpServletRequest request)
17         {
18             String acao = request.getParameter("acao");
19             if (acao.equals("Consulta"))

```

```
20    {
21    String nome = request.getParameter("nome");
22    String descri = request.getParameter("descricao");
23    consulta(nome,descri);
24    }
25    else if (acao.equals("Insere")) insere(request);
26    else if (acao.equals("Apaga")) apaga(request);
27    }
28
29    private void consulta(String nome,String descri)
30    {
31    String consulta = null;
32
33    if ((nome == null||nome.length()<1) &&
34    (descri == null|| descri.length()<1))
35    {
36    retorno = new StringBuffer("Digite o nome ou descricao!");
37    return;
38    }
39
40    if (descri == null|| descri.length()<1)
41        consulta = "SELECT * FROM PESSOA WHERE NOME LIKE '%" +
42            nome+"%'"+" ORDER BY NOME";
43    else if (nome == null|| nome.length()<1)
44        consulta = "SELECT * FROM PESSOA WHERE DESCRICAO LIKE '%" +
45            descri+"%'"+" ORDER BY NOME";
46    else consulta="SELECT * FROM PESSOA WHERE DESCRICAO LIKE '%" +
47        descri+"%' AND NOME LIKE '%" +nome+"%' ORDER BY NOME";
48    try
49    {
50    con=ConnectionBean.getInstance().getConnection();
51    if (con == null)
52    {
```



```
53     retorno = new StringBuffer("Servidor ocupado. Tente mais tarde.!");
54     return;
55 }
56 stmt = con.createStatement();
57 ResultSet rs = stmt.executeQuery(consulta);
58
59 retorno = new StringBuffer();
60 retorno.append("<br><h3>Resultado</h3><br>");
61 while(rs.next())
62 {
63     retorno.append("ID:").append(rs.getString("id"));
64     retorno.append("<br>Nome:").append(rs.getString("Nome"));
65     retorno.append("<br>Telefone:").append(rs.getString("Telefone"));
66     retorno.append("<br>Endereco:").append(rs.getString("Endereco"));
67     retorno.append("<br>email:").append(rs.getString("email"));
68     retorno.append("<br>hp:").append(rs.getString("hp"));
69     retorno.append("<br>celular:").append(rs.getString("celular"));
70     retorno.append("<br>descricao:").append(rs.getString("descricao"));
71     retorno.append("<br><br>");
72 }
73 } catch(Exception e){System.out.println(e.getMessage());}
74 finally {ConnectionBean.getInstance().devolveConnection(con);
75 try{stmt.close();}catch(Exception ee){};
76 }
77 }
78
79 private void insere(javax.servlet.http.HttpServletRequest request)
80 {
81     String[] par = {"telefone","endereco","email","hp","celular","descricao"};
82
83     StringBuffer comando = new StringBuffer("INSERT INTO PESSOA(");
84     StringBuffer values = new StringBuffer(" VALUES(");
85
```

```
86     String aux = request.getParameter("nome");
87     if (aux == null || aux.length()<1)
88     {
89         retorno = new StringBuffer("<br><h3>Digite o nome!</h3><br>");
90         return;
91     }
92
93     try
94     {
95         con=ConnectionBean.getInstance().getConnection();
96         if (con == null)
97         {
98             retorno = new StringBuffer("Servidor ocupado. Tente mais tarde!");
99             return;
100        }
101
102        con.setAutoCommit(false);
103        DatabaseMetaData meta=con.getMetaData();
104
105        if(meta.supportsTransactionIsolationLevel(
106            con.TRANSACTION_REPEATABLE_READ)) {
107            con.setTransactionIsolation(
108                con.TRANSACTION_REPEATABLE_READ);
109        }
110
111
112        int ultimo = obterUltimo(con);
113        if (ultimo==-1) return;
114        ultimo++;
115        comando.append("id,nome");
116        values.append(ultimo+",").append(aux).append("");
117
118        for(int i=0;i<par.length;i++)
```

```
119    {
120    aux = request.getParameter(par[i]);
121    if (aux != null && aux.length()>0)
122    {
123    comando.append(",").append(par[i]);
124    values.append(",").append(aux).append("");
125    }
126    }
127    comando.append("");
128    values.append("");
129    aux = comando.toString()+values.toString();
130    stmt = con.createStatement();
131    stmt.executeUpdate(aux);
132    con.setAutoCommit(true);
133    if(meta.supportsTransactionIsolationLevel(
134    con.TRANSACTION_READ_COMMITTED)) {
135    con.setTransactionIsolation(
136    con.TRANSACTION_READ_COMMITTED);
137    }
138    retorno = new StringBuffer("<br><h3>Inserido!</h3><br>");
139    return;
140    } catch(Exception e)
141    {retorno =
142    new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>"); }
143    finally
144    {
145    ConnectionBean.getInstance().devolveConnection(con);
146    try{stmt.close();}catch(Exception ee){};
147    }
148    }
149
150    private int obtemUltimo(Connection con)
151    {
```

```
152 String consulta = "SELECT MAX(ID) AS MAX FROM PESSOA";
153 try
154 {
155     if (con == null)
156     {
157         retorno = new StringBuffer("Servidor ocupado. Tente mais tarde!");
158         return -1;
159     }
160     stmt = con.createStatement();
161     ResultSet rs = stmt.executeQuery(consulta);
162     if(rs.next())
163         return Integer.parseInt(rs.getString("max"));
164     else return 0;
165     } catch(Exception e) {
166         retorno =
167         new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>");
168         return -1;
169     }
170     finally {try{stmt.close();}catch(Exception ee){};}
171 }
172
173 private void apaga(javax.servlet.http.HttpServletRequest request)
174 {
175     String aux = request.getParameter("nome");
176     if (aux == null || aux.length()<1)
177     {
178         retorno = new StringBuffer("<br><h3>Digite o nome!</h3><br>");
179         return;
180     }
181     String consulta = "DELETE FROM PESSOA WHERE NOME='"+aux+"'";
182     try
183     {
184         con=ConnectionBean.getInstance().getConnection();
```

```
185     if (con == null)
186     {
187         retorno = new StringBuffer("Servidor ocupado. Tente mais tarde!");
188         return;
189     }
190     stmt = con.createStatement();
191     stmt.executeUpdate(consulta);
192
193     retorno = new StringBuffer("<br><h3>Removido!</h3><br>");
194     return;
195     } catch(Exception e){
196     retorno = new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>");
197     }
198     finally {
199         ConnectionBean.getInstance().devolveConnection(con);
200         try{stmt.close();}catch(Exception ee){};}
201     }
202
203     public String[] getLeg(){return legenda;}
204     public String toString(){return retorno.toString();}
205     }
```

Exemplo – AcaoBean.java.

Instalação

Para instalar crie a seguinte estrutura de diretório abaixo do diretório webapps do Tomcat:

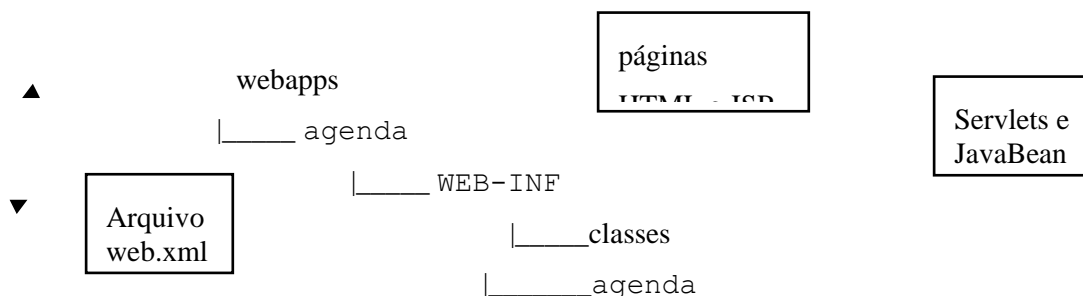


Figura 25. Estrutura de diretórios para a aplicação agenda.

O arquivo web.xml deve ser alterado para conter mapeamento entre a URL agenda e o Servlet AgendaServlet.

