

# Java Web (online)



## CRÉDITOS

### Copyright © Impacta Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este material de estudo (textos, imagens, áudios e vídeos) não pode ser copiado, reproduzido, traduzido, baixado ou convertido em qualquer outra forma eletrônica, digital ou impressa, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Impacta Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

*"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."*

## Java Web (online)

### Coordenação Geral

Marcia M. Rosa

### Coordenação Editorial

Henrique Thomaz Bruscagin

### Supervisão de Desenvolvimento Digital

Alexandre Hideki Chicaoka

### Produção, Gravação, Edição de Vídeo e Finalização

Laís Oliveira Paschoa (Impacta Produtora)  
Xandros Luiz de Oliveira Almeida (Impacta Produtora)

### Roteirização

Edson Marcio Prestes Cordeiro dos Santos

### Curso ministrado por

Edson Marcio Prestes Cordeiro dos Santos

### Edição e Revisão final

Luiz Fernando Oliveira

Edição n. 1774\_0\_EAD  
Mar/2016

*Este material é uma nova obra derivada da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Out/2013:*

*Java Web*

*Autoria: Roárgio Mastropietro*

### Sobre o instrutor do curso:

*Edson Marcio Prestes é bacharel em Informática e programador sênior em Java. Participa de vários projetos Web utilizando JavaServlets/JSP, frameworks e JEE nas áreas financeira, industrial e internet. É instrutor da Impacta desde 2009, ministrando cursos de Java e Android.*

# Sumário

<b>Capítulo 1: Java e a Web .....</b>	<b>9</b>
O mundo da Web e a plataforma Java .....	10
A tecnologia Java.....	10
Como funcionam as requisições Web em geral .....	12
Teste seus conhecimentos .....	15
<b>Capítulo 2: Servidores e containers .....</b>	<b>19</b>
Servidores de aplicação e o Web Container .....	20
Arquitetura de um servidor Java EE em camadas.....	23
Montando o ambiente de desenvolvimento .....	25
Requisições e respostas HTTP .....	27
Teste seus conhecimentos .....	31
Mãos à obra! .....	35
<b>Capítulo 3: Conceitos básicos sobre Servlets .....</b>	<b>43</b>
O que são Servlets? .....	44
O ciclo de vida de um Servlet .....	45
Requisições e respostas no Java Web Container.....	46
Mapeando Servlets no web.xml e anotações na versão Servlet 3.0	49
O projeto Web e o arquivo WAR.....	53
Teste seus conhecimentos .....	57
Mãos à obra! .....	61
<b>Capítulo 4: Comunicação entre cliente e servidor .....</b>	<b>67</b>
Introdução .....	68
Enviando e capturando parâmetros na requisição .....	68
<i>Requisições via GET</i> .....	68
<i>Requisições via POST</i> .....	72
<i>Principais diferenças entre GET e POST</i> .....	75
Encaminhamento e redirecionamento .....	76
<i>Encaminhamento ou Forward</i> .....	76
<i>Redirecionamento ou Redirect</i> .....	78
Parâmetros de inicialização: o ServletConfig e o ServletContext... .....	79
<i>O ServletConfig</i> .....	80
<i>O ServletContext</i> .....	81
Teste seus conhecimentos .....	83
Mãos à obra! .....	87
<b>Capítulo 5: Aplicação avançada de Servlets .....</b>	<b>97</b>

<b>Atributos e listeners .....</b>	<b>98</b>
<i>Atributos.....</i>	<i>98</i>
<i>Listeners .....</i>	<i>101</i>
<b>Uso de cookies e gerenciamento de sessão .....</b>	<b>105</b>
<i>O que são cookies? .....</i>	<i>105</i>
<i>A classe javax.servlet.http.Cookie.....</i>	<i>107</i>
<i>Demonstrando o uso de cookies.....</i>	<i>108</i>
<i>Definindo a necessidade e a utilidade do uso de sessões.....</i>	<i>111</i>
<i>O controle feito pelo Container.....</i>	<i>111</i>
<i>Manipulando a sessão via código .....</i>	<i>113</i>
<i>Invalidando uma sessão .....</i>	<i>114</i>
<b>Filtros.....</b>	<b>116</b>
<i>O que são e para que servem os filtros .....</i>	<i>116</i>
<i>A interface Filter .....</i>	<i>118</i>
<i>Declarando e configurando um filtro.....</i>	<i>118</i>
<b>Teste seus conhecimentos .....</b>	<b>121</b>
<b>Mãos à obra! .....</b>	<b>125</b>
<b>Capítulo 6: Introdução às JSPs .....</b>	<b>135</b>
<b>A necessidade das JSPs e sua inserção no Projeto Web Dinâmico .....</b>	<b>136</b>
<i>As JSPs no contexto do projeto Web Dinâmico .....</i>	<i>137</i>
<b>Elementos de Java Server Pages .....</b>	<b>139</b>
<i>Scriptlets.....</i>	<i>139</i>
<i>Comentários .....</i>	<i>140</i>
<i>Expressões.....</i>	<i>140</i>
<i>Declarações.....</i>	<i>141</i>
<i>Diretivas.....</i>	<i>145</i>
<i>Objetos implícitos .....</i>	<i>148</i>
<b>Teste seus conhecimentos .....</b>	<b>151</b>
<b>Mãos à obra! .....</b>	<b>155</b>
<b>Capítulo 7: Acesso a dados pelas JSPs .....</b>	<b>163</b>
<b>Leitura de dados recebidos de um formulário – GET e POST .....</b>	<b>164</b>
<b>Escrita e leitura de atributos entre Servlets e JSPs .....</b>	<b>165</b>
<b>JavaBeans e Standard Actions .....</b>	<b>169</b>
<i>&lt;jsp:useBean&gt;.....</i>	<i>170</i>
<i>&lt;jsp:setProperty&gt; .....</i>	<i>170</i>
<i>&lt;jsp:getProperty&gt; .....</i>	<i>171</i>
<b>Composição de páginas – Standard Actions &lt;jsp:include&gt; e &lt;jsp:param&gt; .....</b>	<b>172</b>
<b>Teste seus conhecimentos .....</b>	<b>175</b>
<b>Mãos à obra! .....</b>	<b>179</b>
<b>Capítulo 8: Expression Language .....</b>	<b>187</b>
<b>O que é e para que serve a EL? .....</b>	<b>188</b>
<b>Sintaxe e operadores .....</b>	<b>188</b>
<b>Variáveis e acesso a dados .....</b>	<b>191</b>

Navegação em dados .....	194
Configuração de EL e Scriptlets em JSPs .....	196
Teste seus conhecimentos .....	199
Mãos à obra! .....	203
<b>Capítulo 9: Tratamento de erros em aplicações Web.....</b>	<b>211</b>
Introdução .....	212
Tratamento de erros em Servlets .....	213
<i>Criando e configurando uma página de erro personalizada</i> .....	219
<i>Tratando erros HTTP</i> .....	222
Tratamento de erros em JSPs .....	223
Teste seus conhecimentos .....	225
Mãos à obra! .....	229
<b>Capítulo 10: Introdução ao uso de tags.....</b>	<b>233</b>
O que são tags e para que servem as JSP Custom Tags?.....	234
Criando e utilizando Custom Tags em JSPs .....	234
<i>Tags simples</i> .....	235
<i>Tags com atributos</i> .....	238
<i>Acessando o corpo de uma tag</i> .....	240
JSTL – Java Standard Tag Library .....	243
<i>Histórico e definições</i> .....	243
<i>Biblioteca Core</i> .....	246
<i>Biblioteca de internacionalização e mensagens</i> .....	250
Teste seus conhecimentos .....	255
Mãos à obra! .....	259
<b>Capítulo 11: Conhecendo a arquitetura MVC.....</b>	<b>267</b>
Introdução à arquitetura MVC .....	268
Quais as vantagens do uso de um framework? .....	270
O Struts 2 como framework MVC.....	272
Teste seus conhecimentos .....	275
<b>Capítulo 12: Primeira aplicação com Struts 2.....</b>	<b>279</b>
Introdução .....	280
Instalação e configuração inicial .....	280
Criação dos componentes Struts 2 fundamentais da aplicação ..	286
Executando a aplicação e analisando os resultados .....	288
Para onde seguir a partir deste ponto?.....	289
<b>Apêndice 1: Instalação e configuração do Apache Tomcat 7 .....</b>	<b>291</b>
Download do Apache Tomcat.....	292

A estrutura de arquivos do Apache Tomcat.....	295
Execução do Apache Tomcat via prompt de comando.....	296
Execução do Apache Tomcat via IDE Eclipse.....	299
Configuração de usuários no Apache Tomcat.....	303
<b>Apêndice 2: Os objetos Request e Response com detalhes</b>	<b>307</b>
Introdução .....	308
A interface HttpServletRequest.....	309
A interface HttpServletResponse.....	311
Exemplo da utilização dos métodos da interface HttpServletRequest	312

# Apresentação

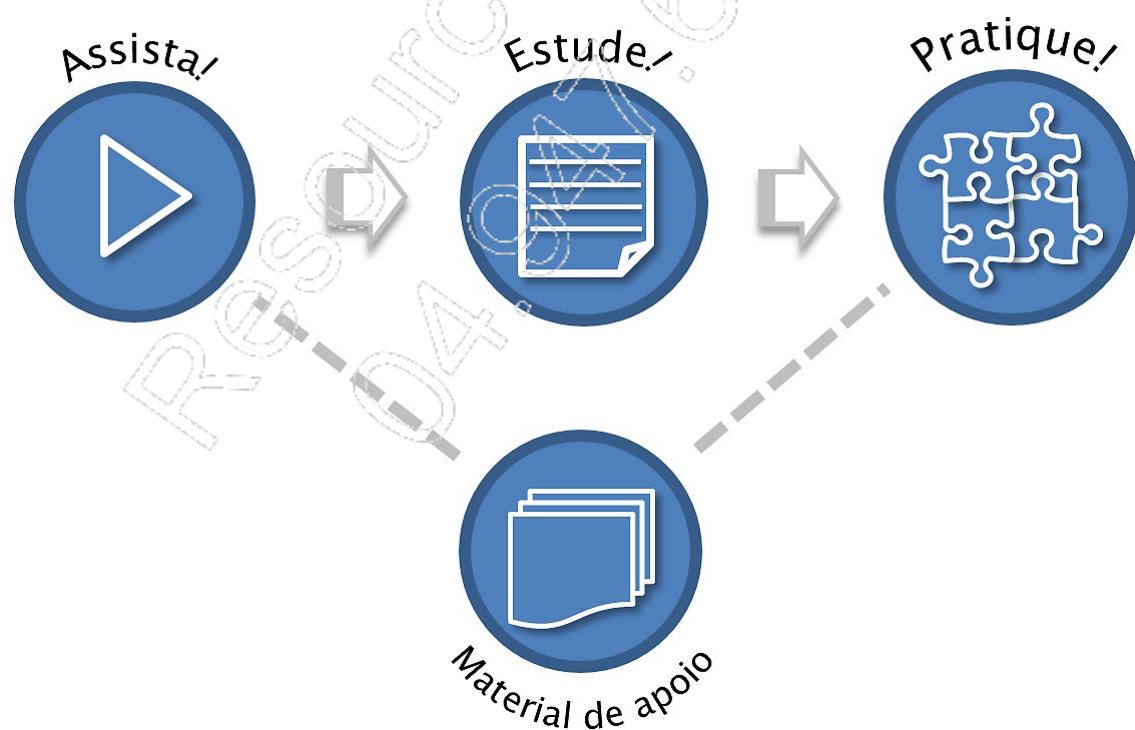
## Bem-vindo!

É um prazer tê-lo como aluno do nosso curso online de Java Web. Este curso é ideal para você que é desenvolvedor, analista, arquiteto, gestor ou entusiasta da plataforma Java atuante na área de TI e deseja adquirir sólido conteúdo referente à tecnologia Java para a Web voltada à construção de aplicações para os ambientes da Internet e Intranet.

Neste curso online, serão apresentados conceitos e aplicações práticas das principais e fundamentais tecnologias que compõem a base da tecnologia Java para a construção de aplicações Web: Servlets e JSP.

Para ter um bom aproveitamento deste curso, é imprescindível que você tenha participado do nosso curso de Java Programmer ou Java Programmer – Módulo III (online), e HTML5 – Fundamentos, ou possua conhecimentos equivalentes. Bom aprendizado!

## Como estudar?



Este curso conta com:



Videoaulas sobre os assuntos que você precisa saber no curso online de Java Web.



Parte teórica, com mais exemplos e detalhes para você que quer se aprofundar no assunto da videoaula.



Exercícios de testes para você pôr à prova o que aprendeu.



Material de apoio para você testar os exemplos e laboratórios da apostila.

# Capítulo 1:

## Java e a Web

# O mundo da Web e a plataforma Java

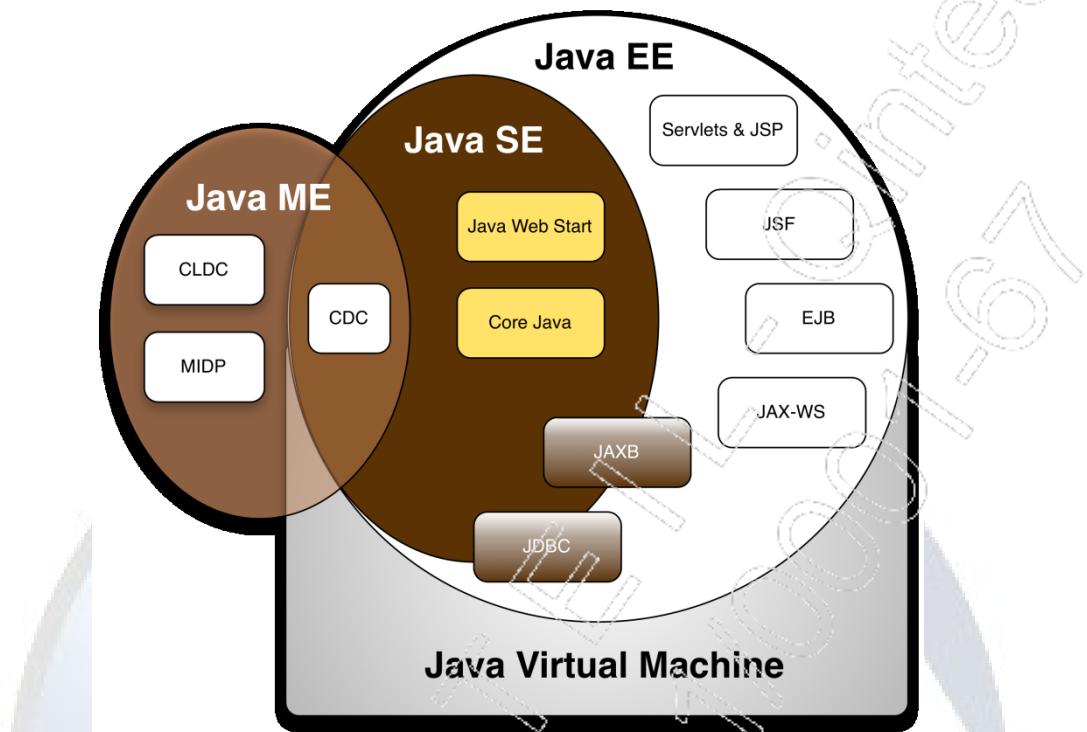
O mundo das aplicações Web é o grande foco das empresas e profissionais de TI no momento. Vivenciamos uma grande expansão dos aplicativos e sistemas baseados na Web em todas as áreas do conhecimento. Grandes quantidades de informações trafegam entre as aplicações, são processadas, moldadas e apresentadas aos mais diversos usuários e nas mais variadas formas: listas, tabelas, gráficos, textos, entre outros.

É nesse cenário que o desenvolvedor Web aparece como profissional-chave, uma ferramenta fundamental para a criação e manutenção de aplicativos baseados na Web. No entanto, ele não é o único atuante nesse meio. A plataforma Java para o desenvolvimento Web constitui-se em um conjunto de ferramentas e de boas práticas, no meio para se construir as aplicações e na forma de realizar o design e a arquitetura engendrados pelo desenvolvedor ou pela equipe de desenvolvimento.

## A tecnologia Java

Java Enterprise Edition é o nome dado à versão da tecnologia Java direcionada às aplicações Web e Enterprise. Didaticamente, a tecnologia Java é dividida em três grandes segmentos: Java EE, Java SE e Java ME.

A seguir, temos um esquema ilustrativo das divisões da plataforma e suas principais APIs:



Nesse esquema, pode-se denotar a presença das citadas áreas, cada qual representando um conjunto de APIs aplicáveis conjunta ou separadamente. São elas:

- **Java SE:** Termo que se refere a Java Standard Edition. É o ramo principal, isto é, representa o núcleo da plataforma. Contém todas as bibliotecas fundamentais e é base para a divisão Enterprise Edition (Java EE) e para parte da divisão Micro Edition (Java ME);
- **Java EE:** Termo que se refere a Java Enterprise Edition. É sem dúvida o maior ramo da plataforma Java e engloba as tecnologias Web e Enterprise (sistemas distribuídos). Baseia-se na versão Java SE e acrescenta uma grande quantidade de APIs específicas ao seu propósito. O Java EE é o foco desta apostila;
- **Java ME:** Termo que se refere a Java Micro Edition. É voltado ao segmento de aplicações Java Embedded, como controladores e robôs, e aos chamados Consumer Devices. Vale lembrar que Java

ME não se refere à tecnologia Android, mas sim a aplicações Java independentes de plataforma. Android é um sistema operacional baseado em Linux para smartphones e tablets. Ele foi concebido por um conjunto de empresas e desenvolvedores, mas foi predominantemente elaborado e é mantido pela empresa Google, que utiliza prioritariamente a tecnologia Java para desenvolvimento de aplicativos. É a menor versão das divisões da plataforma Java. Parte de sua arquitetura baseia-se na JVM como todas as outras versões, porém, abrange algumas tecnologias não baseadas na Java Virtual Machine como MIDP e CLDC.

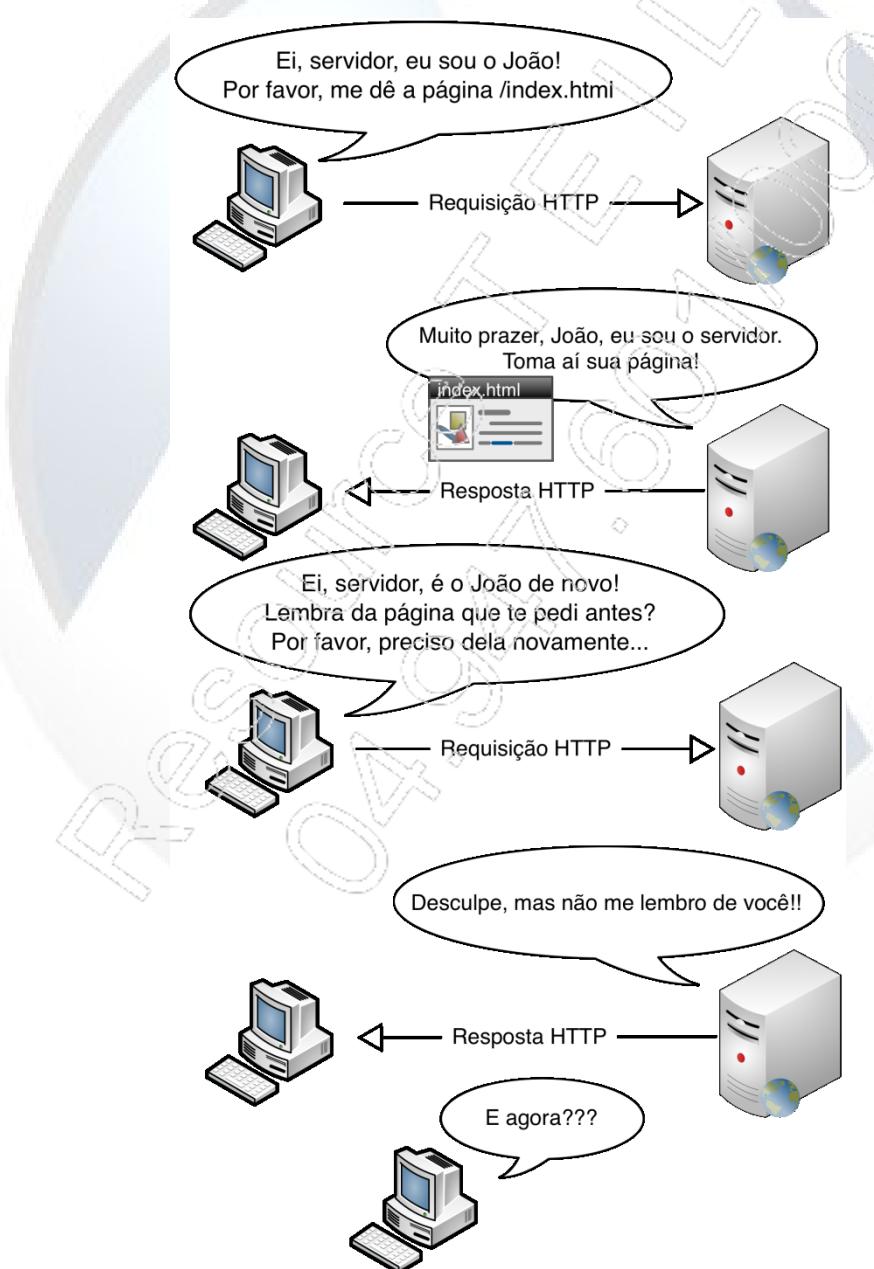
Nosso objeto de estudo são as APIs relacionadas à tecnologia Web da plataforma Java, ou seja, a porção da divisão Java Enterprise Edition (Java EE) utilizada em aplicações Web. Mais especificamente, nosso foco de abordagem recai sobre as especificações Servlet e JSP da versão Java EE 6, a versão mais recente, utilizada mundialmente e preferencial para o ambiente Java Enterprise Edition.

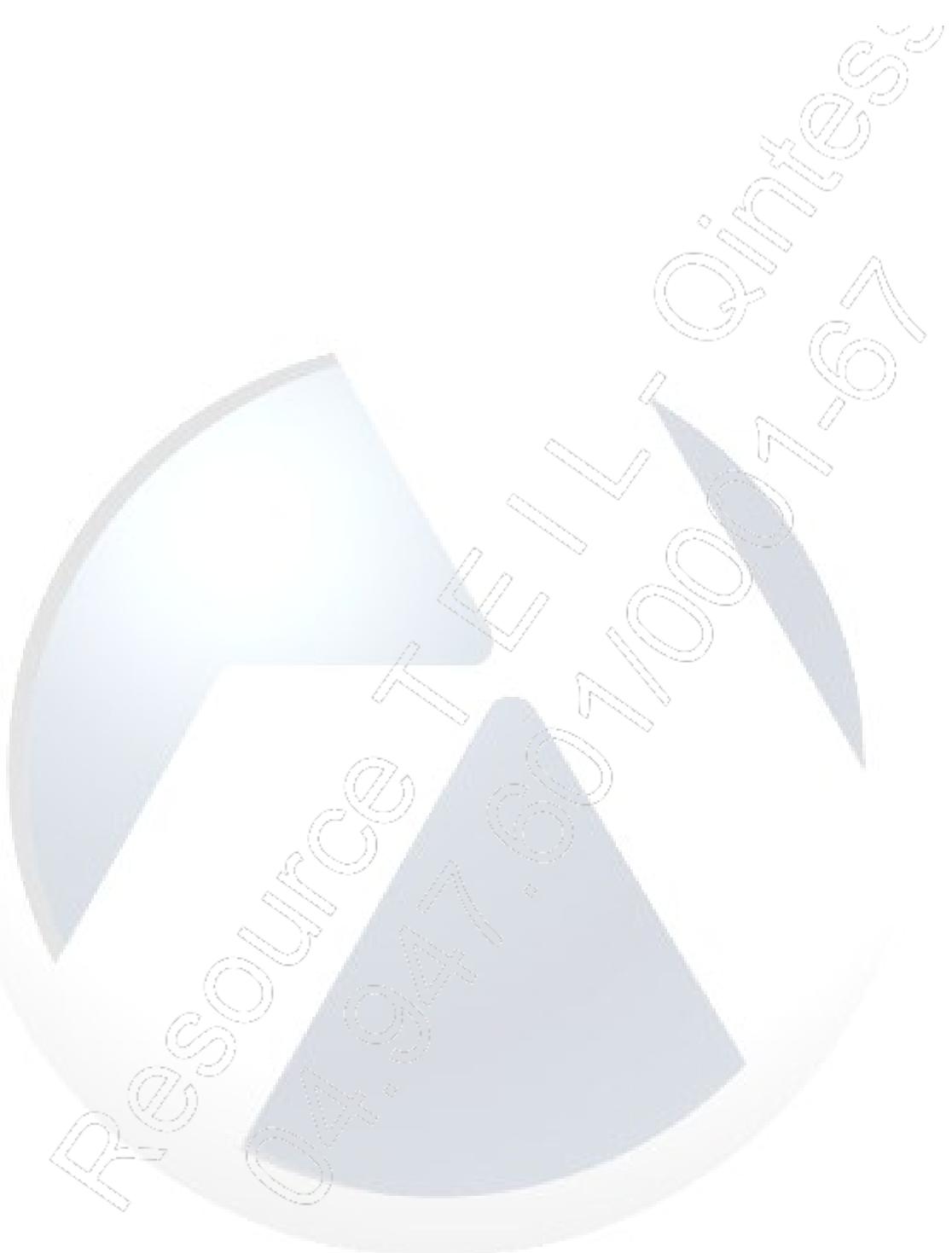
## Como funcionam as requisições Web em geral

Uma primeira e importante pergunta sobre aplicativos para a Web é exatamente como o fluxo de informações ocorre. O conceito de Stateless (sem estado) surge nesse cenário e significa exatamente o que a primeira impressão indica: não há manutenção de estado no mundo Web. O que isso significa?

A cada requisição do usuário, geralmente feita por um browser (navegador) ou por outro aplicativo, as informações passadas ou recebidas restringem-se somente àquela única requisição. O servidor desconhece a identidade ou o histórico das requisições feitas anteriormente por aquele cliente ou outros. A cada pedido ou clique dado em uma página ou aplicação Web, o servidor inicia novamente seu percurso para atender o pedido, mas sem lembranças de pedidos anteriores.

Quer dizer que não há como manter um estado persistente de conversação entre o cliente e o servidor? Na verdade, há diversas formas de se fazer isso, mas é exatamente este o ponto a destacar: o desenvolvedor terá que lidar com essa necessidade e preocupar-se a todo momento com esse estado de conversação, utilizando as ferramentas que possui. Essa característica é fundamental para aplicações Web. A cada dia, surgem formas mais poderosas e elegantes de se gerenciar o estado dos dados e requisições dos clientes. Os modos mais comuns de se realizar tal tarefa são por meio de sessões e cookies. Esses conceitos serão vistos mais adiante.





# Java e a Web

## Teste seus conhecimentos



**1. A tecnologia Java é dividida em três grandes segmentos. Quais são eles?**

- a) Java ME, Java SE e Java Mobile.
- b) Java EE, Java SE e Java Card.
- c) Java ME, Java SE e Java EE.
- d) Java Mobile, Java SE e Java EE.
- e) Java ME, Java Embedded e Java EE.

**2. As aplicações Web são foco da tecnologia Java EE e seu estudo recai basicamente sobre duas especificações e assuntos correlatos. Quais são essas especificações?**

- a) EJBs e Servlets.
- b) JSPs e EJBs.
- c) Classes e Servlets.
- d) Servlets e JSPs.
- e) Nenhuma das alternativas anteriores está correta.

**3. Qual é o nome do ramo da tecnologia Java responsável pelo núcleo da linguagem e que serve de base para as demais áreas do desenvolvimento Java?**

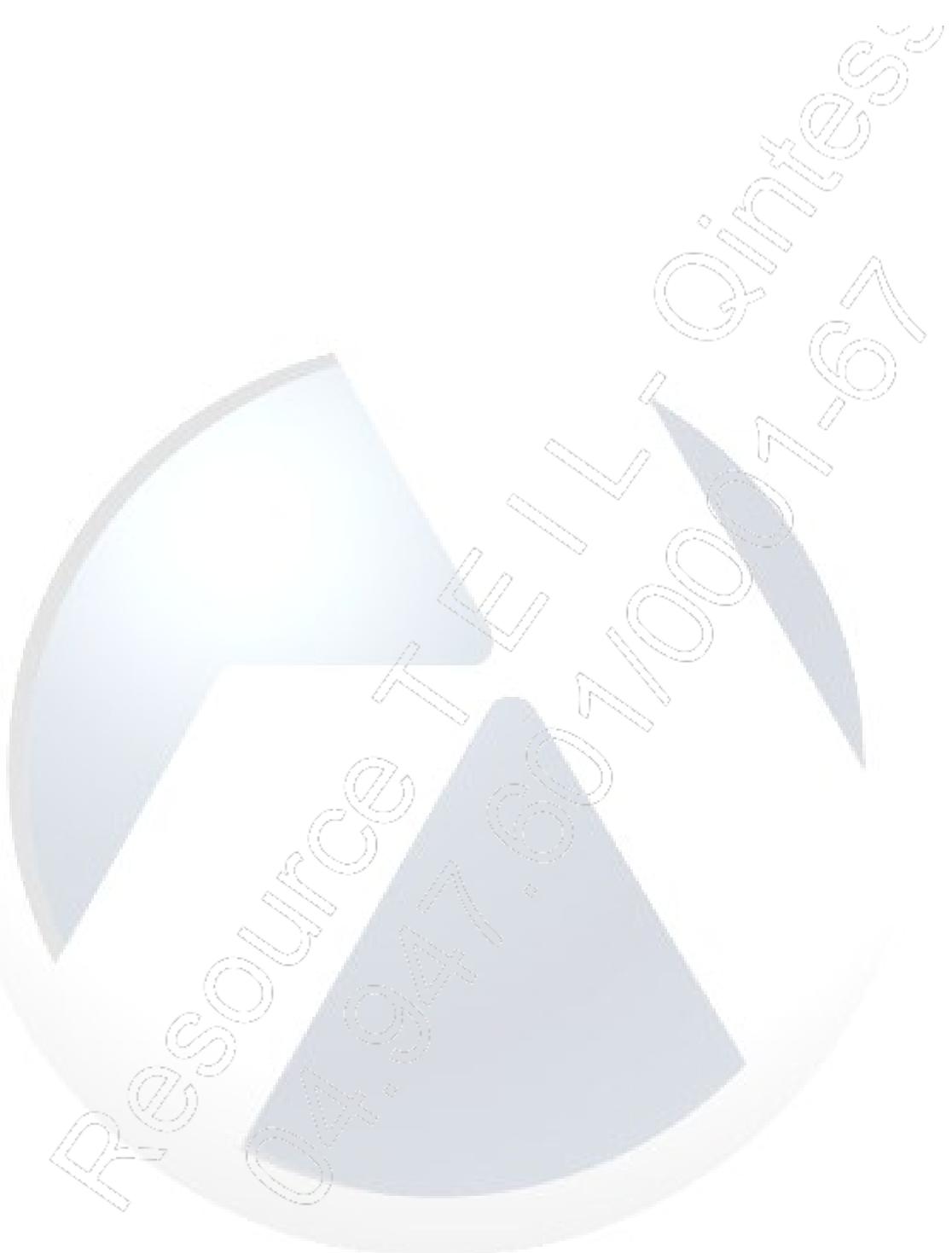
- a) Java SE
- b) Java EE
- c) Java ME
- d) Servlets
- e) JSP

**4. Por que dizemos que a realidade Web é stateless?**

- a) Porque os servidores e os navegadores usados para fazer requisições na Web sempre sabem como se identificar mutuamente, independente da requisição.
- b) Porque os servidores e os navegadores usados para fazer requisições na Web nunca sabem como se identificar mutuamente, independente da requisição.
- c) Porque os servidores e os navegadores usados para fazer requisições na Web não sabem como se identificar mutuamente de forma espontânea, isto é, dependem do desenvolvedor para realizar o tratamento do estado da sessão entre diversas requisições.
- d) Porque o estado do navegador é importante para a requisição a ser feita.
- e) Nenhuma das alternativas anteriores está correta.

**5. Qual(is) é(são) a(s) forma(s) mais comum(ns) de se manter o controle de estado de conversação entre diversas requisições?**

- a) Por meio de sessões e Servlets.
- b) Por meio de sessões e Cookies.
- c) Por meio de cookies.
- d) Usar um banco de dados no servidor.
- e) Nenhuma das alternativas anteriores está correta.



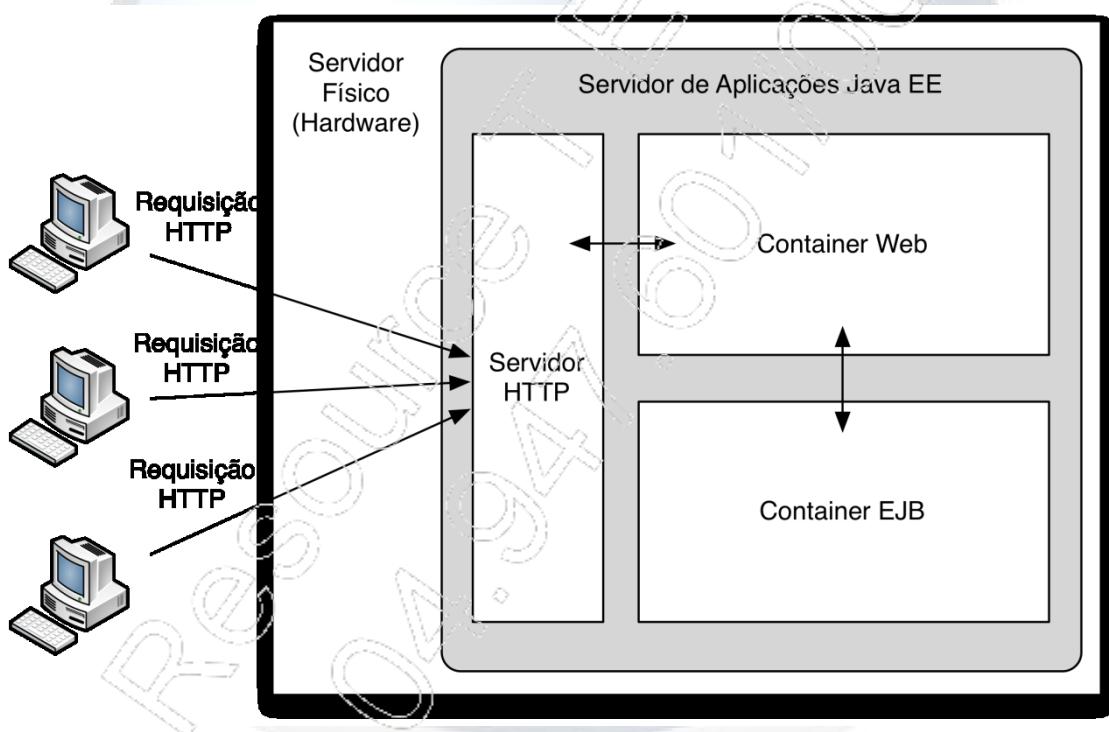
# Capítulo 2:

# Servidores e

# containers

# Servidores de aplicação e o Web Container

As aplicações Web em geral têm alguns requisitos para que possam ser implementadas, seja qual for a tecnologia utilizada. Essas necessidades podem variar conforme a plataforma de desenvolvimento, mas, em geral, algumas são comuns. A existência de um servidor de aplicações é uma realidade que permeia todas as tecnologias de desenvolvimento Web. Um servidor de aplicações trabalha conjuntamente com um servidor Web (Servidor HTTP) para responder a requisições de usuários e clientes. Veja um modelo esquemático a seguir de um servidor Java EE:



No modelo apresentado, as requisições provindas da Web são recebidas por um servidor HTTP (servidor Web), que mapeia o endereço solicitado e encaminha a chamada para o programa registrado como responsável – no caso da plataforma Java, para o Web Container.

Dessa forma, dentro do Web Container, é feito um redirecionamento para o componente registrado a fim de responder às requisições feitas naquele endereço. Comumente, esse componente é um Java Servlet, que pode resolver a requisição de forma isolada ou repassá-la para outros componentes que farão o trabalho útil.

Finalmente, a resposta é montada dinamicamente e redirecionada ao usuário em formato HTML, sendo interpretado e renderizado pelo navegador Web instalado na máquina do usuário.

Em meio a todos esses conceitos, como definir o que é servidor e container? Um servidor, no mundo dos profissionais de infraestrutura de TI, pode significar o conjunto de equipamentos, por vezes distribuídos e escaláveis, que compõem o hardware e o software utilitários, sediados geralmente em um datacenter. Nesse caso, esses equipamentos correspondem ao conceito de servidor físico ou virtual, provendo acesso a arquivos, impressão, aplicativos, Web, banco de dados, etc. Contudo, o servidor é efetivamente o que chamamos de servidor físico, administrado por uma equipe de Engenharia de Redes, com características como memória, espaço em disco e capacidade de processamento.

O conceito de servidor de aplicações supõe apenas um software, um programa de computador sendo executado em um servidor físico e que responde a chamadas em determinada(s) porta(s) de comunicação. Esse tipo de servidor pode atender a chamadas por diversos protocolos de rede, como TCP, HTTP, HTTPS, FTP, SSH, entre outros.

Dentro do servidor de aplicações, na plataforma Java, a especificação da plataforma prevê a existência de dois "containers", para estar de acordo com a definição de um servidor Java EE Completo:

- Web Container;
- EJB Container.

Pela ideia de container, pode-se imaginar um ambiente independente no qual são alocados componentes funcionalmente interligados para um determinado fim. Nesse sentido, trata-se de um espaço feito para alocar outros componentes que sejam específicos para as funcionalidades de Web (Web Container) e componentes de negócios distribuídos (EJB Container).

É importante ressaltar que a diferenciação entre os conceitos de servidor de aplicações e containers pode não ser simples. Alguns preferem unificar os dois conceitos para fins didáticos no início do aprendizado.

Assim como existem requisitos para um Servidor Java EE Completo, há requisitos para servidores que apenas tomam forma de acordo com a especificação do Web Container. Essa especificação já era chamada de Web Profile pela Sun Microsystems e agora esse nome foi mantido pela Oracle, que adquiriu a empresa anterior.

É exatamente neste campo que a plataforma Java para a Web se assenta: será utilizado um dos servidores que concentra apenas o Web Container da plataforma Java EE 6, o Apache Tomcat, largamente utilizado no mundo, dada sua performance, baixa necessidade de recursos para funcionamento e facilidade para instalação e configuração. Outros pontos importantes: ele é open source e gratuito.

Apenas para conhecimento, segue uma lista dos servidores de aplicação Java mais utilizados no mundo hoje em dia, além do Apache Tomcat, já citado:

- **Full Profile (Web Container e EJB Container)**
- Oracle GlassFish Server: Disponível em versão open source gratuita e comercial;
- Oracle WebLogic Server: Somente em versão comercial;
- JBoss Application Server: Red Hat, disponível nas versões open source e comercial;

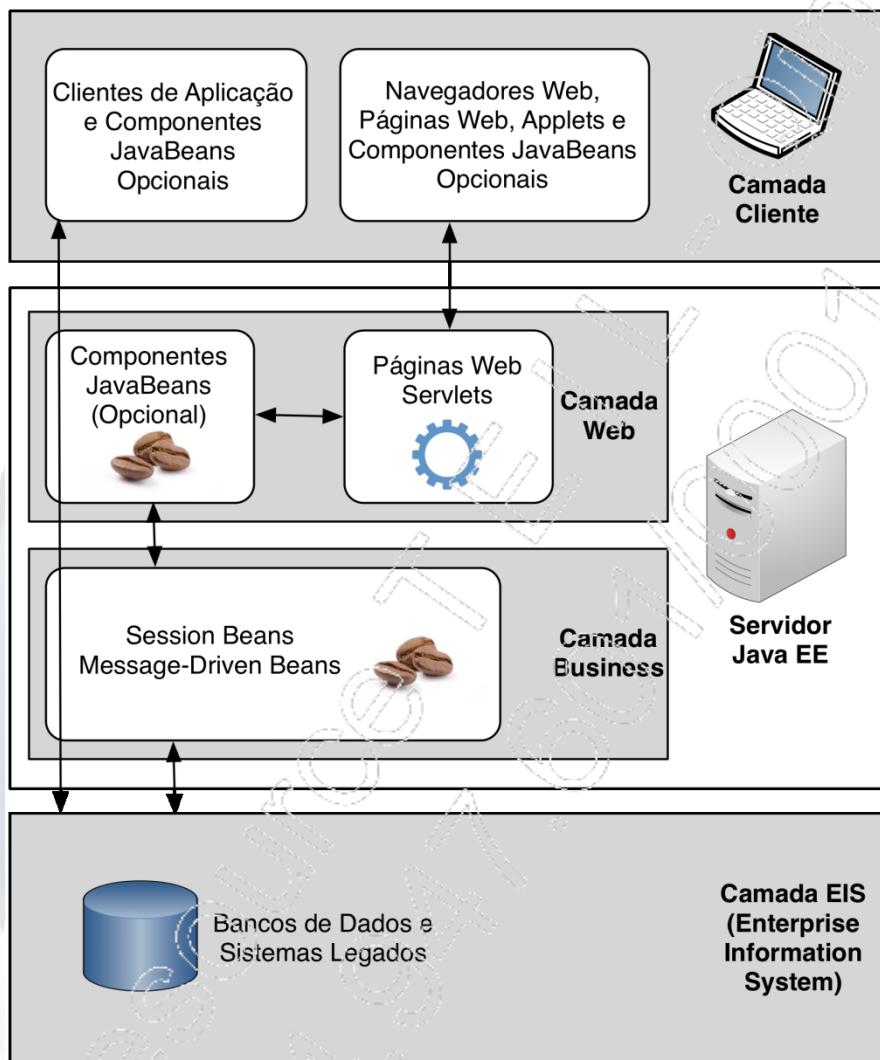
- IBM WebSphere Application Server: Disponível nas versões Community Edition e comercial;
- Apache Geronimo - Apache Software Foundation: Open source.
  - **Web Profile**
- GlassFish Server Open Source Edition;
- JBoss Application Server - Red Hat;
- Caucho Resin;
- Apache TomEE (Versão entre o Tomcat e o Geronimo).

Essa lista traz alguns exemplos, mas existem diversas outras distribuições. Para consultar as distribuições compatíveis com as especificações Java EE e já aprovadas para uso na plataforma Java, consulte o endereço:  
<http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html> (Acesso em 30 ago. 2013).

## Arquitetura de um servidor Java EE em camadas

As aplicações Java EE apresentam diversas possibilidades de configuração, conforme o objetivo e porte do sistema a ser desenvolvido. Em muitas situações, as empresas já possuem sistemas legados ou bancos de dados em uso (chamados EIS - Enterprise Information Services) e o sistema a ser desenvolvido deve comunicar-se e interagir com esse conjunto. É nessa análise de implantação que se define a arquitetura a ser usada.

A fim de observar a colocação da camada Web (representada pelo Web Container) em uma abordagem completa, temos o diagrama que descreve a arquitetura Java EE 6 a seguir:



No diagrama anterior, podemos observar quatro camadas bem definidas: a Cliente, a Web, a Business e a EIS.

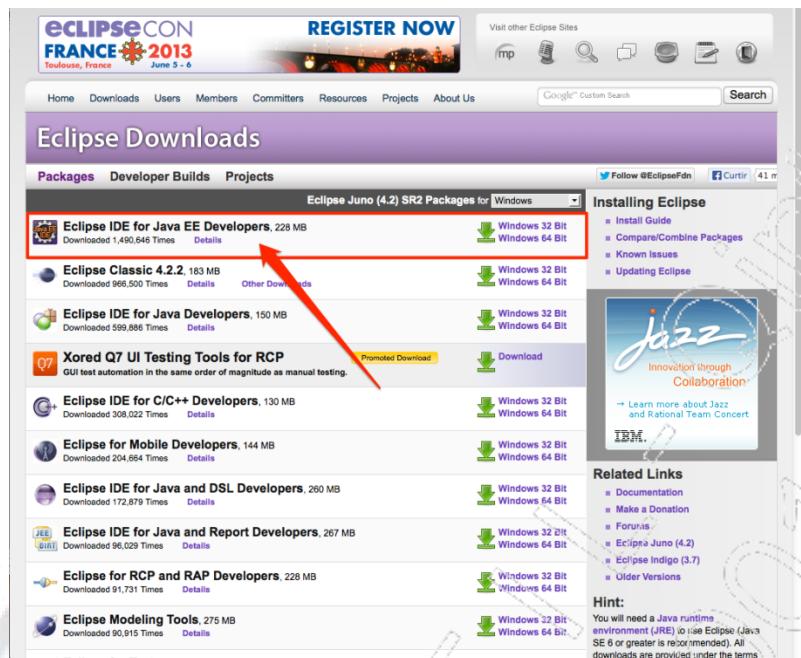
Cabe notar que as camadas são independentes e podem alocar-se de diversas formas, inclusive opcionais. Para o já mencionado Web Profile, utilizado para aplicações Web em Java, o desenho das camadas se reajustaria, de forma que a camada Cliente seria formada pelo browser (navegador) do usuário e não existiria a camada Java EE Business. Nessa situação, toda a lógica de negócio é manipulada por objetos Java simples.

# Montando o ambiente de desenvolvimento

Para montagem do ambiente de desenvolvimento Java Web, são necessários os seguintes itens:

- Um JDK (Java Development Kit), versão 1.6 ou superior, preferencialmente versão 1.7, update mais recente;
- Uma IDE (Ambiente Integrado de Desenvolvimento). Usaremos o Eclipse Juno;
- Um servidor que atenda às especificações de Java Web Container. Usaremos o Apache Tomcat 7.

Considerando que você já conhece a instalação do JDK e do IDE Eclipse (que será usado por padrão), é suficiente lembrar que, para o ambiente Web, você deve efetuar o download do Eclipse na versão Juno (a mais atual), modalidade Eclipse IDE for Java EE Developers, disponível em <http://www.eclipse.org/downloads/> (Acesso em 30 ago. 2013):



Como forma de se manter atualizado, dê preferência a versões do Eclipse iguais ou superiores à versão 3.8 (Juno) e sempre na versão Java EE. Os plug-ins provindos desta versão do eclipse são adequados ao desenvolvimento web.

O último componente necessário à implantação do ambiente de desenvolvimento completo é o Apache Tomcat, em sua versão 7 ou superior, tendo em vista a compatibilidade com a plataforma Java EE 6, adotada como padrão.

O processo de instalação do Apache Tomcat pode ser visto com detalhes no Apêndice 1 desta apostila.

# Requisições e respostas HTTP

O mecanismo de requisições e respostas HTTP é o cerne da compreensão das aplicações Web. Toda a interação do usuário com o aplicativo e vice-versa é feita por meio desse mecanismo. Diversas tecnologias existem tanto no lado do cliente (navegador) quanto no lado do servidor (Web Container) de forma a acrescentar funcionalidades e facilitar o controle de todo o processo ao desenvolvedor, mas, fundamentalmente, as requisições e respostas Web são o mecanismo básico de todas elas. O protocolo HTTP é o conjunto de regras e formatos sob os quais devem ser enviadas e recebidas as requisições. Dentro desse protocolo, existem diversos métodos para o envio de requisições, de acordo com a necessidade do cliente.

De todos os métodos HTTP, indicados adiante, os mais importantes e mais usados em todas as aplicações são o GET e o POST. Recentemente com a grande expansão do uso de Web Services do tipo REST, outros métodos tornaram-se populares como o PUT e o DELETE. Como Web Services estão além do escopo deste curso, focaremos nos dois primeiros.

A seguir, temos uma breve descrição dos métodos HTTP:

- **GET:** Usado para solicitar algum recurso como uma página ou um componente de servidor identificado por meio da URL. Utilizado para passar parâmetros via URL ao servidor de forma explícita. Veja um exemplo:  
`<http://www.impacta.com.br/JavaWeb?Nome=Antonio&Idade=23>;`
- **POST:** Possui a principal função de enviar dados ao servidor, de um formulário, por exemplo. Nesse método, os dados são passados no corpo da requisição e não na URL, como no caso do GET, provendo maior segurança no trâmite de informações;
- **HEAD:** Variação do GET usado para obter metadados, informações por meio do header (cabecalho) da resposta, sem necessidade de recuperar o conteúdo de retorno;
- **PUT:** Envio de algum recurso. Pouco usado no protocolo HTTP. É tradicionalmente usado com FTP;
- **DELETE:** É utilizado para excluir um recurso qualquer;
- **TRACE:** Utilizado em grande parte para debug de forma a obter um eco do pedido e para analisar o que está sendo alterado no pedido pelos servidores intermediários na rota até o recurso pedido;
- **OPTIONS:** Obtém os métodos HTTP aceitos pelo servidor;
- **CONNECT:** Usado para tunelamento SSL, geralmente realizado por um proxy com a finalidade de utilizar conexão segura.

Inicialmente, a solicitação por uma página ou clique feito pelo usuário em uma página Web já aberta gera uma requisição a um servidor Web que possui o seguinte formato simplificado para o método GET:

- Header de requisição GET:

```
GET / HTTP/1.1
Host: www.impacta.com.br
Connection: close
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; de-de)
AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4
Safari/523.10
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-br,en-us;q=0.8,pt;q=0.5,en;q=0.3
Referer:
```

Ao receber essa requisição, o servidor retorna com a resposta, que possui o seguinte cabeçalho exemplificativo:

- Header de resposta do servidor:

```
Status: HTTP/1.1 200 OK
Via: 1.1 MONDEO
Connection: close
Proxy-Connection: close
Content-Length: 47944
Expires: Tue, 07 May 2013 15:35:51 GMT
Date: Mon, 06 May 2013 18:35:51 GMT
Content-type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-Powered-By: PHP/5.3.2
Set-Cookie: PHPSESSID=9b0a1fb88a06355f1fcfdeb2a5ad3589; path=/
Cache-Control: no-store, no-cache, must-revalidate, post-
check=0, pre-check=0
Pragma: no-cache
...
<Conteúdo da página a ser exibido no navegador>
```

Além do exemplo usado anteriormente, outro método de grande importância e que merece tanto destaque quanto o GET é o método POST, cuja requisição está delineada a seguir:

- Header de requisição POST:

```
POST / HTTP/1.1
Host: www.impacta.com.br
Connection: close
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; de-de)
AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4
Safari/523.10
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-br,en-us;q=0.8,pt;q=0.5,en;q=0.3
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Cache-Control: no-cache
Referer:
Content-type: application/x-www-form-urlencoded
Content-length: 0
```

- Header de resposta do servidor:

```
Status: HTTP/1.1 200 OK
Via: 1.1 MONDEO
Connection: close
Proxy-Connection: close
Content-Length: 47944
Expires: Tue, 07 May 2013 15:41:20 GMT
Date: Mon, 06 May 2013 15:41:20 GMT
Content-type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-Powered-By: PHP/5.3.2
Set-Cookie: PHPSESSID=098fc32dd99060413c348d57c456ea29; path=/
Cache-Control: no-store, no-cache, must-revalidate, post-
check=0, pre-check=0
Pragma: no-cache
...
<Conteúdo da página a ser exibido no navegador>
```

Perceba que as respostas são praticamente idênticas, o que altera realmente é a forma de se fazer a requisição.

# Servidores e containers

Teste seus conhecimentos



**1. Quais componentes certamente encontramos em um servidor de aplicações Java EE?**

- a) Web Container e navegador Web.
- b) EJB Container e servidor de impressão.
- c) Web Container e EJB Container.
- d) Servidor HTTP e módulo de segurança dedicado.
- e) Nenhuma das alternativas anteriores está correta.

**2. Sob a ótica de um profissional atuante na área de infraestrutura de TI, como o conceito de servidor pode ser definido?**

- a) Conjunto de equipamentos que compõem o conjunto de hardware e software utilitários, geralmente sediados em um data center, e que fornecem acesso a serviços como arquivos e rede.
- b) Conjunto de hardwares aplicados ao atendimento de demandas de usuários de rede, geralmente de alta capacidade.
- c) Componente de software utilizado para atender requisições HTTP e pedidos de clientes de rede em qualquer protocolo.
- d) Conjunto de softwares, geralmente sediados em um datacenter, utilizados para sustentar a demanda de serviços como impressão e e-mail de uma empresa.
- e) Nenhuma das alternativas anteriores está correta.

**3. Qual é a principal diferença entre um servidor considerado Full Profile e um denominado Web Profile, na plataforma Java EE 6?**

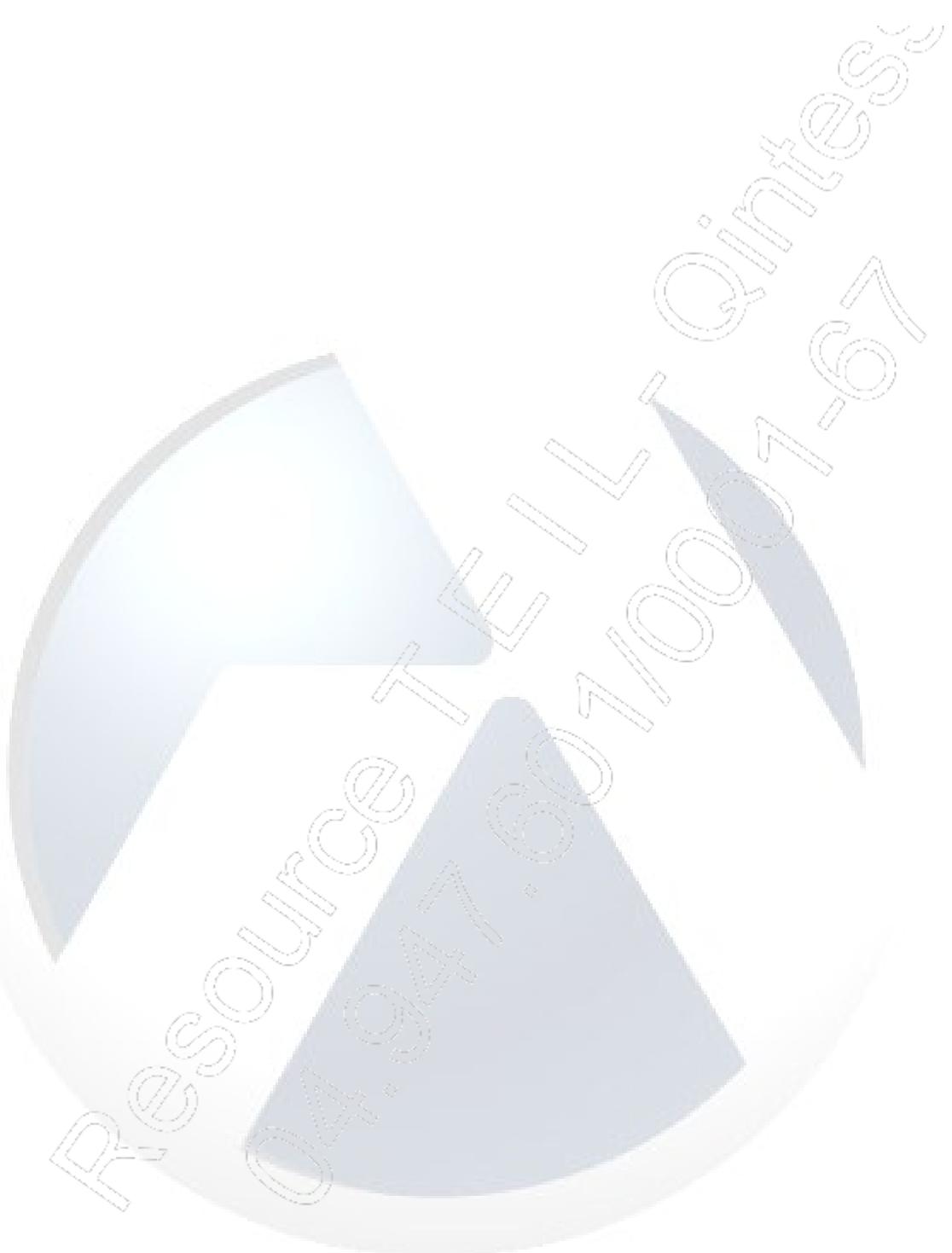
- a) A existência de uma camada extra para atendimento de requisições HTTP.
- b) O servidor Web Profile contém apenas o EJB Container previsto na especificação para servidores de aplicação da plataforma Java EE 6.
- c) O servidor considerado Full Profile contém três containers para lidar com aplicações, uma a mais que o denominado Web Profile.
- d) O servidor Web Profile não está preparado para atender requisições de rede.
- e) Nenhuma das alternativas anteriores está correta.

**4. Dos métodos HTTP existentes para comunicação na Web usando esse protocolo, os mais importantes e usados em aplicações são:**

- a) GET e OPTIONS.
- b) PUT e DELETE.
- c) GET e PUT.
- d) GET e POST.
- e) POST e TRACE.

**5. Qual é a principal característica do método POST que o difere do método GET em relação a requisições HTTP?**

- a) Nenhuma, tanto o método GET quanto o POST podem ser usados para fazer requisições.
- b) O método POST passa os parâmetros no corpo da requisição, ocultando os valores enviados pelos formulários, já o GET os envia diretamente na URL.
- c) O método GET passa os parâmetros no corpo da requisição, ocultando os valores enviados pelos formulários, já o POST os envia diretamente na URL.
- d) O método GET provê criptografia no envio dos dados, já o método POST não.
- e) Nenhuma das alternativas anteriores está correta.



# Servidores e containers

Mãos à obra!

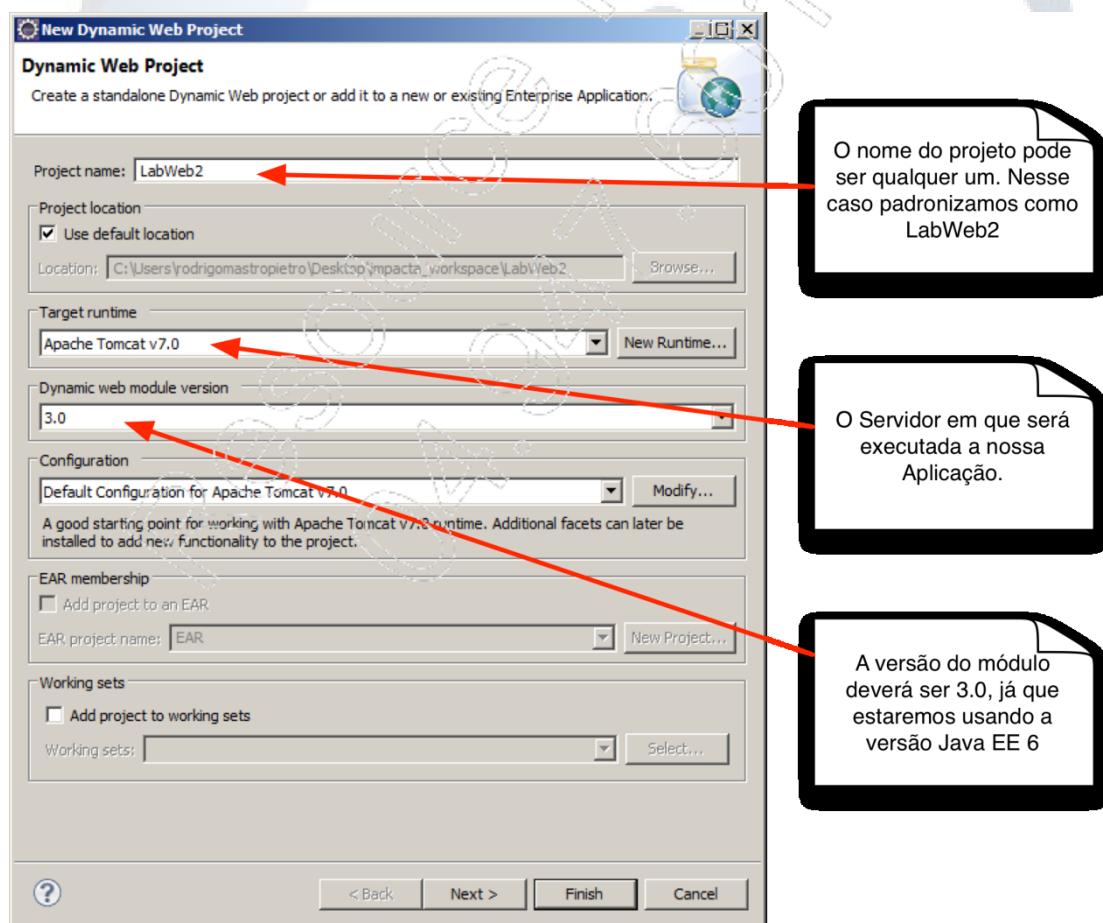
## Laboratório 1

### A - Criando a primeira aplicação Web com Java EE

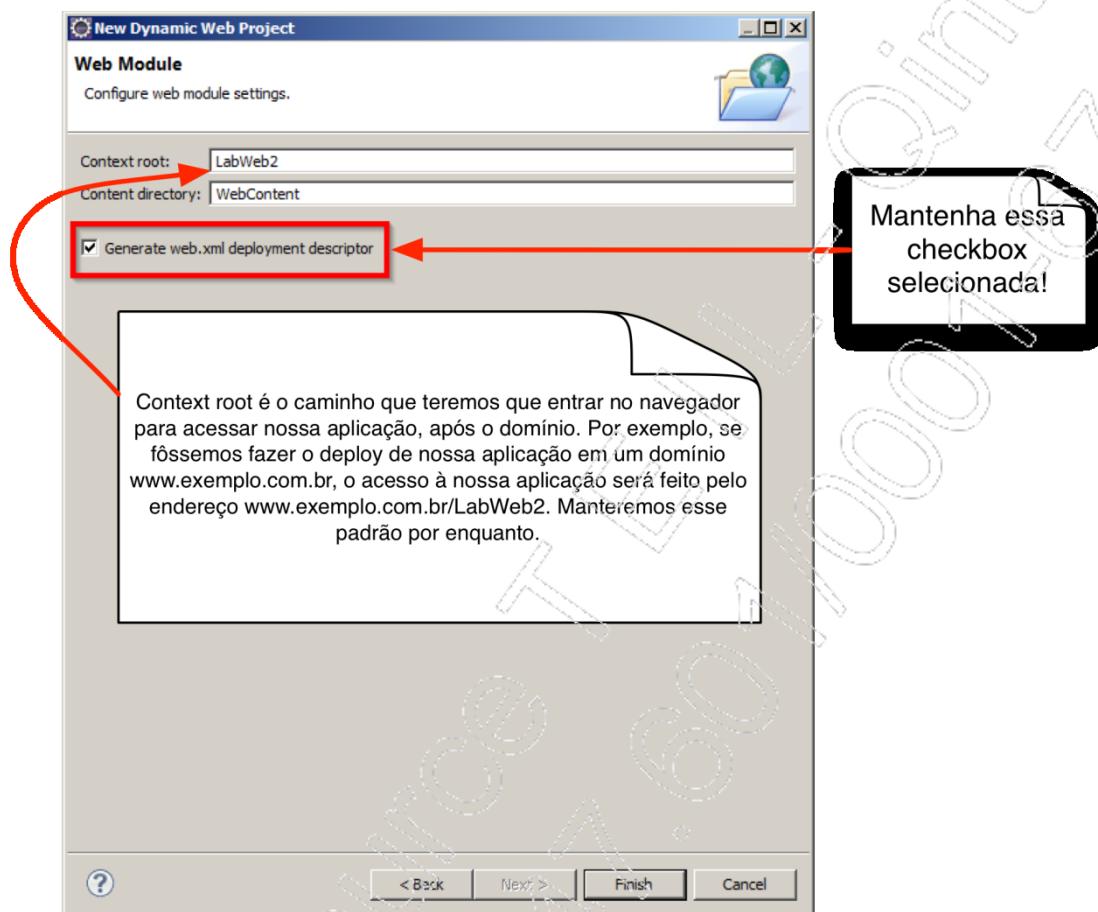
1. Utilizando o Eclipse, assegure-se de estar utilizando a perspectiva Java EE e garanta que o servidor Apache Tomcat 7 esteja instalado na máquina e configurado para utilização no Eclipse. Caso não esteja, consulte o Apêndice 1 para realizar a instalação e as configurações necessárias desse servidor;

2. Crie um novo projeto Web no Eclipse, clicando em **File / New / Dynamic Web Project**;

3. Na tela seguinte, uma série de informações deverá ser inserida. Inicialmente, dê o nome de **LabWeb2** para o projeto e siga as demais informações conforme as instruções a seguir:



4. Clique em **Next**. Na próxima tela, clique em **Next** novamente. A próxima tela é de grande importância para a aplicação. Siga as instruções indicadas nesta imagem:



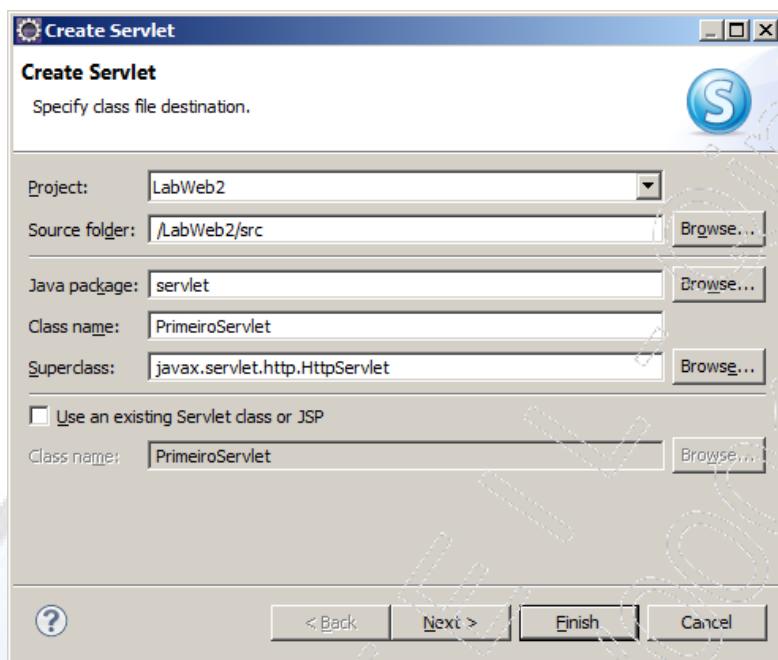
5. Terminados os passos descritos, clique em **Finish** e o projeto será criado;

6. Crie um novo Servlet dentro da aplicação. Para isso, clique com o botão direito do mouse sobre a pasta **src**, localizada dentro da pasta **Java Resources**, e escolha **New / Servlet**;

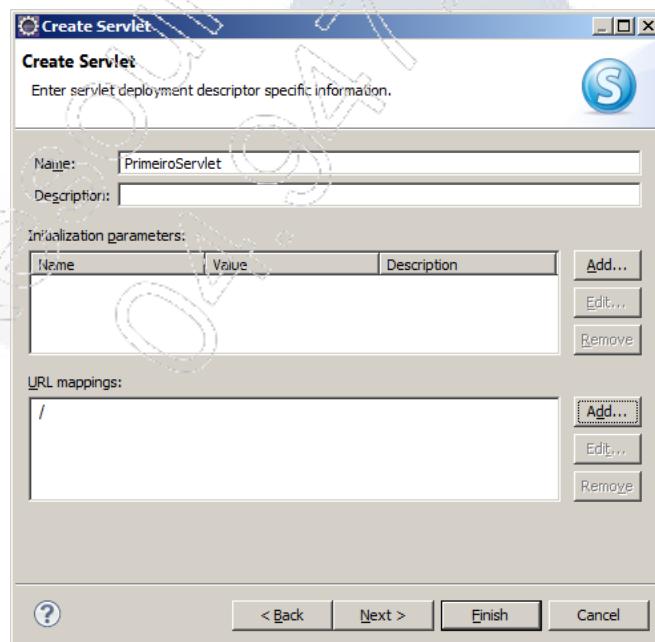
7. Insira os dados para o novo Servlet:

- Em **Java package**, insira o nome **servlet**;
- Em **Class name**, defina o nome **PrimeiroServlet**.

Sua tela deverá ficar desta forma:



8. Na tela anterior, clique em **Next**. Então, na tela seguinte, altere apenas o campo **URL mappings**, selecionando a linha existente e clicando em **Edit...**. Insira apenas o valor **/**, conforme a imagem a seguir. Depois, clique em **Finish**;



9. Com o Servlet criado no Eclipse, ele preenche o editor com o código-fonte modelo para a edição de um Servlet funcional. Nessa classe, vamos nos atentar apenas para um método em especial: o método **doGet(...)**:

```
1 package servlet;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.annotation.WebServlet;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 /**
11  * Servlet implementation class PrimeiroServlet
12 */
13 @WebServlet("/")
14 public class PrimeiroServlet extends HttpServlet {
15     private static final long serialVersionUID = 1L;
16
17 /**
18  * @see HttpServlet#HttpServlet()
19 */
20 public PrimeiroServlet() {
21     super();
22     // TODO Auto-generated constructor stub
23 }
24
25 /**
26  * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
27 */
28 protected void doGet(HttpServletRequest request, HttpServletResponse response)
29         throws ServletException, IOException {
30     // TODO Auto-generated method stub
31 }
32
33 /**
34  * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
35 */
36 protected void doPost(HttpServletRequest request, HttpServletResponse response)
37         throws ServletException, IOException {
38     // TODO Auto-generated method stub
39 }
40
41 }
42 }
```

Observe que esse modelo é cheio de comentários javadoc (`/** ... */`) e comentários de uma linha (`/*...*/`). Se preferir, apague-os para facilitar a visão da estrutura da classe que representa o Servlet criado.

10. Dentro do corpo do método **doGet(...)**, insira o seguinte trecho de código:

```
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>LabWeb2 - Java Web</title></head>");
out.println("<body>");
out.println("<h1>Minha Primeira Aplicacao Web!</h1>");
out.println("</body>");
out.println("</html>");
```

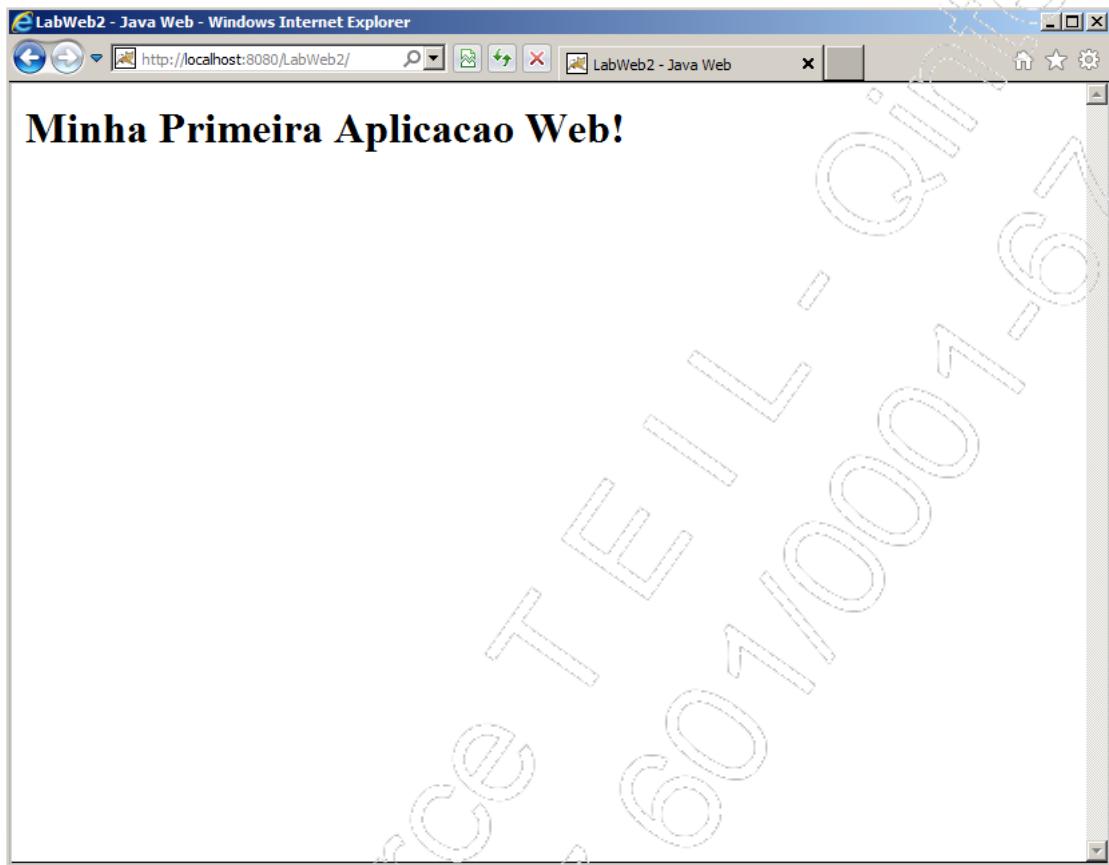
11. Não se esqueça de realizar o import da classe `PrintWriter` no topo de seu arquivo:

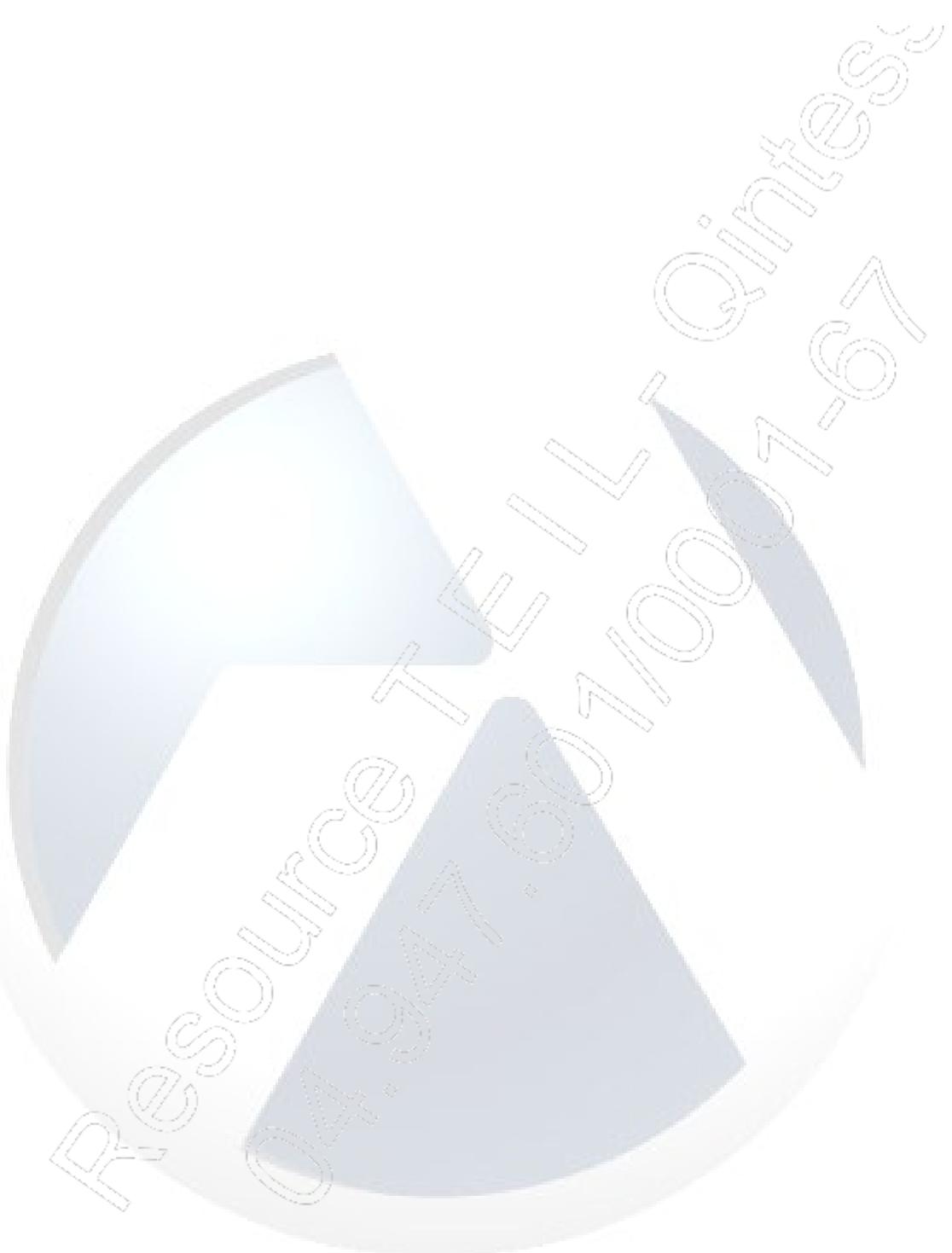
```
import java.io.PrintWriter;
```

12. Execute sua aplicação e teste-a para verificar o código criado em funcionamento. Inicie o Apache Tomcat no Eclipse e certifique-se de que ele não esteja apresentando problemas. Clique com o botão direito do mouse sobre a aplicação e selecione **Run As / Run on Server**. O Eclipse deverá apresentar algumas caixas de diálogo em seguida. Clique em **OK** em todas elas. Se for solicitado reiniciar o servidor, confirme.

13. Será aberta a janela do navegador interno do Eclipse com a URL já acessada: <http://localhost:8080/LabWeb2/> e sua aplicação exibindo a frase: **Minha Primeira Aplicacao Web!**.

14. Faça o teste em um navegador externo. Copie a URL e cole em um navegador externo para obter o mesmo resultado:





# **Capítulo 3: Conceitos básicos sobre Servlets**

# O que são Servlets?

Desde o início de nossa discussão sobre aplicações Web, temos mencionado os Servlets como componentes Java, sem muitos detalhes. Afinal, o que são Servlets?

Servlets são classes Java usadas para responder às necessidades de servidores que hospedam aplicações acessadas por meio de um modelo de Requisição e Resposta. Apesar de serem genéricos o bastante para atenderem diversos protocolos, nosso foco nos direcionará aos servlets especificamente modelados para o protocolo HTTP.

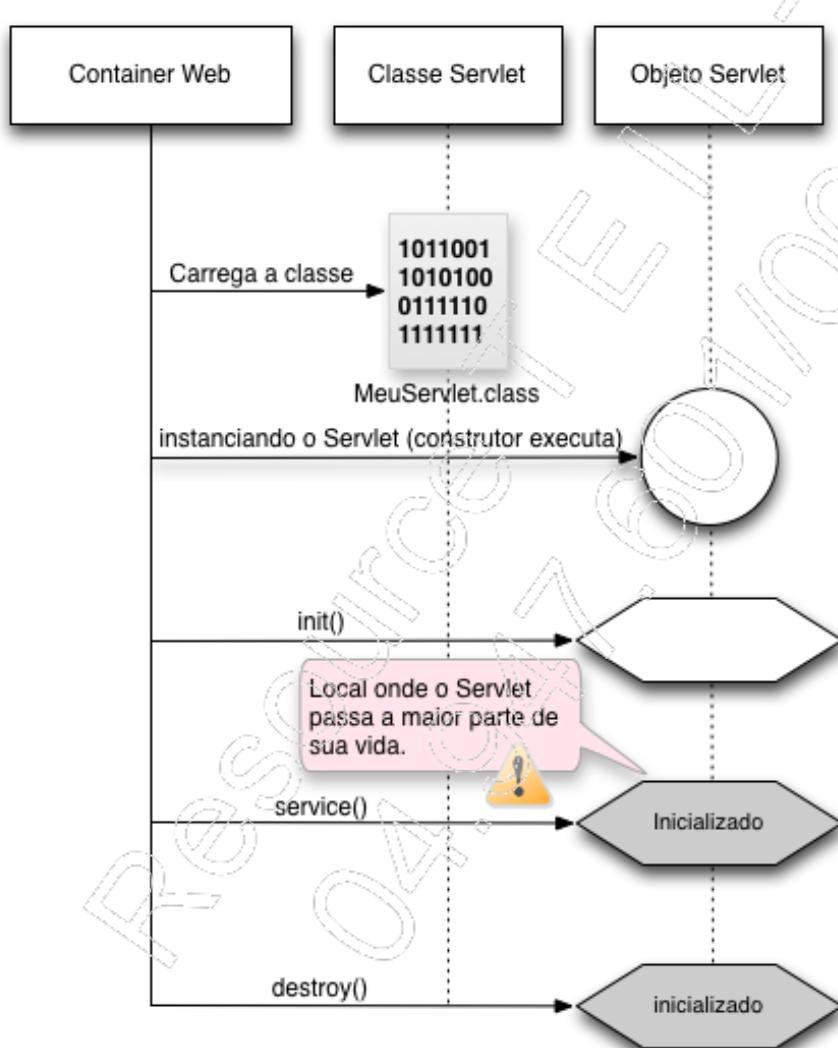
Para a construção de um Servlet, basta criar uma nova classe Java que herde da classe **HttpServlet**. Essa nova classe deverá sobrescrever ao menos um dos métodos Java herdados, correspondentes aos métodos HTTP, conforme a necessidade da aplicação. Em quase todos os casos, esses métodos serão reduzidos ao GET e POST. Os métodos adequados possuem as assinaturas:

- `void doGet(HttpServletRequest req, HttpServletResponse resp);`
- `void doPost(HttpServletRequest req, HttpServletResponse resp).`

Servlets são a forma mais básica de criação de páginas dinâmicas para a Web e estão presentes desde os primeiros passos da tecnologia Java no contexto Web, em meados de 1997. Antigamente, a solução preferida para a criação de páginas dinâmicas na Web eram os chamados CGI (Common Gateway Interfaces), que permitiam a criação de pequenos programas voltados a páginas dinâmicas com linguagens como Perl, C, C++, entre outras. Com a grande limitação dos CGI e a grande expansão da Web, os Servlets acabaram se tornando uma solução viável e, posteriormente, indispensável em certos casos.

# O ciclo de vida de um Servlet

O ciclo de vida do Servlet é bem simples de ser compreendido. Existe apenas um estado principal: inicializado. Se o Servlet não está inicializado, então ou ele está sendo inicializado (com seu construtor ou método **init()** em execução), destruído (com seu método **destroy()** em execução) ou ele simplesmente não existe.



Uma importante observação a ser feita, à qual o desenvolvedor deve estar constantemente atento, é que o Container cria somente uma instância de cada Servlet declarado. Para cada requisição, é criada uma nova thread sobre a mesma instância do Servlet.

Logo, Servlets não constituem um bom lugar para se guardar dados referentes a usuários em particular, dado que a instância é compartilhada entre as requisições. Além disso, não há como se obter uma referência do servlet. Seu controle reside apenas no container.

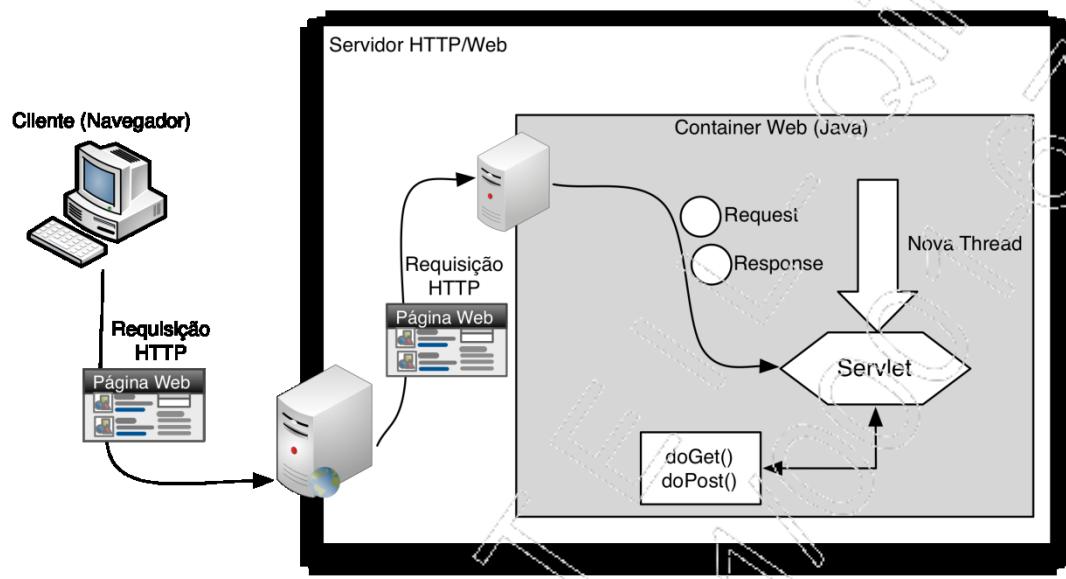
Para a perfeita compreensão do mecanismo de um Servlet, torna-se necessário antes compreender como o Java Web Container lida com a sistemática de requisições e respostas HTTP.