```

<web-app>
  <servlet>
    <servlet-name>
      agenda
    </servlet-name>
    <servlet-class>
      agenda.AgendaServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>
      agenda
    </servlet-name>
    <url-pattern>
      /agenda
    </url-pattern>
  </servlet-mapping>
</web-app>
  
```

Figura . Arquivo web.xml para a agenda.

Considerações sobre a solução

A aplicação acima implementa uma agenda que pode ser acessada por meio da Internet, no entanto, devido à falta de espaço e à necessidade de destacarmos os pontos principais, alguns detalhes foram deixados de lado, como por exemplo uma melhor interface com o usuário. Abaixo seguem alguns comentários sobre algumas particularidades da aplicação:

1. O JavaBean da classe LoginBean é armazenado na sessão para permitir a verificação se o acesso ao site é autorizado. Isto impede que os usuários tentem acessar diretamente a página principal.jsp da agenda. Caso tentem fazer isso, a sessão não conterá um objeto LoginBean associado e, portanto, o acesso será recusado.
2. O JavaBean da classe AcaoBean é armazenado no objeto request uma vez que suas informações são alteradas a cada requisição. Uma forma mais eficiente seria manter o objeto AcaoBean na sessão e cada novo requisição invocar um método do AcaoBean para gerar os resultados. No entanto, o objetivo da nossa implementação não é fazer a aplicação mais eficiente possível, e sim mostrar para o leitor uma aplicação com variadas técnicas.
3. Apesar de termos adotado o padrão MVC de desenvolvimento a aplicação não exibe uma separação total da camada de apresentação (Visão) em relação à camada do modelo. Parte do código HTML da visão é inserido pelo AcaoBean no momento da construção da String contendo o resultado da ação. Isto foi feito para minimizar a quantidade de código Java na página JSP. Pode-se argumentar que neste caso a promessa da separação entre as camadas não é cumprida totalmente. Uma solução para o problema seria gerar o conteúdo em XML e utilizar um analisador de XML para gerar a página de apresentação. No entanto, o uso da tecnologia XML foge ao escopo da apostila.
4. A solução apresenta código redundante para criticar as entradas do usuário. Existe código JavaScript nas páginas, e código Java no Servlet e JavaBeans. O uso de código JavaScript nas páginas para críticas de entrada é indispensável para aliviarmos a carga sobre o servidor. Já o código para crítica no servidor não causa impacto perceptível e útil para evitar tentativas de violação.
5. O código exibe uma preocupação com a concorrência de acessos ao banco de dados que aparentemente não é necessária, uma vez que apenas uma conexão por vez é permitida. No entanto, procuramos fazer o código o mais genérico possível no pouco espaço disponível.

Tag Files

Um dos recursos mais interessantes do JSP é, sem dúvida, a possibilidade de se criar tags customizadas (Custom Tags). Com elas, podemos obter muitas melhorias no desenvolvimento, tais como evitar código repetitivo e criar uma interface web componentizada.

E as taglibs estão soltas por aí, Internet a fora, sempre presentes no mundo Java. A maioria dos frameworks web possui taglibs, para funções como iterações, condições, i18n. Há por aí tags para paginação, exibição de gráficos, e outros recursos visuais. Você já deve ter ouvido falar da JSTL (Java Standard Tag Libraries). O emergente JSF (Java Server Faces) também faz extensivo uso de tags.

Enfim, se você ficou com água na boca, e quer criar suas próprias taglibs, há um recurso muito simples e poderoso para fazer isso: os **Tag Files**.

O que são Tag Files?

Tag Files são arquivos com extensão **.tag**, na qual a custom tag é programada. Não é necessário criar nenhuma classe Java.

Preparação

Geralmente os tag files são empacotados em um arquivo JAR, que é colocado na pasta **WEB-INF/lib** da sua aplicação. Entretanto, também podemos colocar os tag files na pasta **WEB-INF/tags**.

O segundo método é mais adequado para o desenvolvimento, enquanto o primeiro para a distribuição das tags.

Assim, crie a pasta **WEB-INF/tags**. Podemos colocar os nossos tag files diretamente nessa pasta, ou criar subpastas, uma para cada conjunto de tags. Dessa forma, crie a subpasta **mytags**.

Pronto, agora podemos começar a desenvolver nossas tags.

Minha primeira Tag! (não é HelloWorld)

Para começar, vamos criar uma tag que informa a data atual do sistema operacional, de acordo com o Locale default. Dentro de **WEB-INF/tags/mytags**, crie o arquivo **dataDeHoje.tag**.

```
<% @tag description="descricao" pageEncoding="UTF-8" import="java.util.*, java.text.*"%>
<%
DateFormat formatador = DateFormat.getDateInstance(DateFormat.FULL, Locale.getDefault());
out.print(formatador.format(new Date()));
%>
```

Todo tag file deve começar com a diretiva **@tag**. Nela, colocamos informações gerais sobre a tag, sendo que as principais são essas que estão no exemplo. Depois, programamos a tag. Obtemos o formatador e escrevemos a data formatada.

Pois bem, agora criemos a página **exibeDataDeHoje.jsp**.

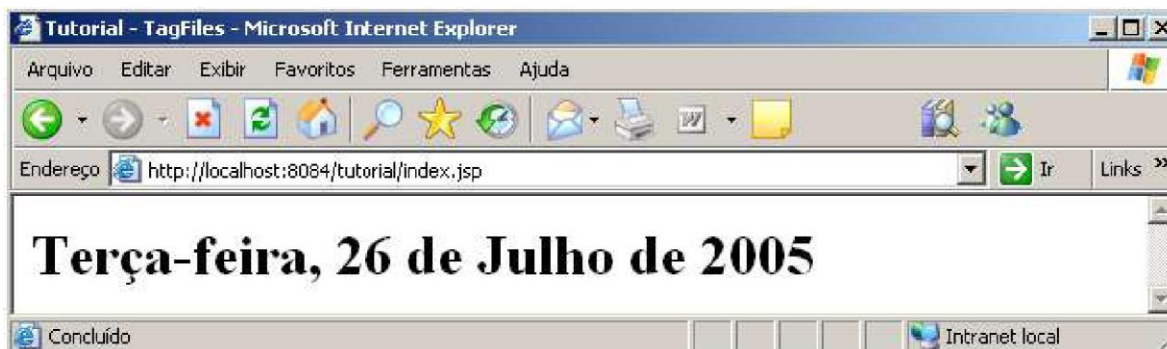
```
<% @taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
<html>
```

```
<head> <title>Tutorial - TagFiles</title> </head>
<body>
<h1><mytags:dataDeHoje/></h1>
</body>
</html>
```

Observe a diretiva **@taglib**. Ela é responsável por definir uma taglib que será usada na página. Podemos ter várias diretivas **@taglib**, que definirão várias taglibs a serem usadas. Informamos o prefixo (**prefix**), que pode ser qualquer um, e depois o diretório das tags (**tagDir**). Quando empacotarmos as tags em um JAR, ao invés de tagDir, usaremos **uri**.

Depois, podemos usar as nossas tags livremente, como demonstrado na parte em negrito do exemplo. Só não devemos nos esquecer de fechar as tags, pois elas seguem o padrão xml.

O resultado disso será:



Tags com atributos

Quando fazemos uma tag, geralmente vamos querer passar atributos, para informar melhor o que queremos. Então, vamos incrementar a nossa dataDeHoje. Vamos informar para a tag qual o idioma no qual queremos ver a data escrita.

```
<% @tag description="descricao" pageEncoding="UTF-8" import="java.util.*, java.text.*"%>
<% @attribute name="language" required="false"%>
<%
Locale locale = Locale.getDefault();
if (language != null) { //Testa se o language foi passado
locale = new Locale(language);
}
DateFormat formatador = DateFormat.getDateInstance(DateFormat.FULL, locale);
out.print(formatador.format(new Date()));
%>
```

Veja a diretiva **@attribute**. Ela é usada para definir um atributo da tag, sendo que cada tag pode conter vários atributos.

Um atributo precisa, necessariamente, ter um **name**. No nosso caso, o name é “language”. Podemos informar também o **required**. Se for true, a passagem deste atributo é obrigatória, e se for false, não é obrigatória (o default é false). Também pode ser informado o tipo, através de **type** (o default é String).

Há outras propriedades de @attribute, mas essas são as mais usadas.

Para acessar o atributos de dentro do tag file, como você deve ter percebido, usamos como identificador o name definido. Podemos também acessar os atributos por meio de EL (Expression Language). No nosso caso, seria **\${language}**.

Agora, modificando a exibeDataDeHoje.jsp, teríamos:

```
<% @taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
<html>
<head> <title>Tutorial - TagFiles</title> </head>
<body>
<h1><mytags:dataDeHoje language="fr"/></h1>
</body>
</html>
```

Queremos a data em francês. Assim, o resultado seria esse:



Uma tag pode ter um corpo: parte compreendida entre a abertura e o fechamento da tag. O acesso ao corpo se dá pela tag **<jsp:doBody/>**. Como exemplo prático, faremos uma tag que, recebendo um título como atributo e um texto qualquer no corpo, gera um quadro. Chamaremos de **quadro.tag**.