## Requisições e respostas no Java Web Container

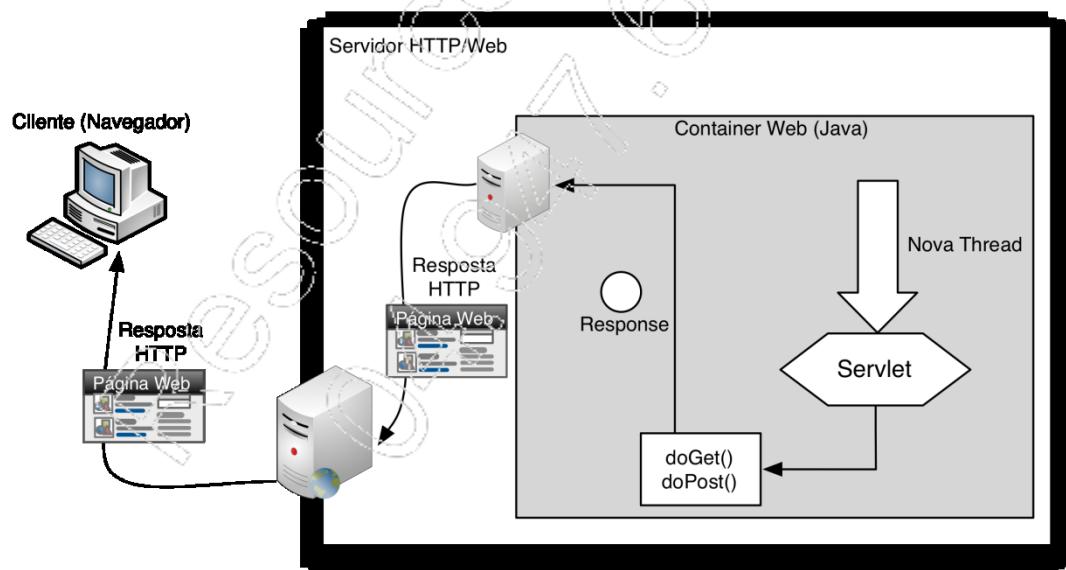
Em um servidor Java EE, o mecanismo de tratamento das requisições e respostas HTTP passa por um processo inicial de montagem, identificação e direcionamento pelo Web Container para, somente depois, repassar o controle para o desenvolvedor, via Servlet ou JSP, que fará o trabalho necessário para entender o pedido e montar a resposta ao cliente.

Basicamente, a sequência adiante ilustra o mecanismo realizado pelo Servidor Web e Java Web Container :

- Requisição Web:



- Resposta Web:



No processo de requisição e resposta Web, em um servidor Java, é possível delinear as seguintes operações realizadas pelo Container:

- O início se dá com o pedido de uma página ou recurso via URL pelo cliente ou por meio do clique em um botão ou link em uma página já disponível ao usuário;
- O navegador então cria a requisição com o pedido do usuário juntamente com os dados passados por ele e os encaminha ao Servidor HTTP (um exemplo é o Apache Web Server, um dos mais utilizados do mundo);
- O servidor HTTP verifica se o recurso solicitado é uma página estática ou um recurso dinâmico. No primeiro caso, obtém a página e a devolve ao usuário, sem intervenção do Java. No segundo caso, tratando-se de um pedido a um recurso Java, ele encaminha a requisição ao Java Web Container para que ele trate a requisição;
- Recebida a solicitação do servidor Web, o Java Web Container localiza o Servlet ou JSP registrado para lidar com a requisição encaminhada e prepara-se para enviar o pedido e os dados recebidos ao respectivo recurso;
- Nessa preparação, o Java Web Container cria dois objetos do tipo **HttpServletRequest** e **HttpServletResponse** que representam, respectivamente, a Requisição recebida e a Resposta a ser enviada ao cliente no retorno;
- Os objetos criados são passados como parâmetros para o método adequado no Servlet (classe que herda de **HttpServlet**), método (Java) esse que representa o recurso HTTP equivalente, usado para fazer a requisição ao servidor. Ao chamar o respectivo método no Servlet, o Container cria uma thread separada para cada requisição (imaginando um cenário em que diversos usuários acessam o mesmo endereço URL ao mesmo tempo);

- É neste ponto que o desenvolvedor entra em ação: o código do método é executado e ele deve retornar uma página HTML ao usuário ou repassar a chamada para algum outro objeto que retorne um HTML válido ao usuário;
- Ao término desse processo, o Container constrói a página HTML e a resposta HTTP ao usuário, retornando-a ao servidor Web para posterior encaminhamento ao usuário final;
- Com uma nova requisição sendo feita pelo usuário, seja por um clique ou por uma nova URL digitada, o processo todo se repete.

## Mapeando Servlets no web.xml e anotações na versão Servlet 3.0

Em toda aplicação Java Web, algumas configurações são necessárias para instruir o Web Container sobre como comportar-se em situações específicas, ou ainda para declarar a existência de um certo componente ou definir seus atributos, além de outras diversas aplicações que serão apresentadas mais adiante. Um arquivo especial é utilizado para a implementação dessas configurações: o **web.xml**. Esse arquivo é conhecido no meio Java como **Deployment Descriptor (Descriptor de Implantação)** ou simplesmente DD, e está presente na grande maioria das aplicações Java Web.

Desde a versão Java EE 6, lançada no final de 2009, esse arquivo é opcional e as configurações que ele implementa podem ser aplicadas por meio de anotações às classes Java que representem os componentes e recursos utilizados pela aplicação.

No mercado, há preferência pelo emprego do arquivo xml em detrimento do uso das anotações. Contudo, as duas formas apresentadas serão abordadas adiante. Na verdade, é possível utilizar ambos os métodos simultaneamente, porém, o arquivo **web.xml** tem prioridade na configuração.

Vejamos um exemplo integral de um típico descritor **web.xml** e onde um Servlet se encaixa em sua declaração:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
7     id="WebApp_ID" version="3.0">
8   <display-name>ProgramaWeb</display-name>
9   <welcome-file-list>
10    <welcome-file>index.html</welcome-file>
11    <welcome-file>index.htm</welcome-file>
12    <welcome-file>index.jsp</welcome-file>
13    <welcome-file>default.html</welcome-file>
14    <welcome-file>default.htm</welcome-file>
15    <welcome-file>default.jsp</welcome-file>
16  </welcome-file-list>
17
18  <servlet>
19    <servlet-name>MeuServlet</servlet-name>
20    <servlet-class>servlet.MeuServlet</servlet-class>
21  </servlet>
22  <servlet-mapping>
23    <servlet-name>MeuServlet</servlet-name>
24    <url-pattern>/meuservlet</url-pattern>
25  </servlet-mapping>
26 </web-app>
```

Basicamente, para cada Servlet que estiver na aplicação deverá existir o seguinte conjunto de tags no DD para descrevê-lo:

```
18  <servlet>
19    <description>Meu primeiro Servlet</description> <!-- Opcional -->
20    <servlet-name>MeuServlet</servlet-name>
21    <servlet-class>servlet.MeuServlet</servlet-class>
22  </servlet>
23  <servlet-mapping>
24    <servlet-name>MeuServlet</servlet-name>
25    <url-pattern>/meuservlet</url-pattern> <!-- Nunca se esqueça da barra! -->
26  </servlet-mapping>
```

Apresentando cada uma das tags do conjunto:

Tag	Descrição
< servlet >	Tag raiz que abre o bloco de declaração para cada Servlet da aplicação.
< description >	Tag opcional que traz uma breve descrição das funcionalidades do Servlet.
< servlet-name >	Nome dado ao Servlet e que serve apenas para referenciá-lo no próprio <b>web.xml</b> . Esse nome será usado como referência para o Servlet na tag < servlet-mapping >.
< servlet-class >	Nome totalmente qualificado da classe que representa o respectivo Servlet. Utiliza a mesma sintaxe empregada para importar classes em Java.
< servlet-mapping >	Tag utilizada para mapear um determinado Servlet a um padrão de URL utilizando, para tanto, a tag < servlet-name > que identifica o servlet específico.
< url-pattern >	Tag usada dentro da anterior para mapear um padrão de URL a ser atendida por um determinado Servlet. A barra tem papel de grande importância nesse contexto.

De forma equivalente, é possível dispensar o uso do arquivo **web.xml** e usar apenas anotações na classe Servlet, de forma a instruir o container acerca da classe a ser controlada e do padrão de URL a ser utilizado. A seguir, temos o exemplo de um Servlet cuja classe está anotada e que dispensa o uso do descriptor de implantação **web.xml**:

```

10 @WebServlet("/meuservlet") ← Perceba a anotação sobre
11 public class MeuServlet extends HttpServlet {
12
13     private static final long serialVersionUID = 1L;
14
15     public MeuServlet() {
16         super();
17     }
18
19     protected void doGet(HttpServletRequest request,
20             HttpServletResponse response) throws ServletException, IOException {
21         //Código para tratar requisição GET
22     }
23
24     protected void doPost(HttpServletRequest request,
25             HttpServletResponse response) throws ServletException, IOException {
26         //Código para tratar requisição POST
27     }
28 }
```

Esse Servlet é, até certo ponto, um objeto regular, sem muita especialização e que fornece implementação para os métodos que tratam de requisições GET e POST. Note a declaração sobre a classe e o parâmetro passado:

```
@WebServlet("/meuservlet")
```

Essa anotação define para o Web Container que o Servlet atual deve ser gerenciado e que seu padrão de URL para atender requisições é "/meuservlet", ou seja, toda requisição feita nessa aplicação, cuja URL atenda ao padrão configurado, será recebida por esse Servlet.

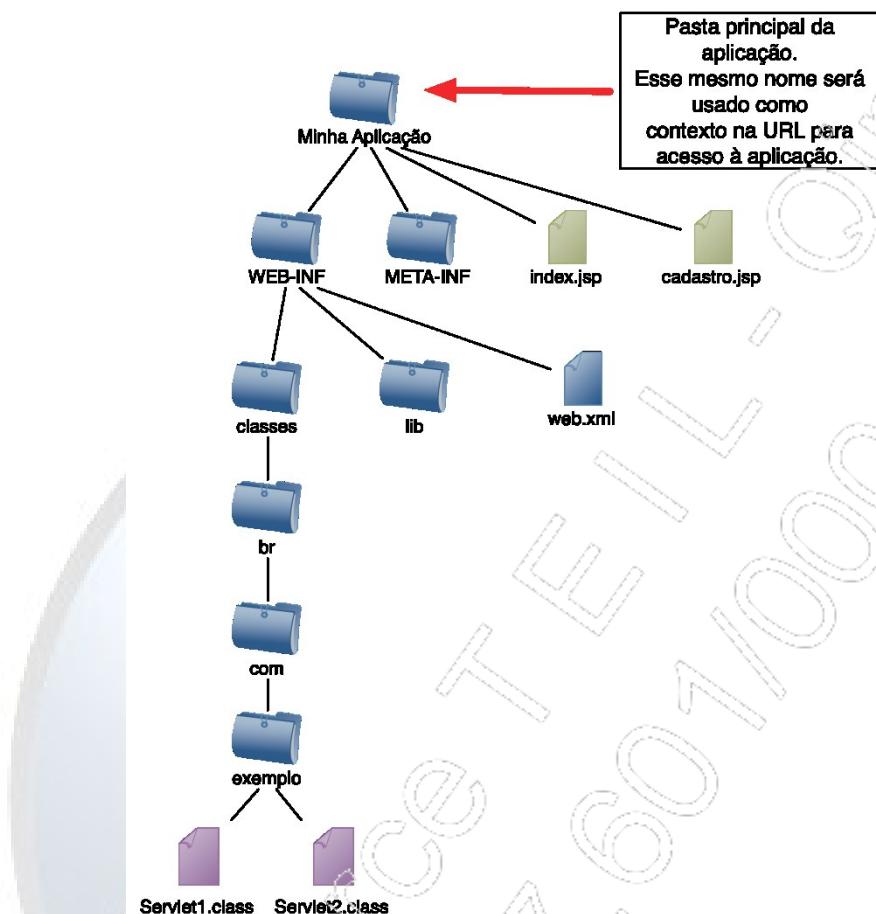
Vale lembrar que ambas as formas podem ser utilizadas, mas que o arquivo XML tem prioridade sobre as Annotations. Esse mecanismo permite que um novo comportamento possa ser atribuído aos componentes da aplicação, sem a necessidade de modificar o código nativo, apenas alterando elementos no arquivo `web.xml`.

# O projeto Web e o arquivo WAR

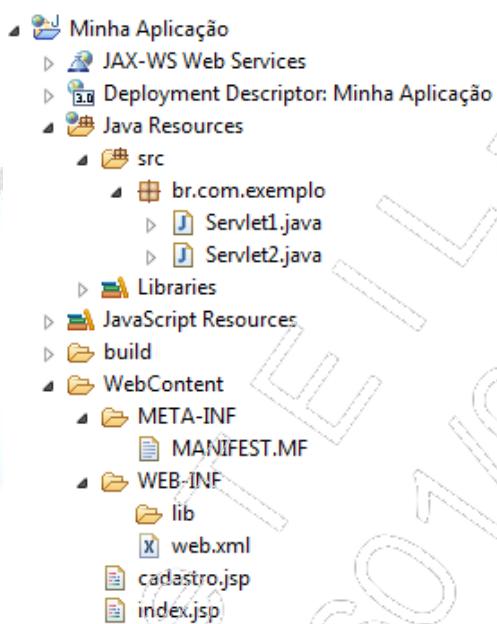
A estrutura de todo projeto Java Web segue um padrão bem definido para deploy nos diversos servidores Java. Esse padrão é, na verdade, a descrição da estrutura de um arquivo compactado, que contém todos os componentes necessários à aplicação, para que possa ser instalada em um Web Container. Denominamos essa estrutura de WAR (Web Archive) e utilizamos a extensão ".war" no arquivo compactado que a representa. Nessa estrutura, podem ser incluídos:

- Servlets, JSP, Tag Libraries, Listeners, Filters;
- Páginas HTML, arquivos CSS e scripts de cliente;
- Deployment Descriptor e outros arquivos de configuração;
- Classes auxiliares e JavaBeans.

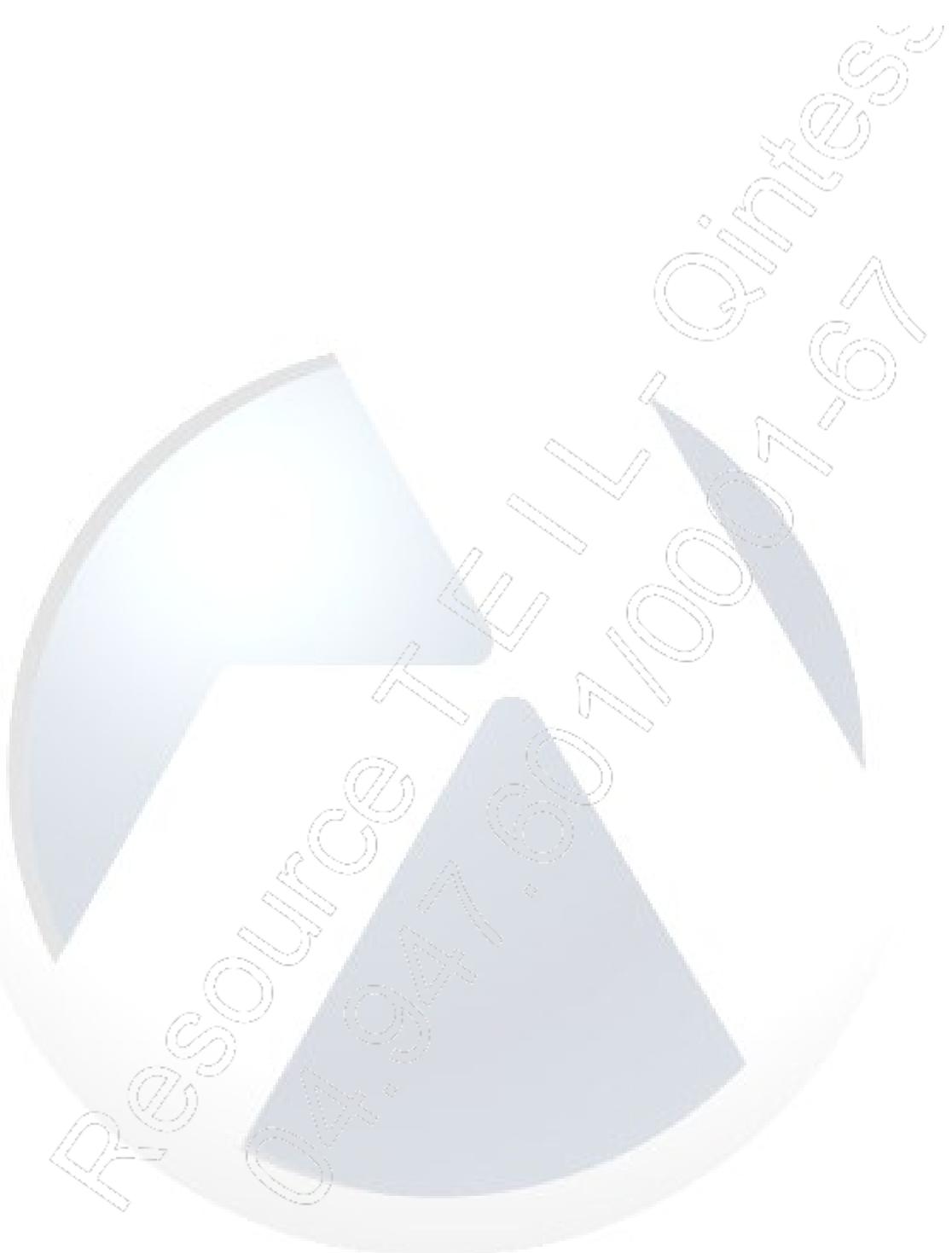
Vejamos a estrutura típica de um arquivo WAR, conforme a especificação Java:



Essa estrutura é a criada dentro de um arquivo .war para deploy ou distribuição de uma suposta aplicação denominada **Minha Aplicação**. Comparando-a com a estrutura criada dentro do Eclipse para cada projeto Web, nota-se a semelhança procurada pela IDE de forma a deixar o desenvolvedor o mais próximo possível do modelo padrão:



Na estrutura do projeto no Eclipse, deve-se considerar a pasta **WebContent** como equivalente à pasta raiz no servidor em que a aplicação será implantada. O conteúdo dessa pasta representa todos os recursos utilizados na aplicação, exceto as classes Java que serão compiladas e executadas durante o runtime do servidor.



# **Conceitos básicos sobre Servlets**

**Teste seus conhecimentos**



**1. Na criação de um Servlet, dois aspectos são fundamentais e norteiam sua utilização no Java Web Container. Quais são eles?**

- a) A implementação de getters e setters apropriados e ao menos um atributo estático que represente seu serialVersionUID, conforme descrito na especificação.
- b) A criação de um construtor sem argumentos e a chamada ao construtor da classe pai como primeira linha a ser executada em sua inicialização.
- c) A implementação adequada dos métodos doGet() e doPost() para atender requisições específicas e a sobrescrição do método equals() de forma a possibilitar a identificação do Servlet pelo Container.
- d) A classe do Servlet deverá herdar da classe HttpServlet e sobrescrever (anular) ao menos um dos métodos relativos à requisição HTTP e equivalente ao método HTTP utilizado. Os mais comuns são o doGet(...) e o doPost(...).
- e) Nenhuma das alternativas anteriores está correta.

**2. No que tange ao seu ciclo de vida, o que ocorre com um Servlet após ser inicializado?**

- a) Passa a maior parte de sua vida útil atendendo requisições de usuários: uma referência do Servlet é passada a cada usuário requisitante durante sua execução.
- b) Passa sua vida útil entre os estados suspenso e ativado, conforme esteja atendendo requisições recebidas ou aguardando no Container para ser utilizado.
- c) Passa a maior parte de sua vida útil atendendo requisições de usuários por meio de threads criadas e alocadas pelo Container.
- d) Pode ser alvo de diversas requisições simultâneas por meio do gerenciamento fornecido pelo Container, fazendo uso de threads distintas para operar sobre uma única instância do Servlet.
- e) As alternativas C e D estão corretas.

**3. Qual(is) das afirmações a seguir está(ão) correta(s) sobre o mecanismo de requisições e respostas executado pelo Java Web Container?**

I - Ao receber a requisição do usuário, o Container trata de obter dois objetos: Request e Response, que residem encapsulados no Header da requisição http.

II - O servidor HTTP é a porta de entrada para todas as requisições HTTP recebidas e é responsável por localizar o recurso estático desejado ou repassar a chamada ao Java Web Container, caso o recurso seja um componente dinâmico.

III - A resposta gerada ao cliente pelo componente Java é dirigida ao servidor HTTP que, por sua vez, encaminha-a ao usuário solicitante.

- a) I
- b) III
- c) II
- d) II e III
- e) Nenhuma das alternativas anteriores está correta.

**4. Qual das alternativas a seguir está correta sobre o Deployment Descriptor?**

- a) A presença do arquivo web.xml é obrigatória em todas as aplicações Java Web.
- b) O arquivo web.xml era obrigatório até antes do lançamento da versão Java EE 6. Nessa versão, foi introduzido o uso de anotações, que tornaram a utilização do web.xml opcional.
- c) A declaração de um Servlet no arquivo web.xml perde a aplicabilidade caso esse servlet possua uma anotação @WebServlet(...).
- d) A declaração de um Servlet no DD é feita com um único bloco de tags: <servlet></servlet>.
- e) O mapeamento de um Servlet é uma tarefa extremamente importante. Caso não seja feito, por padrão, o Container direciona automaticamente todas as requisições a esse Servlet.

**5. O que não é permitido na estrutura interna de um arquivo WAR?**

- a) Classes Java compiladas.
- b) Páginas HTML estáticas.
- c) Arquivos de imagem, som e vídeo.
- d) Servlets, JSP e outros componentes Java.
- e) Nenhuma das alternativas anteriores está correta.

# Conceitos básicos sobre Servlets

Mãos à obra!



## Laboratório 1

### A - Testando padrões de URL via DD e Annotations

1. Crie um novo projeto do tipo **Dynamic Web Project** no Eclipse denominado **LabWeb3**. Não se esqueça de checar a opção **Generate web.xml deployment descriptor**;
2. Examine o projeto criado no Eclipse e navegue nas pastas, buscando identificar a pasta **WebContent** e, dentro dela, a pasta **WEB-INF** contendo o arquivo **web.xml**. Explore também a pasta **Java Resources** e note que este é o local criado pelo Eclipse para armazenar todo o seu código-fonte Java não compilado;
3. Expanda a view denominada **Navigator** no Eclipse, clicando no menu **Window / Show View / Navigator**. Essa view se abrirá ao lado da view **Project Explorer** e representa a árvore e estrutura reais criadas pelo Eclipse no sistema de arquivos. Sempre compare a estrutura dentro do Project Explorer (facilitado para atender às demandas de desenvolvimento) com a estrutura apresentada no Navigator (estrutura real de arquivos);
4. Crie um novo Servlet e especifique como pacote o caminho **br.com.impacta.javaweb**, de nome **ClienteServlet**. Aceite todas as opções padrão, clicando em **Next**, em seguida, em **Next** e depois em **Finish**;

## 5. Analise o código do Servlet criado, que deverá ser semelhante ao seguinte:

```
1 package br.com.impacta.javaweb;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.annotation.WebServlet;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 /**
11  * Servlet implementation class ClienteServlet
12 */
13 @WebServlet("/ClienteServlet")
14 public class ClienteServlet extends HttpServlet {
15     private static final long serialVersionUID = 1L;
16
17 /**
18  * @see HttpServlet#HttpServlet()
19 */
20 public ClienteServlet() {
21     super();
22     // TODO Auto-generated constructor stub
23 }
24
25 /**
26  * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
27 */
28 protected void doGet(HttpServletRequest request, HttpServletResponse response)
29     throws ServletException, IOException {
30     // TODO Auto-generated method stub
31 }
32
33 /**
34  * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
35 */
36 protected void doPost(HttpServletRequest request, HttpServletResponse response)
37     throws ServletException, IOException {
38     // TODO Auto-generated method stub
39 }
40
41 }
```

Note que o Eclipse automaticamente criou a anotação `@WebServlet` (`"/ClienteServlet"`) sobre a classe, o que corresponde à declaração do Servlet e seu mapeamento de URL, que está configurado para `"/ClienteServlet"` neste momento.

6. Substitua o comentário no método **doGet(...)**, gerado por padrão pelo Eclipse, pelo seguinte trecho de código, que imprime a simples frase "**ClienteServlet usando Annotations!**" no navegador do usuário:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>LabWeb3</title></head>");
out.println("<body>");
out.println("<h1>ClienteServlet usando Annotations!</h1>");
out.println("</body>");
out.println("</html>");
```

7. Certifique-se de que o Apache Tomcat não esteja sendo executado (se estiver, interrompa-o) e execute o projeto, aceitando todas as opções padrão que serão exibidas. Note que o servidor automaticamente será iniciado e, ao término do processo, a aplicação já estará implantada e pronta para usar;

8. Abra um navegador e insira a URL <http://localhost:8080/LabWeb3/ClienteServlet>. Você deverá ter uma página vazia no navegador com a saída programada no método **doGet(...)** com a frase "**Cliente Servlet usando Annotations!**". Note que o mapeamento feito no Servlet refere-se ao trecho final da URL, que vem após o Context Root da aplicação (**/LabWeb3**) e, nesse caso, é o mesmo nome do Servlet;

9. Pare o servidor. Altere o mapeamento da URL na anotação à classe para "**/web**". A anotação, depois de alterada, deverá ser semelhante a esta: **@WebServlet("/web")**;

10. Execute a aplicação novamente e insira a URL indicada no passo 8 no navegador. Note que agora surge uma página de erro 404. Experimente mudar a URL para o novo mapeamento <http://localhost:8080/LabWeb3/web>.

Perceba que agora aparece a resposta adequada do Servlet;

11. Deixando o Servlet como está, abra o arquivo **web.xml** e prontamente alterne pela aba inferior do editor para a opção **Source**, para visualização do código XML;

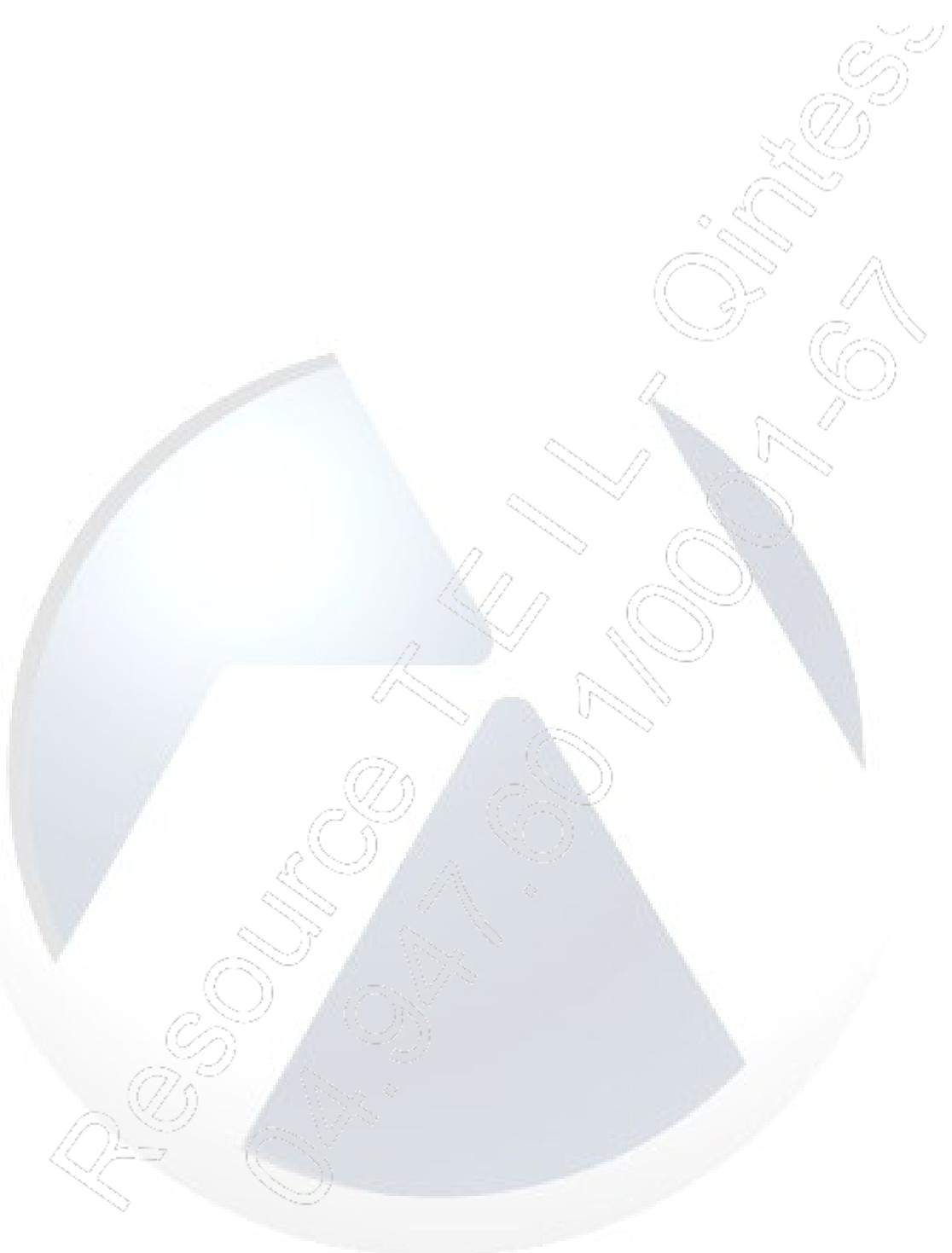
12. Sem alterar o conteúdo já existente do arquivo, insira a configuração para esse Servlet logo após o fechamento da tag "</welcome-file-list>":

```
<servlet>
    <servlet-name>ClienteServlet</servlet-name>
    <servlet-class>br.com.impacta.javaweb.ClienteServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ClienteServlet</servlet-name>
    <url-pattern>/mapeadoviaxml</url-pattern>
</servlet-mapping>
```

13. Antes de executar novamente o programa, interrompa o servidor, comente a linha com a Annotation sobre a classe do Servlet, em seguida, salve e execute a aplicação novamente;

14. Perceba que agora a URL mapeada para "/web" não funciona mais, porém, o mapeamento configurado no web.xml funciona perfeitamente;

15. Remova o comentário colocado na linha referida no passo 13, interrompa o servidor e execute novamente a aplicação. Perceba que tanto o mapeamento feito via Annotation quanto o feito pelo DD funcionam.



# **Capítulo 4:**

# **Comunicação entre**

# **cliente e servidor**

# Introdução

Toda aplicação Web pressupõe a existência de transmissão de dados entre o usuário e o sistema, seja por um período mínimo de tempo ou para armazenamento por diversos anos. Essa troca de informações ocorre por meio das requisições HTTP, que utilizam basicamente os dois métodos HTTP mais conhecidos: o GET e o POST.

Nesse contexto, abordaremos as decorrências e necessidades apresentadas para uma comunicação eficaz entre o cliente e o servidor utilizando os escopos e formas de manutenção de estado disponíveis.

## Enviando e capturando parâmetros na requisição

Toda requisição HTTP feita pelo cliente pode carregar um ou mais parâmetros que serão capturados pelo servidor e aplicados a alguma lógica ou armazenamento e possivelmente um posterior redirecionamento na página do usuário.

### Requisições via GET

Nas requisições utilizando o método GET, os parâmetros são anexados ao final da URL, por meio de uma sintaxe específica. Supondo que a URL utilizada para acesso ao nosso sistema seja <http://www.exemplo.com.br>, temos:

<http://www.exemplo.com.br?id=2&categoria=5>.

Nesse exemplo, são passados dois parâmetros para o servidor diretamente no corpo da URL (típico de uma requisição GET): **id = 2** e **categoria = 5**. Note que a lista de parâmetros é iniciada logo após a URL por meio do caractere "?" e cada parâmetro é separado do outro pelo caractere "&".

Em nossos exemplos, utilizaremos a URL **http://localhost:8080** seguida dos parâmetros, porém, a sintaxe continua a mesma que foi demonstrada anteriormente.

O trecho da URL que contém os parâmetros (a porção iniciada em "?") recebe o nome de **query string**, por ser tão usual aos desenvolvedores Web.

Para análise prática, vamos imaginar uma aplicação Web (Dynamic Web Project) denominada **PassagemdeParametrosGET** e implantada em um servidor Tomcat local. Vamos supor que ela tenha sido criada no Eclipse com um único Servlet com o nome **ParamServlet**, cujo código é o seguinte:

```
1 package servlet;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10
11 public class ParamServlet extends HttpServlet {
12     private static final long serialVersionUID = 1L;
13
14     public ParamServlet() {
15         super();
16     }
17
18     protected void doGet(HttpServletRequest request,
19             HttpServletResponse response) throws ServletException, IOException {
20         response.setContentType("text/html");
21         PrintWriter out = response.getWriter();
22         out.println("<html>");
23         out.println("<head><title>Passagem de Parâmetros</title></head>");
24         out.println("<body>");
25         out.println("<h1>Parâmetros passados:</h1>");
26         out.println("<h3>id: " + request.getParameter("id") + "</h3>");
27         out.println("<h3>categoria: " + request.getParameter("categoria") + "</h3>");
28         out.println("</body>");
29         out.println("</html>");
30     }
31 }
```

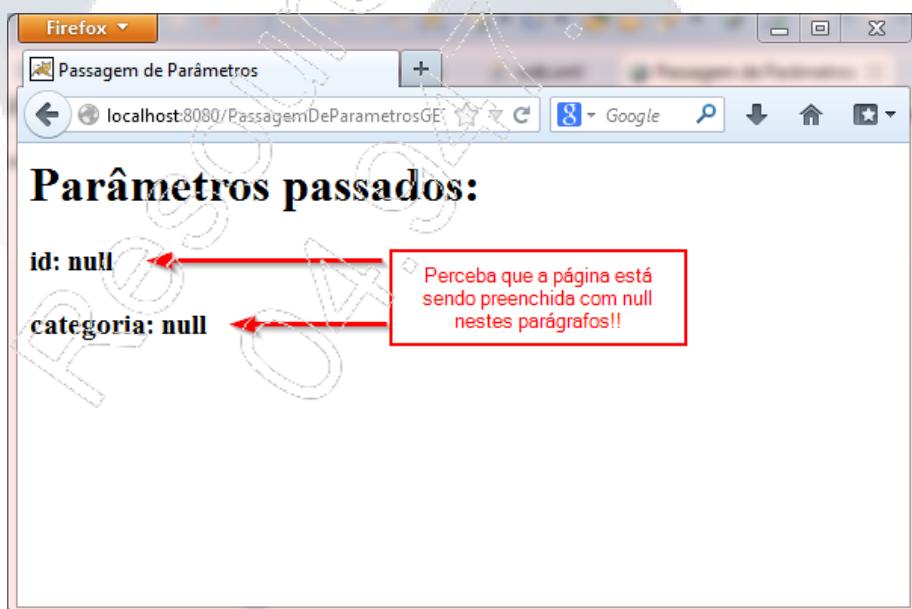
Ao inspecionarmos o DD dessa aplicação, notamos o seguinte trecho de configuração e mapeamento feito para esse Servlet:

```
13<!--<servlet>
14    <servlet-name>Param Servlet</servlet-name>
15    <servlet-class>servlet.ParamServlet</servlet-class>
16 </servlet>
17<!--<servlet-mapping>
18    <servlet-name>Param Servlet</servlet-name>
19    <url-pattern>/ParamServlet</url-pattern>
20 </servlet-mapping>
```

Logo, a URL para efetuarmos uma requisição GET nesse Servlet é composta pelo Context Root da aplicação - supondo que o Context Root seja o próprio nome do Projeto no Eclipse - mais o mapeamento para o qual o Servlet está constrito na declaração:

<http://localhost:8080/PassagemDeParametrosGET/ParamServlet>

Ao receber essa requisição, o Container automaticamente passará a execução para o método **doGet(...)** do Servlet, que retornará uma página ao usuário com o seguinte conteúdo:



Perceba que a palavra **null** aparece exatamente no ponto da página em que o Servlet adiciona a chamada:

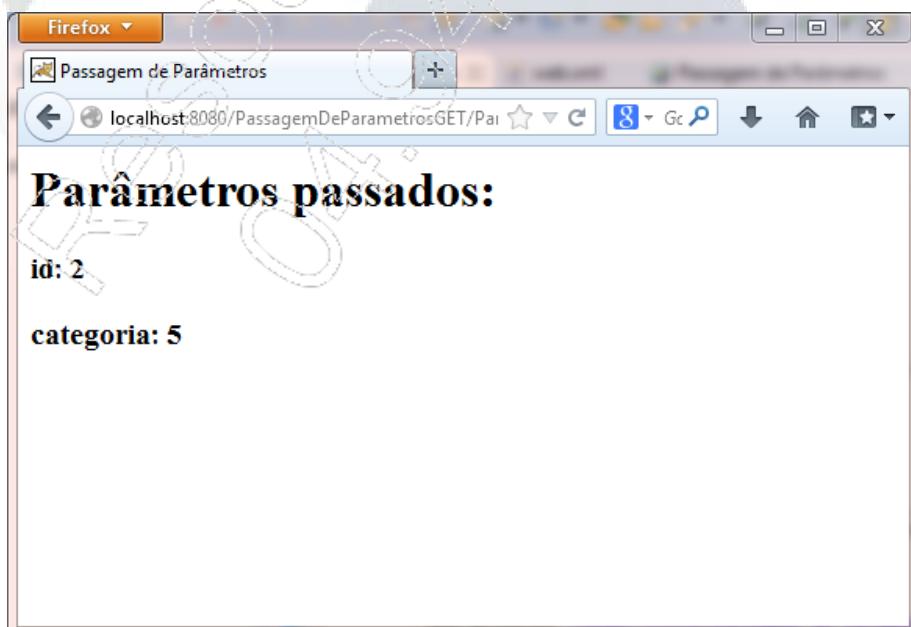
```
out.println("<h3>id: " + request.getParameter("id") + "</h3>");  
out.println("<h3>categoria: " + request.getParameter("categoria") + "</h3>");
```

O que ocorre é que o Servlet tenta adicionar à página os parâmetros cujas chaves são **"id"** e **"categoria"**, porém, nenhum parâmetro está sendo passado. Dessa forma, os parâmetros que nativamente são tratados como strings acabam recebendo o valor default para objetos: **null**, incluídos na página de resposta e enviados ao usuário.

Façamos agora a chamada de forma adequada, passando os parâmetros esperados na URL, ou seja, inserindo a query string esperada pelo Servlet:

**http://localhost:8080/PassagemDeParametrosGET/ParamServlet?id=2&categoria=5**

Enviando essa requisição, é possível notar que agora o Servlet pôde capturar os parâmetros passados pela requisição e reinseri-los em uma página para retornar ao usuário:



## Requisições via POST

As requisições que utilizam o método POST são as mais comuns de se encontrar nas páginas em que se expõe um formulário para preenchimento pelo usuário. Geralmente, os dados passados por POST requerem um sigilo maior e, por esse motivo, não podem trafegar abertamente na URL como é feito com o método GET.

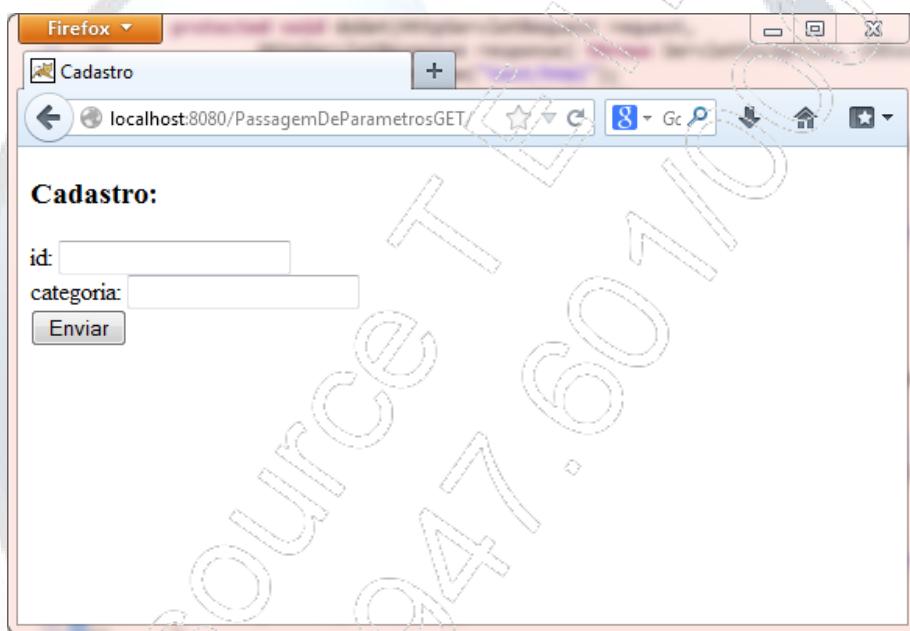
Nas requisições feitas via POST, os dados vêm anexados ao corpo da requisição, indisponível via URL. As requisições via POST vêm definidas em geral pelo par de tags `<form>...</form>` nos formulários HTML. Vejamos um exemplo de formulário a seguir para passar os mesmos parâmetros passados anteriormente por GET:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Cadastro</title>
5 </head>
6 <body>
7     <h3>Cadastro:</h3>
8     <form method="post" action="ParamServlet">
9         id: <input type="text" name="id" /><br />
10        categoria: <input type="text" name="categoria" /> <br />
11        <input type="submit" value="Enviar" />
12     </form>
13 </body>
14 </html>
```

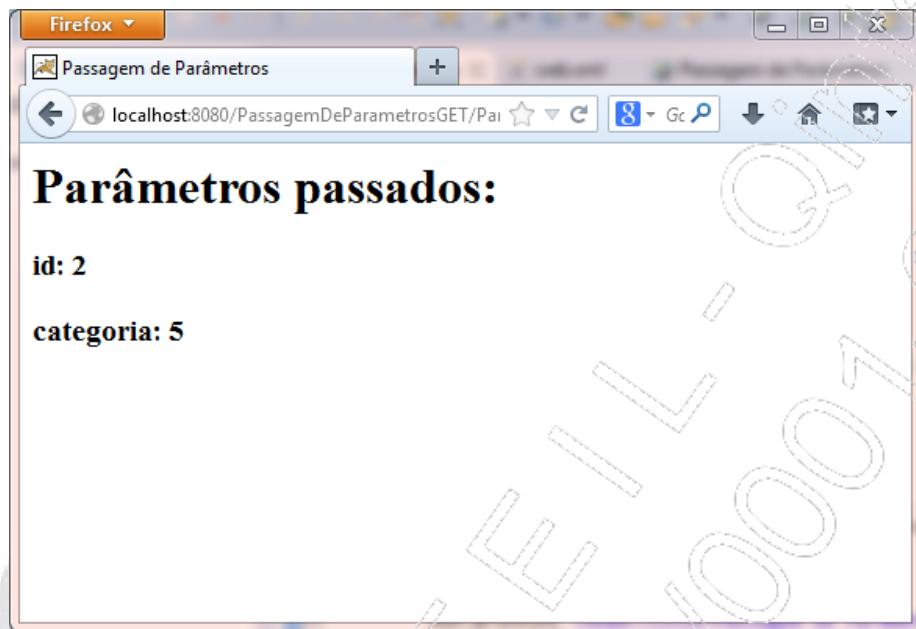
Note a tag `<form method="post" action="ParamServlet">` de abertura e perceba que os atributos definem o método e a porção da URL que será invocada quando o botão **Enviar** for clicado. É essencial a passagem de "ParamServlet" para o atributo **action**, sem a barra /, pois assim o caminho de URL atual é mantido e apenas o conteúdo **ParamServlet** é anexado ao final, exatamente o que desejamos com a instrução. Com a barra, todo o Context Root deve ser passado novamente.

Criamos uma página HTML denominada **index.html** com o conteúdo descrito. O próximo passo é definir código para o método **doPost(...)** em nosso Serviet, pois, do contrário, um erro do tipo **HTTP 405 - Método Post não é suportado por essa URL** será lançado. Neste exemplo, usamos o mesmo código anteriormente utilizado para o método **doGet(...)**, porém, agora para atender requisições Post.

Ao executar o programa, obtemos **index.html** como página inicial. Inspecione o **web.xml** e entenda por que essa página foi chamada automaticamente pelo Container:



Insira os valores 2 para **id** e 5 para **categoria** e clique em **Enviar**. Em seguida, você obterá este resultado:



O código no Servlet não precisou ser alterado para atender ambas as formas de requisição. O que foi preciso efetivamente foi a criação de um método para atender cada tipo de requisição. É necessário ter sempre **doGet(...)** para atender requisições GET e **doPost(...)** para atender requisições POST.

# Principais diferenças entre GET e POST

Entre os diversos motivos que permeiam a decisão pelo uso de GET ou POST, alguns são marcantes e devem ser sempre focados pelo desenvolvedor de forma a atingir o objetivo desejado sem efeitos colaterais:

	GET	POST
Segurança	Menos seguro	Mais seguro
Bookmarking	Permite	Não permite
Idempotência	Sim	Não
Tamanho	Limitado	Ilimitado

Quanto à segurança, o POST se destaca, pois seus parâmetros são passados de forma oculta ao Servlet, no corpo da requisição. O método GET, por passar todos os parâmetros pela URL, permite que o usuário guarde a URL para consulta específica posterior. Esse bookmarking é desejado quando o conteúdo do site puder ser acessado diretamente em certo ponto.

O método GET, pela definição HTTP, deve ser utilizado para se obter apenas coisas, nunca para alterar dados no servidor. Por esse motivo, caso o usuário clique diversas vezes em um botão **Enviar** ou acesse repetidas vezes a mesma URL, essas repetidas requisições NÃO devem provocar qualquer efeito colateral, por isso, o método GET é dito IDEMPOTENTE. Ao contrário, o método POST deve ser utilizado para atualizar dados no servidor, caracterizando-se como NÃO IDEMPOTENTE. Cabe ao desenvolvedor trabalhar para evitar efeitos colaterais quando uma requisição POST é enviada diversas vezes. Outro fator a ser lembrado é que o desenvolvedor é livre para inserir o código que desejar no método **doGet(...)**, inclusive código que manipule dados e atualize dados no servidor. Como já dito, essas são especificações do protocolo HTTP, que, no entanto, podem ser burladas pelo desenvolvedor.

Por fim, como o GET anexa os parâmetros à URL, essa solicitação tem tamanho limitado, já o POST não tem essa limitação.

# Encaminhamento e redirecionamento

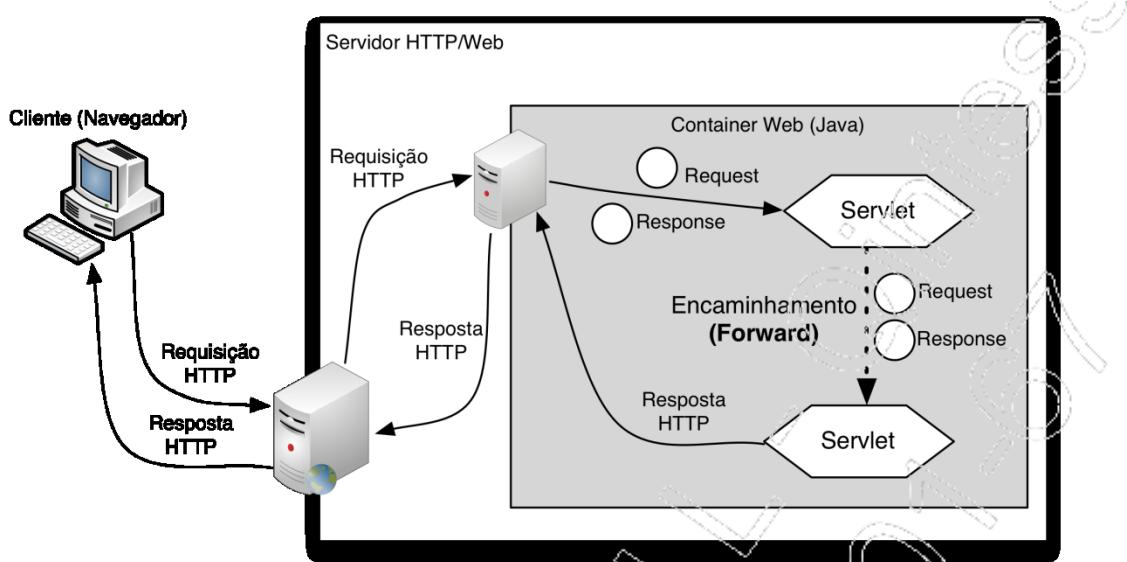
A navegação entre componentes Java e entre páginas é uma tarefa extremamente importante em toda aplicação Web. A todo momento será preciso encaminhar uma requisição para outro Servlet ou JSP que se encarregue de resolver o problema, seja por sua especialização ou por sua responsabilidade, e retornar ao cliente. Essa comunicação pode ser feita com a **preservação** dos dados passados na Requisição pelo usuário e é chamada de encaminhamento ou **Forward**.

Por vezes, pode ser necessário apenas encaminhar o usuário a uma outra URL, absoluta ou relativa, que aponte para um endereço em nossa própria aplicação ou mesmo em outro servidor. Esse tipo de comunicação é chamado de Redirecionamento ou **Redirect** e não preserva os dados passados por parâmetro na Requisição.

As formas de se realizar um Forward ou um Redirect são diferentes. Conheceremos os dois métodos de forma esquemática.

## Encaminhamento ou Forward

Quando se realiza um encaminhamento (ou Forward), o Servlet receptor da requisição decide que a requisição deve ser repassada para uma outra parte ou objeto da aplicação, como outro Servlet ou JSP. O usuário recebe a resposta sem saber o que ocorreu em background, pois a barra de endereços do navegador dele não muda.



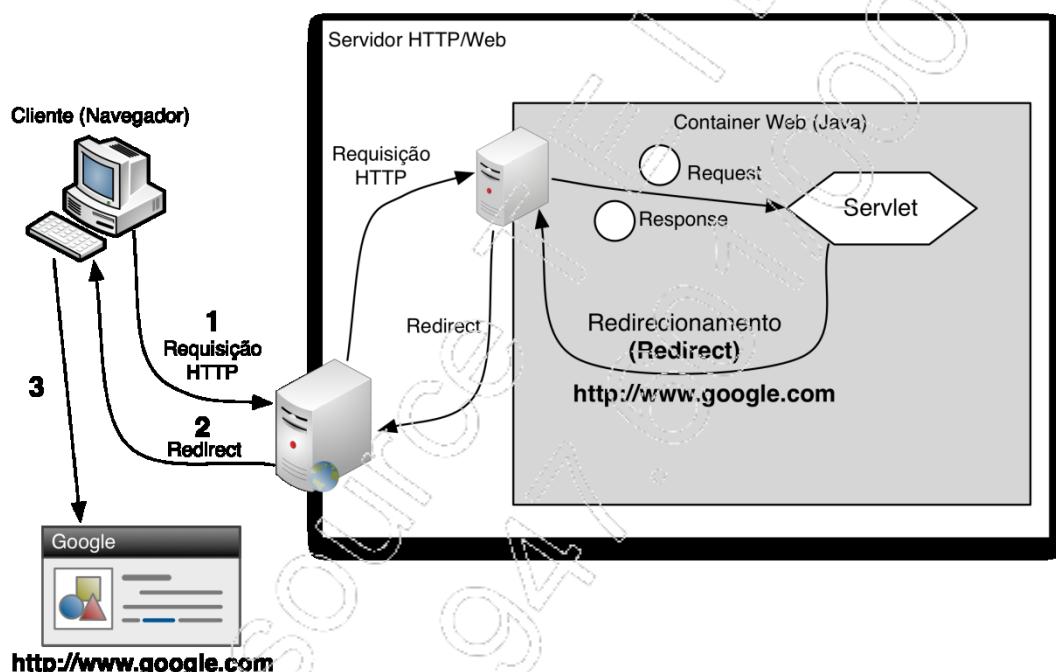
Vejamos como realizar um encaminhamento a partir do código de um Servlet (no corpo de um método de serviço como `doGet(...)` ou `doPost(...)`):

```

25   protected void doGet(HttpServletRequest request,
26                         HttpServletResponse response) throws ServletException, IOException {
27     //código de validação do encaminhamento
28     RequestDispatcher encaminhamento = request.getRequestDispatcher("/outroServlet");
29     encaminhamento.forward(request, response);
30 }
```

# Redirecionamento ou Redirect

No redirecionamento (ou Redirect), o Servlet ou outro componente percebe que não é responsável pela requisição e decide encaminhar o usuário para outra URL. Nesse caso, o endereço do navegador muda para o endereço de redirecionamento e todo o trabalho de requisição é feito novamente pelo navegador. O redirecionamento pode ser feito para uma URL da própria aplicação ou para uma URL completamente distinta, conforme mostrado no esquema adiante, no qual o usuário é redirecionado para a URL do Google.



Vejamos como realizar um redirecionamento a partir do código de um Servlet (no corpo de um método de serviço como `doGet(...)` ou `doPost(...)`):

```

25  protected void doGet(HttpServletRequest request,
26      HttpServletResponse response) throws ServletException, IOException {
27      response.sendRedirect("http://www.google.com");
28  }
  
```

# Parâmetros de inicialização: o ServletConfig e o ServletContext

Toda aplicação Java Web, ao ter seu deploy feito em um servidor, adquire dois tipos de contextos de extrema utilidade, ainda que esse servidor possua somente o Web Container, como o Apache Tomcat. Esses contextos, que devem ser entendidos como áreas de depósito de informações comuns para serem acessadas pelos componentes devidos, estão disponíveis para inicialização de objetos e/ou para compartilhamento:

- **ServletConfig:** Existe um para cada Servlet da aplicação. O ServletConfig possibilita a criação de parâmetros de inicialização recuperáveis programaticamente no Deployment Descriptor individualmente para cada Servlet;
- **ServletContext:** Trata-se de um contexto compartilhado por todos os Servlets na aplicação que possibilita a criação de parâmetros de inicialização recuperáveis programaticamente no Deployment Descriptor, e de atributos diversos alocados e consumidos por diversos Servlets.

É possível obter uma referência ao objeto **ServletConfig** por meio do método `getServletConfig()` e, da mesma forma, um **ServletContext** por meio do método `getServletContext()`, ambos métodos herdados.

## O ServletConfig

A principal razão de um **ServletConfig** existir é fornecer acesso aos parâmetros de inicialização do Servlet. O objeto é criado no método `init()` do Servlet e, em seguida, passado ao Servlet recém-criado para utilização.

O método para obter uma referência ao **ServletConfig** é o `getServletConfig()` herdado pelo Servlet e, portanto, disponível diretamente em qualquer lugar do Servlet, com exceção do construtor. Nunca chame esse método através do construtor, pois ele ainda não estará pronto para ser utilizado.

Segue o código para declarar um parâmetro de inicialização no Deployment Descriptor chamado "email". Como esse parâmetro é individual por Servlet, ele deve estar dentro das tags `<servlet>...</servlet>` do componente correspondente:

```
12<servlet>
13  <servlet-name>Param Servlet</servlet-name>
14  <servlet-class>servlet.ParamServlet</servlet-class>
15<init-param>
16  <param-name>email</param-name>
17  <param-value>aluno@impacta.com.br</param-value>
18</init-param>
19</servlet>
20<servlet-mapping>
21  <servlet-name>Param Servlet</servlet-name>
22  <url-pattern>/ParamServlet</url-pattern>
23</servlet-mapping>
```

Recuperando esse parâmetro a partir do código de um Servlet (do interior do bloco de algum método do Servlet):

```
30  out.println("Email: " + getServletConfig().getInitParameter("email"));
```

# O ServletContext

O ServletContext é o objeto que representa o contexto global da aplicação, compartilhado por todos os Servlets e JSPs. Quando a necessidade de um parâmetro de inicialização extrapolar os limites de um Servlet, certamente é o momento de utilizar esse contexto.

Ele ainda é utilizado como área comum para compartilhamento de objetos e elementos (denominados atributos) entre os componentes da aplicação.

A seguir, temos o código para declarar um parâmetro de inicialização no Deployment Descriptor chamado "email". Como esse parâmetro agora pertence ao ServletContext, ele deve estar dentro das tags `<context-param>...</context-param>`, que localizam-se diretamente abaixo das tags-raiz `<web-app>...</web-app>`:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 ⊕ <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
7     <display-name>PassagemDeParametrosGET</display-name>
8 ⊕     <welcome-file-list>
9         <welcome-file>index.html</welcome-file>
10        <welcome-file>index.htm</welcome-file>
11        <welcome-file>index.jsp</welcome-file>
12        <welcome-file>default.html</welcome-file>
13        <welcome-file>default.htm</welcome-file>
14        <welcome-file>default.jsp</welcome-file>
15     </welcome-file-list>
16 ⊕     <servlet>
17         <servlet-name>Param Servlet</servlet-name>
18         <servlet-class>servlet.ParamServlet</servlet-class>
19     </servlet>
20 ⊕     <servlet-mapping>
21         <servlet-name>Param Servlet</servlet-name>
22         <url-pattern>/ParamServlet</url-pattern>
23     </servlet-mapping>
24     <context-param>
25         <param-name>email</param-name>
26         <param-value>aluno@impacta.com.br</param-value>
27     </context-param>
28 </web-app>

```

De forma semelhante ao caso anterior, podemos recuperar esse parâmetro a partir do código de um Servlet (do interior do bloco de algum método do Servlet) com o seguinte comando:

```
30 |     out.println("Email: " + getServletContext().getInitParameter("email"));
```

A utilização do **ServletContext** para a passagem de atributos será examinada mais adiante, juntamente com o estudo de Listeners.

# Comunicação entre cliente e servidor

Teste seus conhecimentos



**1. O que deve ser feito com os parâmetros quando são passados ao servidor por meio do método GET?**

- a) Devem ser anexados ao fim da URL de chamada da aplicação, exatamente na ordem em que serão utilizados, a fim de evitar o casting indevido dos valores no Servlet.
- b) Podem ser passados no corpo da requisição assim como o método POST. O desenvolvedor pode optar por esse modo para explorar a idempotência do método GET.
- c) Devem ser anexados ao fim da URL, iniciados após o símbolo & e separados por um "?" a cada par de "chave=valor".
- d) Devem ser necessariamente passados via URL e pressupõem o caráter público dos parâmetros passados, sem ter a segurança das informações como foco principal.
- e) Nenhuma das alternativas anteriores está correta.

**2. Considerando as diferenças existentes entre os métodos GET e POST, qual das alternativas a seguir está correta?**

- a) O método POST, além de ser mais seguro, é o indicado tanto para obter dados do servidor quanto para atualizar dados no servidor, em qualquer situação.
- b) Diversos CMS (Content Management System) atuais trabalham fortemente o conceito de bookmarking, que prevê a possibilidade de que os usuários guardem as URLs consultadas para posterior consulta. Essa possibilidade é tipicamente fornecida pelo método GET.
- c) O método GET é considerado não idempotente porque geralmente não é utilizado para realizar alterações no lado servidor, somente consultas.
- d) Dentro de um Servlet, o Web Container do Java encarrega-se de garantir a idempotência do código, que lidará com requisições GET, e do conceito oposto, para o código que lidará com requisições POST.
- e) Com relação ao tamanho e à quantidade de parâmetros, não há diferença entre os métodos GET e POST.

**3. O que o desenvolvedor deve ter em mente ao realizar um encaminhamento ou Forward?**

- a) Por meio desse tipo de navegação, o usuário não poderá realizar o bookmarking da página para a qual foi redirecionado.
- b) Os parâmetros passados na requisição inicial serão perdidos quando a navegação ocorrer.
- c) O ServletContext será reiniciado toda vez que uma navegação deste tipo ocorrer.
- d) O objeto utilizado para navegação é obtido somente por meio do objeto HttpServletResponse e recebido pelo Servlet no momento da chegada de uma nova requisição.
- e) Nenhuma das alternativas anteriores está correta.

**4. O que é possível afirmar acerca do objeto ServletConfig?**

- a) Só pode ser obtido por meio de um objeto HttpServletRequest, uma vez que o Servlet só é executado quando chega uma nova requisição.
- b) É herdado pelo Servlet e, portanto, pode ser invocado de qualquer lugar do componente.
- c) Tem como função principal a possibilidade de compartilhar atributos entre Servlets da mesma aplicação, desde que sejam inseridos via web.xml.
- d) Pode possuir parâmetros de inicialização que são declarados em um par de tags <config-param>...<config-param>.
- e) Nenhuma das alternativas anteriores está correta.

**5. Na comparação entre os objetos ServletConfig e ServletContext, podemos afirmar que:**

- a) ambos os objetos podem ser obtidos por métodos próprios, herdados por todos os Servlets, diretamente de um dos métodos de serviço do componente.
- b) o ServletContext é individual por Servlet e o ServletConfig é unitário por aplicação.
- c) é possível alocar atributos em ambos os objetos como forma de compartilhamento entre os componentes da aplicação.
- d) o ServletContext só pode ser acessado por Servlets declarados no mesmo pacote.
- e) Nenhuma das alternativas anteriores está correta.

# Comunicação entre cliente e servidor

Mãos à obra!



## Laboratório 1

### A - Passando parâmetros na requisição

1. Crie um novo projeto do tipo **Dynamic Web Project** no Eclipse denominado **LabWeb4**, assegurando-se de checar a opção **Generate web.xml deployment descriptor**;
2. Crie um novo arquivo HTML no projeto com o nome **index.html**. Ao visualizar o diálogo de escolha do template, escolha HTML 5 como padrão e finalize;
3. Perceba que o Eclipse cria o arquivo dentro da pasta **WebContent**, pois é lá que todo o conteúdo que irá compor o pacote para deploy no servidor estará. Inspecione o código desse arquivo;
4. O formulário que será inserido nessa página deverá estar dentro das tags **<body>...</body>**, portanto observe e digite o código a seguir. O trecho em destaque é o que será acrescentado ao template padrão da página;

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>Insert title here</title>
6  </head>
7  <body>
8      <h3>Cadastro de Usuário</h3>
9      <form method="post" action="capturadados">
10         Nome: <br />
11         <input type="Text" name="nome" size="40"> <br />
12
13         E-mail: <br />
14         <input type="Text" name="email" size="40">
15
16         <p>
17             Deseja receber nossa newsletter? <br />
18             <input type="Radio" name="newsletter" value="sim">Sim
19             <input type="Radio" name="newsletter" value="nao">Não
20         </p>
21         <p>
22             Informe qual a sua posição na empresa em que trabalha: <br />
23             <select name="posicao">
24                 <option value="CEO">CEO</option>
25                 <option value="Diretor Executivo">Diretor Executivo</option>
26                 <option value="Gerente de TI">Gerente de TI</option>
27                 <option value="Analista de Produção">Analista de Produção</option>
28                 <option value="Arquiteto de Software">Arquiteto de Software</option>
29                 <option value="Desenvolvedor">Desenvolvedor</option>
30             </select>
31         </p>
32         <p>
33             Deixe suas impressões sobre nosso serviço: <br />
34             <textarea name="impressoes" cols="28" rows="5"></textarea>
35         </p>
36         <p>
37             <input type="Submit" value="Enviar ">
38             <input type="Reset" value="Limpar Dados">
39         </p>
40     </form>
41 </body>
42 </html>
```

5. Aproveite para analisar o código HTML utilizado. Note que o formulário em questão tem como método configurado o POST e a action configurada aponta para um padrão de URL cujo Servlet responsável ainda não foi criado: "**capturadados**". Crie agora um Servlet com o nome **CapturaDados** e que responda à URL de padrão "/**capturadados**";

6. Insira o seguinte código no Servlet:

```
1 package servlet;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.annotation.WebServlet;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11 @WebServlet("/capturadados")
12 public class CapturaDados extends HttpServlet {
13     private static final long serialVersionUID = 1L;
14
15     public CapturaDados() {
16         super();
17     }
18
19     protected void doPost(HttpServletRequest request,
20             HttpServletResponse response) throws ServletException, IOException {
21         if (request.getCharacterEncoding() == null) {
22             request.setCharacterEncoding("UTF-8");
23         }
24         response.setCharacterEncoding("UTF-8");
25         response.setContentType("text/html");
26         PrintWriter out = response.getWriter();
27         out.println("<html>");
28         out.println("<head>");
29         out.println("<title>Cadastro realizado com sucesso!</title>");
30         out.println("</head>");
31         out.println("<body>");
32         out.println("<h1>Cadastro realizado com sucesso!</h1>");
33         out.println("<h4>Nome:</h4> " + request.getParameter("nome"));
34         out.println("<h4>E-mail:</h4> " + request.getParameter("email"));
35         out.println("<h4>Deseja receber nossa newsletter?:</h4> " +
36             request.getParameter("newsletter"));
37         out.println("<h4>Posição em que trabalha:</h4> " +
38             request.getParameter("posicao"));
39         out.println("<h4>Impressões:</h4> \n" + request.getParameter("impressoes"));
40         out.println("</body>");
41         out.println("</html>");
42     }
43 }
```

Na parte destacada, note que estamos utilizando um trecho novo de código. Esse trecho garante que a página gerada pelo Servlet seja compatível com o padrão UTF-8, que abrange os caracteres especiais da língua portuguesa (e todos os outros idiomas do mundo!). Perceba também que estamos configurando e realizando o mapeamento desse Servlet por meio da anotação `@WebServlet("/capturadados")`.

7. Inspecione o **web.xml** e certifique-se de que no nó **<welcome-file-list>...</welcome-file-list >** exista uma linha para o valor **index.html**. Agora já é possível perceber que o Container irá procurar um arquivo que possua um dos nomes descritos e executá-lo como página inicial em nossa aplicação;
8. Execute a aplicação. Será possível visualizar a seguinte página com o formulário:

The screenshot shows a Firefox browser window with the title bar 'Firefox'. The address bar displays 'localhost:8080/LabWeb4/'. The main content area is a form titled 'Cadastro de Usuário'. The form fields include:

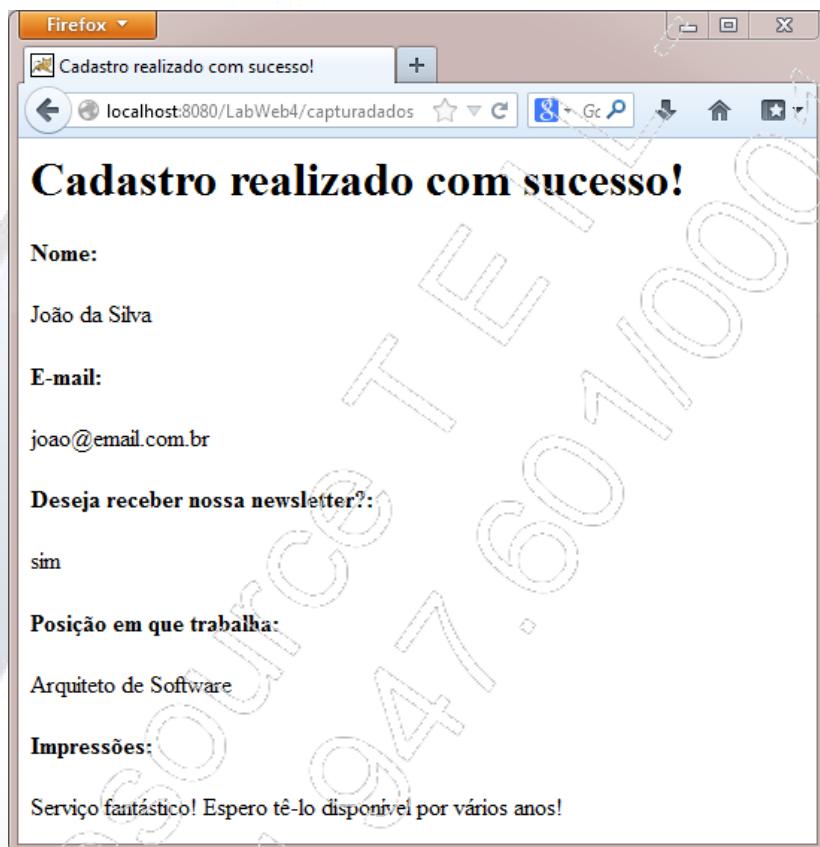
- Nome: (text input field)
- E-mail: (text input field)
- Deseja receber nossa newsletter?  
○ Sim ○ Não (radio buttons)
- Informe qual a sua posição na empresa em que trabalha:  
CEO (dropdown menu)
- Deixe suas impressões sobre nosso serviço:  
 (text area)
- Enviar (button)
- Limpar Dados (button)

9. Preencha o formulário com os dados a seguir:

- Nome: **João da Silva**;
- E-mail: **joao@email.com.br**;
- Deseja receber nossa newsletter? **Sim**;

- Informe qual a sua posição na empresa em que trabalha: **Arquiteto de Software**;
- Deixe suas impressões sobre nosso serviço: **Serviço fantástico! Espero tê-lo disponível por vários anos!**.

10. Clique em **Enviar**. Será exibida a seguinte página:



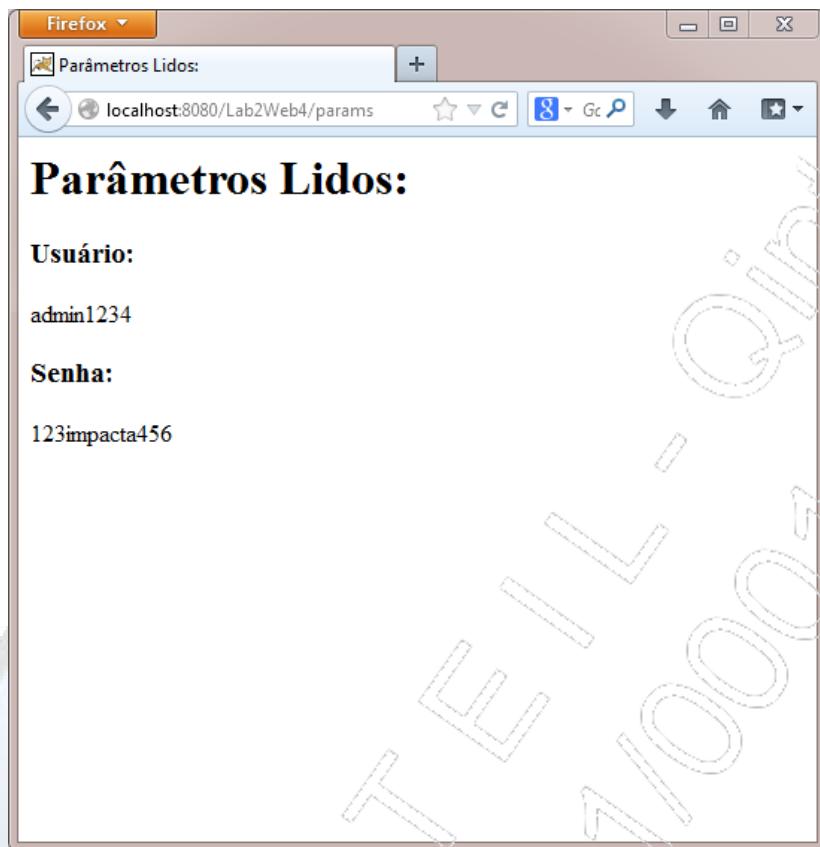
## Laboratório 2

### A - Configurando parâmetros de inicialização

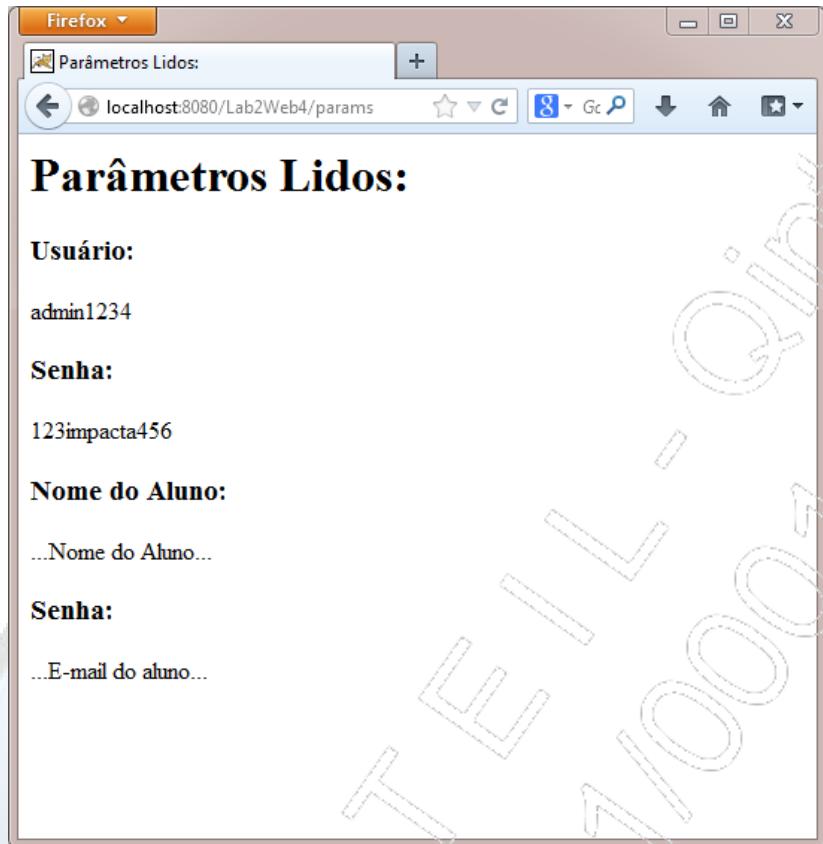
1. Utilizando o projeto criado no Laboratório 1, crie dois parâmetros de inicialização no **ServletContext** da aplicação para guardar os valores de usuário e senha temporários da aplicação com os seguintes pares de nome/valor:

```
param-name: usuario  
param-value: admin1234  
  
param-name: senha  
param-value: 123impacta456
```

2. Crie um novo Servlet denominado **ParamServlet**, mapeado para o padrão de URL "/params", que exiba os parâmetros de inicialização no método específico para tratar requisições GET. Ao ser executada a aplicação e chamada a URL <http://localhost:8080/Lab2Web4/params>, a seguinte página deverá ser exibida:



3. Crie mais dois parâmetros, um com seu nome e outro com seu e-mail, desta vez mapeados dentro do **ServletConfig** do Servlet **ParamServlet** criado no passo anterior. Para essa tarefa, faça as devidas alterações no **web.xml** e apague a Annotation existente no Servlet, de forma a instruir o Container a usar o DD para declaração e mapeamento;
4. Adicione o código HTML à saída do método responsável para exibir esses parâmetros capturados ao fim da página já criada e utilizada para exibir os parâmetros definidos no passo 1. A página final deverá ser semelhante a esta:



## Laboratório 3

### A - Realizando navegação por meio de encaminhamento

1. Utilizando o projeto do Laboratório anterior, crie um novo Servlet denominado **ForwardServlet**, mapeado para o padrão de URL **"/forward"**. Escolha a forma que preferir para declaração e mapeamento;

2. Esse Servlet espera receber um parâmetro denominado **"chave"**, que poderá conter os valores 1 ou 2, por meio de chamada do tipo GET. Adicione código ao método adequado do Servlet que verifique o valor do parâmetro recebido:

- Caso tenha sido 1, encaminhe (Forward) o usuário para o Servlet **ParamServlet**, criado no exercício anterior, lembrando de utilizar o mapeamento correto criado para aquele componente ("**/params**");
- Caso tenha sido 2, redirecione (Redirect) o usuário para a homepage da Impacta: <http://www.impacta.com.br>;
- Caso seja qualquer outro valor, retorne uma página em branco ao usuário com a seguinte mensagem de erro: **O parâmetro passado é inválido!**.

3. Execute a aplicação e teste as três possíveis URLs:

- <http://localhost:8080/Lab3Web4/forward?chave=1>
- <http://localhost:8080/Lab3Web4/forward?chave=2>
- <http://localhost:8080/Lab3Web4/forward?chave=235>

# Capítulo 5: Aplicação avançada de Servlets



# Atributos e listeners

Neste tópico, vamos conhecer as características e as funcionalidades de atributos e listeners.

## Atributos

No que diz respeito aos componentes Java em uma aplicação Web (vimos Servlets, até o momento), abordamos as diversas formas de comunicação e passagem de parâmetros entre cliente e servidor, além da utilização e aplicação de parâmetros de inicialização nos escopos do **ServletConfig** (um por Servlet) e **ServletContext** (um por aplicação).

O mecanismo de passagem de parâmetros atende apenas uma parte das necessidades de uma aplicação Web tendo em vista algumas limitações inerentes a essa funcionalidade, como:

- Permitir apenas a passagem e captura de Strings;
- Não facilitar a passagem de grande quantidade de informações;
- Ser um mecanismo ineficiente quando existe a necessidade de comunicação entre componentes no mesmo servidor.

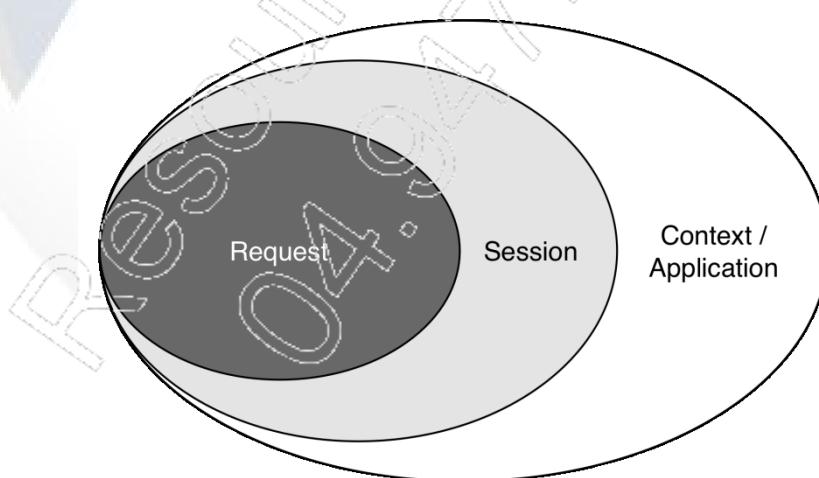
Nesse contexto, surge a necessidade de um outro mecanismo, mais eficiente e maleável, focado na comunicação interna entre componentes do servidor: os atributos.

A plataforma Java fornece uma API voltada à manipulação de atributos. Entende-se por API um conjunto de métodos que compõem uma interface pública comum. É possível encontrar essa API nas interfaces **ServletContext**, **HttpServletRequest** e **HttpSession**. Esse conjunto de métodos permite a inclusão, recuperação e remoção de quaisquer objetos em todos os contextos anteriormente descritos com a finalidade de compartilhamento e comunicação entre os componentes de uma aplicação Java Web.

A chamada "API de manipulação de atributos" compreende os seguintes métodos:

- **Object getAttribute(String name);**
- **void setAttribute(String name, Object value);**
- **void removeAttribute(String name);**
- **Enumeration <String>getAttributeNames();**

Dentro de qualquer aplicação Java Web, é importante que o desenvolvedor tenha bem clara a noção dos escopos disponíveis para trabalho. O seguinte diagrama demonstra quais são e como se relacionam esses contextos:



- **Request ou Escopo de requisição:** O menor ou o mais volátil dos escopos. É renovado a cada requisição nova do usuário. Qualquer atributo inserido neste escopo tem a duração de uma requisição. Existem diversos escopos desse tipo por usuário;

- Inserindo e lendo um atributo do escopo Request:

```
46     request.setAttribute("dataNascimento", new Date());  
47  
48     Date data = (Date)request.getAttribute("dataNascimento");  
49
```

- **Session ou Escopo de sessão:** Escopo intermediário e o mais usado por desenvolvedores Web. Atributos inseridos neste escopo perduram por várias requisições do mesmo usuário até que a sessão seja invalidada por uma das causas possíveis. Deve ser usado com cautela, pois pode sobrecarregar o servidor quando um número grande de usuários estiver acessando a aplicação simultaneamente. Geralmente existe um escopo desse tipo por usuário;
- **Context / Application ou Escopo de aplicação:** Escopo de maior abrangência. É único por aplicação. Atributos inseridos neste escopo permanecem por toda a vida da aplicação. Deve ser usado com extrema cautela, pois quaisquer elementos inseridos aqui permanecerão por toda a vida da aplicação se não forem removidos manualmente. Existe apenas um escopo desse tipo compartilhado por todos os usuários e suas sessões e requisições.
- Inserindo e lendo um atributo do escopo Application:

```
46     getServletContext().setAttribute("dataNascimento", new Date());  
47  
48     Date data = (Date)getServletContext().getAttribute("dataNascimento");  
49
```

# Listeners

Existem situações nas aplicações Web em que é importante poder executar um código customizado na ocorrência de certos eventos. Por exemplo, pode-se querer ler o conteúdo de um parâmetro de inicialização do contexto da aplicação logo no momento de sua criação, de forma a preparar a infraestrutura para os demais componentes da aplicação. Uma string que represente a conexão com um banco de dados pode estar colocada em um context-param no **web.xml**. Dessa forma, é importante lê-la logo no início da aplicação, antes da criação de qualquer Servlet ou JSP, de forma a deixar a conexão com o banco de dados já preparada. Esse é um caso típico em que não cabe o uso de Servlets ou JSPs. Existe a necessidade de algo independente, um componente que receba alerta dos eventos que ocorrem durante o ciclo de vida da aplicação e que seja exterior aos componentes de negócios. Tal aplicação é tipicamente atribuída a um Listener.

Um Listener ou "ouvinte" nada mais é do que uma classe Java que aplica uma das interfaces definidas na API, fornecendo implementação concreta aos seus métodos, chamados pelo Container, conforme o evento de disparo ocorra.

Existem Listeners para uma gama de eventos no Web Container. A seguir, temos uma tabela com uma lista de Listeners disponíveis:

Listener	Aplicação prática
<b>ServletContextAttributeListener</b>	Capturar eventos de adição, remoção ou substituição de atributos no <b>ServletContext</b> .
<b>HttpSessionListener</b>	Capturar eventos que permitem rastrear as sessões e usuários ativos.
<b>ServletRequestListener</b>	Captura um evento gerado toda vez que uma nova requisição chegar.
<b>ServletRequestAttributeListener</b>	Captura eventos de adição, remoção ou substituição de atributos no contexto Request.
<b>HttpSessionBindingListener</b>	Notifica uma classe de objeto-atributo de

Listener	Aplicação prática
	Sessão sobre modificações ocorridas nesta classe de objetos.
<b>HttpSessionAttributeListener</b>	Capturar eventos de adição, remoção ou substituição de atributos no contexto Session.
<b>ServletContextListener</b>	Capturar o evento lançado quando um <b>ServletContext</b> foi criado ou destruído.
<b>HttpSessionActivationListener</b>	Notifica uma classe de objeto-atributo de sessão quando a sessão à qual esse objeto está associado estiver migrando entre JVMs.

Um Listener que implementa **ServletContextAttributeListener** possui o seguinte código:

```

1 package listeners;
2
3 import javax.servlet.ServletContextAttributeEvent;
4 import javax.servlet.ServletContextAttributeListener;
5 import javax.servlet.annotation.WebListener;
6
7 @WebListener
8 public class ServletContextListener implements ServletContextAttributeListener {
9
10    public ServletContextListener() {
11
12    }
13
14    public void attributeAdded(ServletContextAttributeEvent event) {
15        System.out.println("Atributo " + event.getName() +
16            " adicionado ao ServletContext!");
17    }
18
19    public void attributeReplaced(ServletContextAttributeEvent event) {
20    }
21
22
23    public void attributeRemoved(ServletContextAttributeEvent event) {
24        System.out.println("Atributo " + event.getName() +
25            " removido do ServletContext!");
26    }
27 }
```

No Listener criado, os eventos de atributo adicionado e removido do **ServletContext** estão sendo capturados. Quando esses eventos ocorrerem, os devidos métodos serão chamados.

Todo Listener, para ser tratado como tal pelo Container, deve ter aplicada a Annotation **@WebListener** à classe devida ou uma declaração equivalente no **web.xml** que tem a seguinte sintaxe (semelhante ao que ocorre com Servlets):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
7   id="WebApp_ID" version="3.0">
8
9 <display-name>ListenersTest</display-name>
10<welcome-file-list>
11   <welcome-file>index.html</welcome-file>
12   <welcome-file>index.htm</welcome-file>
13   <welcome-file>index.jsp</welcome-file>
14   <welcome-file>default.html</welcome-file>
15   <welcome-file>default.htm</welcome-file>
16   <welcome-file>default.jsp</welcome-file>
17 </welcome-file-list>
18
19<servlet>
20   <servlet-name>AddAttributeServlet</servlet-name>
21   <servlet-class>servlet.AddAttributeServlet</servlet-class>
22 </servlet>
23<servlet-mapping>
24   <servlet-name>AddAttributeServlet</servlet-name>
25   <url-pattern>/AddAttributeServlet</url-pattern>
26 </servlet-mapping>
27
28<servlet>
29   <servlet-name>RemoveAttributeServlet</servlet-name>
30   <servlet-class>servlet.RemoveAttributeServlet</servlet-class>
31 </servlet>
32<servlet-mapping>
33   <servlet-name>RemoveAttributeServlet</servlet-name>
34   <url-pattern>/RemoveAttributeServlet</url-pattern>
35 </servlet-mapping>
36
37<listener>
38   <listener-class>listeners.ServletContextListener</listener-class>
39 </listener>
40
41 </web-app>
```

Para o Listener e o **web.xml** fornecidos, segue o código empregado em dois Servlets nessa aplicação. O primeiro adiciona um atributo ao **ServletContext** ao receber uma requisição GET e o segundo remove o mesmo atributo do **ServletContext** ao receber uma requisição GET:

- Servlet que adiciona o atributo:

```

1 package servlet;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.http.HttpServlet;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8
9 public class AttributeServlet extends HttpServlet {
10     private static final long serialVersionUID = 1L;
11
12     public AttributeServlet() {
13         super();
14     }
15
16     protected void doGet(HttpServletRequest request,
17             HttpServletResponse response) throws ServletException, IOException {
18
19         getServletContext().setAttribute("email", "aluno@impacta.com.br");
20
21     }
22 }
```

- Servlet que remove o atributo:

```

1 package servlet;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.http.HttpServlet;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8
9 public class RemoveAttributeServlet extends HttpServlet {
10     private static final long serialVersionUID = 1L;
11
12     public RemoveAttributeServlet() {
13         super();
14     }
15
16     protected void doGet(HttpServletRequest request,
17             HttpServletResponse response) throws ServletException, IOException {
18
19         getServletContext().removeAttribute("email");
20
21     }
22 }
```

Ao executar a aplicação e inserir a URL

**http://localhost:8080/ListenersTest/AddAttributeServlet**, obtemos a saída esperada no console do servidor:

```
INFO: Server startup in 705 ms
Atributo email adicionado ao ServletContext!
```

Quando inserimos a URL

**http://localhost:8080/ListenersTest/RemoveAttributeServlet**, obtemos a saída complementar no console do servidor:

```
INFO: Server startup in 705 ms
Atributo email adicionado ao ServletContext!
Atributo email removido do ServletContext!
```

## Uso de cookies e gerenciamento de sessão

Neste tópico, vamos abordar a utilização de cookies e sessões. Em diversas oportunidades nota-se o uso de cookies por várias aplicações ou sites visitados na Internet. Já as sessões compõem o mais importante escopo de toda aplicação Java Web e estão presentes em quase todas as tecnologias de servidor conhecidas atualmente.