```
<% @tag description="descricao" pageEncoding="UTF-8"%>
```

```
<% @attribute name="title" required="true"%>
```

```
<table>
```

```
<tr><td>
```

```
<fieldset>
```

```
<legend>${title}</legend>
```

```
<jsp:doBody/>
```

```
</fieldset>
```

```
</td></tr>
```

```
</table>
```

O **<jsp:doBody/>** acessa e escreve o conteúdo do corpo da tag. Agora, para testarmos, criaremos a pagina **testaQuadro.jsp**:

```
<% @taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
```

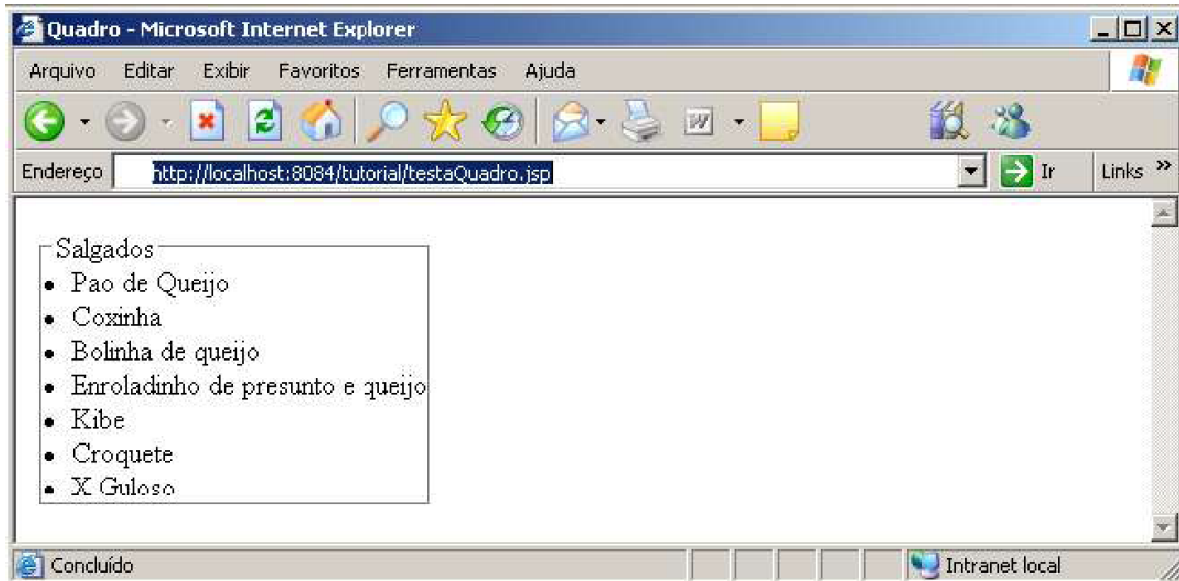
```
<html>
```

```
<head>
```

```
<title>Quadro</title>
```

```
</head>
<body>
<mytags:quadro title="Salgados">
<li> Pao de Queijo
<li> Coxinha
<li> Bolinha de queijo
<li> Enroladinho de presunto e queijo
<li> Kibe
<li> Croquete
<li> X Guloso
</mytags:quadro>
</body>
</html>
```

O resultado deverá ser algo assim:



Se quisermos manipular o conteúdo do corpo podemos, ao invés de escrevê-lo na tela, armazená-lo em alguma variável, através dos atributos **var** e **scope** de `<jsp:doBody/>`. Dentro do tag file, faríamos:

```
<jsp:doBody var="texto" scope="request"/>
```

Nesse caso, gravaríamos o corpo na variável de requisição **texto**. Depois, se quiséssemos acessá-lo, usaríamos `${texto}` ou `request.getAttribute("texto")`.

Dependendo da sua tag, você pode querer que ela não aceite corpo. Para fazer isso, é simples:

```
<%@tag description="descricao" pageEncoding="UTF-8" body-content="empty"%>
```


Empacotando suas tags para distribuição

Se você quiser distribuir suas tags, para que outros usem, você deve empacotá-las em um arquivo JAR, formando assim uma biblioteca de tags. Aí, a pessoa que for usar coloca o jar em WEB-INF/lib e pronto.

Assim, vamos empacotar as nossas duas tags. Para fazer isso, devemos criar um arquivo **TLD** (Tag Library Descriptor), que ficará numa pasta chamada **META-INF**, e daremos o nome de **taglibs.tld** (qualquer outro

nome poderia ser dado).

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<uri>taglibs</uri>
<tag-file>
<name>today</name>
<path>/META-INF/tags/mytags/dataDeHoje.tag</path>
</tag-file>
<tag-file>
<name>quadro</name>
<path>/META-INF/tags/mytags/quadro.tag</path>
</tag-file>
</taglib>
```

Dentro de taglibs.tld, podemos destacar:

<uri>taglibs</uri>: Especifica a uri da taglib, de forma que quando ela for usada em uma página, faremos `<%@taglib prefix="mytags" uri="taglibs"%>`.

<tag-file> ... </tagfile> : Aqui, declaramos as nossas tag files, informando o nome e o arquivo com caminho completo. O nome definido será utilizado na hora de fazer uso da tag, como por exemplo:

```
<mytags:today/>
```

Agora, como você de ter concluído, colocamos os nossos tagfiles em **/META-**

INF/tags/mytags.

Depois disso, é só criar o JAR, na linha de comando ou no seu IDE favorito. Criado o JAR, você já pode distribuir suas tags sem problemas.

JSTL

No ano 2000, o Java Community Process (JCP), selecionou um expert group para JSTL. Desde de então o expert group definiu a especificação da JSTL e produziu uma referência de implementação.

A JSTL é projetada para trabalhar com servlets containers com suporte as APIs Servlet 2.3 e JSP 1.2 ou acima.

A JSTL é uma coleção de custom tags que executam funcionalidades comuns em aplicações WEB, incluindo iteração e seleção, formatação de dados, manipulação de XML e acesso a banco de dados. A JSTL permite que os desenvolvedores JSP focalizem em necessidades específicas do desenvolvimento, ao invés de reinventar a roda.

A JSTL é composta de:

- Uma linguagem de Expressão
- Bibliotecas de Ações padrão (42 ações em quatro bibliotecas)
- Validators (2 validators)

Instalação da JSTL

Para que possamos utilizar os exemplos citados nesse tutorial, utilizaremos o pacote fornecido pelo projeto Apache Jakarta, o Taglib Standard que é a implementação de referência da JSTL.

Você pode baixar o pacote [em http://jakarta.apache.org/taglibs/index.html](http://jakarta.apache.org/taglibs/index.html)

Nota:

JSTL 1.1 requer um container que suporte as especificações Servlet 2.4 e JSP 2.0

JSTL 1.0 requer um container que suporte as especificações Servlet 2.3 e JSP 1.2

Para utilizar a JSTL em suas aplicações, copie os arquivos *.jar* que estão dentro da pasta lib do pacote da JSTL para a pasta WEB-INF do seu projeto.

TLD

Há muitas recomendações para copiar os arquivos TLD para a pasta WEB-INF, para que possa se utilizar as tags. Mas uma boa prática é utilizar os TLDs que estão embutidos nos JARs do Taglibs Standard, dentro da pasta META-INF.

A Expression Language (EL)

A EL é uma simples linguagem baseada em ECMAScript (também conhecida como JavaScript) e XPath. Ela provém expressões e identificadores; aritméticos, lógicos, operadores relacionais; e conversão de tipos.

A EL torna simples o acesso a objetos implícitos tal como o servlet request/response, variáveis de escopo e objetos armazenados no escopo JSP (page, request, session e application). A EL reduz drasticamente a necessidade de utilizar expressões JSP e scripts, aumentando a manutenibilidade e extensibilidade de aplicações WEB.

Expressões EL são invocadas com essa sintaxe: `${expressão}`. As expressões podem consistir em:

Identificadores

Identificadores representam o nome dos objetos armazenados em um escopo JSP: *page*, *request*, *session* ou *application*.

Quando a EL encontra um identificador, ela procura por uma variável de escopo com o mesmo nome no escopo JSP, na ordem citada acima.

Por exemplo, o seguinte trecho de código armazena uma String em um escopo *page* e é acessada através de uma expressão EL:

```
<%
// cria a String
String s = "Portal Java";
// armazena a String no escopo page
pageContext.setAttribute("name", s);
%>
```

E para acessar a String com a expressão EL:

```
<c:out value='${name}'/>
```

Operadores

São estes os operadores da EL:

Tipo	Operador
------	----------

Aritimético	+ - * / (div) % (mod)
Grupo	()
Identificador de Acesso	. []
Lógico	&& (and) (or) ! (not) empty
Relacional	== (eq) != (ne) < (lt) > (gt) <= (le) >= (ge)
Unário	-

Precedência de operadores:

Os operadores são listados da esquerda para a direita. Por exemplo, o operador [] tem a precedência sobre o operador .

- [] .
- ()
- - (unary) not ! empty
- * / div % mod
- + - (binary)
- < > <= >= lt gt le ge
- == != eq ne
- && and
- || or =

Os operadores [] e .

A EL prove dois operadores que permitem você acessar variáveis de escopo e suas propriedades. O operador . é similar ao operador . do Java, mas em vez de acessar métodos, você acessará propriedades de Beans. Por exemplo:

Você tem um Bean chamado Pessoa armazenado em uma variável de escopo chamada pessoa e o Bean contém as propriedades nome e idade, você pode acessar essas propriedades assim:

```
Nome: <c:out value='${pessoa.nome}'/>
Idade: <c:out value='${pessoa.idade}'/>
```

O operador [] é utilizado para acessar objetos em maps, lists e arrays.