### O que são cookies?

Cookies são pequenas porções de dados armazenados na máquina do usuário. Eles podem ser criados e manipulados por um aplicativo ou site da Web, caso haja permissão do usuário para tanto.

Esses pequenos conjuntos de dados são muito usados para se obter alguma informação útil acerca do usuário conectado ou de histórico de conexões já realizadas. Exemplos desses dados são: código de identificação da sessão do usuário, últimas páginas visitadas na aplicação, assuntos de interesse, entre outros.

Em Java, esses "pequenos conjuntos de dados" são representados por objetos específicos, instâncias da classe **javax.servlet.http.Cookie**. Eles podem ser enviados para o cliente por meio do já conhecido objeto **HttpServletResponse** e recuperados por meio de um objeto **HttpServletRequest**.

Esse acesso é feito por meio dos seguintes métodos:

- Na classe **HttpServletRequest**:
- **getCookies():Cookies[]**: Retorna um array de todos os objetos cookie que o cliente enviou na requisição;
- Na classe **HttpServletResponse**:
- **addCookie(cookie:Cookie):void**: Adiciona o cookie passado como parâmetro na resposta.

## A classe javax.servlet.http.Cookie

Métodos	Descrição
<b>getComment():String</b> <b>setComment(comment:String):void</b>	Métodos de acesso ao atributo <b>comment</b> , que descreve o cookie e seu propósito.
<b>getDomain():String</b> <b>setDomain(domain:String):void</b>	Métodos de acesso ao String que representa o domínio em que todos os componentes terão acesso ao cookie.
<b>getMaxAge():int</b> <b>setMaxAge(maxAge:int):void</b>	Métodos de acesso para o tempo de vida do cookie, em segundos. -1 significa que o cookie deve expirar assim que o browser for fechado.
<b>getName():String</b> <b>setName(name:String):void</b>	Métodos de acesso ao atributo <b>name</b> do cookie, usado como identificador.
<b>getPath():String</b> <b>setPath(path:String):void</b>	Métodos de acesso à URL de onde todos os componentes poderão acessar o cookie.
<b>getSecure():boolean</b> <b>setSecure(secure:boolean):void</b>	Define se será utilizado https para acesso ao cookie.
<b>getValue():String</b> <b>setValue(value:String):void</b>	Métodos de acesso ao atributo <b>value</b> do cookie. É efetivamente o dado que esse cookie carrega consigo.

# Demonstrando o uso de cookies

Em um exemplo em que são utilizados cookies para armazenar os dados de usuário e senha, temos dois Servlets: um para adicionar dois cookies à máquina do usuário e outro para lê-los e exibi-los na tela.

Vejamos o primeiro Servlet, que é denominado **CriandoCookieServlet**:

```

1 package servlets;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.annotation.WebServlet;
7 import javax.servlet.http.Cookie;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 @WebServlet("/CriandoCookieServlet")
13 public class CriandoCookieServlet extends HttpServlet {
14     private static final long serialVersionUID = 1L;
15
16     protected void doGet(HttpServletRequest request,
17             HttpServletResponse response) throws ServletException, IOException {
18         response.setContentType("text/html;charset=UTF-8");
19
20         String usuario = "aluno";
21         String senha = "impacta123";
22
23         response.addCookie(new Cookie("usuario", usuario));
24         response.addCookie(new Cookie("senha", senha));
25
26         PrintWriter out = response.getWriter();
27         out.println("<html>");
28         out.println("<head><title>Adicionando cookies na resposta</title></head>");
29         out.println("<body>");
30         out.println("<h1>Foram adicionados os seguintes Cookies à resposta:</h1>");
31         out.println("<h3>Usuário: " + usuario + "</h3> <br />");
32         out.println("<h3>Senha: " + senha + "</h3> <br />");
33         out.println("</body>");
34         out.println("</html>");
35         out.close();
36     }
37 }
```

Vale destacar o método utilizado para se adicionar um Cookie à máquina do usuário, **void addCookie(Cookie cookie)**, do objeto response. É importante lembrar que toda configuração feita no objeto response deve ocorrer antes que se faça qualquer tipo de redirecionamento ou encaminhamento.

Agora vejamos o código do segundo Servlet, utilizado para ler os Cookies e mostrar o conteúdo em uma página HTML:

```

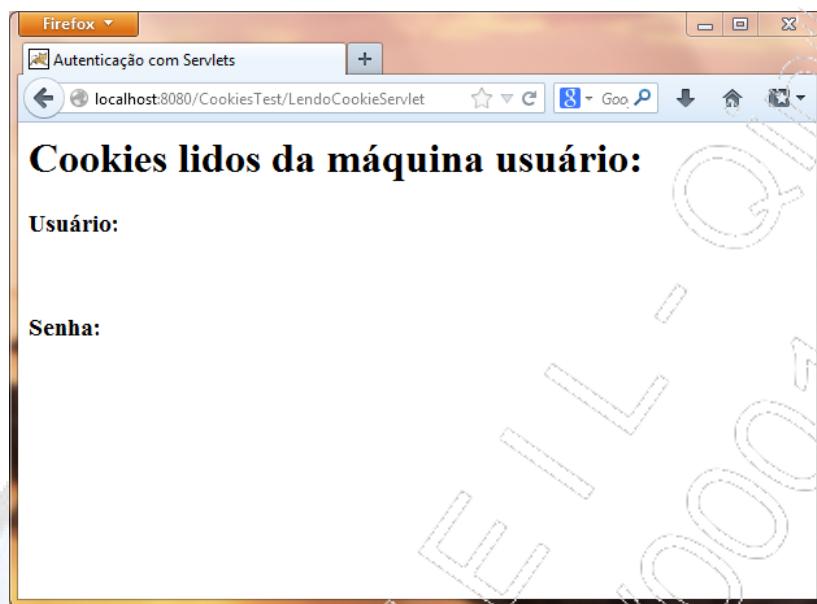
1 package servlets;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.annotation.WebServlet;
7 import javax.servlet.http.Cookie;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 @WebServlet("/LendoCookieServlet")
13 public class LendoCookieServlet extends HttpServlet {
14     private static final long serialVersionUID = 1L;
15
16     protected void doGet(HttpServletRequest request,
17             HttpServletResponse response) throws ServletException, IOException {
18         response.setContentType("text/html;charset=UTF-8");
19
20         String usuario = "";
21         String senha = "";
22
23         Cookie[] cookies = request.getCookies();
24
25         if (cookies != null) {
26             for (int i = 0; i < cookies.length; i++) {
27                 if (cookies[i].getName().equals("usuario")) {
28                     usuario = cookies[i].getValue();
29                 }
30
31                 if (cookies[i].getName().equals("senha")) {
32                     senha = cookies[i].getValue();
33                 }
34             }
35         }
36
37         PrintWriter out = response.getWriter();
38         out.println("<html>");
39         out.println("<head><title>Autenticação com Servlets</title></head>");
40         out.println("<body>");
41         out.println("<h1>Cookies lidos da máquina usuário:</h1>");
42         out.println("<h3>Usuário: " + usuario + "</h3> <br />");
43         out.println("<h3>Senha: " + senha + "</h3> <br />");
44         out.println("</body>");
45         out.println("</html>");
46         out.close();
47     }

```

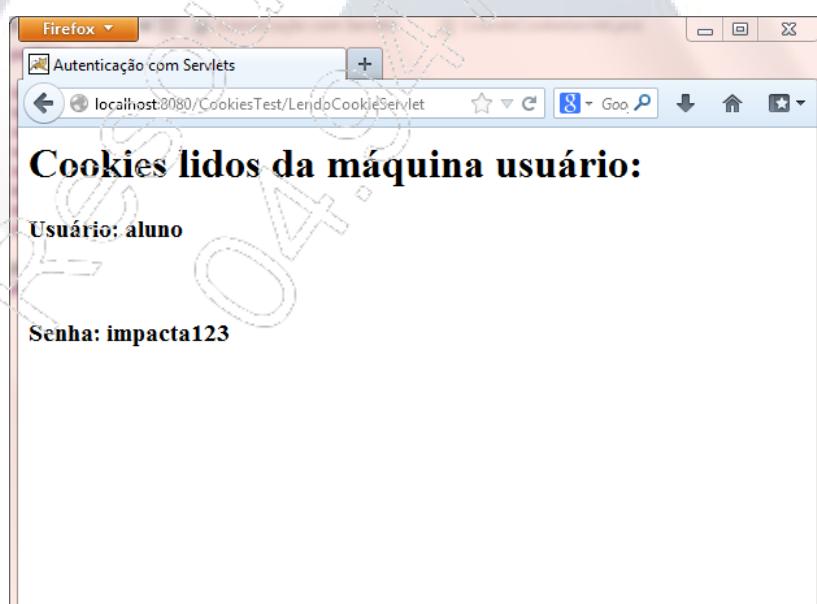
Nesse Servlet, é importante destacar o método utilizado para recuperar os cookies do objeto `request`, `Cookie[] getCookies()`. Esse método retorna um array contendo todos os cookies encontrados para o contexto da aplicação ou null, caso não haja nenhum.

Tendo sido feito o deploy da aplicação no servidor, ao invocar a URL `http://localhost:8080/CookiesTest/LendoCookieServlet` relativa à requisição GET para o Servlet de leitura de cookies, sem tê-los implantado ainda, obtemos

o resultado já esperado para a página: os campos de usuário e senha estão vazios:



Posteriormente à URL de leitura, fazendo a chamada ao Servlet de criação dos cookies, **CriandoCookieServlet**, por meio da URL <http://localhost:8080/CookiesTest/CriandoCookieServlet>, obtemos, agora com os cookies já criados, os valores devidos para os dois cookies implantados na máquina do usuário:



# Definindo a necessidade e a utilidade do uso de sessões

Conforme explicitado anteriormente, as sessões compõem, juntamente com as Requisições e o **ServletContext**, os três escopos mais comuns disponíveis aos desenvolvedor.

As sessões persistem por várias requisições de um mesmo usuário e são fundamentais para individualizar a experiência que a aplicação propicia aos seus usuários. São elas que permitem a memorização de dados consultados ou inseridos em requisições anteriores e que possibilitam, por exemplo, guardar os itens adicionados a um carrinho de compras em uma loja virtual.

Algumas considerações são essenciais para o bom uso de sessões:

- As sessões são individuais, existe uma para cada cliente conectado;
- As sessões persistem a diversas requisições, mas, uma vez encerradas, seus dados se perdem caso não sejam armazenadas em um banco de dados, em um arquivo físico ou transferidas para um contexto maior, como o de aplicação (**ServletContext**);
- O uso excessivo de dados em sessão pode acarretar uma grande demanda de memória no servidor. Por esse motivo, o desenvolvedor deve estar atento a isso.

## O controle feito pelo Container

O container utiliza-se de um identificador único para localizar e mapear os usuários e suas respectivas sessões. Esse identificador é encaminhado via Response, geralmente armazenado no cliente via Cookie e recuperado via Request. O formato desse identificador varia conforme o Java Container / Servidor de aplicações que está sendo utilizado. Para o caso do Apache Tomcat, esse identificador é semelhante a **JSESSIONID=AS435H878JG12477**.

As estratégias para gerenciamento da identificação do usuário são variadas e dependem da situação de acesso permitido pelo usuário e das preferências do Container. Vamos conhecê-las a seguir:

Estratégia	Descrição
Cookies	São trocados pelo header da <b>request</b> e <b>response</b> .
Reescrita de URL	Nessa estratégia, o ID de identificação da sessão é acoplado à URL de todas as requisições e links nas páginas.
Campos escondidos (Hidden fields)	O servidor insere campos ocultos nos FORMs como um modo de manter controle da sessão.
SSL (Secure Sockets Layer)	As chaves de sessão geradas pelo SSL em conexões HTTPS podem ser usadas como identificação para a sessão do usuário.

# Manipulando a sessão via código

Uma sessão em Java é representada pela classe `javax.servlet.http.HttpSession` e possui alguns métodos de grande utilidade para o desenvolvedor Java.

Para se obter uma sessão, utiliza-se o método `getSession()`:

```
HttpSession session = request.getSession();
```

Perceba que é possível obter uma referência para o objeto `HttpSession`, que representa uma sessão em Java, diretamente da Request. O método `getSession()`, no entanto, é sobrecarregado:

Método	Descrição
<code>getSession():HttpSession</code>	Retorna a sessão já existente com essa <code>request</code> ou cria uma nova, caso não exista nenhuma.
<code>getSession(create:Boolean):HttpSession</code>	Retorna a sessão já existente com essa <code>request</code> , ou cria uma nova se não existir uma sessão e o argumento <code>create</code> for igual a <code>true</code> . Caso o argumento seja <code>false</code> e não exista nenhuma sessão associada, retorna <code>null</code> .

Criada a sessão, é possível obter seu ID e sua data de criação por meio destes dois importantes métodos:

Método	Descrição
<code>getId():String</code>	Retorna uma String com o identificador único da sessão criado pelo Container.
<code>getCreationTime():long</code>	Retorna a data de criação da sessão em milisegundos que podem ser convertidos em um objeto do tipo Date para manipulação no código.

As sessões possuem os métodos da API de atributos e oferecem grande facilidade na manipulação de objetos neste escopo. Descrevemos esses métodos, já citados, para uma melhor compreensão:

Método	Descrição
<code>getAttribute(name:String):Object</code>	Retorna o objeto associado ao nome passado.
<code>getAttributeNames():Enumeration</code>	Retorna uma Enumeration com os nomes dos atributos existentes na sessão.
<code>setAttribute(name:String, valor:Object)</code>	Adiciona o objeto na sessão associado ao nome passado no parâmetro do método.
<code>removeAttribute(name:String):void</code>	Remove o objeto especificado.

## Invalidando uma sessão

Uma sessão deve ser invalidada quando não estiver mais em uso, de forma a preservar recursos no servidor e os dados do usuário que precisem ser persistidos. Comumente, ocorre quando o usuário clica no botão **Sair** da aplicação ou quando permanece muito tempo inativo. O desenvolvedor pode utilizar duas formas para invalidar sessões:

- Explicitamente, adotando o método **void invalidate()**;
- Pela configuração do tempo máximo de inatividade antes de invalidar automaticamente a sessão. Nesse caso, pode-se usar o método **void setMaxInactiveInterval(int interval)** ou configurar esse tempo no Deployment Descriptor, como será demonstrado adiante.

Vamos conhecer, a seguir, a relação de métodos e utilitários para invalidar a sessão:

Método	Descrição
<b>invalidate():void</b>	Invalida a sessão e desassocia todos os objetos que possuía.
<b>setMaxInactiveInterval(interval:int): void</b>	Especifica o intervalo de tempo máximo que o cliente – neste caso, o Navegador – pode ficar inativo até que sessão seja invalidada no servidor. Esse valor é definido em segundos.
<b>getLastAccessedTime():long</b>	Retorna a representação em long da data da última requisição deste cliente.

Trecho do arquivo **web.xml** em que o timeout da sessão é configurado:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
7   id="WebApp_ID" version="3.0">
8 <welcome-file-list>
9   <welcome-file>index.html</welcome-file>
10 </welcome-file-list>
11
12 <session-config>
13   <session-timeout>10</session-timeout>
14 </session-config>
15
16 </web-app>
```

Quando a configuração for feita no Deployment Descriptor, deve-se atentar para inserir o tempo em minutos, não em segundos como na forma programática.

# Filtros

Neste tópico, vamos tratar dos componentes especiais chamados filtros. Vamos aprender a declarar e configurar um filtro, além de conhecer a interface `Filter`.

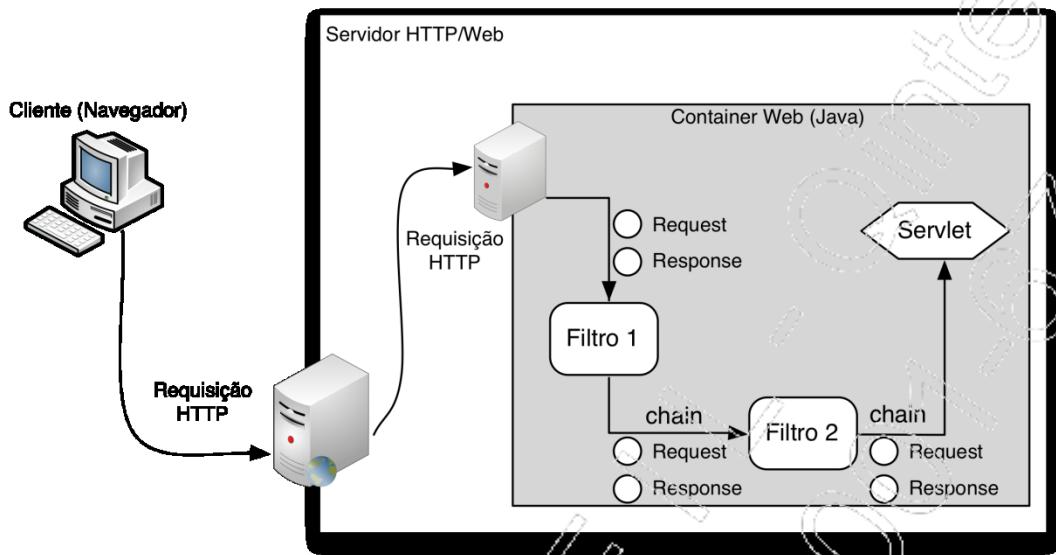
## O que são e para que servem os filtros

Filtros são componentes especiais introduzidos na versão 2.3 da API Servlet. Eles compõem um mecanismo de interceptação de Requests ou Responses, de forma que o desenvolvedor possa realizar ações nesses eventos. Efetivamente, os filtros interceptam e disponibilizam a request ou a response antes que seja encaminhada aos seus respectivos destinos: Servlet ou usuário.

O Container é quem gerencia a criação e extinção desses componentes, cabendo ao desenvolvedor apenas delinear suas aplicações e encaminhamentos de forma eficaz.

Filtros podem ser encadeados, o que é um recurso muito útil quando for necessário separar as funções de cada filtro aplicados a uma mesma Request.

Vejamos o mecanismo de atuação dos filtros:



Algumas aplicações possíveis e mais usuais para os filtros:

- **Autenticação:** É possível verificar se o usuário solicitante está autenticado ou não antes de encaminhar a request para o Servlet correspondente, tomando uma ação diversa quando necessário;
- **Log:** Caso haja interesse, um filtro pode ser usado para criar um log de todas as requests recebidas ou responses encaminhadas;
- **Validação:** Existe a possibilidade de validar dados provindos da request, como parâmetros, e encaminhar o usuário para diferentes Servlets, dependendo de suas escolhas;
- **Conexão:** É possível realizar uma conexão com o banco de dados assim que a request chegar e for repassada ao Servlet pelo desenvolvedor no código do filtro, já com a configuração de acesso ao banco de dados feita. Essa não é uma prática muito recomendada, mas é possível.

## A interface Filter

Para a criação de um filtro, basta que seja criada uma classe que implemente a interface **Filter**. Essa interface prevê a necessidade de implementação de três métodos, analisados a seguir:

Métodos	Descrição
<b>init(fc:FilterConfig):void throws ServletException</b>	Método chamado somente uma vez, quando o Container é iniciado.
<b>destroy():void</b>	Método executado somente uma vez, quando o Container é desativado.
<b>doFilter(request:ServletRequest, response:ServletResponse, chain:FilterChain):void</b>	Método principal onde reside a lógica do filtro. Chamado a cada interceptação feita.

## Declarando e configurando um filtro

Assim como os Servlets, os filtros devem ser declarados no **web.xml** para que sejam mapeados e controlados pelo Container, ou ainda podem ter sua declaração feita por meio de anotações nas classes e implementação:

```
@WebFilter(...parâmetros...)
```

No Deployment Descriptor, um conjunto de atributos é usado para definir os filtros, de forma bem semelhante aos Servlets. Vamos conhecê-lo a seguir:

Tag	Descrição
<filter>	Tag raiz que abre o bloco de declaração para cada filtro da aplicação.
<filter-name>	Nome dado ao filtro que serve somente para referenciá-lo no próprio <b>web.xml</b> . Esse nome será usado como referência para o filtro na tag <filter-mapping>.
<filter-class>	Nome totalmente qualificado da classe que representa o respectivo filtro. Utiliza a mesma sintaxe que o filtro utilizado para importar classes em Java.
<filter-mapping>	Tag empregada para mapear um determinado filtro a um padrão de URL, utilizando para tanto, a tag <filter-name> que identifica o filtro específico.
<url-pattern> ou <servlet-name>	Tag usada dentro da anterior para mapear um padrão de URL ou um servlet de destino/origem a ser atendido por um determinado filtro.
<dispatcher>	Tag usada para identificar quais origens de solicitações deverão ser interceptadas. Possibilidades: REQUEST, FORWARD, INCLUDE e ERROR.

Um exemplo do trecho de declaração para um filtro dentro do Deployment Descriptor:

```

16<filter>
17  <display-name>MeuFiltro</display-name>
18  <filter-name>MeuFiltro</filter-name>
19  <filter-class>filter.Meufiltro</filter-class>
20</filter>
21<filter-mapping>
22  <filter-name>MeuFiltro</filter-name>
23  <url-pattern>/*</url-pattern>
24  <dispatcher>REQUEST</dispatcher>
25  <dispatcher>FORWARD</dispatcher>
26</filter-mapping>

```

Os atributos dispatcher podem ser mais de um e determinam os tipos de comunicação a serem interceptadas pelo filtro. Nesse caso, todas as requisições e Encaminhamentos serão interceptados.

A seguir, temos o código de um filtro simples, configurado com o uso de anotações, que intercepta todas as requisições e forwards e imprime no console do servidor frases diferentes conforme seus eventos ocorram:

```

1 package filter;
2
3 import java.io.IOException;
4 import java.util.Date;
5 import javax.servlet.DispatcherType;
6 import javax.servlet.Filter;
7 import javax.servlet.FilterChain;
8 import javax.servlet.FilterConfig;
9 import javax.servlet.ServletException;
10 import javax.servlet.ServletRequest;
11 import javax.servlet.ServletResponse;
12 import javax.servlet.annotation.WebFilter;
13 import javax.servlet.http.HttpServletRequest;
14
15 @WebFilter(dispatcherTypes = { DispatcherType.REQUEST, DispatcherType.FORWARD },
16     urlPatterns = { "/" })
17 public class MeuFiltro implements Filter {
18
19     public void destroy() {
20         System.out.println(new Date() + " - Filtro sendo destruído!");
21     }
22
23     public void doFilter(ServletRequest request, ServletResponse response,
24             FilterChain chain) throws IOException, ServletException {
25
26         System.out.println(new Date() + "Filtro sendo executado após"
27             + " uma request!");
28         HttpServletRequest req = (HttpServletRequest) request;
29         System.out.println("URL interceptada pelo filtro: "
30             + req.getRequestURI());
31
32         // prossegue na chamada ao Servlet devido ou ao próximo filtro
33         chain.doFilter(request, response);
34     }
35
36     public void init(FilterConfig fConfig) throws ServletException {
37         System.out.println(new Date() + " - Filtro sendo criado!");
38     }
39
40 }
```

# Aplicação avançada de Servlets

Teste seus conhecimentos



**1. Qual das alternativas completa adequadamente a afirmação a seguir?**

**Em uma aplicação Web, o uso de atributos é essencial tendo em vista que o mecanismo de passagem de parâmetros**

- a) permite apenas a passagem e a captura de objetos genéricos.
- b) possui características de segurança incompatíveis com a maioria dos componentes internos do Web Container.
- c) facilita apenas a passagem de pequenos blocos de informação.
- d) realiza a comunicação entre componentes no mesmo servidor com eficiência, porém, toma grande parcela de memória em uso.
- e) Nenhuma das alternativas anteriores está correta.

**2. O que podemos afirmar quanto aos escopos definidos em um Java Web Container?**

- a) O menor deles é o contexto de sessão, usado para capturar cada solicitação do usuário.
- b) Na maior parte das vezes, o contexto de sessão deve ser substituído pelo contexto de aplicação, pois a garantia de persistir o dado por mais tempo é maior e mais segura.
- c) É uma boa prática salvar um atributo de sessão no contexto de requisição para garantir boa performance sem perda de dados.
- d) O contexto de aplicação persiste por inúmeras sessões de usuários e representa um estado que se mantém por toda a vida da aplicação, sendo renovado apenas quando o Container é desativado e ativado novamente.
- e) Mover objetos entre contextos nunca é considerado boa prática.

**3. Qual das alternativas a seguir está correta sobre os cookies presentes em uma aplicação Web?**

- a) Representam uma solução muito boa para a persistência de dados no lado do cliente, sendo muito utilizados para fazer cache de páginas visitadas anteriormente.
- b) Guardam texto sem criptografia e sem segurança, sendo acessíveis a qualquer usuário na máquina do cliente e, por esse motivo, inviáveis para dados sigilosos.
- c) É possível reabilitar o uso de cookies pelo navegador do cliente por via programática, usando objetos da API Java EE 6.
- d) São a única forma de guardar o estado de uma sessão na máquina do cliente e recuperá-la quando ele retornar à aplicação Web.
- e) Nenhuma das alternativas anteriores está correta.

**4. O que ocorre ao obtermos a referência para uma sessão, em um método do Servlet, por meio do comando HttpSession session = request.getSession();?**

- a) O container pode retornar null, caso não exista uma sessão criada para esse cliente.
- b) Sempre obteremos um objeto válido de sessão, mas devemos realizar casting do objeto retornado de forma a não obter erro pelo compilador.
- c) Ele pode ser substituído por um método semelhante no objeto ServletContext, que retorna uma sessão.
- d) Não deve ser usado quando detectar-se que o navegador do cliente está configurado para não receber Cookies.
- e) Retorna uma referência para a sessão existente do cliente em questão e, caso ela não exista, cria uma nova e retorna essa nova referência.

**5. Qual(is) das afirmações a seguir apresenta(m) somente aplicações para filtros válidas em aplicações Java para Web?**

- I. Autenticação do usuário solicitante na aplicação.**
  - II. Invalidação de sessão aberta para o usuário solicitante.**
  - III. Geração de registro de Log em um local estratégico no servidor.**
- 
- a) Somente a afirmação I está correta.
  - b) As afirmações I e II estão corretas.
  - c) Somente a afirmação III está correta.
  - d) As afirmações I e III estão corretas.
  - e) Nenhuma das alternativas anteriores está correta.

# Aplicação avançada de Servlets

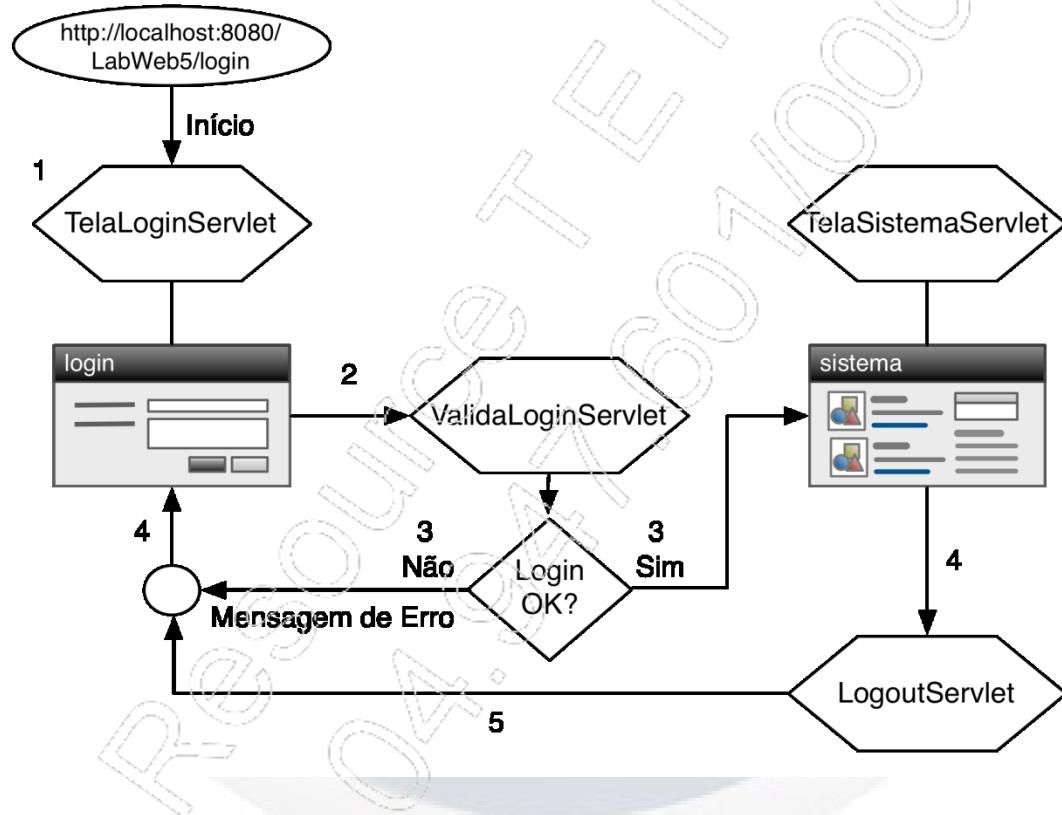
Mãos à obra!



## Laboratório 1

### A - Criando página de login, capturando, validando e compartilhando dados no Web Container

Neste laboratório, será criada uma aplicação de validação e invalidação de acesso de usuários em um contexto bem simples. Para entender o fluxo a ser desenvolvido nessa aplicação, observe o digrama a seguir. Ele apresenta o esquema de todas as etapas a serem realizadas neste exercício:



1. Crie um novo projeto do tipo **Dynamic Web Project** denominado **LabWeb5** no Eclipse. Em seguida, cheque a opção **Generate web.xml deployment descriptor**;
2. Crie um Servlet chamado **TelaLoginServlet**, que será responsável por montar a tela de login em seu método GET e capturar os dados no servidor.

Mapeie esse Servlet para atender requisições no padrão de URL "/login". O código do Servlet deverá ser de acordo com o exibido a seguir:

```
1 package servlet;
2
3④ import java.io.IOException;
4
5
6 @WebServlet("/login")
7 public class TelaLoginServlet extends HttpServlet {
8     private static final long serialVersionUID = 1L;
9
10
11     public TelaLoginServlet() {
12         super();
13     }
14
15     protected void doGet(HttpServletRequest request,
16             HttpServletResponse response) throws ServletException, IOException {
17         doPost(request, response);
18     }
19
20     protected void doPost(HttpServletRequest request,
21             HttpServletResponse response) throws ServletException, IOException {
22         response.setContentType("text/html");
23         PrintWriter out = response.getWriter();
24
25         String method = "POST";
26         String action = request.getContextPath() + "/validalogin";
27         String mensagem = (String)request.getAttribute("mensagem");
28         out.println("<html>");
29         out.println("<head>");
30         out.println("<title>Atenticação de usuário</title>");
31         out.println("</head>");
32         out.println("<body>");
33         out.println("<h3>Entre com seu usuário e senha: </h3>");
34         if (mensagem != null && !mensagem.equals("")) {
35             out.println("<h4>" + mensagem + "</h4>");
36         }
37         out.println("<form method=\"" + method + "\" action=\"" +
38             action + "\">\n");
39         out.println("<table> <tr>");
40         out.println("<td>Usuário: </td><td><input type=\"text\" " +
41             "name=\"usuario\" size=\"20\" > </td>\"");
42         out.println("</tr><tr>");
43         out.println("<td>Senha: </td><td><input type=\"password\" " +
44             "name=\"senha\" size=\"20\" > </td>\"");
45         out.println("</tr><tr>");
46         out.println("<td></td><td><input type=\"submit\" value=\"Enviar\">" +
47             "</td>\"");
48         out.println("</tr></table>");
49         out.println("</form>");
50         out.println("</body>");
51         out.println("</html>");
52         out.close();
53     }
54 }
55 }
```

Perceba que esse Servlet monta uma tela de Login por meio de um formulário HTTP e envia os dados coletados, via método POST, para o Servlet registrado como ouvinte do padrão de URL "/validaLogin". Note ainda que o atributo **mensagem** da request está sendo capturado nesse Servlet e, caso ele exista, será adicionado à página como forma de atualizar o usuário sobre algum resultado do login. Esse atributo será inserido pelo Servlet responsável pela validação dos dados.

3. Crie o Servlet **ValidaLoginServlet**, que atenderá a requisições no padrão "/validaLogin". Esse Servlet deverá conter o seguinte código:

```
1 package servlet;
2
3 import java.io.IOException;
4
5 @WebServlet("/validaLogin")
6 public class ValidaLoginServlet extends HttpServlet {
7     private static final long serialVersionUID = 1L;
8
9     public ValidaLoginServlet() {
10         super();
11     }
12
13     protected void doPost(HttpServletRequest request,
14                         HttpServletResponse response) throws ServletException, IOException {
15
16         String usuario = request.getParameter("usuario");
17         String senha = request.getParameter("senha");
18         String urlRedirecionamento = "/login";
19         String mensagem = null;
20
21
22         if (usuario.equals("") || usuario == null) {
23             mensagem = "Usuário inválido!";
24         } else if (senha.equals("") || senha == null) {
25             mensagem = "Senha inválida!";
26         } else if (usuario.equals("aluno") && senha.equals("impacta123")) {
27             mensagem = "Usuário autenticado em nosso sistema!";
28             Usuario user = new Usuario(usuario, senha);
29             request.getSession().setAttribute("usuario", user);
30             urlRedirecionamento = "/sistema";
31         } else {
32             mensagem = "Dados de acesso inválidos!";
33         }
34
35         request.setAttribute("mensagem", mensagem);
36         RequestDispatcher dispatcher = request.getRequestDispatcher(urlRedirecionamento);
37         dispatcher.forward(request, response);
38     }
39
40 }
```

Nesse Servlet reside a lógica de validação do usuário digitado e a montagem do mecanismo de mensagem de erro. O usuário esperado para a aplicação é **aluno** e a senha é **impacta123**. Caso sejam digitados valores diferentes desses ou os campos sejam deixados em branco, uma mensagem de erro específica será criada. Posteriormente, essa mensagem será inserida como um atributo no objeto Request.

O redirecionamento do fluxo também é condicional: em caso de autenticação falha, o usuário é direcionado novamente à página de login, porém, dessa vez uma mensagem será exibida. Em caso de sucesso, o usuário é direcionado à página de sistema ("**/sistema**"), cujo Servlet correspondente será criado em breve.

Quando a autenticação é feita com sucesso, um objeto de tipo **Usuario** é criado e armazenado como atributo na Sessão, de forma a ser utilizado posteriormente. Vejamos a classe que o define.

4. Crie a classe **Usuario** dentro de um pacote denominado **entidades**, conforme segue, para representar um usuário no sistema. Essa é uma classe plana, contendo dois atributos encapsulados, seus respectivos getters e setters e um construtor;

```
1 package entidades;
2
3 public class Usuario {
4
5     private String usuario;
6     private String senha;
7
8     public Usuario(String usuario, String senha) {
9         super();
10        this.usuario = usuario;
11        this.senha = senha;
12    }
13
14    public String getUsuario() {
15        return usuario;
16    }
17    public void setUsuario(String usuario) {
18        this.usuario = usuario;
19    }
20    public String getSenha() {
21        return senha;
22    }
23    public void setSenha(String senha) {
24        this.senha = senha;
25    }
26 }
```

5. Crie o Servlet **TelaSistemaServlet**, que representará a tela inicial de acesso ao sistema após a autenticação ter sido feita com sucesso. Será mapeado para o padrão **"/sistema"** e deverá possuir o seguinte código:

```
1 package servlet;
2
3+ import java.io.IOException;
13
14 @WebServlet("/sistema")
15 public class TelaSistemaServlet extends HttpServlet {
16     private static final long serialVersionUID = 1L;
17
18     public TelaSistemaServlet() {
19         super();
20     }
21
22     protected void doGet(HttpServletRequest request,
23             HttpServletResponse response) throws ServletException, IOException {
24         doPost(request, response);
25     }
26
27     protected void doPost(HttpServletRequest request,
28             HttpServletResponse response) throws ServletException, IOException {
29
30         response.setContentType("text/html;charset=UTF-8");
31
32         Usuario user = (Usuario)request.getSession().getAttribute("usuario");
33
34         PrintWriter out = response.getWriter();
35         out.println("<html>");
36         out.println("<head>");
37         out.println("<title>Sistema de Cadastro</title>");
38         out.println("</head>");
39         out.println("<body>");
40         out.println("<h1>" + request.getAttribute("mensagem") + "</h1>");
41         out.println("Bem vindo ao sistema, Sr./Sra. " + user.getUsuario());
42         out.println("<br /><br /><br /><br />");
43         out.println("<a href='logout'>Sair do Sistema</a>");
44         out.println("</body>");
45         out.println("</html>");
46         out.close();
47     }
48 }
```

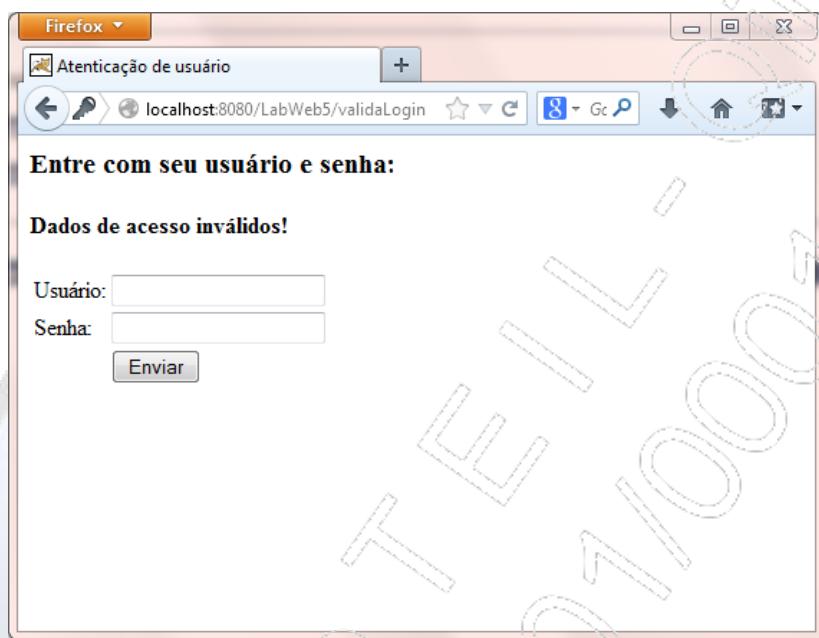
Nesse Servlet, estamos recuperando o objeto **Usuario**, salvo na Sessão no momento da autenticação, para utilizá-lo mais adiante na impressão da frase de boas-vindas.

Ainda nessa página, imprimimos um link de Logout, apontando para o padrão de URL **"/logout"**. Esse Servlet, responsável pela saída do usuário do sistema, será criado em seguida.

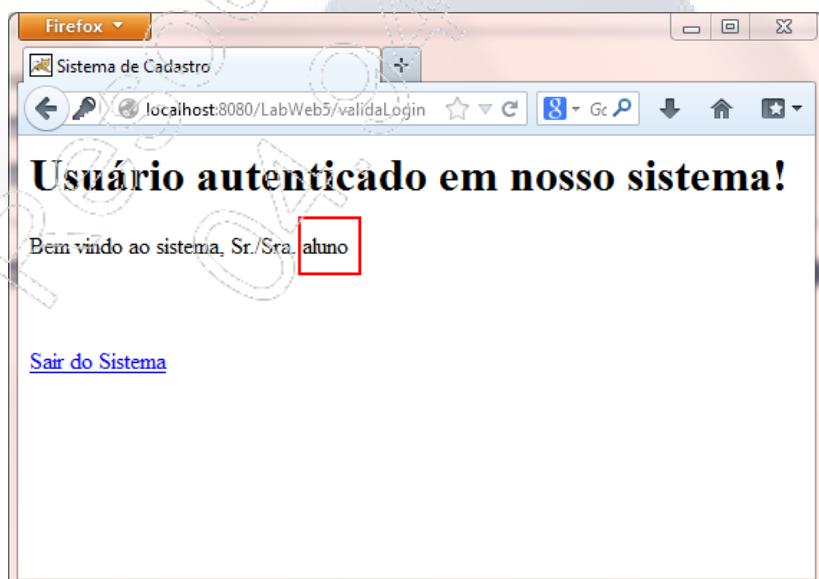
6. Crie o Servlet denominado **LogoutServlet**, que será responsável pela invalidação da sessão do usuário e redirecionamento para a página inicial. Seu código é bem simples:

```
1 package servlet;
2
3+ import java.io.IOException;
10
11 @.WebServlet("/logout")
12 public class LogoutServlet extends HttpServlet {
13     private static final long serialVersionUID = 1L;
14
15+     public LogoutServlet() {
16         super();
17     }
18
19+     protected void doGet(HttpServletRequest request,
20             HttpServletResponse response) throws ServletException, IOException {
21         doPost(request, response);
22     }
23
24+     protected void doPost(HttpServletRequest request,
25             HttpServletResponse response) throws ServletException, IOException {
26         HttpSession session = request.getSession(false);
27         if (session != null) {
28             session.invalidate();
29         }
30         response.sendRedirect(request.getContextPath() + "/login");
31     }
32
33 }
```

7. Execute a aplicação e teste-a por meio da URL **http://localhost:8080/LabWeb5/login**. Insira valores inválidos de usuário e senha e experimente deixar os campos em branco para testar as mensagens de validação. A seguinte mensagem aparecerá na parte superior do formulário:



8. Insira os valores esperados de usuário e senha (**aluno** e **impacta123**) e clique em **Enviar**. A seguinte tela de sistema deverá aparecer:



Note que o usuário, nesse caso, **aluno** foi recuperado com sucesso do objeto de tipo **Usuario**, colocado em sessão anteriormente.

Em seguida, clique no link **Sair do Sistema**. Verifique que a sessão passa a ser invalidada e que o usuário é redirecionado para a tela inicial de login.

# Capítulo 6: Introdução às JSPs

# A necessidade das JSPs e sua inserção no Projeto Web Dinâmico

Os Servlets representaram uma grande mudança na forma de tratar requisições Web e na criação de páginas dinâmicas. Ainda são extensamente usados em diversas aplicações, websites e serviços remotos.

Conforme sua aplicação foi aumentando, um problema começou a surgir dado o constante acréscimo do número de páginas e da complexidade das aplicações: nos Servlets, o código HTML é gerado dentro de classes Java. Não é preciso mencionar o grande trabalho para lidar com todas as aspas simples e duplas, além de outros símbolos diversos dentro de Strings Java. Em pouco tempo, uma página HTML de média complexidade pode se tornar um grande desafio para a geração, manutenção e extensão de código de marcação encapsulados em Strings Java.

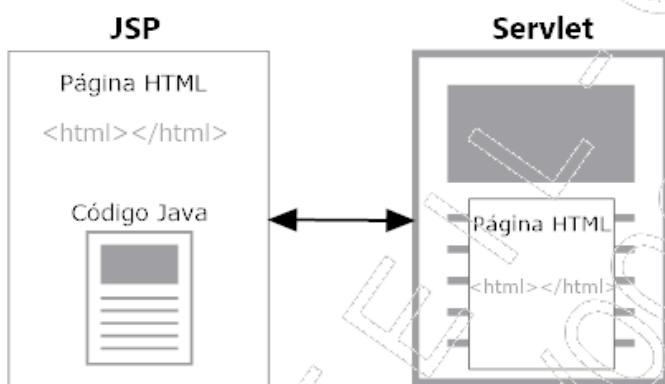
Um outro problema decorrente desse último é a integração entre Web designers e outros profissionais relacionados ao ecossistema Web, que não conhecem os aspectos técnicos de uma aplicação Java Web. Seria de difícil compreensão, para profissionais não familiarizados com a linguagem Java, a análise de um código HTML gerado dentro de Servlets.

A saída mais adequada para esse problema seria algo que possibilitasse o caminho inverso: a criação de páginas HTML com código Java embutido. Dessa forma, diminui-se a dependência da sintaxe Java para a compreensão de marcação estrutural e estilização de páginas.

Esta solução, ainda baseada na tecnologia Servlet, é chamada de Java Server Pages, que consiste na criação de páginas HTML comuns com trechos de código Java ou marcação especializada embutida.

A Java Server Pages não só possibilitou uma melhor integração entre profissionais de diversas áreas em um mesmo projeto Web, mas também permitiu a criação de aplicações mais organizadas, focadas na separação de responsabilidade (arquitetura MVC, por exemplo) e na robustez exigida por aplicações de grande escopo.

Visualmente, temos o seguinte esquema:



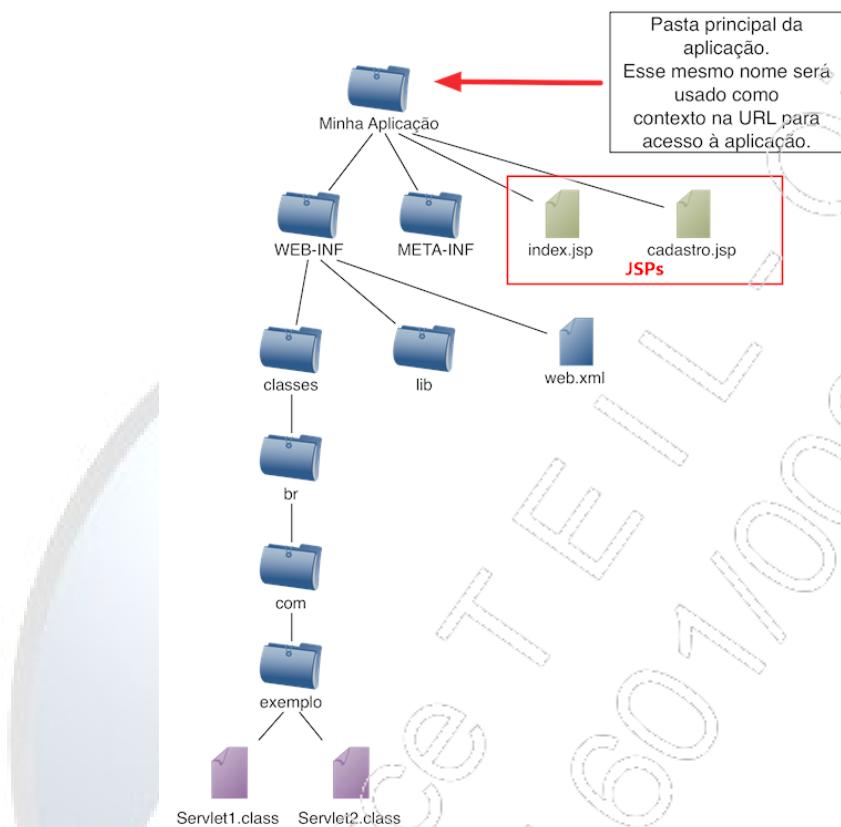
As JSPs tornaram possível a criação de páginas por meio da utilização de ferramentas especializadas nesse quesito, o que permitiu ao desenvolvedor, posteriormente, acrescentar somente os trechos de código responsáveis pelo comportamento dinâmico da página.

Quando executada, a JSP criada transforma-se em um Servlet pelo Web Container e é compilada. A tecnologia Servlet continua sendo totalmente aplicada, porém, o código agora é gerado pelo Container, permitindo que o foco do desenvolvimento seja aplicado em um nível mais elevado, propiciando a interação entre design e codificação de forma transparente e independente.

## As JSPs no contexto do projeto Web Dinâmico

Em um Projeto Web Dinâmico, as JSPs são alocadas juntamente com as demais páginas Web, caso existam, diretamente na pasta raiz do projeto. Essa estrutura é encapsulada em um arquivo do tipo **.war**, conforme visto anteriormente, para efetiva implantação em um servidor de aplicações Java.

Utilizando o esquema da estrutura de um arquivo **war**, já apresentado anteriormente, destacamos agora a posição das JSPs na árvore de diretórios:



Conforme apontado nesse esquema, as JSPs podem ocupar o mesmo lugar que as demais páginas Web existentes no projeto, contudo, é comum utilizarem uma pasta denominada **jsp** para armazenar as páginas e, dessa forma, separá-las das demais pastas de conteúdo da aplicação Web.

Estruturalmente, as JSPs nada mais são do que páginas HTML (com algumas marcações específicas) e extensão de arquivo **.jsp**.

# Elementos de Java Server Pages

Neste tópico, conhceremos todos os elementos dinâmicos utilizados em páginas JSP, suas características, funções e sintaxe. Cada elemento possui uma sintaxe própria, que é interpretada pelo Container na hora de gerar o HTML final de retorno ao cliente.

## Scriptlets

Scriptlets são blocos de código Java que podem ser alocados em qualquer ponto da página e em qualquer quantidade necessária. É sugerido o uso limitado de Scriptlets de forma a não tornar a integração entre sintaxe Java e HTML algo muito complexo de ser lido.

Sua sintaxe utiliza o par `<% ...Código Java... %>`. Por exemplo:

```
11<h1>
12    <% out.println("Teste com JSPs!"); %>
13 </h1>
```

No código apresentado, a string “**Teste com JSPs!**” será impressa na página dentro de tags `<h1></h1>`.

É importante ressaltar que, nesse caso, estamos utilizando o objeto **out** sem nem mesmo tê-lo declarado. Isso é possível porque, em toda JSP que for utilizada, alguns objetos implícitos estarão disponíveis para imediata aplicação no código. Um deles é o objeto **out**, já conhecido e utilizado anteriormente. Este e os demais objetos implícitos serão vistos adiante neste capítulo.

## Comentários

Dois tipos de comentários são possíveis em uma JSP: os comentários nativos do HTML, que usam a notação padrão de comentários em código de marcação:

```
15 <!-- Comentário nativo HTML -->
```

E os comentários específicos de notação JSP, que aderem à sintaxe:

```
17 <%-- Comentário específico JSP --%>
```

A principal diferença entre as opções apresentadas é que a primeira é ignorada pelo browser, porém, visível ao cliente juntamente com o código-fonte da página. A segunda notação, específica do JSP, não é inserida no código-fonte enviado ao cliente. Na verdade, ela é ignorada pelo compilador Java e permanece oculta à saída.

## Expressões

As expressões constituem um atalho para impressão na página de saída enviada ao cliente.

Dada a ampla e necessária utilização dos comandos `out.print(...)` e `out.println(...)`, ambos provindos de um objeto derivado de `PrintWriter`, a especificação JSP prevê uma sintaxe especial como forma de abreviar e facilitar a utilização de impressão na construção de páginas Web.

As expressões possuem a seguinte sintaxe:

```
<%=variável ou valor>
```

A seguir, temos um exemplo prático da aplicação de expressões para impressão de uma variável Java:

```

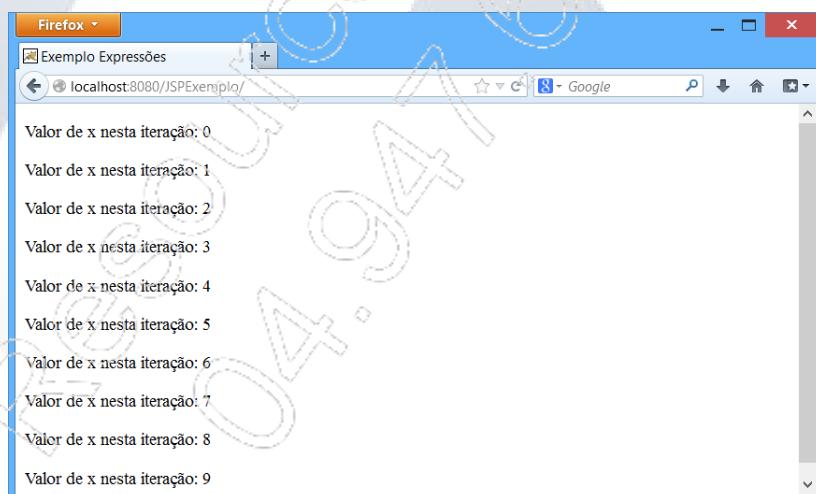
11 <%-- Impressão usando expressões em um laço for --%>
12
13 <% for (int x = 0; x < 10; x++) { %>
14     <p>Valor de x nesta iteração: <%=x %> </p>
15 <% } %>

```

No laço anterior, note que a expressão foi circundada por dois blocos Scriptlet que guardam o código Java embutido na página HTML. Perceba ainda a utilização intercalada da tag `<p>...</p>` com a expressão de forma transparente.

A expressão indicada equivale a um scriptlet `<% out.print(x) %>` no local em que foi inserida.

A saída produzida é:



## Declarações

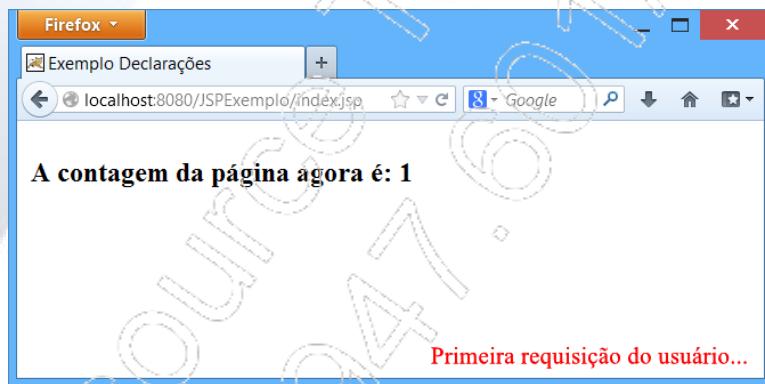
Vimos o uso de Scriptlets Java em uma página HTML como forma de inserir blocos de código Java onde quer que se deseje acrescentar conteúdo dinâmico, sem a preocupação com o real e maior escopo por trás de toda a definição dos Scriptlets.

Para toda JSP criada, o Web Container cria um Servlet em background, utilizado para compilação e atendimento de requisições à respectiva página. A forma de criação desse Servlet varia de servidor para servidor e não atende a um caminho ou formato específico.

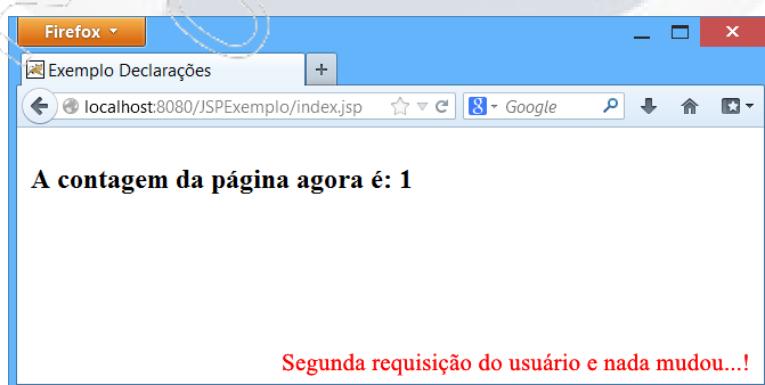
Utilizando Scriptlets, seria possível supor qual a saída do seguinte trecho de código?

```
11 <% int contador = 0; %>
12 <h4>
13     A contagem da página agora é:
14     <%= ++contador %>
15 </h4>
```

Confira a saída:



Agora, em uma segunda requisição:



Se considerarmos que o trecho de código anterior primeiramente declara uma variável e, em seguida, incrementa-a, era de se esperar que a contagem estivesse em 2.

Ocorre que, ao utilizarmos Scriptlets e expressões, estamos inserindo código dentro de um método de serviço do Servlet gerado, chamado a cada requisição sem armazenamento de estado. Inspecionando o código desse Servlet no Apache Tomcat, obtemos algo que à primeira vista parece muito complexo. Retirando o código que não nos interessa nesse momento, ficamos com o seguinte trecho a ser analisado:

```

1④ import java.io.IOException;
2
3 public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
4     implements org.apache.jasper.runtime.JspSourceDependent {
5
6     private static final long serialVersionUID = 1L;
7
8     @Override
9     public Map<String, Long> getDependents() {
10         //.. código gerado pelo Container...
11         return null;
12     }
13
14     @Override
15     public void _jspService(HttpServletRequest request, HttpServletResponse response)
16         throws ServletException, IOException {
17         PrintWriter out = response.getWriter();
18         //... código de montagem da página...
19         int contador = 0; ←
20         out.write("\r\n");
21         out.write("<h4>\r\n");
22         out.write("A contagem da página agora é: \r\n");
23         out.write("\r\n");
24         out.print( ++contador ); ←
25         out.write("\r\n");
26         out.write("</h4>\r\n");
27         //... código de fechamento da página...
28     }
29
30 }
31
32 }
```

Essa classe representa o Servlet gerado pelo Apache Tomcat 7 para nossa página **index.jsp**. É possível localizar esse código no seguinte endereço, que varia conforme o Container utilizado: **C:\<caminho da pasta de instalação do Apache Tomcat>\work\Catalina\localhost\ <nome do projeto criado>\org\apache\jsp\<nome da página>\_jsp.java**.

É possível perceber, nas linhas indicadas pelas setas, que o código inserido na JSP acabou tendo escopo local ao método de serviço.

Recorremos às declarações quando é preciso inseri-las em atributos de instâncias, além de declarar métodos e quaisquer outros membros no corpo do Servlet. Elas garantem que o código gerado possuirá escopo de instância, perdurando entre requisições distintas.

As declarações possuem a seguinte sintaxe:

```
<%! ...declarações... %>
```

Note o uso de exclamação no início do bloco.

Alterando o exemplo dado anteriormente, de forma a usar declarações no código:

```
11 <%! int contador = 0; %> ←  
12<h4>  
13     A contagem da página agora é:  
14     <%= ++contador %>  
15 </h4>
```

O efeito de contagem entre as requisições agora seria atingido a contento.

# Diretivas

São instruções de configuração da JSP em geral usadas para importar classes, configurar o tipo de saída, incluir componentes ou outros arquivos, entre outras funcionalidades. A seguir, temos uma relação dos três tipos de diretivas e seus principais atributos:

- **Diretivas Page**

É a diretiva com a maior variedade de aplicações: possui onze atributos diferentes. A seguir, temos a sintaxe de declaração da diretiva **page**:

```
<%@ page atributo1="valor1" atributo2="valor2" ... %>
```

Note o uso de @ após a abertura do bloco.

Dada sua vasta abrangência, pode ocorrer mais de uma vez na mesma página JSP.

- **Atributo Info**

É utilizado quando se deseja inserir informações sobre a página de forma resumida. Este atributo não possui restrições de tamanho. Por exemplo:

```
<%@ page info="Estudo sobre Diretivas" %>
```

- **Atributo Language**

Atributo empregado para definir a linguagem que será usada na criação de script para a página, no caso, a linguagem Java. Por exemplo:

```
<%@ page language="java" %>
```

- **Atributo ContentType**

Especifica o tipo MIME (Multipurpose Internet Mail Extensions) da resposta que a JSP está fornecendo.

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

- **Atributo Extends**

Determina a superclasse que o container utilizará quando a página estiver sendo traduzida em um Servlet Java.

```
<%@ page extends="com.taglib.jsp.primeirapagina" %>
```

- **Atributo Import**

Um dos atributos mais usuais e importantes da diretiva page. Com o atributo **import**, é possível importar um conjunto de classes Java que poderão ser usadas nas páginas JSPs. Equivale aos comandos import usados em classes Java usuais.

```
<%@ page import="java.util.List" %>
```

- **Atributo Session**

Informa se a página faz parte do gerenciamento de sessão. Este atributo é do tipo boolean.

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

- **Atributo Buffer**

Monitora a saída “bufferizada” para uma página JSP. Utiliza-se kilobytes para especificar o tamanho do buffer. Para passar o conteúdo de uma JSP para a resposta HTTP de forma rápida, este atributo deve ser definido como “none”.

```
<%@ page buffer="12kb" %>  
ou  
<%@ page buffer="none" %>
```

- **Atributo AutoFlush**

Possui a mesma função do atributo Buffer, porém, monitora especificamente como o container JSP reage quando o buffer de saída está cheio. Quando isso acontece, todo o conteúdo desse buffer de saída é removido e, em seguida, enviado ao servidor HTTP. Este, por sua vez, leva o conteúdo até o Browser.

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

- **Atributo isThreadSafe**

Para que uma página JSP compilada em Servlet seja capaz de responder a várias solicitações, deve-se configurar o atributo **isThreadSafe** como true. Caso contrário, pode-se defini-lo como false.

```
<%@ page isThreadSafe="false" %>
```

- **Atributo errorPage**

Permite definir uma nova página para ser mostrada quando ocorrer um erro inesperado no processamento de uma página JSP no container.

```
<%@ page errorPage="/trabalho/paginas/erro.jsp" %>
```

- **Atributo isErrorPage**

Especifica uma página JSP como página de erro a ser utilizada como padrão pelo sistema Web em Java ou por um grupo de páginas JSP.

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

- **Diretiva Include**

Insere o conteúdo de um arquivo (página) em outro (página). Esse conteúdo, que pode ser ilimitado, toma o lugar da diretiva.

```
<%@ include file="URL do Arquivo" %>  
ou  
<jsp:directive.include file="URL do Arquivo" />
```

- **Diretiva Taglib**

Se esta diretiva for usada em uma página, um conjunto de tags personalizadas será disponibilizado pela página.

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

## Objetos implícitos

A API aplicada aos JSPs disponibiliza uma série de objetos muito úteis ao desenvolvimento Web e de fácil acesso ao desenvolvedor. Esses objetos representam os contextos da aplicação, o Stream de impressão na saída ao requisitante e outros objetos de conveniência. Eles são instanciados por default, retirando essa responsabilidade do desenvolvedor.

Segue a lista de objetos implícitos nas JSPs:

- **request:** Objeto que implementa a interface `javax.servlet.http.HttpServletRequest`. Já conhecido por ser largamente utilizado nos métodos de serviços dos Servlets, especialmente nos métodos `doGet(...)` e `doPost(...)`. Como se trata do mesmo objeto, dispõe de todos os métodos usuais: `request.getAttribute()`, `request.getParameter()`, etc;
- **response:** Objeto que implementa a interface `javax.servlet.http.HttpServletResponse`. Já conhecido por ser amplamente utilizado nos métodos de serviços dos Servlets, especialmente os métodos `doGet(...)` e `doPost(...)`, assim como descrito no item anterior;
- **out:** Representa a instância da classe `java.servlet.jsp.JspWriter`, que é subclasse da classe abstrata `Writer`, da qual também herda `PrintWriter`, usado nos exemplos com Servlets. É usada para imprimir caracteres na página;
- **session:** Objeto que representa uma instância da classe `javax.servlet.http.HttpSession`, de forma a permitir a manipulação da sessão do usuário diretamente da JSP;
- **config:** Referência ao objeto da interface `javax.servlet.ServletConfig`, que designa a configuração do Servlet JSP ajustado no deployment descriptor (`web.xml`). Pode ser utilizado para recuperar parâmetros de inicialização do Servlet, por exemplo;
- **application:** Referência ao objeto que implementa a interface `javax.servlet.ServletContext`, que reflete o contexto da aplicação mais amplo disponível;

- **page**: Objeto tipificado como **java.lang.Object** que representa a página atual. Fundamentalmente guarda informações da página corrente;
- **pageContext**: Referência ao objeto **javax.servlet.jsp.PageContext**, que oferece acesso a todos os escopos disponíveis no JSP e a diversos atributos úteis de página, como o **request** e **response** atual, o **ServletConfig**, **HttpSession** e **ServletContext** ;
- **exception**: Instância de objeto do tipo **java.lang.Throwable**, que indica uma exceção e estará disponível para utilização principalmente nas páginas de erros da aplicação.

# Introdução às JSPs

## Teste seus conhecimentos



**1. Qual das alternativas a seguir não está correta sobre as diversas facilidades oferecidas pelas JSPs?**

- a) Facilitaram a criação de código HTML, especialmente em páginas complexas.
- b) Otimizaram a integração entre profissionais de Design e desenvolvedores.
- c) Tornaram mais fácil a separação de responsabilidades em aplicações Java Web, facilitando, por exemplo, a implementação da arquitetura MVC.
- d) Possibilitaram aos desenvolvedores e designers a utilização de ferramentas especializadas em Web Design para criação de páginas.
- e) Tornaram mais ágil a transmissão de parâmetros via Request inseridos em formulários pelos usuários.

**2. Leia as afirmações a seguir sobre a sintaxe e a utilização de Scriptlets e assinale a alternativa correta:**

- I. Scriptlets são blocos de código Java inseridos dentro de uma página montada com linguagem de marcação.
- II. Scriptlets devem ser inseridos somente entre as tags <BODY></BODY> da página, tendo em vista que o header dessa página ainda não estaria montado no momento de sua compilação no servidor.
- III. O uso de Scriptlets em páginas JSP é recomendado e considera-se uma boa prática utilizá-los em grande quantidade para separar as responsabilidades de cada trecho de código, aderindo às bases da Programação Orientada a Objetos.

- a) Somente a afirmação I está correta.
- b) Somente as afirmações I e II estão corretas.
- c) Somente a afirmação III está correta.
- d) Somente as afirmações I e III estão corretas.
- e) Todas as afirmações estão corretas.

**3. Qual das alternativas a seguir está correta sobre o trecho de código a seguir, que foi inserido no corpo de uma página JSP?**

1. <%-- Declaração e utilização de variável --%>
2. <% String nome = "Impacta Java Web"; %>
3. <h3>Escola Responsável: <%=nome %> </h3>

- a) A declaração da variável nome na linha 2 somente poderia ser feita no interior de uma declaração.
- b) A sintaxe utilizada para o comentário utilizado na linha 1 pode expor dados indesejados no código de marcação da página gerada e enviada ao cliente.
- c) O uso da declaração na linha 3 evidencia que uma impressão da variável nome ocorrerá em algum ponto da tela.
- d) A variável nome declarada no Scriptlet da linha 2 terá escopo local ao método de serviço do Servlet gerado e será recriada a cada requisição para essa página.
- e) A expressão utilizada na linha 3 está sintaticamente incorreta.

**4. Qual das alternativas a seguir não está correta sobre declarações e diretivas em JSPs?**

- a) As declarações são blocos semelhantes aos Scriptlets, porém, atuam em um escopo mais amplo.
- b) As declarações são utilizadas quando existe a necessidade de se declarar campos ou métodos no corpo do Servlet gerado, obtendo, dessa forma, membros que persistem entre requisições à mesma página.
- c) As diretivas são blocos que permitem a inserção de código de configuração das JSPs e devem vir no cabeçalho da página Web.
- d) A page é a mais longa das diretivas e seus atributos mais comuns são ContentType e Import.
- e) A diretiva include é largamente utilizada para a montagem fragmentada de páginas JSP, facilitando a organização final.

**5. Qual das alternativas a seguir está correta sobre os objetos implícitos?**

- a) O objeto out representa uma instância de objeto derivado da classe abstrata StreamWriter.
- b) O objeto response não pode ser utilizado para se obter um Writer por meio do método getWriter(), antes usado com Servlets, dado que já existe um objeto implícito para essa finalidade.
- c) O objeto adequado para se obter os parâmetros de inicialização do contexto da aplicação é o pageContext.
- d) O objeto application é uma referência para o ServletContext da aplicação.
- e) Não é possível manipular a sessão de um usuário conectado por meio de objetos implícitos.

# Introdução às JSPs

Mãos à obra!



## Laboratório 1

### A – Analisando um sistema Web de cadastro e identificando as funcionalidades de JSP vistas até o momento

A partir deste laboratório, será utilizado o mesmo projeto nos laboratórios seguintes, até o término desta parte da apostila. Cada funcionalidade vista será implementada em um sistema de forma funcional. Ao final, um sistema real de cadastro terá sido construído, o qual poderá servir de base para aplicações concretas. Para manter o foco deste curso, nem todas as reais funcionalidades de um aplicativo desse porte serão abordadas. De qualquer forma, a partir do conhecimento que será adquirido, o aluno estará apto a estender e atribuir novas características ao sistema livremente.

1. Inicialize o Eclipse e localize o projeto chamado **LabWeb6\_Cadastro**. Copie-o para o workspace atual do Eclipse e importe-o seguindo as orientações fornecidas pelo instrutor;
2. Navegue pelo projeto e verifique seus componentes Java, as páginas JSP e o conteúdo Web separados por pastas: **css**, **images** e **js**. Essas pastas contêm, respectivamente, as folhas de estilo em cascata, as imagens usadas no projeto e os scripts Javascript inseridos nas páginas criadas;

Páginas HTML, CSS e scripts Javascript são todos itens concernentes à atividade de Web Design e não são foco de nosso curso. Caso haja interesse em conhecer sua natureza em detalhes, seu código pode ser explorado e estudado. O código fornecido é emblemático e serve de suporte para as aplicações a serem criadas no curso.

3. Confirme ainda, a existência de dois pacotes denominados **datasource** e **servlets**. O primeiro pacote contém duas classes de negócio, denominadas **Aluno** e **Dados**. O segundo pacote contém dois servlets utilizados: **ValidaServlet** e **SairServlet**;
4. Analise o código das classes de negócio e note que a classe **Dados** é responsável por fornecer dados válidos à nossa aplicação. Os dados são gerados fixamente via código, porém, seria perfeitamente possível e razoável que fosse utilizada uma fonte de dados estruturada como um banco de dados ou mesmo um arquivo XML ou JSON provindo de algum serviço Web. Para fins didáticos, a fonte atual serve aos nossos propósitos;
5. Analise o código dos dois servlets existentes e perceba que eles possuem apenas a estrutura codificada, sem lógica alguma, com comentários de instrução sobre as tarefas a serem realizadas. Esses itens serão abordados em tarefas ou laboratórios futuros;
6. Analise as páginas JSP **login.jsp** e **sistema.jsp**, procurando localizar os elementos vistos no capítulo: diretivas, expressões, scriptlets, comentários, etc.

## Laboratório 2

### A – Construindo e compreendendo a interação das JSP em um sistema Web de cadastro

1. Localize a página **login.jsp** e identifique o comentário JSP que abre a página. Crie uma diretiva page logo abaixo do comentário, de forma que a JSP fique adequadamente formatada. O seguinte código deve ser adicionado:

```
1 <%--Diretiva page - Lab 2 Tarefa 1 --%>
2 <%@ page language="java" contentType="text/html; charset=utf-8"
3     pageEncoding="utf-8"%>
```

2. No bloco **<head>** dessa mesma página, é necessário incluir uma diretiva include para anexar o arquivo **header.jsp**, presente no projeto, logo abaixo do item **<title>**. Localize-se pelo comentário adequado indicando a tarefa a ser feita. Insira a diretiva adequadamente, conforme o código a seguir:

```
5<head>
6     <title>Cadastro de Alunos - Login</title>
7     <%@include file="header.jsp" %>
8 </head>
```

3. Logo após a linha 48, localize o comentário relativo à tarefa 3. Nesse ponto, será inserido um bloco de código que imprimirá na tela uma mensagem de erro caso o login tenha falhado. Para tanto, insira o seguinte bloco de código logo abaixo do comentário:

```
50<
51     <%
52         String message = (String)request.getAttribute("message");
53         if (!(message == null || message.equals("")))) {
54             %>
55                 <br /> <br />
56                 <div class="error-box round">
57                     <%=message %>
58                 </div>
59             }
60         %>
```

4. Note que temos dois blocos scriptlet nesse trecho: o primeiro captura um atributo “**message**” passado pela requisição (e, portanto, obtido pelo objeto `request`) para essa página. Caso esse atributo não seja nulo ou vazio, ele imprime na tela uma mensagem de erro formatada. Perceba a integração entre o código Java e o HTML. Como temos um `if` nesse trecho, o segundo bloco scriptlet apenas fecha seu escopo. Todo o conteúdo HTML e Java existente entre os blocos são alocados dentro do escopo desse `if`. Dentro do `if`, estamos utilizando uma expressão JSP que imprime na tela o conteúdo da variável “**message**” obtida anteriormente;

5. Ainda na mesma página, localize a tag `<form ...>` que abre o formulário de login e note que ela declara como atributo `action` o caminho “**validaServlet**”. Esse caminho refere-se ao servlet **ValidaServlet**, existente, porém, ainda não implementado em nossa aplicação. O código atual do servlet é o seguinte:

```
1 package servlets;
2
3 import java.io.IOException;
4
5 @WebServlet("/validaServlet")
6 public class ValidaServlet extends HttpServlet {
7     private static final long serialVersionUID = 1L;
8
9
10    public ValidaServlet() {
11        super();
12    }
13
14    protected void doPost(HttpServletRequest request,
15                          HttpServletResponse response)
16                          throws ServletException, IOException {
17        //Passos para implementação do Servlet:
18        //Captura de parâmetros passados
19        //Validação dos parâmetros e configuração de mensagem de retorno
20        //Configuração da url para navegação conforme seja a validação:
21        //Caso seja com sucesso, encaminhamento para a página inicial do sistema
22        //Caso seja invalidada, encaminhamento para a tela de login com envio de
23        //mensagem de erro anexada como atributo.
24    }
25 }
```

6. Analise os requisitos descritos no corpo do método **doPost(...)** e passe à implementação desse método de modo a atender às necessidades de validação trazidas pelo login criado anteriormente. Insira o seguinte código ou equivalente no corpo do método **doPost(...)**:

```
String usuario = request.getParameter("login");
String senha = request.getParameter("senha");
String message = null;
String urlForward = "/login.jsp";

if (usuario == null || usuario.equals("")) {
    message = "Insira um usuário válido!";
} else if (senha == null || senha.equals("")) {
    message = "Insira uma senha válida!";
} else if (!(usuario.equals("aluno") & senha.equals("impacta123"))) {
    message = "Usuário ou senha digitados não conferem!";
} else {
    urlForward = "sistema.jsp";
}

request.setAttribute("message", message);
RequestDispatcher dispatcher = request.getRequestDispatcher(urlForward);
dispatcher.forward(request, response);
```

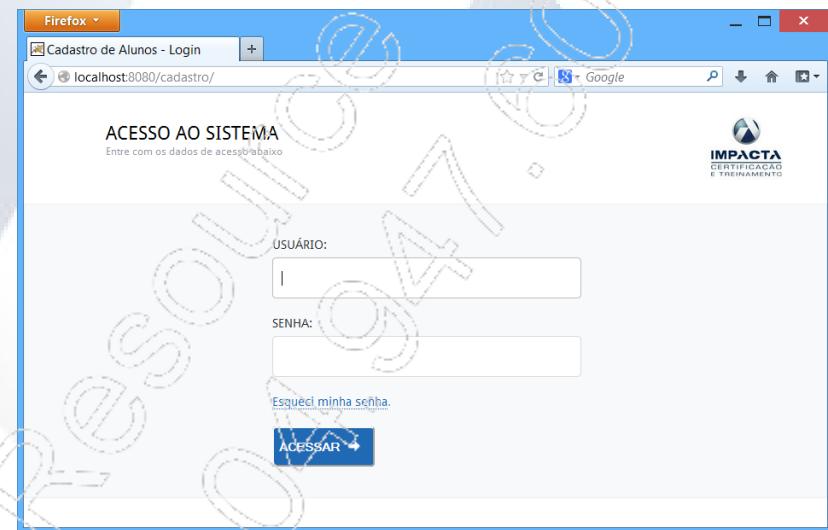
7. Analise a lógica do código anterior: estamos obtendo os parâmetros para usuário e senha passados pelo formulário na página de login para o servlet. Declaramos variáveis auxiliares para guardar a mensagem de erro, caso exista, e para guardar o destino do **Forward**, que faremos ao final. Posteriormente, ocorre uma série de verificações, um atributo que representa a mensagem de erro é inserido na **request** e um encaminhamento é feito para a página de login, em caso de erro, ou para a página de sistema, em caso de sucesso. O usuário e senha de acesso permanecem os mesmos usados em aplicações anteriores:

- Usuário: **aluno**;
- Senha: **impacta123**.

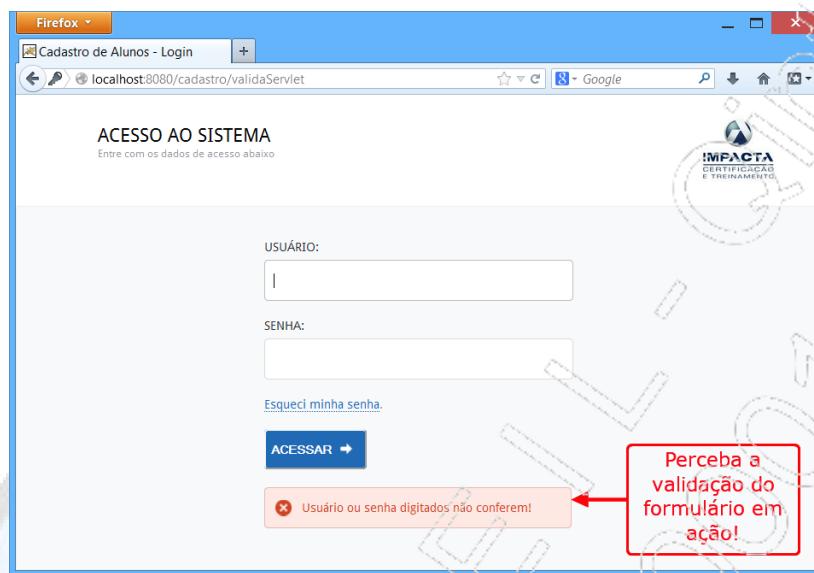
8. Altere o Deployment Descriptor para apontar à página **login.jsp** como página inicial na aplicação:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
6   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
7   id="WebApp_ID" version="3.0">
8     <display-name>LabWeb6_Completo</display-name>
9     <welcome-file-list>
10       <welcome-file>login.jsp</welcome-file> ←
11     </welcome-file-list>
12   </web-app>
```

9. Teste a aplicação. Apesar de ainda não estar totalmente funcional, já é possível testar a tela de login e sua validação. A seguinte tela inicial é apresentada:



10. Inserindo um usuário e uma senha quaisquer, a mensagem de erro é exibida logo abaixo do botão **Acessar**, em que colocamos o código Java.



# Capítulo 7: Acesso a dados pelas JSPs

# Leitura de dados recebidos de um formulário – GET e POST

Na parte destinada aos Servlets, vimos como capturar parâmetros passados por requisições GET e POST, provindos de formulários de dados HTML, e como tratar as informações no lado do servidor.

Como componentes Web, as Java Server Pages têm capacidades semelhantes às dos Servlets – na verdade, são definidas por Servlets em background – e características até mais facilitadas ao desenvolvedor, tendo em vista sua natureza de página HTML.

Uma requisição não precisa, necessariamente, passar por um Servlet antes de ser encaminhada a uma JSP. Ela pode ser diretamente direcionada à JSP responsável e lá ser tratada para montar a página de retorno ao usuário ou ainda encaminhar a requisição a outro componente.

A forma de se capturar os dados provindos da requisição é ainda mais fácil na JSP do que no Servlet propriamente dito. Utiliza-se o objeto implícito `request` para invocar o método `getParameter("...")`. Vejamos um exemplo:

```
<h3><%= request.getParameter("login") %></h3>
<h3><%= request.getParameter("senha") %></h3>
```

O trecho anterior apresenta duas expressões JSP que imprimirão na tela os valores dos parâmetros recuperados com as chaves `login` e `senha`. Vale notar o uso do objeto implícito `request`, criado e disponibilizado pelo container automaticamente. Esse código pode estar em qualquer lugar da página para ser usado e, independentemente do método utilizado pela requisição (GET ou POST), o objeto implícito cumpre em retornar os parâmetros recuperados.

# Escrita e leitura de atributos entre Servlets e JSPs

Em aplicações Web de maior importância, é comum a preocupação com a separação da lógica e individualização das responsabilidades funcionais do programa. É um grande problema, ainda que seja comum encontrar aplicações no mercado com essa característica, por exemplo, inserir código que trate de regras de negócio da aplicação em uma página JSP. As JSPs têm a fundamental missão de representar a camada de apresentação de uma aplicação Web e, por esse motivo, devem pautar-se por obter, moldar e exibir dados ao usuário. A lógica cabível a uma JSP deve ser aquela relacionada com a apresentação dos componentes da tela. Posteriormente, será abordado o padrão (ou a metodologia) MVC, muito comum e usado em aplicações Web das mais diversas naturezas e tecnologias.

Nesse contexto, é fundamental conhecer os mecanismos de passagem e recebimento de atributos entre os componentes de uma aplicação Web Java. Um Servlet deve ser responsabilizado por capturar requisições, decidir as ações a tomar, acionar objetos de negócio e repassar dados para exibição pela JSP. Nesse papel, a JSP limita-se a capturar os dados e exibi-los ao usuário. Vejamos como fazer essa comunicação entre os componentes.

Em uma aplicação de autenticação bem simples, temos a seguinte JSP, que representa o formulário de login:

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
2 pageEncoding="utf-8"%>
3
4 <!DOCTYPE html>
5 <html lang="pt">
6 <head>
7   <title>Cadastro de Alunos - Login</title>
8 </head>
9 <body>
10 <h3>Login de usuário</h3>
11 <form action="validaservlet" method="post">
12   <p>
13     Nome:<br/>
14     <input type="text" name="login"/>
15   </p>
16   <p>
17     Senha:<br/>
18     <input type="password" name="senha"/>
19   </p>
20   <input type="submit" value="Acessar"/>
21 </form>
22 </body>
23 </html>
```

Nessa página, basta chamar a atenção para o atributo **action** do formulário, que se refere a um Servlet já mapeado em nossa aplicação, o qual fará a captura dos parâmetros passados por POST.

Analisemos agora o código do Servlet que capturará essa requisição:

```

1 package servlets;
2
3@import java.io.IOException;
13
14 @WebServlet("/validaservlet")
15 public class ValidaServlet extends HttpServlet {
16     private static final long serialVersionUID = 1L;
17
18
19@     public ValidaServlet() {
20         super();
21     }
22
23@     protected void doPost(HttpServletRequest request,
24             HttpServletResponse response) throws ServletException, IOException {
25
26         String login = request.getParameter("login");
27         String senha = request.getParameter("senha");
28         HttpSession session = request.getSession(false);
29
30         Usuario user = new Usuario(login, senha);
31         session.setAttribute("usuario", user);
32
33         RequestDispatcher dispatcher = request.getRequestDispatcher("dados.jsp");
34         dispatcher.forward(request, response);
35     }
36
37 }
```

Vale a pena denotar nesse código as ações tomadas linha a linha. Nas linhas 26 e 27, o Servlet captura os parâmetros de login passados pela requisição. Na linha 28, cria-se uma nova sessão para esse usuário. As sessões são, sem dúvida, os objetos de escopo mais usados em aplicações Web.

Na linha 30, um novo objeto do tipo **Usuario** é instanciado para representar o usuário que acaba de se autenticar no sistema. A classe **Usuario** é bem simples e seu código será exposto em seguida. Na linha 31, esse objeto é inserido no escopo de sessão, onde estará disponível para qualquer outro componente da aplicação.

Por fim, o Servlet realiza um encaminhamento para a JSP **dados.jsp**, que será responsável por exibir os dados configurados na sessão a respeito do login.

A classe **Usuario** representa um usuário de forma bem simplificada, contendo apenas dois campos, um construtor, getters e setters.