O seguinte código acessa o primeiro objeto em um Array:

```
<%  
String[] num = { "1", "2", "3" };  
pageContext.setAttribute("array", num);  
%>  
  
<c:out value='${array[0]}'/>
```

Valores Literais

São estes os valores literais da EL:

Tipo	Exemplo
Boolean	true ou false
Integer	143 +3 -4
Double	1.43 -2.35
String	strings com “” e ‘,’
Null	null

Objetos implícitos

A característica mais útil da EL é o acesso a todos os objetos implícitos da aplicação:

Objeto	Tipo	Descrição	Valor
Cookie	Map	Cookie name	Cookie
Header	Map	Request header name	Request header value
headerValues	Map	Request header name	String[] of request header values
initParam	Map	Request header name	Initialization parameter value
param	Map	Request parameter name	Request parameter value
paramValues	Map	Request parameter name	String[] of request parameter values
pageContext	PageContext	--	--
pageScope	Map	Page-scoped attribute name	Page-scoped attribute value
requestScope	Map	Request-scoped attribute name	Request-scoped attribute value
sessionScope	Map	Session-scoped attribute name	Session-scoped attribute value
applicationScope	Map	Application-scoped attribute name	Application-scoped attribute value

Um dos objetos implícitos mais utilizado é o `param`, que acessa parâmetros do request.

Por exemplo:

Esse formulário apenas exibe dois campos a serem preenchidos e sua action aponta para a página *parametros.jsp*.

form.jsp

```
<html>
<head>
<title>Objetos Implícitos - EL</title>
</head>
<body>
<form action='parametros.jsp'>
<table>
<tr>
<td>Nome:</td>
<td><input type="text" name="nome"></td>
</tr>
<tr>
<td>Idade:</td>
<td><input type="text" name="idade"></td>
</tr>
</table>
<input type="submit" value="Enviar">
</form>
</body>
</html>
```

parametros.jsp

```
<% @ taglib uri='http://java.sun.com/jstl/core' prefix='c'
%>
<html>
<head>
<title>Acessando Request Parameters com EL</title>
</head>
<body>
<font size='5'>Parâmetros</font>
<c:out value='${param.nome}'/>,
<c:out value='${param.idade}'/>
</body>
</html>
```


Utilizando uma biblioteca de tags

Há dois passos simples que devem ser seguidos para utilizar uma biblioteca de tags existente em um JSP:

- Disponibilize a biblioteca de tags para o JSP
- Utilizar a tag requerida no momento em que precisar

Importando a bibliotecas de tags

O processo é realmente muito simples:

Para importar a biblioteca de tags em um JSP, basta utilizar a diretiva taglib. .

```
<%@ taglib uri="" prefix=""  
%>
```

Detalhando os atributos:

- **uri** – permite especificar a localização do TLD (*tag lib descriptor*)
- **prefix** – é uma string que é utilizada unicamente para identificar as tags personalizadas que vc deseja utilizar.

TLD

É um arquivo XML que descreve o mapeamento entre as tags e seu tratador de tags.

Há duas maneiras básicas de especificar esse mapeamento. Uma opção é especificar um URL (relativo ou absoluto) para o arquivo TLD. A outra opção é fornecer um mapeamento de nome/URL lógico dentro do arquivo web.xml da aplicação.

Caso queira utilizar a primeira opção:

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

Uma desvantagem de utilizar uma URL para localizar um arquivo TLD torna-se aparente se o nome ou a localização do arquivo mudar. É por essa razão que a outra forma é preferida.

As URLs são muito comuns e, ao utilizar JSTL, você verá diretivas como essa:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"  
%>
```


Ações de Finalidades Gerais

A JSTL fornece quatro ações de finalidades gerais:

- `<c:out>`
- `<c:set`
- `<c:remove>`
- `<c:catch>`

As ações listadas acima representam ações fundamentais. Vamos detalhar cada uma das ações:

`<c:out>`

A sintaxe da ação `<c:out>` é:

```
<c:out value [default]
[escapeXml]/>
```

Essa ação avalia uma expressão EL, converte o resultado para uma String e envia essa String para o `JspWriter` corrente.

Considere o seguinte trecho de código:

```
<%= request.getParameter("nome")
%>
```

A ação `<c:out>` é o equivalente a expressão JSP citada acima:

```
<c:out
value='${param.nome}'/>
```

O atributo *default* (opcional) especifica um valor padrão que deve ser utilizado quando a expressão é avaliada como *null* ou se a expressão não conseguir ser avaliada e lançar uma exceção.

Exemplo:

```
<c:out value='${param.nome}' default="João Java da Silva"  
/>
```

Caso o parâmetro nome não for passado, o valor que será exibido será:

João Java da Silva!!!

O atributo *escapeXml* (opcional) controla certos caracteres de XML que são convertidos em referências de entidade ou em referências de caractere. Por padrão, essa conversão acontece para os seguintes caracteres:

- Um caractere < é substituído por <
- Um caractere > é substituído por >
- Um caractere & é substituído por &
- Um caractere ' é substituído por '
- Um caractere " é substituído por ".

O atributo *escapeXml* recebe um valor *boolean* (*true* ou *false*), por padrão o valor é *true*.

<c:set>

A ação <c:set> é completamente versátil, permitindo :

- Armazenar um valor em uma variável de escopo
- Apagar uma variável de escopo
- Ajustar a propriedade de um Bean a um valor especificado
- Ajustar a propriedade de um Bean para null
- Armazenar uma entrada (chave/par de valor) em um MAP Java
- Modificar uma entrada em um MAP Java
- Remover uma entrada em um MAP Java

A ação <c:set> suporta quatro sintaxes, duas dessas sintaxes permitem você manipular variáveis de escopo e as outras duas, permitem você manipular Beans e MAPs.

A primeira sintaxe da ação <c:set> é:

```
<c:set value var  
[scope]/>
```

Essa forma de ação <c:set> especifica o nome e o valor da variável exportada junto com o escopo opcional (por padrão assume o valor *page*).

A segunda sintaxe:

```
<c:set value var  
[scope]/>  
conteúdo do corpo  
</c:set>
```

Essa forma permite especificar o valor utilizando qualquer código válido JSP como conteúdo do corpo da ação `<c:set>`

Nas duas formas, o atributo *scope* é opcional e por padrão assume o valor *page*.

As outras duas formas da ação `<c:set>` são utilizadas para configurar as propriedades de objetos. Essa ação tem a mesma funcionalidade da tag `<jsp:setProperty>` que permite configurar os valores das propriedades de um Bean. Você também pode utilizar a ação `<c:set>` para esse propósito.

A sintaxe é:

```
<c:set target property  
value/>
```

e

```
<c:set target  
property>  
conteúdo do corpo  
</c:set>
```

Vamos a um exemplo de uso da utilização da tags `<c:set>` para configurar propriedades de um Bean.

Este trecho de código apenas ajusta a propriedade nome do Bean Usuário para o parâmetro passado:

```
<jsp:useBean id='usuario'  
class='beans.Usuario'/>  
  
<c:set target='${usuario}' property='nome'  
value='${param.nome}'/>
```

<c:remove>

Como podemos observar, a ação `<c:set>` permite a criação de variáveis de escopo. Algumas vezes é necessário remover essas variáveis.

Para essa funcionalidade a JSTL disponibiliza a ação `<c:remove>`

A sintaxe da ação `<c:remove>` é:

```
<c:remove var  
[scope]/>
```

Você precisa especificar o nome da variável de escopo que você quer remover com o atributo `var`. Opcionalmente você pode especificar a variável de escopo com o atributo `scope`.

Se você não especificar o atributo `escopo`, a ação `<c:remove>` irá buscar nos escopos por uma variável de escopo com o nome que você especificou no atributo `var`.

A ordem da busca nos escopos é:

- page,
- request
- session,
- application

<c:catch>

A ação `<c:catch>` disponibiliza um mecanismo de tratamento de erros.

A sintaxe da ação `<c:catch>` é:

```
<c:catch [var]>  
conteúdo do corpo, com ações que podem lançar  
exceções
```



```
</c:catch>
```


Ações Condicionais

Ações condicionais são essenciais em qualquer linguagem de programação.

Elas foram projetadas para facilitar a realização de programação condicional no JSP e são também uma alternativa uma à utilização de scriptlets.

A JSTL dispõe as seguintes tags condicionais:

- `<c:if>`
- `<c:choose>`
- `<c:when>`
- `<c:otherwise>`

Elas permitem realizar dois tipos diferentes de processamento: Processamento condicional simples e processamento mutuamente exclusivo.

<c:if>

A ação `<c:if>` realiza o processamento condicional simples. Ou seja, ela fornece uma condição de teste que é avaliada com um valor *boolean*. Se a ação for avaliada como um valor *true*, o conteúdo do corpo da ação `<c:if>` é processado. Caso contrário o conteúdo do corpo é ignorado.

O conteúdo do corpo das tags pode ser qualquer código JSP válido!

Vamos a sintaxe da tag:

```
<c:if test='condição' var='nome da variável' [scope='{page || request || session || application}']  
/>
```

O atributo *test*, que é requerido, é uma expressão *boolean* que determina quando o conteúdo do corpo da tag será processado.

Os atributos opcionais *var* e *scope* especificam a variável de escopo que armazenará o valor *boolean* da expressão especificada com o atributo *test*.

O seguinte trecho de código testa se um parâmetro é menor que 30, se o teste for *true* o parâmetro tamanho será exibido.