```

1 package entities;
2
3 public class Usuario {
4
5     private String login;
6     private String senha;
7
8     public Usuario(String login, String senha) {
9         super();
10        this.login = login;
11        this.senha = senha;
12    }
13
14    public String getLogin() {
15        return login;
16    }
17    public void setLogin(String login) {
18        this.login = login;
19    }
20    public String getSenha() {
21        return senha;
22    }
23    public void setSenha(String senha) {
24        this.senha = senha;
25    }
26 }

```

A página **dados.jsp**, ao final, recebe o encaminhamento do Servlet e é retornada ao usuário. Este código é bem simples e direto, sendo responsável apenas por exibir os dados obtidos da sessão na tela para o usuário:

```

1 <%@ page language="java" contentType="text/html; charset=utf-8"
2     pageEncoding="utf-8" import="entities.Usuario"%>
3
4 <!DOCTYPE html>
5 <html lang="pt">
6 <head>
7     <title>Dados de login capturados</title>
8 </head>
9 <body>
10 <h3>Login de usuário</h3>
11
12 <%
13     Usuario usuario = (Usuario) session.getAttribute("usuario");
14 %>
15
16     <h4>Login capturado:</h4> <%=usuario.getLogin()%>
17     <h4>Senha capturada:</h4> <%=usuario.getSenha()%>
18
19 </body>
20 </html>

```

Vale notar a obtenção do objeto **Usuário** do escopo de sessão, seu casting obrigatório e, logo após, seu uso nas expressões apresentadas.

Da mesma forma apresentada para o compartilhamento de dados na sessão, pode-se estender o conceito para os escopos de requisição e aplicação, conforme a situação demande.

## JavaBeans e Standard Actions

Os JavaBeans são classes simples utilizadas como porções reutilizáveis em diversas partes do código da aplicação Java. Geralmente representam dados que circulam por entre as camadas de uma aplicação livremente.

A manipulação de JavaBeans em JSPs é algo muito comum, como podemos notar analisando nosso exemplo anterior: a classe **Usuario** é um excelente candidato a JavaBean. Ela apenas não o é por não cumprir com as exigências previstas para uma classe dessas, apesar de possuir a mesma função. São características obrigatórias de JavaBeans:

- Possuir o construtor default, também conhecido como no-args;
- Possuir atributos marcados como private e expor getters e setters para cada campo;
- Deve implementar a interface **java.io.Serializable**.

Grande quantidade de código em JSPs se faz necessária para as simples funções de instanciar, acessar e configurar propriedades em JavaBeans, dado que sempre será essencial a criação de novos objetos para trafegar entre os componentes da aplicação. Eles representam entidades e dados do domínio da aplicação.

Com o constante objetivo de tornar as JSPs mais limpas e com a menor quantidade de código Java possível, algumas tags especializadas estão disponíveis ao desenvolvedor. Elas visam automatizar as atividades de instanciação, obtenção e configuração de JavaBeans. Vamos conhecê-las a seguir:

- <jsp:useBean>;
- <jsp:setProperty>;
- <jsp:getProperty>.

## <jsp:useBean>

Esta tag é utilizada para criar uma nova instância de um JavaBean ou atribuir uma instância existente a uma nova referência. Nessa situação, sempre é chamado o construtor padrão do JavaBean. Sua sintaxe é a seguinte:

```
<jsp:useBean id="user" scope="session" class="entities.Usuario" />
```

A <jsp:useBean> referenciada cria uma nova instância da classe **entities.Usuario**, armazena a referência em uma variável denominada **user** e insere essa referência no contexto de sessão do usuário.

## <jsp:setProperty>

Esta tag é responsável por atribuir valores aos atributos do JavaBean criado. Geralmente é usada conjuntamente com a tag descrita anteriormente e depende primordialmente de que a sintaxe de encapsulamento dos campos na classe esteja feita de forma adequada (campos private com métodos assensores). Sua sintaxe é a seguinte:

```
<jsp:setProperty property="Login" name="user" param="Login"/>
<jsp:setProperty property="senha" name="user" param="senha"/>
```

Considerando o exemplo da tag descrita no item anterior, perceba o uso da referência atribuída na criação pela referência **id** e sua utilização aqui pelo atributo **name**.

O atributo **param** dessa tag obtém o valor de um parâmetro recebido da requisição atual e equivale ao seguinte código equivalente:

```
<%  
    user.setLogin(request.getParameter("login"));  
    user.setSenha(request.getParameter("senha"));  
%>
```

O **param** daquela tag poderia ser substituído pelo atributo **value** que, nesse caso, receberia um valor literal ou o resultado de uma expressão que poderia ser embutida inline na tag.

## <jsp:getProperty>

Esta tag possui a função de obter valores dos atributos de um JavaBean criado anteriormente. Semelhante à tag anterior, porém, semanticamente oposta, essa tag depende da existência de um objeto javaBean prévio e, sobretudo, de que a sintaxe de encapsulamento dos campos na classe esteja feita de forma adequada (campos private com métodos assessores).

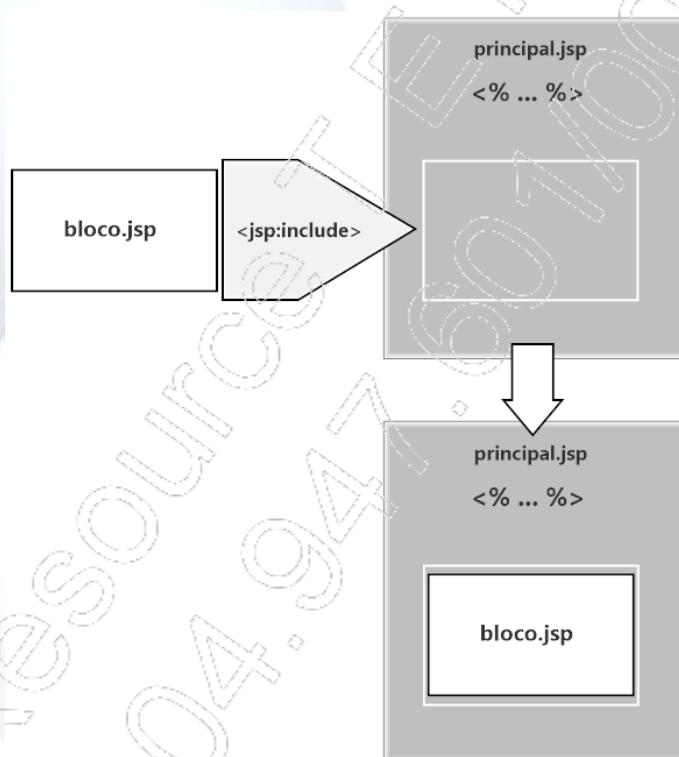
Sua sintaxe, em um contexto Web, é a seguinte:

```
<h4>Login capturado:</h4>  
<jsp:getProperty property="login" name="user"/>  
  
<h4>Senha capturada:</h4>  
<jsp:getProperty property="senha" name="user"/>
```

Perceba a aplicação de duas tags em código que recupera os valores das propriedades do JavaBean criado e inicializado nos dois itens anteriores.

# Composição de páginas – Standard Actions <jsp:include> e <jsp:param>

A composição de páginas é um recurso de grande utilidade quando uma aplicação passa a trabalhar com volumosas quantidades de telas e código do lado cliente, pois permite a reutilização de partes comuns em páginas, como cabeçalhos e rodapés, que tendem a se repetir em todas as páginas da aplicação. Em um esquema visual, a composição pode ser apresentada da seguinte forma:



Anteriormente, foram apresentadas a diretiva **page** e seu atributo **include** como formas de realizar composição em uma página. Ao utilizar a diretiva, a inclusão ocorre em tempo de compilação, não podendo ser usada de forma dinâmica. Com a Standard Action **<jsp:include>**, a inclusão acontece dinamicamente, em tempo de execução e, por esse motivo, é possível passar parâmetros às páginas que recebem o trecho incluído, de forma que estes sejam acessíveis pelo objeto implícito **request**. Esses parâmetros são passados por meio da Standard Action **<jsp:param>**. Vejamos a sintaxe (em um exemplo funcional de uso da tag):

```
12<jsp:include page="bloco.jsp" flush="true">
13    <jsp:param name="nome" value="Antonio da Silva"/>
14 </jsp:include>
```

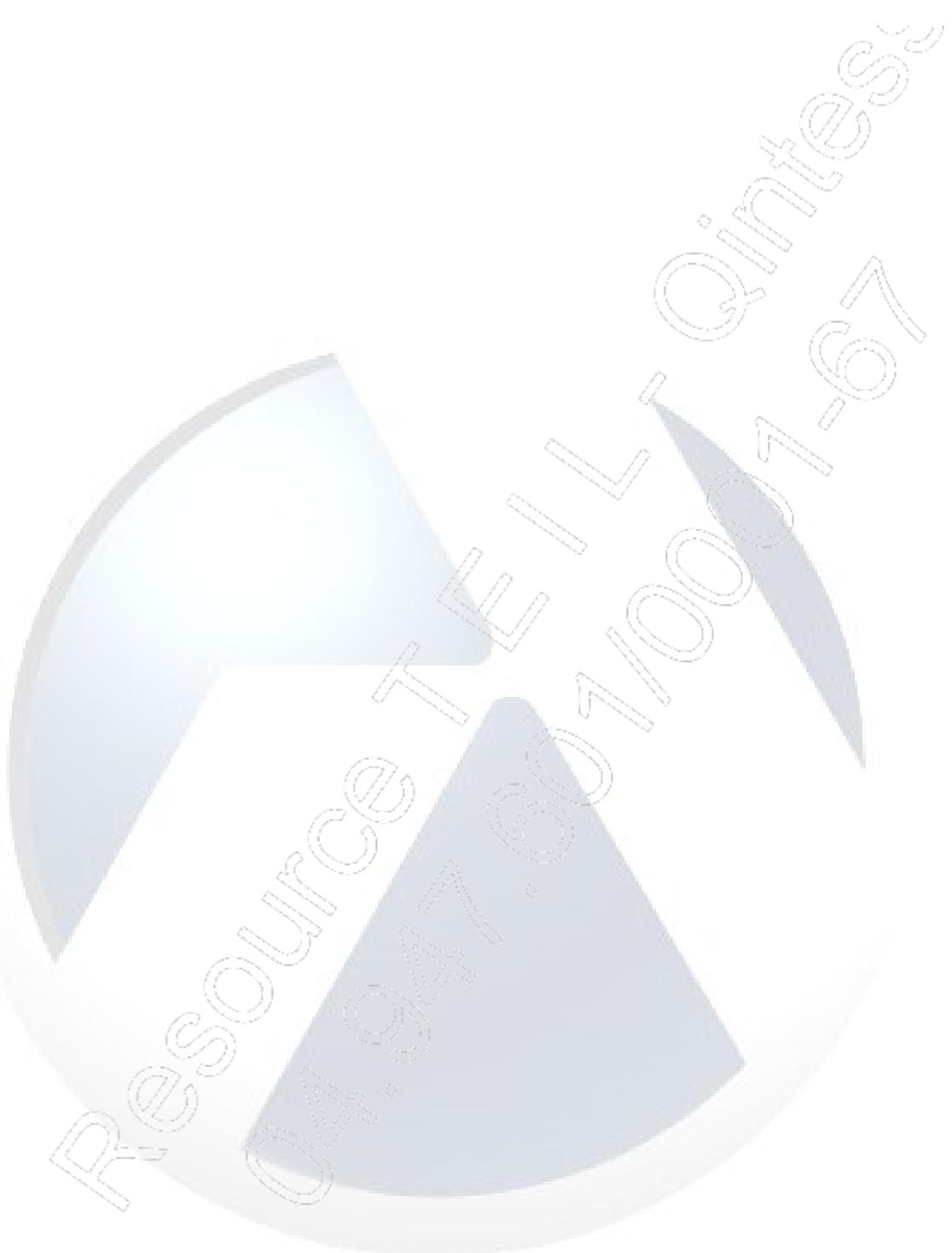
Note na tag **include** o atributo **flush**, que, por default, é definido como **false**. Atribuímos **true** para garantir que a página seja atualizada em todas as situações, caso ainda exista algum fragmento de dados seus no buffer.

Perceba ainda que estamos passando um parâmetro cuja chave é definida como **name** e o valor é indicado como **Antonio da Silva** (porém, poderia ser qualquer nome.). Esse parâmetro será recepcionado na página hospedeira por meio do objeto implícito **request**.

Não há muito a se comentar sobre a tag **<jsp:param>** além do exemplo já apresentado. Vejamos, então, o código da página **bloco.jsp**, incluída por meio da tag exibida anteriormente:

```
6 <%@ page import="java.util.Date" %>
7<h3> Bom Dia, Sr.
8    <%= request.getParameter("nome") %> <br />
9 </h3>
10
11 Data atual: <%= new Date() %>
```

Esse pequeno bloco de código representa o trecho inserido na página por meio da Standard Action **include**. Perceba a expressão que utiliza o parâmetro de **request** cuja chave é **nome**. Esse parâmetro é exatamente aquele configurado na tag anterior.



# Acesso a dados pelas JSPs

Teste seus conhecimentos



**1. Qual das alternativas a seguir completa adequadamente a frase a seguir?**

**Quando uma requisição provinda de algum formulário HTML chega diretamente a uma JSP, \_\_\_\_\_.**

- a) provavelmente houve um erro por parte do desenvolvedor, uma vez que JSPs não podem receber requisições diretamente.
- b) os parâmetros nessa requisição só são passíveis de captura pela JSP, caso seja utilizado o método POST.
- c) os parâmetros nessa requisição só são passíveis de captura pela JSP, caso seja utilizado o método GET.
- d) as JSPs não possuem meios de capturar parâmetros passados por métodos diferentes de GET e POST, como PUT ou DELETE.
- e) Nenhuma das alternativas anteriores está correta.

**2. O que ocorre quando uma Java Server Page, após capturar os dados de login de usuário, precisar passar esses dados para outro componente no servidor a fim de validá-los?**

- a) Esses dados só podem ser passados por meio de parâmetros de requisição por se tratar de uma exigência do padrão de marcação HTML.
- b) Esses dados podem ser passados por meio de um atributo em qualquer um dos contextos existentes.
- c) Esses dados só podem ser direcionados para um Servlet, independente da forma de passagem dos dados, tendo em vista que a validação de dados é assunto exclusivo de abordagem por Servlets.
- d) Esses dados podem ser passados por parâmetro. A vantagem disso é que se torna possível a tipificação dos valores para seus reais formatos, como números, por exemplo.
- e) Além de parâmetros na requisição, só é possível utilizar atributos no objeto HttpServletRequest.

**3. Qual das alternativas a seguir está correta sobre os padrões que são exigidos de um JavaBean e sua real utilidade em aplicações Web Java?**

- a) Deve possuir somente um construtor e ele deve declarar como parâmetros todos os campos da classe.
- b) Deve ser marcado com a interface Cloneable, tendo em vista que será copiado quando da passagem entre as camadas da aplicação.
- c) Deve privilegiar o encapsulamento, sendo obrigatória a existência de métodos assessores (getter e setter), e ter todos os seus campos marcados como private.
- d) Deve implementar a interface Serializable e, portanto, implementar os dois métodos obrigatórios nela definidos.
- e) São objetos comumente utilizados para representar a implementação da camada de negócios, contendo métodos utilitários.

**4. Qual das alternativas a seguir não está correta sobre as Standard Actions voltadas para os JavaBeans?**

- a) Foram criadas com o objetivo de facilitar a compreensão do código das JSPs para os profissionais não desenvolvedores de Java, como os Designers, por exemplo.
- b) Facilitam a instanciação de JavaBeans para os desenvolvedores, sem o uso de Scriptlets.
- c) São comumente utilizadas para se obter e configurar propriedades de JavaBeans previamente instanciados.
- d) Permitem eliminar o uso de Scriptlets nas JSPs em geral, pois todo o código relativo a acesso a dados pode ser realizado por meio delas.
- e) Facilitam o desacoplamento entre as camadas de uma aplicação Web, prática muito valorizada nas aplicações atuais.

**5. Qual das alternativas a seguir está correta sobre a técnica de composição de páginas JSP?**

- a) É feita exclusivamente com a Standard Action <jsp:include> e deve ser colocada logo no início da JSP hospedeira.
- b) Garante o reaproveitamento de código entre páginas e facilita a manutenção do sistema, diminuindo os pontos de alteração caso surjam problemas.
- c) Pode ser feita tanto com a Standard Action quanto com a diretiva page, sendo que, no último caso, a composição é dita dinâmica.
- d) Utilizando a diretiva page, ao invés da Standard Action, torna-se possível a utilização de parâmetros na página que será inserida como fragmento em outra.
- e) Nenhuma das alternativas anteriores está correta.

# Acesso a dados pelas JSPs

Mãos à obra!



## Laboratório 1

### A – Utilizando Standard Actions, composição e uso de lógica para exibir dados em uma JSP

1. No laboratório anterior, iniciamos o desenvolvimento de um projeto de cadastro, ao qual daremos sequência nesta atividade. Para tanto, localize o projeto chamado **LabWeb7\_Cadastro\_Inicial**. Copie-o para o workspace atual do Eclipse e importe-o seguindo as orientações fornecidas pelo instrutor;

Esse projeto é exatamente igual ao do laboratório do capítulo 6, com todas as modificações daquele laboratório já realizadas.

2. Iniciaremos utilizando a Standard Action `<jsp:include>` para substituir a composição feita antes com a página `header.jsp` dentro de `login.jsp` e incluir uma outra na página `sistema.jsp`, que será a página com o efetivo conteúdo de nossa aplicação Web. Para essa substituição, utilizaremos a passagem de parâmetros com a Standard Action `<jsp:param>` para a definição do título da página. Para isso, siga estes passos:

2.1. Na página `login.jsp`, localize o seguinte trecho de código:

```
7<head>
8    <title>Cadastro de Alunos - Login</title>
9    <%--Diretiva include - Lab 2 Tarefa 2 --%>
10   <%@include file="header.jsp" %>
11 </head>
```

2.2. Todo o conteúdo interno às tags `<head>...</head>` será trocado pelas Standard Actions de composição e o título da página será passado por parâmetro à página `header.jsp`, que obterá esse parâmetro por meio de acesso ao objeto implícito `request` e o posicionará entre as tags `<title>...</title>` presentes no corpo da página incluída. Vejamos como ficará o trecho anterior com o código substituído:

```
6@<head>
7@    <jsp:include page="header.jsp">
8@        <jsp:param value="Cadastro de Alunos - Login" name="title"/>
9@    </jsp:include>
10</head>
```

2.3. Altere o código da página `header.jsp`, incluída no passo anterior, de forma que ela acesse o parâmetro passado pela Standard Action e fique com o seguinte código:

```
1 <title><%=request.getParameter("title")%></title>
2 <meta charset="utf-8">
3
4 <!-- CSS -->
5 <link href='http://fonts.googleapis.com/css?family=Droid+Sans:400,700'
6   rel='stylesheet'>
7 <link rel="stylesheet" href="css/style.css">
8
9 <!-- Otimização para mobile -->
10 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
11
12 <!-- jQuery & arquivos JS -->
13@<script
14    src="http://ajax.googleapis.com/ajax/Libs/jquery/1.7.1/jquery.min.js"></script>
15 <script src="js/script.js"></script>
```

2.4. Execute o projeto e insira o usuário e senha programados para efetivamente acessar o aplicativo:

- Usuário: **aluno**;
- Senha: **impacta123**.

2.5. Note que a página **sistema.jsp**, para a qual é direcionado o usuário após a realização do login, aparece sem qualquer formatação e isso ocorre porque ainda não foram importados os componentes necessários de css e javascript, presentes na página **header.jsp**. Essa página será inserida na página **sistema** no próximo passo. Vejamos como está a página **sistema.jsp** neste ponto:



2.6. Assim como no passo 2.1., faça a inclusão da página **header.jsp** na página **sistema.jsp** usando as Standard Actions de composição, de forma que o trecho de código entre as tags **<head>...</head>** fique com o seguinte código:

```

8<head>
9<jsp:include page="header.jsp">
10<jsp:param value="Cadastro de Alunos - Sistema" name="title"/>
11</jsp:include>
12</head>

```

2.7. Execute a aplicação, caso ainda não esteja sendo executada, e confira a aparência da página **sistema.jsp** após a inclusão da página **header** em seu cabeçalho.

3. Vamos implementar agora o Servlet responsável pela saída do usuário do sistema, acionado pelo botão **Sair** da página **sistema.jsp**. Esse Servlet está inicialmente comentado com instruções passo a passo sobre como proceder para realizar a desconexão do usuário do sistema. Analise o seu código inicial:

```
1 package servlets;
2
3 import java.io.IOException;
11
12 @WebServlet("/sair")
13 public class SairServlet extends HttpServlet {
14     private static final long serialVersionUID = 1L;
15
16     public SairServlet() {
17         super();
18     }
19
20     protected void doPost(HttpServletRequest request,
21             HttpServletResponse response) throws ServletException, IOException {
22         //Passos para implementação do Servlet:
23         //Obtenha a sessão atual do usuário logado
24         //Verifique se a sessão capturada existe ou é válida e,
25         //nesse segundo caso, invalide-a.
26         //Redirecione o usuário para a página de login inicial da aplicação.
27     }
28 }
```

3.1. Digite o seguinte código no método POST desse Servlet, como forma de implementar as regras delineadas nos comentários. Dessa forma, você garante a invalidação da sessão e a liberação de recursos quando o usuário sair do sistema.

```
23     HttpSession session = request.getSession(false);
24     if (session != null){
25         session.invalidate();
26     }
27     response.sendRedirect(request.getContextPath() + "/login.jsp");
```

4. Na página **sistema.jsp**, localize o trecho de código adiante, iniciado na linha 81. Atente aos comentários existentes no corpo de cada linha que compõe uma coluna e aos detalhes dessa tabela. Note ainda os comentários que denotam o início e o fim do bloco de repetição. Para popular esse componente, serão utilizados os dados retornados pela classe **Dados**, criada no pacote **datasource**, e cada dado terá o formato de um JavaBean, nesse caso, a classe **Aluno**, criada no mesmo pacote;

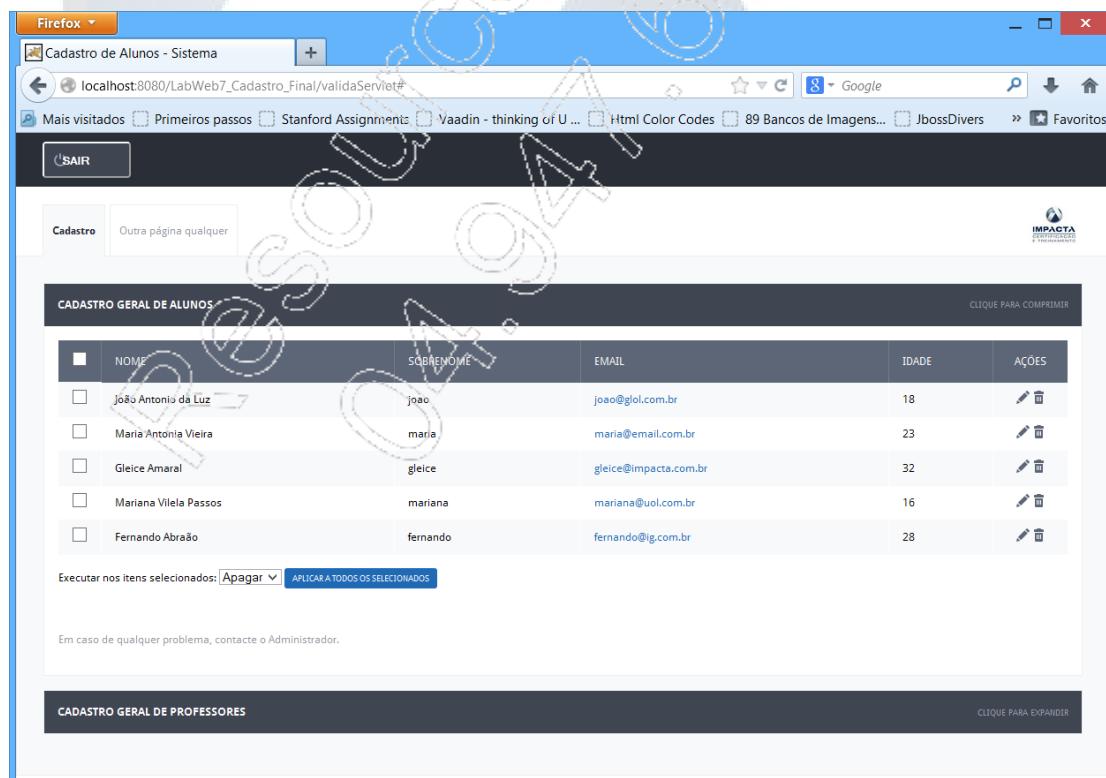
```
<tbody>
    <%--Início do bloco de repetição --%
    <tr>
        <td><input type="checkbox"></td>
        <td><%--Nome do aluno vai aqui --%></td>
        <td><%--Usuário do aluno vai aqui --%></td>
        <td><a href="#"><%--Email do aluno vai aqui --%></a></td>
        <td><%--Idade do aluno vai aqui --%></td>
        <td><a href="#" class="table-actions-button ic-table-edit"></a>
            <a href="#" class="table-actions-button ic-table-delete"></a></td>
    </tr>
    <%--Fim do bloco de repetição --%
</tbody>
```

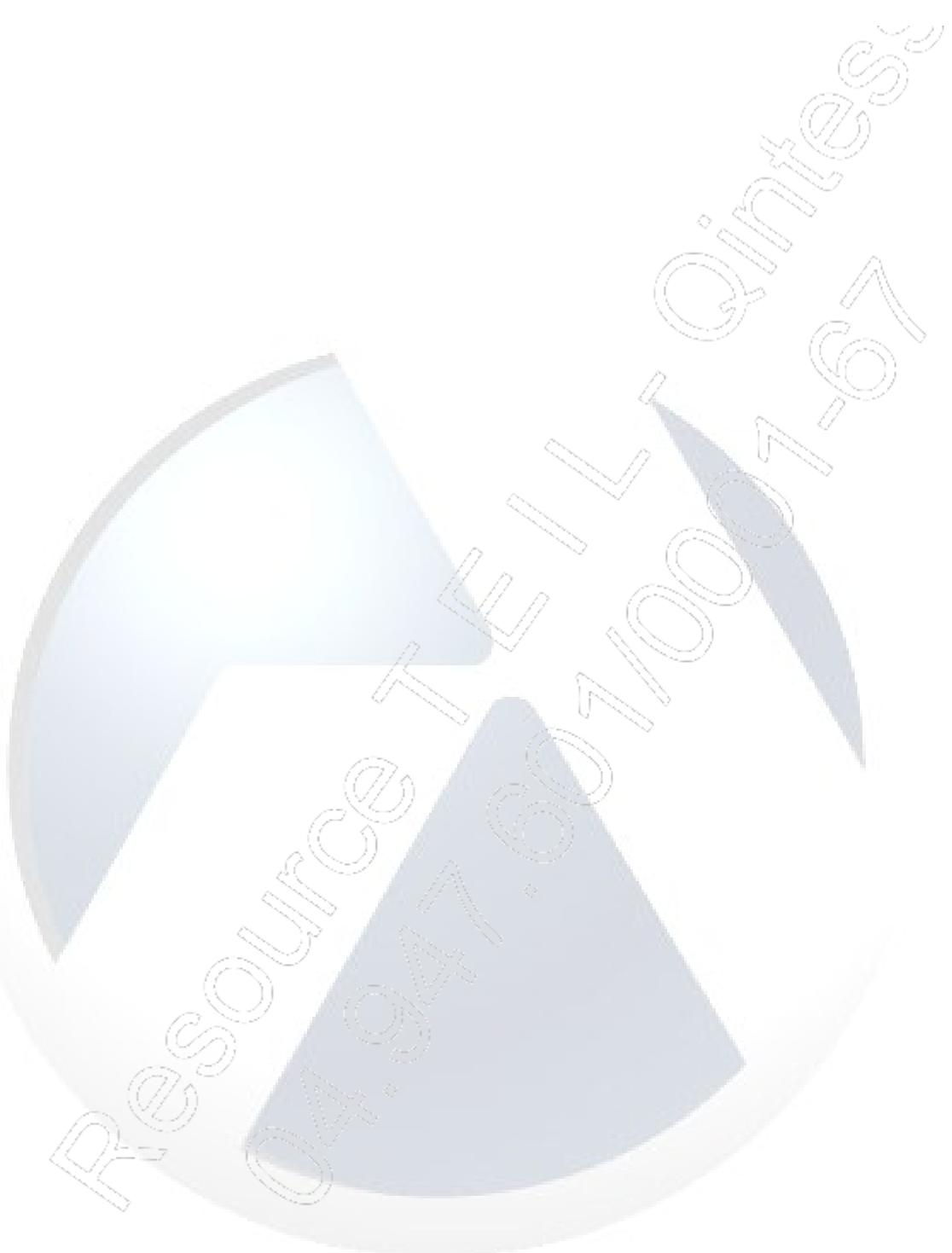
- 4.1. Analise o código das classes **Aluno** e **Dados** minuciosamente;
- 4.2. Crie, antes do primeiro comentário alertando sobre o início do bloco de repetição, um bloco Scriptlet de forma a instanciar a classe **Dados** e obter uma lista populada de objetos **Aluno** por meio do método **getDados()**;
- 4.3. Crie, abaixo do comentário acerca do início do laço de repetição, um laço for-enhanced cujo corpo seja todo o conteúdo das tags **<tr>...</tr>**, incluindo as próprias tags;
- 4.4. Dentro desse laço, substitua cada comentário pela expressão que imprime o retorno das chamadas ao método getter adequado do objeto **Aluno**, objeto controle do laço de repetição. O código deverá estar semelhante ao exposto a seguir:

```
<tbody>
<%
    Dados dados = new Dados();
    List<Aluno> alunos = dados.getDados();
%>
<%--Início do bloco de repetição --%
<%for (Aluno aluno : alunos) { %>
<tr>
    <td><input type="checkbox"></td>
    <td><%=aluno.getNome()%></td>
    <td><%=aluno.getUserName()%></td>
    <td><a href="#"><%=aluno.getEmail()%></a></td>
    <td><%=aluno.getIdade()%></td>
    <td><a href="#" class="table-actions-button ic-table-edit"></a>
        <a href="#" class="table-actions-button ic-table-delete"></a></td>
</tr>
<%} %>
<%--Fim do bloco de repetição --%
</tbody>
```

4.5. Como tarefa opcional final, substitua todas as expressões que contêm chamadas a métodos getter no corpo do laço por Standard Actions equivalentes a JavaBeans.

Ao final, a página **sistema.jsp** deverá estar semelhante a esta imagem:





# Capítulo 8: Expression Language

# O que é e para que serve a EL?

A Expression Language foi incorporada à especificação JSP em sua versão 2.0, apresenta uma sintaxe que segue as linhas de linguagens como o ECMAScript e o XPath e procura trazer clareza, facilidade e versatilidade para as Java Server Pages.

Até o capítulo anterior, vimos o acesso a dados, parâmetros e atributos pelas JSPs por meio de Scriptlets, Expressões e Standard Actions. Todas as formas apresentadas funcionam perfeitamente, porém, exigem certo conhecimento em linguagem Java para poder operar com as Java Server Pages, componentes essenciais da camada de apresentação das aplicações Web.

De forma a simplificar e tornar mais intuitivo o tratamento da lógica e acesso a dados no interior de uma JSP, especialmente para profissionais voltados à área de design gráfico, a EL traz uma sintaxe unificada e simples.

## Sintaxe e operadores

A sintaxe da Expression Language baseia-se na criação e alocação de expressões entre as chaves no seguinte trecho: \${ ... }. Essa sintaxe pode ser usada para inserir texto e conteúdo na página, bem como substituir dinamicamente o valor e/ou atributos de uma tag. Vale lembrar que uma expressão EL não deve ser colocada dentro de Scriptlets, e sim estar diretamente alocada no código da página JSP.

Vejamos, a seguir, um exemplo do uso de EL:

```
<h3>Usuário: ${usuario.login}</h3>
<h3>Senha: ${usuario.senha}</h3>
```

Esse código pode ser utilizado para recuperar o valor armazenado nos campos login e senha de um objeto instanciado cuja referência é **usuario**. Se fôssemos realizar a mesma tarefa com meios alternativos, teríamos as seguintes opções:

- Usando Standard Actions:

```
<h3>Usuário: <jsp:getProperty property="Login" name="usuario"/></h3>
<h3>Senha: <jsp:getProperty property="senha" name="usuario"/></h3>
```

- Usando Expressions:

```
<h3>Usuário: <%= ((Usuario)session.getAttribute("usuario")).getLogin() %></h3>
<h3>Senha: <%= ((Usuario)session.getAttribute("usuario")).getSenha() %></h3>
```

- Usando Scriptlet:

```
<%
    Usuario usuario = (Usuario)session.getAttribute("usuario");
    out.println("<h3>Usuário: " + usuario.getLogin() + "</h3>");
    out.println("<h3>Senha: " + usuario.getSenha() + "</h3>");
%>
```

Comparando as possibilidades apresentadas, fica bem clara a economia de código e a facilidade de compreensão da EL quando comparada com as demais formas disponíveis para lidar com dados, especialmente JavaBeans.

A forma de uso dos operadores disponíveis em Expressões EL é bem semelhante na semântica ao uso feito diretamente em linguagem Java literal. O diferencial na EL é a possibilidade do uso de palavras textuais para substituir alguns operadores como forma de tornar ainda mais legível o intento do código da JSP ao profissional não desenvolvedor.

Vejamos uma tabela com os principais operadores existentes em EL:

Descrição	Sintaxe EL
<b>Aritméticos</b>	
<b>Adição</b>	+
<b>Subtração</b>	-
<b>Multiplicação</b>	*
<b>Divisão</b>	/ e div
<b>Resto</b>	% e mod
<b>Lógicos</b>	
<b>AND</b>	&& e and
<b>OR</b>	e or
<b>NOT</b>	! e not
<b>Relacionais</b>	
<b>Igual</b>	== e eq
<b>Diferente</b>	!= e ne
<b>Menor que</b>	< e lt
<b>Maior que</b>	> e gt
<b>Menor ou igual a</b>	<= e le
<b>Maior ou igual a</b>	>= e ge
<b>Teste null</b>	empty

Segue um exemplo prático do uso de operadores em EL:

```
10 - 4 = <span>${10 - 4}</span> <br />
20 % 5 = <span>${20 mod 5}</span> <br />
true && (!false) = <span>${true and not(false)}</span> <br />
10 <= 11 = <span>${10 le 11}</span> <br />
```

Confira a saída no navegador:

```
10 - 4 = 6
20 % 5 = 0
true && (!false) = true
10 <= 11 = true
```

# Variáveis e acesso a dados

Em EL, as variáveis ou dados representados ou não por objetos são acessados diretamente pelo nome. Qualquer dado recebido na JSP, seja por parâmetro ou mesmo por meio de atributo existente em qualquer um dos escopos válidos disponíveis, pode ser acessado diretamente pelo nome.

Como exemplo, citamos o objeto **Usuario** usado como parâmetro anteriormente. Em um Servlet qualquer, o atributo **usuario** é inserido no escopo de sessão:

```
Usuario usuario = new Usuario();
HttpSession session = request.getSession();
session.setAttribute("usuario", usuario);
```

A obtenção desse atributo a partir de um JSP, cujo encaminhamento ou mesmo outra requisição tenha sido realizada, pode ser feita de formas diferentes, conforme demonstrado a seguir:

- Sem o uso de EL:

```
Usuário: <%= (Usuario)session.getAttribute("usuario") %>
```

- Com o uso de EL:

```
Usuário: ${usuario}
```

No código anterior, é possível perceber que basta a indicação da chave do atributo para que a EL consiga localizá-lo no escopo devido. Isso ocorre porque é feita uma busca em todos os escopos válidos, iniciando por **page**, depois **request**, **session** e, por último, **servletContext (application)**. É possível determinar explicitamente, por meio de acesso a um objeto específico qual o escopo a se procurar, porém, antes é necessário conhecer os objetos implícitos da EL.

A Expression Language possui objetos implícitos, assim como as JSPs, porém, seus objetos são diferentes daqueles das JSPs de diversas maneiras. O próprio significado dos objetos varia – lembre-se de que para se obter um atributo de um escopo não é necessário sequer mencionar o escopo, basta o identificador do atributo. Vejamos esses objetos um a um com os seus significados nesta tabela:

Objeto implícito	Descrição e significado
<b>requestScope</b>	Mapa de atributos do escopo de request (não é o objeto Request, como em JSP).
<b>sessionScope</b>	Mapa de atributos do escopo de session (não é o objeto HttpSession, como em JSP).
<b>applicationScope</b>	Mapa de atributos do escopo de aplicação (não é o objeto ServletContext, como em JSP).
<b>pageScope</b>	Mapa de atributos do escopo de página, PageContext.
<b>pageContext</b>	Referência para o objeto PageContext.
<b>param</b>	Mapa de parâmetros da request obtidos por <code>request.getParameter(...);</code>
<b>paramValues</b>	Mapa de parâmetros da request, no formato de array, obtidos por <code>request.getParameterValues();</code>
<b>header</b>	Mapa de headers da request obtidos por <code>request.getHeader();</code>
<b>headerValues</b>	Mapa de headers da request em formato de array, obtidos por <code>request.getHeaders();</code>
<b>cookie</b>	Mapa de cookies obtidos da request.
<b>initParam</b>	Mapa de parâmetros de inicialização do contexto da aplicação.

Grande parte dos objetos implícitos são, na verdade, mapas populados com o conteúdo obtido de cada um dos escopos e com o conteúdo recebido da requisição em andamento.

O objeto **pageContext**, nesse ponto, merece destaque, pois possibilita ao desenvolvedor acessar os objetos de escopo da mesma maneira como em JSP. Há situações em que podemos precisar de uma referência para o próprio objeto referente ao escopo e não apenas de um mapa que contenha seus atributos. Nessas situações, o objeto **pageContext** pode ser utilizado como meio para se obter tais referências. Por exemplo:

- **pageContext.request** equivale ao objeto implícito **request** das JSP;
- **pageContext.session** equivale ao objeto implícito **session** das JSP.

# Navegação em dados

A Expression Language possui uma sintaxe que emprega o operador de acesso indexado, muito usado com arrays: [...]. Com o uso do indexador, a EL apresenta grande versatilidade em acesso e oferece uma maior gama de possibilidades no acesso a dados.

Fazendo uma comparação com o operador ponto (“.”):

The diagram shows two EL expressions side-by-side: \${usuario.login} and \${usuario["login"]}. A large curly brace groups both expressions together, with the text "São equivalentes!" written next to it, indicating they represent the same thing.

A sintaxe de ambas as expressões são equivalentes em função, porém, apresentam diferenças sintáticas. A primeira grande vantagem do uso de colchetes é a sua abrangência de uso:

- Usando o operador **ponto**, o que estiver à sua esquerda tem que ser um bean ou um Map, e o que estiver à direita do ponto, por consequência, deve ser uma propriedade bean ou uma chave do mapa e, em ambos os casos, inclusive o lado direito, deve seguir as regras de nomenclatura Java;
- Usando o operador **colchete**, a variável da esquerda (na qual se aplicam os colchetes) pode ser um Map, um bean, um array ou uma List. Perceba que o escopo, nesse caso, fica bem maior.
- Se o valor que vem dentro dos colchetes for uma String literal, pode tratar-se de uma propriedade de um bean, de uma chave de Map e até ser utilizado para trazer índices de arrays e Lists.

Vejamos, a seguir, alguns exemplos de utilização:

- Acessando um elemento de um array ou List:
- `${nomes}`: Chamando `toString()` no array ou List;
- `${nomes[1]}`: Imprimindo o elemento de posição 1 no array ou List;
- `${nomes["1"]}`: Também imprimindo o elemento de posição 1 no array ou List.
  - Acessando os elementos de um bean ou mapa:
- Nesse caso, é possível tanto o uso do operador ponto como do operador colchetes;
- `${usuario.login}`: Retorna o valor da propriedade `login` do bean usuário ou o valor do mapa usuário cuja chave seja `login`. É importante ressaltar que o operador ponto exige o uso de um identificador Java adequado após o ponto;
- `${usuario["login"]}`: Também retorna o valor da propriedade `login` do bean usuário ou o valor do mapa usuário cuja chave seja `login`.

Vale lembrar, para complementar os exemplos citados, que a EL é tolerante no que tange à exceção `NullPointerException`, ou seja, ela não devolve erro, caso o objeto não esteja instanciado antes de acessar suas propriedades. Ainda, a EL é tolerante à tentativa de acesso a elementos por meio de índices equivocados. Em casos de acesso indexado indevidamente, não será gerada uma exceção do tipo `ArrayIndexOutOfBoundsException`. Nesse caso, a EL apenas retornará o valor vazio.

# Configuração de EL e Scriptlets em JSPs

Podem existir situações em que seja importante para o desenvolvedor ou arquiteto de aplicações habilitar ou desabilitar scripts em páginas JSP.

É possível habilitar ou desabilitar scripts e o uso de EL em páginas JSP por meio de diretivas de configuração inseridas dentro do arquivo **web.xml**. Por padrão, versões Java EE 1.4 e posteriores vêm com EL e scripting em geral (Scriptlets, expressões e declarações) habilitados de fábrica.

A proibição de scripts na página pode ser, de certa forma, uma medida de controle que garanta o uso de tecnologia do lado cliente apenas. JSPs com scripting desabilitado geram erro, caso possuam código Java. Esse erro ocorre no acesso à página.

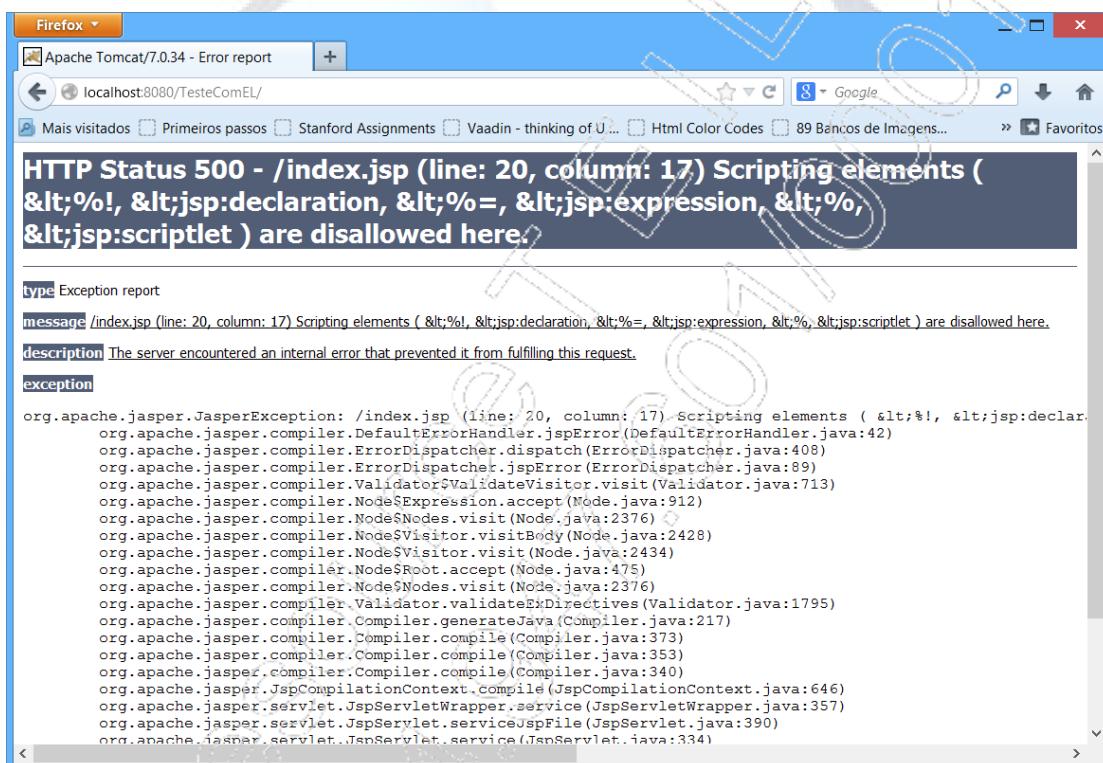
Vejamos o arquivo **web.xml** com trecho que contém estrutura XML para inabilitar Scripting em um padrão genérico de URLs para JSPs:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://java.sun.com/xml/ns/javaee"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
6   id="WebApp_ID" version="3.0">
7
8 <welcome-file-list>
9   <welcome-file>index.jsp</welcome-file>
10 </welcome-file-list>
11
12 <jsp-config>
13   <jsp-property-group>
14     <url-pattern>*.jsp</url-pattern>
15     <scripting-invalid>true</scripting-invalid>
16   </jsp-property-group>
17 </jsp-config>
18
19 </web-app>
```

Do trecho de código anterior, destaca-se o valor `true` inserido na tag de invalidação de Scripting. Por este trecho, proíbe-se todas as JSPs da aplicação (o padrão `*.jsp`) de possuírem qualquer bloco de Scriptlet, Expressões ou Declarações. Para se atingir o resultado esperado, é possível alocar múltiplas tags `<url-pattern>`.