```
<c:if test='${param.tamanho < 30}'>  
  <c:out value='${param.tamanho}'/>  
</c:if>
```

Nota:

Não há nenhuma funcionalidade do tipo *if-then-else* fornecida pela ação `<c:if>`. Nos casos mais simples, você pode utilizar um número pequeno de ações `<c:if>` mutuamente para abranger os diferentes casos nos quais está interessado. Uma idéia melhor é utilizar a ação `<c:choose>`.

Condições Mutuamente Exclusivas

<c:choose>

Algumas vezes você precisa executar uma ação se uma de diversas circunstâncias for verdadeira.

Quando você especifica uma condição mutuamente exclusiva com JSTL, você utilizará a ação <c:choose>

A sintaxe da ação <c:choose> é:

```
<c:choose>
conteúdo do
corpo
</c:choose>
```

O conteúdo do corpo da ação <c:choose> pode ser:

- Uma ou muitas ações <c:when>
- Nenhuma ação <c:otherwise>. Essa ação deve aparecer depois as ações <c:when>
- Caracteres de espaço em branco entre as ações <c:when> e <c:otherwise>

Você pode pensar sobre a ação <c:choose> como uma *if/else if/else if/else* em Java:

Código Java	Ações JSTL
if(condicao1)	<c:choose> <c:when test="condicao1"> </c:when>
else if(condicao2)	<c:when test="condicao2"> </c:when>
else	<c:otherwise> <c:otherwise> <c:choose>

A ação <c:choose> processa no máximo uma de suas ações aninhadas. A regra é que a ação <c:choose> processe a primeira ação <c:when> cuja condição de teste é avaliada como *true*. Entretanto, se nenhuma das ações <c:when> for aplicável, a ação <c:otherwise> é processada, se estiver presente. Se fornecer uma ação <c:otherwise>, você deverá certificar-se de que ela é a última ação dentro da ação <c:choose> pai.

<c:when>

Se você necessitar escolher entre mais de duas condições, você pode simular um *switch* simplesmente adicionando mais ações <c:when> no corpo da ação <c:choose>.

A sintaxe da ação <c:when> é:

```
<c:when test=">  
  conteúdo do  
  corpo  
</c:when>
```

A ação <c:when> é similar a ação <c:if>, ambas as ações tem um atributo *test* que determina se o conteúdo do corpo da ação é avaliado.

Exemplo:

```
<c:when test='${sexo == 1}'>  
  <img src='<c:out  
  value="masculino.jpg"/>'>  
</c:when>
```

<c:otherwise>

A ação <c:otherwise> deve avaliar se nenhuma das suas ações <c:when> irmãs for avaliada como *true*. Ela deve ser a última ação dentro de uma ação <c:choose>.

A sintaxe da ação <c:otherwise> é:

```
<c:otherwise>  
  conteúdo do  
  corpo  
</c:otherwise>
```

Exemplo:


```
<c:otherwise>  
Olá usuário !!!  
</c:otherwise  
>
```

O conteúdo do corpo pode ser qualquer código JSP válido.

A seguir, um exemplo de uso de algumas tags citadas.

Esse trecho de código apenas cria um formulário com os campos nome , sexo e um campo oculto chamado submitted que é utilizado para identificar se o form já foi submetido alguma vez.

```
<form name="form1" method="post" action="exemplo.jsp">  
<table width="37%" border="1" cellpadding="0" cellspacing="0" bordercolor="WhiteSmoke">  
<tr>  
<td width="15%"><b>Nome:</b></td>  
<td width="85%">  
<input name="nome" type="text" value="<c:out value='${param.nome}'/>">  
</td>  
</tr>  
<tr>  
<td><b>Sexo:</b></td>  
<td>  
<select name="sexo">  
<option></option>  
<option value="1" <c:if test='${param.sexo == 1}'>selected</c:if>>Masculino</option>  
<option value="2" <c:if test='${param.sexo == 2}'>selected</c:if>>Feminino</option>  
</select>  
</td>  
</tr>  
<tr>  
<td colspan="2">  
<div align="center">
```

```
<input name="bt" type="submit" value="Enviar">
<input type="hidden" value="true" name="submitted">
</div>
</td>
</tr>
<tr>
<td colspan="2"><b>Nome: </b><i><c:out value='${param.nome}'/></i></td>
</tr>
<tr>
<td colspan="2"><b>Sexo:</b>
<c:if test='${param.sexo == 1}'>
<i>Masculino</i>
</c:if>
<c:if test='${param.sexo == 2}'>
<i>Feminino</i>
</c:if>
</td>
</tr>
</table>
</form>
```

Exemplo do uso de JSTL

```
<% @page contentType="text/html"%>
```

```
<% @page pageEncoding="UTF-8"%>
```

<!--taglib é uma tag JSP para declaração dos pacotes específicos JSTL através da URI apropriada e da sugestão de prefixo padrão, lembrando que essa etapa é importante, pois, definirá o prefixo de chamada das tags de cada pacote.-->

```
<% @taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<% @taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
```

```
<% @taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
```

```
<html>
```

```
<head>
```

<!-- setDataSource é uma tag JSTL para configuração de acesso ao servidor de banco de dados criando um objeto dataSource que será consumido pelas outras tags SQL, onde devem ser definidos os atributos necessários para especificar: o nome da variável dataSource, driver, url(caminho do servidor), usuário, senha e principalmente o "escopo" de visibilidade do objeto na aplicação, em nosso caso, definimos que nosso objeto será compartilhado por todo o tempo de vida da sessão.-->

```
<sql:setDataSource var="ds"
```

```
    driver="org.firebirdsql.jdbc.FBDriver"
```

```
url="jdbc:firebirdsql:localhost:c:\ACADEMICO.FDB"
```

```
    user="sysdba"
```

```
    password="masterkey"
```

```
    scope="session" />
```

```
<title>Exemplo JSTL - Manoel Pimentel </title>
```

```
</head>
```

```
<body>
```

```
<h1>Listagem de Produtos</h1>
```

<hr>

<!--A tag query é usada para processar uma sentença SQL de seleção de registros e gerar um objeto ResultSet internamente na memória, conforme especificado no atributo var, usando a conexão aberta chamada "ds", que neste caso está definido no atributo dataSource através do uso de EL(Expression Language). -->

<sql:query var="ResultadoProdutos" dataSource="{ds}">

select * from PRODUTOS

</sql:query>

<table border="1">

<thead>

<th>Código</th>

<th>Nome</th>

<th>Ultima Compra</th>

<th>Preco Custo</th>

<th>Preco Venda</th>

<th>Margem</th>

<th>Avaliação</th>

</thead>

<!--forEach, implementa um laço para fazer a interação no ResultSet gerado pela tag query conforme o atributo items.

-->

<c:forEach var="listaProdutos" items="{ResultadoProdutos.rows}">

<tr>

<!--

A tag out é responsável por gerar uma String de saída na tela -->

<td><c:out value="{listaProdutos.CODPRD}"/></td>

<td><c:out value="{listaProdutos.NOME}"/></td>

```
<td>

<fmt:formatDate pattern="dd/MM/yyyy"
Value="${listaProdutos.DATA_ULTIMA_COMPRA}"/>
</td>

<td><c:out value="${listaProdutos.PRECO_CUSTO}"/></td>
<td><c:out value="${listaProdutos.PRECO_VENDA}"/></td>

<c:set var="totalPrecoCusto" value="${totalPrecoCusto+listaProdutos.PRECO_CUSTO}"/>
<c:set var="totalPrecoVenda" value="${totalPrecoVenda+listaProdutos.PRECO_VENDA}"/>
<c:set var="valorMargem" value="${listaProdutos.PRECO_VENDA-
listaProdutos.PRECO_CUSTO}"/>

<td><c:out value="${valorMargem}"/></td>

<!--As tags choose, when e otherwise, aplicam um conjunto de estrutura de decisão. -->
```

```
<c:choose>
    <c:when test="${valorMargem<=350}">
        <td>Baixa</td>
    </c:when>
    <c:when test="${valorMargem<=400}">
        <td>Media</td>
    </c:when>
    <c:otherwise>
```

```
<td>Alta</td>
</c:otherwise>
</c:choose>
</tr>
</c:forEach>

<tfoot>
  <th colspan="3">
    Total:
  </th>
  <th>

<!--formatNumber aplica uma formatação de decimais no atributo value conforme o atributo
pattern. -->
<fmt:formatNumber value="${totalPrecoCusto}" pattern="#,#00.00#"/>
  </th>

  <th>
<fmt:formatNumber value="${totalPrecoVenda}" pattern="#,#00.00#"/>
  </th>
</tfoot>
</table>
</body>
</html>
```

Exemplo JSTL - Manoel Pimentel - Microsoft Internet Explorer

Arquivo Editar Exibir Favoritos Ferramentas Ajuda

Endereço http://localhost:8084/ExemploJSTL/

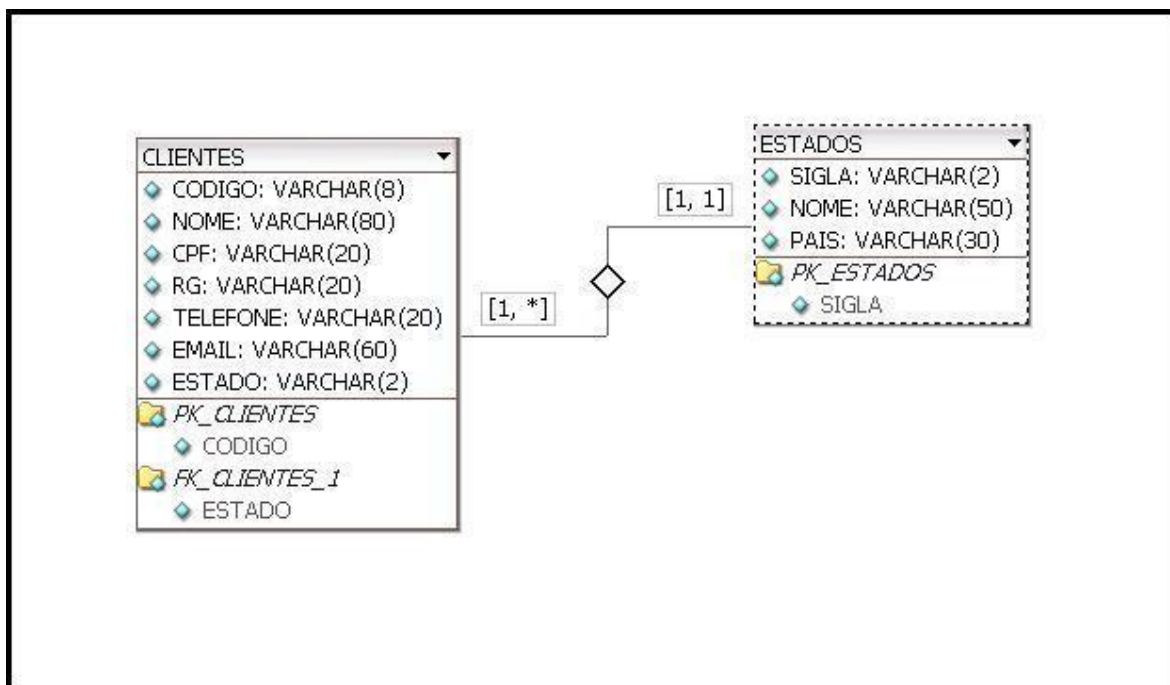
Listagem de Produtos

Código	Nome	Ultima Compra	Preco Custo	Preco Venda	Margem	Avaliação
1	Maracujá In-Natura	25/10/1999	300.20	600.40	300.20	Baixa
2	Abacaxi In-Natura	25/10/1999	200.20	500.40	300.20	Baixa
3	Acerola In-Natura	25/10/1999	400.20	700.40	300.20	Baixa
4	Laranja In-Natura	15/05/2006	240.20	600.40	360.20	Media
5	Açaí In-Natura	15/05/2006	100.20	500.40	400.20	Alta
6	Jerimum	15/05/2006	200.20	500.40	300.20	Baixa
Total:			1.441,20	3.402,40		

Exemplo de Cadastro Web com JSTL

Nossa pequena aplicação terá 2(duas) páginas **JSP**, uma contendo um formulário com alguns campos para cadastro de cliente, e outra com o código **JSTL** para processamento dos dados enviados via requisição **POST** e gravação dos mesmos no banco de dados.

Para melhor aproveitamento do exemplo que será exposto neste artigo, observe na **Figura 01**, as tabelas **CLIENTES** e **ESTADOS** que servirão de base para nossa pequena aplicação.



Agora usando a IDE de sua familiaridade, crie um novo projeto web, de preferência usando o container TomCat(estou usando no exemplo a versão 5.5.17) e crie dois arquivos JSP, um chamado **index.jsp** contendo um formulário com os campos referentes ao cadastro simples de clientes(**ver listagem 02**) e outro chamado **gravaCliente.jsp**, que executar o inserção na tabela do banco de dados, usando os valores enviados pela página index.jsp por método POST (**ver listagem 03**).

É importante lembrar que a tecnologia JSTL, é uma biblioteca de extensão da tecnologia JSP, por isso, criamos arquivos “.jsp” para que através de chamadas específicas das taglibs, passemos a usar os recursos oferecidos pela JSTL.

Note que os exemplos abaixo, estão comentados em formato web com os marcadores

<!-- e -->.

Index.jsp – Tela de Cadastro

```
<% @page contentType="text/html"%>
```

```
<% @page pageEncoding="UTF-8"%>
```

```
<!--Chamada aos TLD's de cada pacote JSTL -->
```

```
<% @taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<% @taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
"http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
```

```
<head>
```

<!--Criação de um dataSource, que proverá uma conexão ao servidor de banco dados, note que o escopo de conexão está definido como sendo de sessão, dessa forma poderemos usar o mesmo objeto chamado "ds" de qualquer outra página JSP participante do mesmo gerenciamento de sessão.-->

```
<sql:setDataSource var="ds"
```

```
    driver="org.firebirdsql.jdbc.FBDriver"
```

```
    url="jdbc:firebirdsql:localhost:c:\ACADEMICO.FDB"
```

```
    user="sysdba"
```

```
    password="masterkey"
```

```
    scope="session"/>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
<title>Tela Cadastro - Artigo Manoel Pimentel</title>
```

```
</head>
```

```
<body>
```

```
<h1>Exemplo JSTL - Tela de Cadastro</h1>

<hr>

<form action="gravaCliente.jsp" method="post">
    <label>Código </label><br>
    <input type="text" name="edtCodigo" size="10"/><br>
    <label>Nome: </label><br>
    <input type="text" name="edtNome" size="60"/><br>
    <label>CPF: </label><br>
    <input type="text" name="edtRG" size="15"/><br>
    <label>RG: </label><br>
    <input type="text" name="edtCPF" size="10"/><br>
    <label>Estado: </label><br>

    <!--Executa um comando SQL de seleção, gerando um objeto do tipo Result, que é
    semelhante a classe ResultSet da API JDBC -->

    <sql:query var="qryEstados" dataSource="${ds}">
        select * from ESTADOS
        order by
            NOME
    </sql:query>

    <!--Cria um objeto select (estilo comboBox), preenchendo suas opções com um laço
    forEach na coleção contida em "qryEstados.rows", armazenando cada registro, na variável estado, e
    acessando a valor de um determinado campo usando a EL(Expression
    Language) ${estado.nome} -->

    <select name="cmbEstados">
        <c:forEach var="estado" items="${qryEstados.rows}">
            <option value="PA">${estado.nome}</option>
        </c:forEach>
    </select><br>
```

```

<label>Telefone: </label><br>
<input type="text" id="edtTelefone" size="15"/><br>
<label>E-mail: </label><br>
<input type="text" name="edtEmail" size="50"/><br>
<hr>
<input accesskey="o" type="submit" name="btnOK" value="OK">
<input accesskey="c" type="reset" name="btnCancelar" value="Limpar">
</form>
</body>
</html>

```

The screenshot shows a web browser window with the title 'Tela Cadastro - Artigo Manoel Pimentel - Mozilla Firefox'. The address bar shows 'http://localhost:8084/Exemplo/JSTL2/'. The main content area has the heading 'Exemplo JSTL - Tela de Cadastro'. Below the heading is a registration form with the following fields and values:

- Código: C002
- Nome: Emanuel Silva Pimentel
- CPF: 989898
- RG: 8888888
- Estado: Pará (selected from a dropdown menu showing options: Pará, Amapá, Amazonas)
- E-mail: emanuel@pimentel.com

At the bottom of the form are two buttons: 'OK' and 'Limpar'.

gravaCliente.jsp – Para gravação na tabela CLIENTES

```

<% @page contentType="text/html"%>
<% @page pageEncoding="UTF-8"%>

<!-- Chamada aos TLD's de cada pacote JSTL -->
<% @taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<% @taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Tela Cadastro - Artigo Manoel Pimentel</title>
  </head>
  <body>
```

<!--Equivalente ao try/catch, a tag catch tenta executar o que estiver dentro de seu corpo, e caso ocorra alguma exceção, será capturada e armazenada na variável "ex". -->

```
<c:catch var="ex">
```

<!--Cria uma transação com o banco dados, onde podemos executar de forma mais protegida e sequencial, várias atualizações ou inserções. -->

```
<sql:transaction dataSource="${ds}">
```

<!--Executa algum comando como insert, update ou delete e armazena o resultado na variável "gravaCli". Note que estamos usando a tag sql:param para passar dinamicamente os valores em cada sinal de interrogação da cláusula values, vale lembrar que dessa forma, estaremos gerando uma espécie de sentença preparada, e dessa forma ganharemos performance na execução do comando SQL. -->

```
<sql:update var="gravaCli">
  insert into CLIENTES (CODIGO,NOME,CPF,RG,TELEFONE,EMAIL,ESTADO)
  values(?,?,?,?,?,?,?)
  <sql:param value="${param['edtCodigo']}" />
  <sql:param value="${param['edtNome']}" />
  <sql:param value="${param['edtCPF']}" />
  <sql:param value="${param['edtRG']}" />
```

```
<sql:param value="${param['edtTelefone']}" />
<sql:param value="${param['edtEmail']}" />
<sql:param value="${param['cmbEstados']}" />
```

```
</sql:update>
</sql:transaction>
</c:catch>
```

<!--Essa é uma sacada legal, pois na tag “out” abaixo, caso o objeto “ex” esteja nulo(ou seja, sem exceção), será exibido o valor contido no atributo default, dessa forma a mensagem de sucesso só será exibida caso não tenha ocorrido nenhuma exceção. -->

```
<h1>
  <c:out value="${ex}" default="Gravação executada com sucesso!" />
</h1>
<hr>
  <input type="button" value="Voltar" name="btnVoltar" onclick="history.back();"
</body>
</html>
```



JavaServer Faces – JSF

JSF é uma tecnologia que incorpora características de um framework MVC para WEB e de um modelo de interfaces gráficas baseado em eventos. Por basear-se no padrão de projeto MVC, uma de suas melhores vantagens é a clara separação entre a visualização e regras de negócio (modelo).

A idéia do padrão MVC é dividir uma aplicação em três camadas: modelo, visualização e controle.

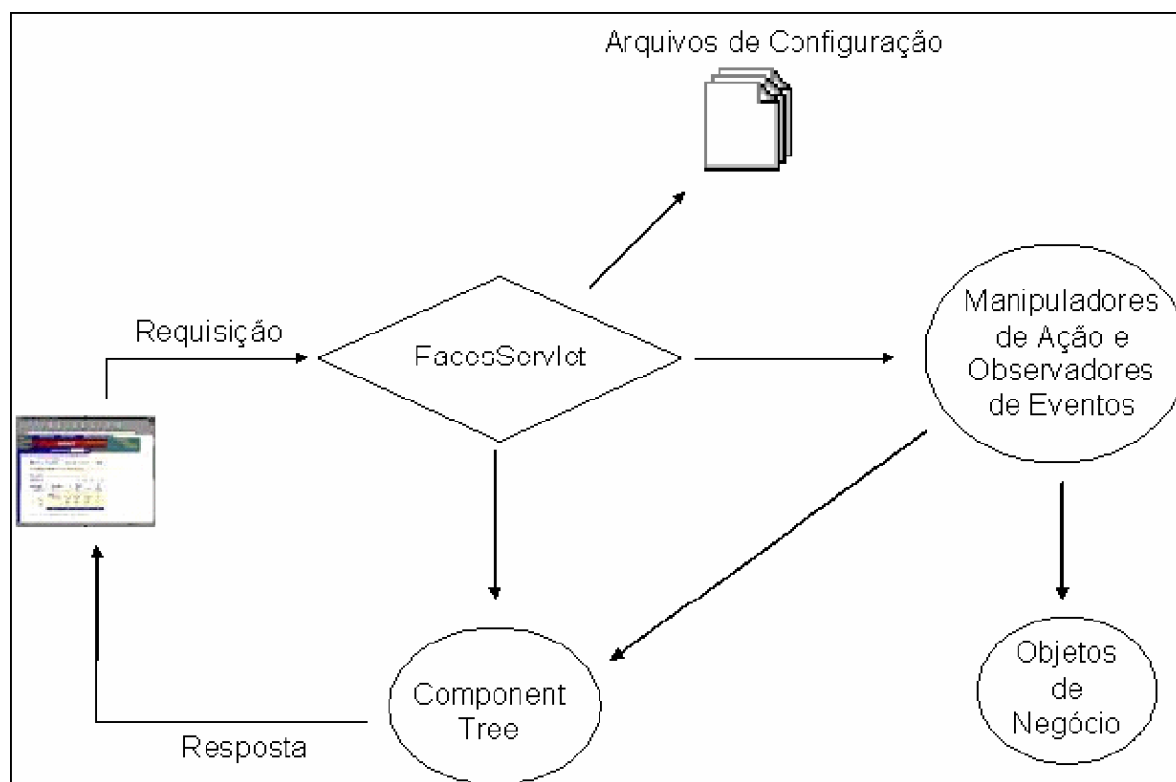
O modelo é responsável por representar os objetos de negócio, manter o estado da aplicação e fornecer ao controlador o acesso aos dados. A visualização representa a interface com o usuário, sendo responsável por definir a forma como os dados serão apresentados e encaminhar as ações dos usuários para o controlador. Já a camada de controle é responsável por fazer a ligação entre o modelo e a visualização, além de interpretar as ações do usuário e as traduzir para uma operação sobre o modelo, onde são realizadas mudanças e, então, gerar uma visualização apropriada.

No JSF, o controle é composto por um servlet denominado *FacesServlet*, por arquivos de configuração e por um conjunto de manipuladores de ações e observadores de eventos. O *FacesServlet* é responsável por receber requisições da WEB, redirecioná-las para o modelo e então remeter uma resposta.

Os arquivos de configuração são responsáveis por realizar associações e mapeamentos de ações e pela definição de regras de navegação. Os manipuladores de eventos são responsáveis por receber os dados vindos da camada de visualização, acessar o modelo, e então devolver o resultado para o *FacesServlet*.

O modelo representa os objetos de negócio e executa uma lógica de negócio ao receber os dados vindos da camada de visualização. Finalmente, a visualização é composta por *component trees* (hierarquia de componentes UI), tornando possível unir um componente ao outro para formar interfaces mais complexas.

A Figura 1 mostra a arquitetura do JavaServer Faces baseada no modelo MVC.



JavaServer Faces oferece ganhos no desenvolvimento de aplicações WEB por diversos motivos:

- Permite que o desenvolvedor crie UIs através de um conjunto de componentes UIs prédefinidos;
- Fornece um conjunto de tags JSP para acessar os componentes;
- Reusa componentes da página;
- Associa os eventos do lado cliente com os manipuladores dos eventos do lado servidor (os componentes de entrada possuem um valor local representando o estado no lado servidor);

- Fornece separação de funções que envolvem a construção de aplicações WEB.

Embora JavaServer Faces forneça tags JSP para representar os componentes em uma página, ele foi projetado para ser flexível, sem limitar-se a nenhuma linguagem markup em particular, nem a protocolos ou tipo de clientes. Ele também permite a criação de componentes próprios a partir de classes de componentes, conforme mencionado anteriormente.

JSF possui dois principais componentes: Java APIs para a representação de componentes UI e o gerenciamento de seus estados, manipulação/observação de eventos, validação de entrada, conversão de dados, internacionalização e acessibilidade; e taglibs JSP que expressam a interface JSF em uma página JSP e que realizam a conexão dos objetos no lado servidor.

É claro que existe muito mais a ser dito sobre JavaServer Faces. Esse artigo apenas fornece uma visão geral, mas espero ter criado uma certa curiosidade a respeito dessa nova tecnologia.

Implementando um exemplo com JSF

Será uma aplicação bem simples para demonstrar o uso dessa nova tecnologia. O nosso exemplo consiste de uma página inicial contendo 2 links: um para a inserção de dados e outro para a busca.

A página de inserção consiste de um formulário onde o usuário entrará com o nome, endereço, cidade e telefone. Os dados serão guardados em um banco de dados (no meu caso, eu uso o PostgreSQL) para uma posterior consulta. Se o nome a ser inserido já existir no banco de dados, uma mensagem será exibida informando ao usuário que o nome já está cadastrado (no nosso exemplo, o nome é a chave primária da tabela). Caso contrário, uma mensagem de sucesso será exibida ao usuário.

A busca se dará pelo nome da pessoa. Se o nome a ser buscado estiver cadastrado no banco, então uma página com os dados relativos ao nome buscado serão exibidos. Caso contrário, será informado ao usuário que o nome buscado não existe no banco.

Para a criação da tabela da base de dados foi utilizado o script apresentado abaixo.

```
CREATE TABLE pessoa
(
nome varchar(30) NOT NULL,
endereço varchar(50),
cidade varchar(20),
telefone varchar(10),
PRIMARY KEY (nome)
);
```

Index.jsf

```
<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head>
<title>Exemplo JSF</title>
</head>
<body>
<f:view>
<h:form>
```

```
<center>
<h1>Agenda</h1>
<br>
<h3>
<h:outputLink value="inserir.jsf">
<f:verbatim>Inserir</f:verbatim>
</h:outputLink>
<br><br>
<h:outputLink value="buscar.jsf">
<f:verbatim>Buscar</f:verbatim>
</h:outputLink>
</h3>
</center>
</h:form>
</f:view>
</body>
</html>
```

Algumas tags aqui merecem ser comentadas:

- `<h:form>` gera um formulário.
- `<h:outputLink>` cria um link para a página definida pelo campo *value*. O texto que compõe o link é colocado utilizando-se a tag `<f:verbatim>`.

O usuário terá a opção de buscar ou inserir dados. Os códigos das páginas de busca e inserção são mostrados a seguir, respectivamente.

Buscar.jsf

```
<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
```

```
<f:view>
<h:form>
<center><h2> Busca </h2></center>
<br>
Digite o nome:
<h:inputText id="nome" value="#{agenda.nome}"/>
<h:commandButton value="OK" action="#{agenda.buscar}"/>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Inserir.jsf

```
<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<center><h2> Inserção </h2></center>
<br>
<h3>Entre com os dados abaixo</h3>
<table>
<tr>
<td>Nome:</td>
<td>
<h:inputText value="#{agenda.nome}"/>
</td>
```

```

</tr>
<tr>
<td>Endereço:</td>
<td>
<h:inputText value="#{agenda.endereco}"/>
</td>
</tr>
<tr>
<td>Cidade:</td>
<td>
<h:inputText value="#{agenda.cidade}"/>
</td>
</tr>
<tr>
<td>Telefone:</td>
<td>
<h:inputText value="#{agenda.telefone}"/>
</td>
</tr>
</table>
<p>
<h:commandButton value="Inserir" action="#{agenda.inserir}"/>
</p>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body></html>

```

A tag `<h:inputText>` cria uma caixa de texto onde o valor digitado é guardado em *value*. Para criar um botão, é utilizada a tag `<h:commandButton>`. O label do botão é colocado em *value*. *Action*

determina qual a ação que o botão deve tomar. Na página buscar.jsp, ao clicar no botão OK, o método buscar() da classe AgendaDB é chamado.

O código da classe bean **AgendaDB** com os métodos getters e setters das variáveis nome, endereço, cidade e telefone e com os métodos inserir e buscar ficará conforme mostrado abaixo. É nesse arquivo onde a conexão com o banco é feita. Nas aplicações JSF, os beans são usados para que dados possam ser acessados através de uma página. O código java referente a essa classe deve estar localizado no diretório JavaSource. Já que estamos utilizando um Java bean, um *managed-bean* deverá ser criado no arquivo *faces-config.xml* que está presente no diretório *WEB-INF*.

```
import java.sql.*;

public class AgendaDB
{
    private String nome = blank;
    private String endereco = blank;
    private String cidade = blank;
    private String telefone = blank;
    private String result_busca = blank;
    private String result_inserir = blank;
    public static final String BUSCA_INVALIDA = "failure";
    public static final String BUSCA_VALIDA = "success";
    public static final String SUCESSO_INSERTAO = "success";
    public static final String FALHA_INSERTAO = "failure";
    static Connection con = null;
    static Statement stm = null;
    static ResultSet rs;
    static private String blank = "";

    public AgendaDB()
    {
        if (con==null) {
            try {
                Class.forName("org.postgresql.Driver");
                con =
```

```
DriverManager.getConnection("jdbc:postgresql://localhost:5432/talita","talita","tata");
} catch (SQLException e) {
System.err.println ("Erro: "+e);
con = null;
} catch (ClassNotFoundException e) {
System.out.println("ClassNotFoundException");
e.printStackTrace();
}
}
}
public String getNome() {
return nome;
}
public void setNome(String nome) {
this.nome = nome;
}
public String getCidade() {
return cidade;
}
public void setCidade(String cidade) {
this.cidade = cidade;
}
public String getEndereco() {
return endereco;
}
public void setEndereco(String endereco) {
this.endereco = endereco;
}
public String getTelefone() {
return telefone;
}
public void setTelefone(String telefone) {
this.telefone = telefone;
```



```
}

public String inserir() {
    String result_inserir = FALHA_INSERCAO;
    try {
        stm = con.createStatement();
        stm.execute("INSERT INTO pessoa(nome,endereco,cidade,telefone) VALUES ('" + nome + "','" +
            endereco + "','" + cidade + "','" + telefone + "')");
        stm.close();
        result_inserir = SUCESSO_INSERCAO;
    } catch (SQLException e) {
        System.err.println ("Erro: "+e);
        result_inserir = FALHA_INSERCAO;
    }
    return result_inserir;
}

public String buscar() throws SQLException {
    String result_busca = BUSCA_INVALIDA;
    try {
        stm = con.createStatement();
        rs = stm.executeQuery("SELECT * FROM pessoa WHERE nome = '" + nome + "'");
        if (rs.next()) {
            nome = rs.getString(1);
            endereco = rs.getString(2);
            cidade = rs.getString(3);
            telefone = rs.getString(4);
            result_busca = BUSCA_VALIDA;
        }
        else
            result_busca = BUSCA_INVALIDA;
        rs.close();
        stm.close();
    } catch (SQLException e) {
```

```
System.err.println ("Erro: "+e);  
}  
return result_busca;  
}  
}
```

Ainda existirão mais 4 páginas JSP em nosso exemplo:

Sucesso_busca.jsf

esta página informa ao usuário que a busca foi bem sucedida, apresentando os dados referentes ao nome procurado:

```
<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>  
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>  
<html>  
<body>  
<f:view>  
<h:form>  
<center><h2> Resultado da Busca </h2></center>  
<br>  
<table>  
<tr>  
<td>Nome:</td>  
<td>  
<h:outputText value="#{agenda.nome}"/>  
</td>  
</tr>  
<tr>  
<td>Endereço:</td>  
<td>  
<h:outputText value="#{agenda.endereco}"/>  
</td>  
</tr>
```

```
<tr>
<td>Cidade:</td>
<td>
<h:outputText value="#{agenda.cidade}"/>
</td>
</tr>
<tr>
<td>Telefone:</td>
<td>
<h:outputText value="#{agenda.telefone}"/>
</td>
</tr>
</table>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Falha_busca.jsf

Esta página informa ao usuário que o nome buscado não existe no banco de dados:

```
<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<h3>
<h:outputText value="#{agenda.nome}"/>
não foi encontrado(a)!
```

```

</h3>
</h:form>
<br>
<h:outputLink value="buscar.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>

```

Sucesso_insercao.jsf

Informa que os dados foram inseridos com sucesso no banco:

```

<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<% @ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
Dados Inseridos com Sucesso!
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>

```

Falha_insercao.jsf

Informa que os dados não foram inseridos porque já existe no banco um nome igual ao que está se tentando inserir.

```

<% @ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

```

```
<% @ taglib uri="http://java.sun.com/jsp/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<h3>
<h:outputText value="#{agenda.nome}"/>
já está cadastrado!Entre com outro nome!
</h3>
</h:form>
<br>
<h:outputLink value="inserir.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```


Arquivos de Configuração

Finalmente faltam os arquivos de configuração *faces-config.xml* e *web.xml*. No arquivo *facesconfig.xml* nós vamos definir as regras de navegação e o managed-bean relativo à nossa classe AgendaDB.

O código está mostrado logo abaixo.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
<navigation-rule>
<from-view-id>/buscar.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/sucesso_busca.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/falha_busca.jsp</to-view-id>
</navigation-case>
</navigation-rule>
<navigation-rule>
<from-view-id>/inserir.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/sucesso_insercao.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/falha_insercao.jsp</to-view-id>
</navigation-case>
```

```
</navigation-rule>
<managed-bean>
<managed-bean-name>agenda</managed-bean-name>
<managed-bean-class>AgendaDB</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```


O código do arquivo *web.xml* pode ser visto abaixo.

```
<?xml version="1.0"?>
<!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>
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name>
<param-value>client</param-value>
</context-param>
<context-param>
<param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>/WEB-INF/faces-config.xml</param-value>
</context-param>
<listener>
<listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
<!-- Faces Servlet -->
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup> 1 </load-on-startup>
</servlet>
<!-- Faces Servlet Mapping -->
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
</web-app>
```

A linha `<url-pattern>*.jsf</url-pattern>` presente no último bloco desse arquivo, indica que as páginas terão a extensão jsf. Por esse motivo que qualquer chamada de uma página dentro de outra a extensão não é jsp e sim jsf.