Ao tentar acessar uma página com essa proibição, obtém-se como resultado no navegador a página de erro do Tomcat, que pode variar conforme o servidor usado:



Assim como vimos anteriormente, é possível desabilitar o uso de Expression Language nas JSPs. A sintaxe para se fazer isso via `web.xml` é semelhante à anterior, porém, a tag a ser usada é `<el-ignored>`. Vejamos o trecho do `web.xml` correspondente:

```

12<jsp-config>
13  <jsp-property-group>
14    <url-pattern>*.jsp</url-pattern>
15    <el-ignored>true</el-ignored>
16  </jsp-property-group>
17</jsp-config>

```

Com EL, é possível fazer o controle de habilitação por página, através de um atributo da diretiva `@page`. Esse atributo determina se naquela pagina específica será utilizada EL ou se deverá ser ignorada qualquer expressão desse tipo. Essa configuração recebe prioridade sobre aquela configurada no Deployment Descriptor. Vejamos um exemplo do uso desse atributo:

```
<%@ page language="java" isELIgnored="true" import="java.util.*"%>
```



# Expression Language

Teste seus conhecimentos



### 1. O que é a Expression Language?

- a) Uma modalidade de Scriptlet Java mais resumida que permite acessar com maior facilidade dados encontrados em arrays e coleções em geral.
- b) Uma característica das JSPs que permitem acessar dados de JavaBeans.
- c) Trata-se apenas de expressões Java com uma sintaxe mais agradável e flexível, permitindo ao profissional de Design uma melhor interação com a página.
- d) Uma linguagem de sintaxe próxima ao ECMAScript que veio para facilitar o acesso a dados e retirar o excesso de código Java das páginas JSP, viabilizando a interação entre desenvolvimento e design.
- e) Nenhuma das alternativas anteriores está correta.

### 2. Quais destes operadores não estão disponíveis em EL?

- a) / e div.
- b) % e mod.
- c) null e ge.
- d) empty e ne.
- e) + e eq.

### 3. Qual das alternativas a seguir está correta sobre os objetos implícitos trazidos pela EL?

- a) São equivalentes aos trazidos pelas JSP com alguns outros novos que propiciam uma maior gama de facilidades e acessos aos escopos ou contextos válidos.
- b) Na maioria, são mapas que representam atributos, dados ou parâmetros de escopo.
- c) O objeto implícito requestScope pode ser usado em substituição ao objeto implícito Request, disponibilizado pelas JSPs.
- d) Os parâmetros recebidos por uma requisição HTTP podem ser obtidos, via EL, por uma chamada ao objeto param, dentro do objeto requestScope.
- e) Nenhuma das alternativas anteriores está correta.

4. Assinale a alternativa que apresenta a circunstância que o desenvolvedor deve analisar para decidir quando usar o operador ponto (exemplo: “\${usuario.login}”) e/ou o operador colchete (exemplo: “\${usuario[“login”]}”):
- a) Analisar, quando usar ponto, se trata-se de um JavaBean ou List à esquerda do ponto, pois somente esses tipos permitem tal sintaxe.
  - b) Decidir se prefere acessar os elementos de um array por operador ponto ou colchete, já que este último é o que tem mais elementos e demanda mais digitação.
  - c) Prever a necessidade futura de alteração estrutural na fonte dos dados usada na JSP, pois, caso a coleção utilizada hoje em uma EL seja um Map e mude futuramente para uma List, somente o operador colchete pode garantir a compatibilidade, o ponto não.
  - d) Conhecer o perfil dos profissionais envolvidos no projeto, pois, apesar de ambos os operadores serem completamente intercambiáveis, profissionais familiarizados com linguagens de Script podem se acostumar melhor com o operador colchete.
  - e) Nenhuma das alternativas anteriores está correta.

**5. O que ocorre ao desabilitar a possibilidade de Scripting via web.xml para uma JSP e ativar o atributo isELIgnored dessa mesma página via diretiva @page?**

- a) Fica desabilitado o uso de Scriptlets e expressões na página e, em caso de um acesso a essa JSP, os trechos incidentes serão ignorados.
- b) Fica proibido o uso de EL na JSP, causando e retornando o erro HTTP 500 para qualquer tentativa de acesso àquela página.
- c) As expressões EL e os Scriptlets são ignorados, não gerando erro para o usuário.
- d) Obtém-se um erro de compilação, pois não é permitido desabilitar scripts via web.xml e a EL via página. Os métodos devem ser os mesmos.
- e) O desenvolvedor fica proibido de utilizar qualquer bloco de Scriptlet, Expressão ou Declaração, sob a pena da geração de erro HTTP 500 ao usuário que tente o acesso a essa JSP, e todas as EL da página são ignoradas.

# Expression Language

Mãos à obra!



## Laboratório 1

### A – Inserção da lista de cadastro de professores e uso de EL com a biblioteca JSTL

Neste laboratório, criaremos e popularemos a tabela de professores já existente na página nos mesmos moldes da tabela de alunos. Será utilizada uma coleção do tipo **List** como fonte de dados de uma forma simples, porém, eficaz para atender aos objetivos desta atividade.

1. Localize o projeto chamado **LabWeb8\_Cadastro\_EL\_Inicial**. Copie-o para o workspace atual do Eclipse e importe-o seguindo as orientações fornecidas pelo instrutor;

Este projeto é exatamente igual ao do laboratório do capítulo 7, com todas as modificações desse laboratório já realizadas.

2. Crie uma nova classe denominada **Professor** no pacote **datasource**. Da mesma forma que a classe **Aluno**, essa classe será utilizada como um ajudante em nosso pequeno sistema. Ela será usada primariamente para guardar dados e transitar entre as camadas da aplicação. Para tanto, insira o seguinte código para a classe **Professor**:

```
1 package datasource;
2 public class Professor {
3     private String nome;
4     private String disciplina;
5     private String email;
6
7     public Professor() {}
8
9     public Professor(String nome, String disciplina, String email){
10         super();
11         this.nome = nome;
12         this.disciplina = disciplina;
13         this.email = email;
14     }
15     public String getNome() {
16         return nome;
17     }
18     public void setNome(String nome) {
19         this.nome = nome;
20     }
21     public String getDisciplina() {
22         return disciplina;
23     }
24     public void setDisciplina(String disciplina) {
25         this.disciplina = disciplina;
26     }
27     public String getEmail() {
28         return email;
29     }
30     public void setEmail(String email) {
31         this.email = email;
32     }
33 }
```

3. Na classe **Dados**, já existente, faça as seguintes alterações de forma a oferecer suporte para a página **sistema.jsp**:

3.1. Crie um campo na classe, do tipo **List<Professor>** para armazenar os dados de objetos **Professor** que serão utilizados na página;

```
private List<Professor> professores;
```

3.2. Crie um método denominado **populaListaProfessores()** que será responsável pela instanciação tardia (Lazy Instantiation) do campo criado no passo anterior e por popular a lista com seis novos professores;

```
private void populaListaProfessores() {  
    if (this.professores == null) {  
        professores = new ArrayList<>();  
    }  
    professores.add(new Professor("Nicolau Copernico", "Astronomia", "copernico@email.com"));  
    professores.add(new Professor("Albert Einstein", "Física", "einstein@email.com"));  
    professores.add(new Professor("Isaac Newton", "Matemática", "newton@email.com"));  
    professores.add(new Professor("James Gosling", "Programação Java", "gosling@email.com"));  
    professores.add(new Professor("Marie Curie", "Química", "mcurie@email.com"));  
    professores.add(new Professor("Charles Darwin", "Biologia", "darwin@email.com"));  
}
```

3.3. Crie um método getter, denominado **getDadosProfessores()** para a coleção de objetos **Professor** que invoca o método criado anteriormente e retorna a referência da lista;

```
public List<Professor> getDadosProfessores() {  
    this.populaListaProfessores();  
    return professores;  
}
```

3.4. Altere o nome dos métodos criados anteriormente para, respectivamente, popular a lista de alunos e obter sua referência. Esta alteração visa organizar a classe **Dados** e fornecer identificadores significativos aos métodos que serão usados na página **sistema.jsp**.

- De **getDados()** para **getDadosAlunos()**;
- De **populaLista()** para **populaListaAlunos()**;

4. Faremos agora algumas alterações na página **sistema.jsp** de forma a popular a tabela de professores, pois, até o momento, temos apenas a tabela de alunos preenchida. Além disso, substituiremos o uso de Scriptlets pelo uso de EL e introduziremos mais uma funcionalidade na nossa JSP: o uso da biblioteca JSTL (Java Server Pages Standard Template / Tag Library). Tag Libraries serão vistas em um capítulo posterior, porém, utilizaremos uma de suas tags em nosso código. Para isso, siga estes passos:

4.1. Inicie copiando os dois arquivos **.jar** existentes na pasta **JSTL jars** para a pasta **/WebContent/WEB-INF/lib** do projeto. Esses são os arquivos da biblioteca JSTL, que será utilizada em breve;

4.2. Insira a diretiva adiante no início da página, logo após a diretiva `@page`. Esta diretiva garantirá a disponibilidade da biblioteca de tags JSTL para uso nessa página;

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

4.3. Altere o código Scriptlet usado para instanciar a classe **Dados** e obter a lista de alunos como estava, para agora, obter a lista de alunos e a lista de professores e inseri-las como atributos no contexto de Sessão. O seguinte trecho de código deverá substituir o Scriptlet antes posto a partir da linha 82. Utilizaremos as listas no escopo de sessão, pois, para trabalharmos com EL, não é possível compartilhar objetos diretamente com Scriptlets. Essa divergência é devida ao lapso de tecnologia entre as duas alternativas: os Scriptlets mais antigos e a EL mais nova e recomendada;

```
<%
    Dados dados = new Dados();
    session.setAttribute("alunos", dados.getDadosAlunos());
    session.setAttribute("professores", dados.getDadosProfessores());
%>
```

4.4. Substitua o laço de repetição **for** para a inserção de dados na tabela HTML, antes feito por meio de Scriptlets, pela tag `<c:forEach>` da JSTL e todos os itens inseridos nas células, antes por meio de Expressions, agora por Expressões EL, da seguinte forma:

```
<%--Início do bloco de repetição --%>
<c:forEach items="${alunos}" var="aluno">
    <tr>
        <td><input type="checkbox"></td>
        <td> ${aluno.nome} </td>
        <td> ${aluno.userName} </td>
        <td><a href="#"> ${aluno.email} </a></td>
        <td> ${aluno.idade} </td>
        <td><a href="#" class="table-actions-button ic-table-edit"></a>
            <a href="#" class="table-actions-button ic-table-delete"></a></td>
    </tr>
</c:forEach>
<%--Fim do bloco de repetição --%>
```

4.5. Perceba o uso de EL para a obtenção de referência à lista **alunos**, inserida no escopo de sessão anteriormente. A tag **<c:forEach>** utiliza-se do atributo **items** para referenciar a coleção e do atributo **var** para definir o identificador do objeto alocado a cada iteração do laço, parecido com o Enhanced-for usado na linguagem Java. Note a facilidade de leitura do código quando se usa uma tag como essa para automatizar a tarefa de iteração, especialmente para profissionais não convededores da linguagem Java;

4.6. Criaremos agora a tabela de professores nos mesmos moldes da tabela de alunos, já existente na página. Para tanto, localize o seguinte conjunto de tags na página **sistema.jsp**, a partir da linha 118:

```
<div class="content-module-main">
    <h3>Funcionalidades a serem inseridas no futuro...</h3>
</div>
```

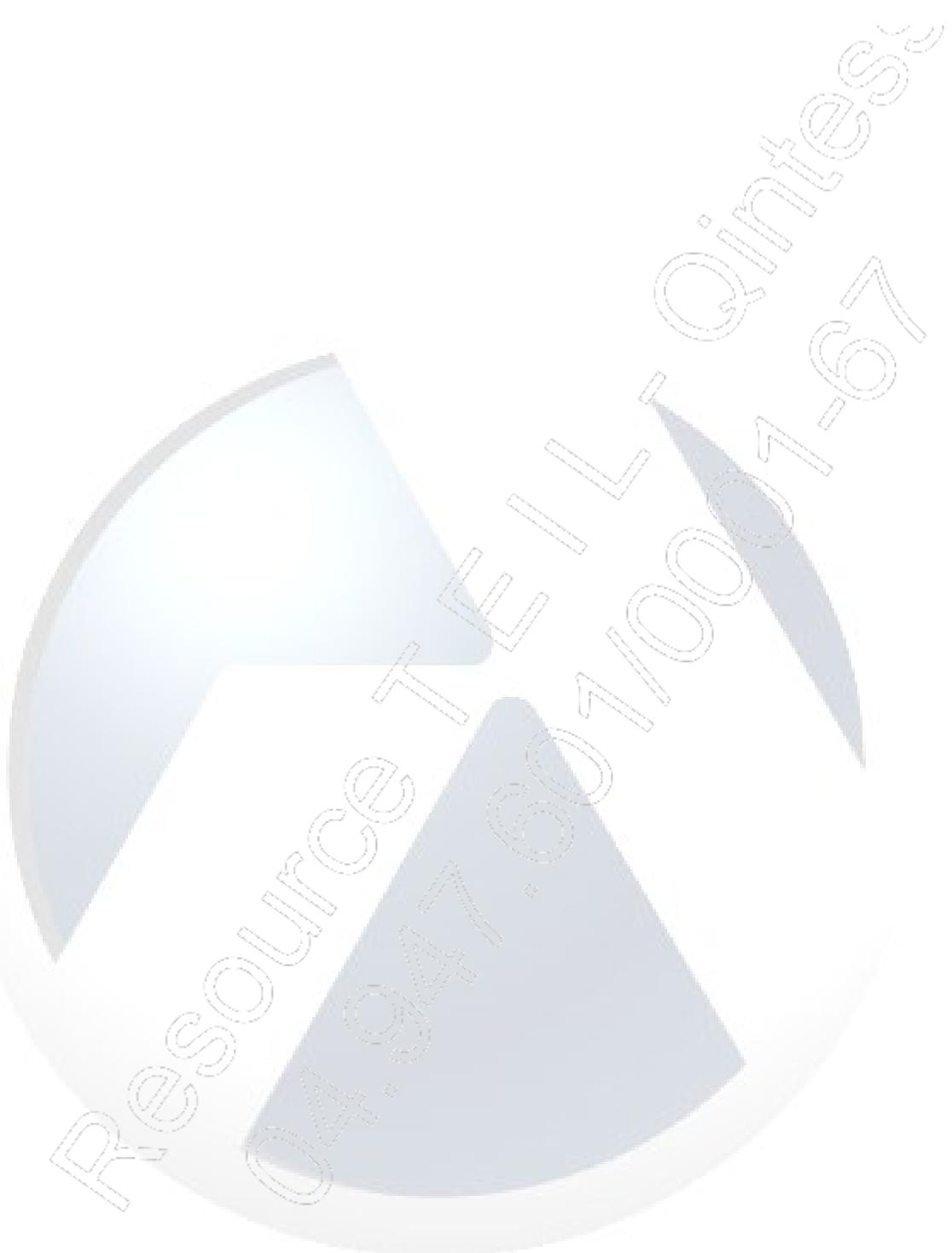
4.7. Inicie apagando o conteúdo interno da div. Substitua-o pelo código indicado a seguir e compare o código já existente com a tabela de alunos, de forma a compreender sua funcionalidade:

```
<div class="content-module-main">
    <table>
        <thead>
            <tr>
                <th><input type="checkbox" id="table-select-all"></th>
                <th>Nome</th>
                <th>Disciplina</th>
                <th>Email</th>
                <th>Ações</th>
            </tr>
        </thead>
        <tfoot>
            <tr>
                <td colspan="5" class="table-footer"><label
                    for="table-select-actions">Executar nos itens selecionados:</label>
                    <select id="table-select-actions">
                        <option value="option1">Apagar</option>
                        <option value="option2">Exportar</option>
                        <option value="option3">Arquivar</option>
                    </select>
                    <a href="#" class="round button blue text-upper small-button">
                        Aplicar a todos os selecionados
                    </a>
                </td>
            </tr>
        </tfoot>

        <tbody>
            <%-- Início do bloco de repetição --%>
            <c:forEach items="${professores}" var="prof">
                <tr>
                    <td><input type="checkbox"></td>
                    <td>${prof.nome}</td>
                    <td>${prof.disciplina}</td>
                    <td><a href="#">${prof.email}</a></td>
                    <td><a href="#" class="table-actions-button ic-table-edit"></a>
                        <a href="#" class="table-actions-button ic-table-delete"></a></td>
                </tr>
            </c:forEach>
            <%-- Fim do bloco de repetição --%>
        </tbody>
    </table>
</div>
```

4.8. Perceba que a tabela é moldada para os campos da entidade **Professor** e, após montada, terá aparência semelhante à de **alunos**, porém, com colunas específicas. O uso da tag de iteração **<c:forEach>** e de EL no bloco é semelhante ao da tabela de alunos.

5. Execute a aplicação e analise o resultado. Agora temos a página **sistema.jsp** com as tabelas de alunos e professores já populadas e todo o código baseado em EL.



# Capítulo 9:

# Tratamento de erros

# em aplicações Web



# Introdução

Em toda aplicação Web, é fundamental a preocupação com o tratamento de erros, de forma a guiar o usuário na direção certa para perfeito uso do sistema.

A primeira ideia que surge quando se toca no assunto de tratamento de erros é o uso de exceções e mensagens personalizadas reproduzidas na interface para ciência do usuário. No ambiente Web, esse conceito vai além da simples ciência da natureza do erro. É preciso lidar com erros de sintaxe e semântica de programação, erros de validação por entrada de dados inválidos e erros HTTP, lançados pelo servidor Web.

Na plataforma Java, é possível configurar a aplicação Web para mapear tanto os erros provindos de exceções quanto os erros de HTTP para uma determinada página JSP, e então personalizar seu conteúdo de forma a entreter e informar o usuário em uma situação anormal. Além disso, é possível prever a ocorrência de erros em quaisquer dos componentes Java do lado servidor, sejam Servlets, JSPs, filtros, listeners, etc. Veremos o tratamento e mapeamento de erros em Servlets e os procedimentos aplicados servirão de base para os demais componentes, com exceção da JSP, cujo tratamento será apresentado mais adiante.

# Tratamento de erros em Servlets

Em um Servlet, todo o código e lógica de aplicação tendem a localizar-se nos métodos de serviço voltados a cada uma das modalidades de requisição HTTP. Os mais utilizados são o **doGet(...)** e **doPost(...)**. Ao se atentar para a assinatura desses métodos, percebe-se que, por padrão, eles lançam dois tipos de exceções apenas: **ServletException** e **IOException**.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

Dessa forma, qualquer exceção que se queira lançar de dentro de um dos métodos de serviço de um Servlet, deverá conformar-se com um dos dois tipos descritos. A tentativa de sobrescrever um dos métodos alterando a quantidade ou o tipo da Exception gera erro de compilação.

A forma de se contornar esse problema e criar exceções personalizadas com o fim de lançá-las via Servlet é criar uma classe que herde de um dos dois tipos previstos pelos métodos de serviço. Vejamos como é o código de uma exceção personalizada, somente com a implementação padrão dos construtores da classe pai, nesse caso, **ServletException**:

```
package exceptions;

import javax.servlet.ServletException;

public class AcessoProibidoException extends ServletException {

    private static final long serialVersionUID = 5687104463928675470L;

    public AcessoProibidoException() {
    }

    public AcessoProibidoException(String message) {
        super(message);
    }

    public AcessoProibidoException(Throwable rootCause) {
        super(rootCause);
    }

    public AcessoProibidoException(String message, Throwable rootCause) {
        super(message, rootCause);
    }
}
```

Com uma exceção particular, encapsulada no formato de uma **ServletException**, é possível lançá-la e configurar nossa aplicação para capturá-la, direcionando o usuário a uma página de erro personalizada.

A seguir, o código de um Servlet que lança a exceção criada, quando o acesso do usuário não corresponder ao nível desejado:

```
package servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import exceptions.AcessoProibidoException;

@WebServlet("/validaLogin")
public class ValidaLoginServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public ValidaLoginServlet() {
        super();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String nivelAcesso = request.getParameter("nivelAcesso");

        if (!nivelAcesso.equals("admin")) {
            throw new AcessoProibidoException("Você não tem permissão de acesso!");
        }
    }
}
```

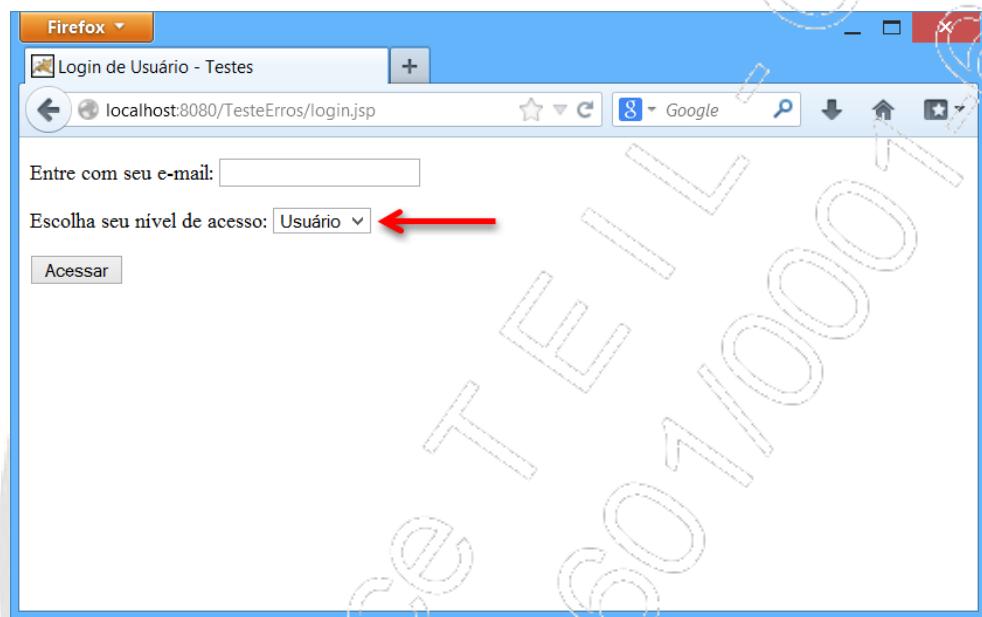
Para utilizar a funcionalidade descrita, será criada uma página de login simples, com o intuito de possibilitar o envio de parâmetros via POST para o Servlet anterior e confirmar o lançamento da exceção diretamente pelo navegador.

Vejamos o código desta página de login criada:

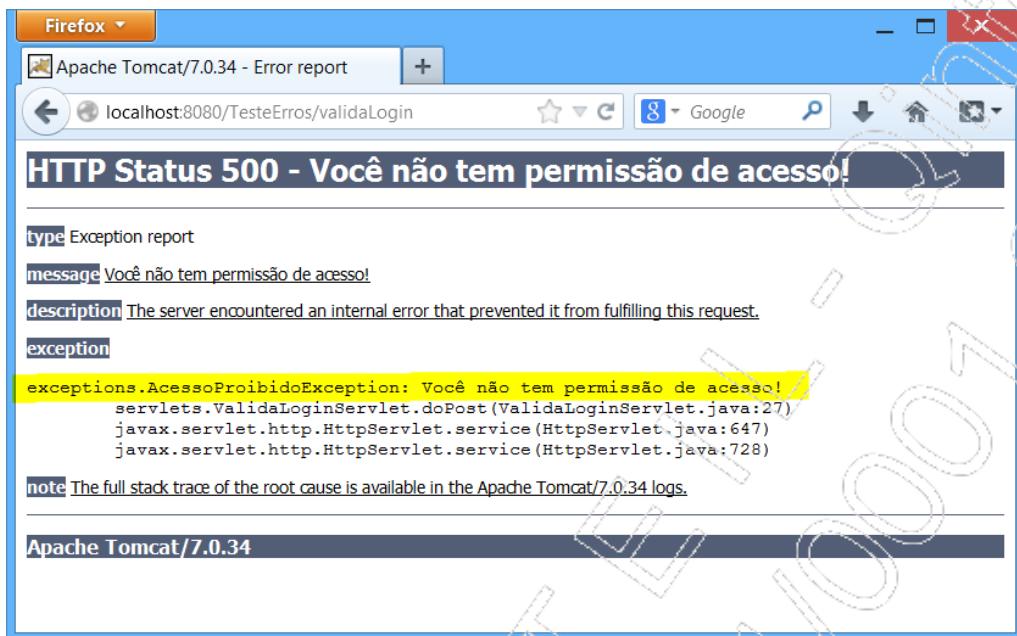
```
<%@page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>

<!DOCTYPE html>
<html lang="pt">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Login de Usuário - Testes</title>
</head>
<body>
<form action="validaLogin" method="post">
<p>Entre com seu e-mail: <input type="text" name="email" /> </p>
<p>Escolha seu nível de acesso:
<select name="nivelAcesso">
<option value="admin">Admin</option>
<option value="user">Usuário</option>
</select> </p>
<p><input type="submit" value="Acessar"></p>
</form>
</body>
</html>
```

Esta página possui um componente do tipo caixa de seleção, que permite a escolha das opções **Admin** ou **Usuário**. Com a última opção encaminhada por parâmetro, o Servlet lançará a Exception **AcessoProibidoException**, que será exibida na janela do navegador para o usuário, tendo em vista que nenhum tratamento foi feito ainda. Vejamos a tela de login após uma execução do aplicativo:



Com a opção **Usuário** já selecionada, clicando-se em **Acessar**, obtém-se a janela com a descrição do erro: a exceção lançada pelo Servlet:



Essa página pode variar de servidor para servidor. Para o Apache Tomcat, essa é a aparência padrão. Note que a pilha de rastreamento da chamada de métodos é mostrada resumidamente na tela do usuário e a mensagem configurada para a exceção em questão foi apresentada automaticamente na ocorrência do erro. Note ainda que o servidor encapsula o erro gerado em um status HTTP de erro código 500 – Erro interno do servidor.

# Criando e configurando uma página de erro personalizada

Para poder transmitir uma sensação maior de segurança e estabilidade ao usuário de nossa aplicação, é sempre importante impor uma padronização às páginas e seções das interfaces. Com o mecanismo de tratamento de erros não deve ser diferente.

Uma página de erro personalizada passa uma ideia de controle e planejamento prévio. O usuário sente que o erro que está enfrentando é algo que já havia sido previsto e mapeado pela equipe de desenvolvedores.

Criaremos uma página de erro personalizada, baseada no layout da aplicação de cadastro que estivemos construindo nos últimos laboratórios. Essa página possuirá o intuito de ser exibida sempre que um erro ocorrer, apresentando a descrição do erro ao usuário final de uma forma compreensível.

A página criada foi denominada de `Erro.jsp` e possui exatamente os mesmos requisitos que toda JSP. Uma única configuração, no entanto, faz-se necessária: a inserção de um atributo na diretiva `@page`.

```
isErrorPage="true"
```

Esse atributo é necessário para que o container saiba que essa página deverá ser tratada como uma página de erro e, a partir dessa declaração, tornar disponível à página o objeto implícito `exception`, que encapsula a exceção capturada na aplicação e a torna disponível para utilização na página.

Vejamos o código dessa página:

```

1 <%@page language="java" contentType="text/html; charset=utf-8"
2     isErrorPage="true" pageEncoding="utf-8"%>
3 <!DOCTYPE html>
4 <html lang="pt">
5 <head>
6     <jsp:include page="header.jsp">
7         <jsp:param value="Cadastro de Alunos - Erro" name="title"/>
8     </jsp:include>
9 </head>
10 <body>
11     <!-- Header -->
12     <div id="header">
13
14         <div class="page-full-width cf">
15
16             <div id="login-intro" class="fl">
17
18                 <h1>Erro!</h1>
19                 <h5>Ocorreu um erro na aplicação.</h5>
20
21             </div> <!-- login-intro -->
22
23             <!-- Logo da Aplicação -->
24             <!-- Esse logo será automaticamente redimensionado para 39px height. -->
25             <a href="#" id="company-branding" class="fr">
26                 
27             </a>
28         </div>
29
30     </div> <!-- fim header -->
31
32     <!-- Formulário Principal -->
33     <div id="content">
34
35         <h1>Erro: <%=exception.getMessage()%></h1>
36         <h4>Detalhe: <%=exception.getClass().getName()%></h4>
37
38     </div>
39     <!-- fim formulário principal-->
40 </body>
41 </html>
```

Vale a pena notar a utilização do objeto implícito **exception** nas linhas 35 e 36 como forma de exibir o erro para o usuário.

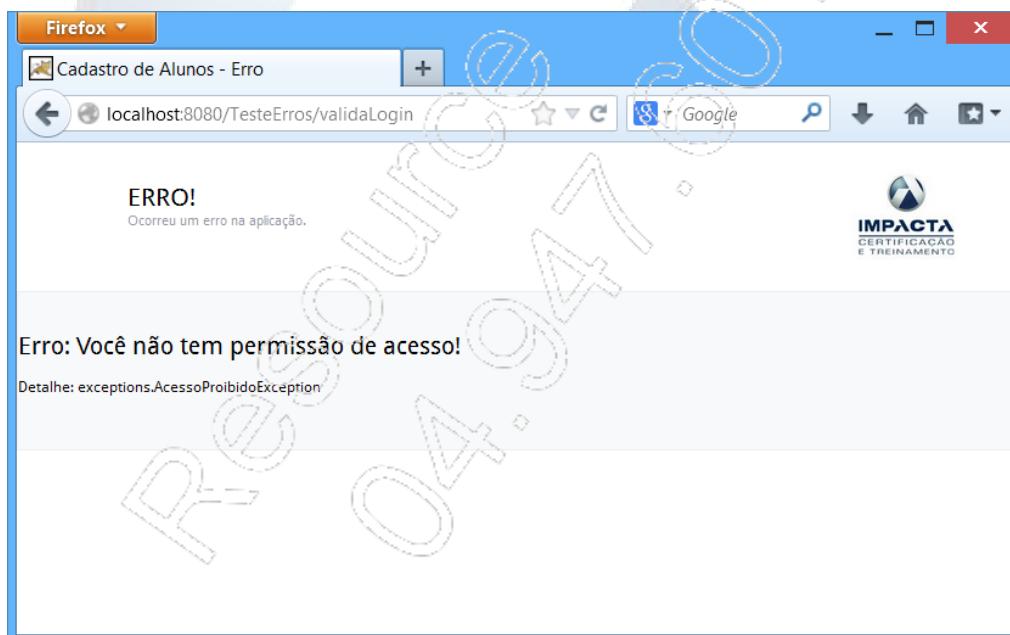
Criada a página, é preciso realizar as necessárias configurações no arquivo **web.xml**, de forma a instruir o servidor para redirecionar o usuário para a página criada, sempre que ocorrer uma exceção deste tipo.

O seguinte conteúdo deverá ser inserido no Deployment Descriptor de forma a possibilitar o redirecionamento por erro:

```
<error-page>
    <exception-type>exceptions.AcessoProibidoException</exception-type>
    <location>/Erro.jsp</location>
</error-page>
```

O par de tags `<error-page>` deve ser inserido diretamente no interior das tags `<web-app>` e basicamente é composto por duas tags mais internas que representam, respectivamente, o tipo de exceção a ser lançado (pode ser usado qualquer caractere curinga nesse campo) e a sua localização exata, em relação à raiz da aplicação.

Com a configuração feita e após executada a aplicação, na ocorrência do erro criado anteriormente, obtemos como saída a seguinte página:



## Tratando erros HTTP

Da mesma forma que foi feito o mapeamento no Deployment Descriptor para o tratamento de exceções, é possível fazê-lo também para erros HTTP. Os erros HTTP seguem códigos internacionalmente padronizados que denotam estados de erro no servidor Web. Os mais comuns são:

Código do Erro	Significado
400	<b>Bad Request:</b> O pedido não pode ser entregue porque a sintaxe está incorreta.
401	<b>Unauthorized:</b> Utilizado especificamente quando a autenticação é possível mas não foi aprovada. Similar ao erro 403 <b>Forbidden</b> .
403	<b>Forbidden:</b> O pedido foi legal, mas o servidor está se recusando a responder.
404	<b>Not Found:</b> O recurso solicitado não foi localizado.
407	Autenticação de proxy necessária.
408	<b>Timeout perdido:</b> O usuário não apresentou uma requisição dentro do período esperado pelo servidor.
409	<b>Conflito:</b> Ocorreu conflito no pedido ou requisição.
500	<b>Internal Error:</b> Ocorreu um erro interno no servidor.
501	<b>Not Implemented:</b> O servidor não implementa a funcionalidade requerida para atender a requisição.
503	<b>Server Unavailable:</b> Servidor sobrecarregado sem condições de atender a requisição no momento.

O código no DD para a configuração do redirecionamento para um erro comum como o 404, por exemplo, segue esta sintaxe:

```
<error-page>
  <error-code>404</error-code>
  <location>/Erro.jsp</location>
</error-page>
```

A partir de um Servlet, é possível lançar um erro HTTP no objeto response, de forma a encaminhá-lo via servidor HTTP ao usuário. Por meio da referência ao objeto **HttpServletResponse**, dois métodos estão disponíveis:

```
sendError(int statusCode);  
sendError(int statusCode, String message);
```

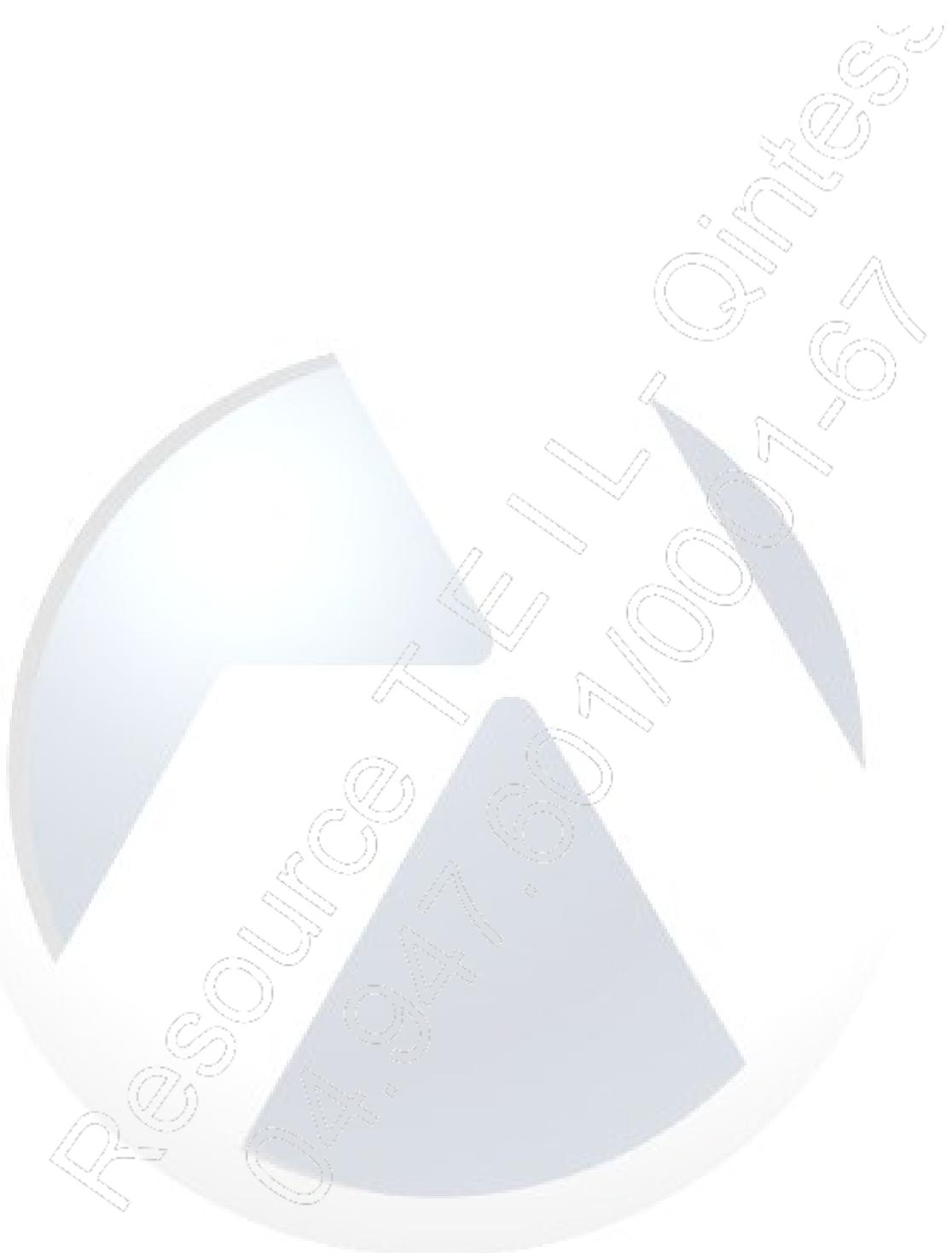
## Tratamento de erros em JSPs

Os erros em JSPs podem ser tratados da mesma forma vista anteriormente, usando uma configuração padrão para redirecionamento via DD, ou pode-se utilizar um atributo especial da diretiva **@page** das JSPs, que permite dispensar o uso de configurações no arquivo **web.xml**. Esse atributo é o **errorPage** e tem a seguinte sintaxe:

```
errorPage="Erro.jsp"
```

Usando esse atributo, os erros ocorridos na JSP em questão geram redirecionamento automático para a página de erro indicada. Uma vez na página de erro, o uso do objeto implícito ocorre da mesma forma descrita anteriormente para o caso de Servlets.

O uso desse atributo é geralmente voltado para a captura de exceções de Runtime, aquelas não checadas pelo compilador e que podem ocorrer por descuidos graves em tempo de execução.



# Tratamento de erros em aplicações Web

Teste seus conhecimentos



**1. O que significa tratar erros em uma aplicação Web?**

- a) Cuidar para que todas as exceções potencialmente prejudiciais sejam tratadas em todos os Servlets da aplicação.
- b) Garantir que todas as JSPs da aplicação usem preferencialmente a EL em vez de acesso a dados por Scriptlets, pois a EL é complacente com o acesso a objetos não iniciados (null).
- c) Garantir que a aplicação contenha uma ou mais páginas de erro personalizadas de forma a redirecionar o usuário na ocorrência de qualquer tipo de erro, causando uma sensação maior de segurança e credibilidade.
- d) Não encaminhar de forma alguma parâmetros de requisição que demandem segurança por meio do método GET, uma vez que o Servlet responsável pode não ter condições de capturar erros gerados por esse método HTTP.
- e) Nenhuma das alternativas anteriores está correta.

**2. Qual das alternativas a seguir está correta sobre as exceções lançadas pelos métodos de serviço dos Servlets?**

- a) Podem ser trocadas por tipos mais genéricos de forma a englobar um maior número de erros capturáveis.
- b) Só podem ser de dois tipos, previamente definidos na hierarquia: ServletException e RuntimeException.
- c) Não é possível utilizar uma Exception de tipo personalizado no lugar dos tipos já definidos nos métodos.
- d) Para criar um tipo específico de exceção a ser lançado nos métodos de serviço, é necessário criar uma classe que herde de ServletException ou de IOException.
- e) Podem ser suprimidas se a classe do Servlet for declarada abstrata.

**3. O que se utiliza para mapear uma exceção para uma página JSP via DD?**

- a) O conjunto de tags <error-page> e, dentro destas, um conjunto <url-pattern> e outro conjunto <error-code>.
- b) O conjunto de tags <error-page> e, dentro destas, um conjunto <exception-type> e outro conjunto <location>.
- c) O conjunto de tags <error-pages> e, dentro destas, um conjunto <url-pattern> e outro conjunto <exception-type>.
- d) O conjunto de tags <error-page> na raiz do arquivo XML e, dentro destas, um conjunto <location> e outro conjunto <error-code>.
- e) O conjunto de tags <error-page> na raiz do arquivo XML e, dentro destas, um conjunto <url-pattern> e outro conjunto <exception-type>.

**4. Ao efetuar duas requisições seguidas, o usuário recebeu como resposta inicialmente o código de erro HTTP 404 e, em seguida, o código de erro HTTP 500. O que esse usuário pode concluir?**

- a) O servidor não encontrou o recurso solicitado na primeira requisição e negou-se a retornar o recurso solicitado na segunda requisição.
- b) O servidor não encontrou o recurso solicitado na primeira requisição e relatou erro de indisponibilidade na segunda requisição.
- c) O servidor relatou a existência de conflito na primeira requisição e erro interno na segunda requisição.
- d) O servidor relatou erro interno na primeira requisição e recurso não encontrado na segunda requisição.
- e) Nenhuma das alternativas anteriores está correta.

**5. Qual das alternativas a seguir está correta sobre tratamento de erros e JSPs?**

- a) Na página JSP que representa a efetiva página de erro a ser exibida ao usuário, é obrigatório uso do atributo isErrorPage = “true”.
- b) O principal motivo que justifica o uso do atributo isErrorPage = “true” é a disponibilização do objeto implícito exception à JSP, a partir deste momento.
- c) Nas JSPs cujo intuito seja tratar exceções lançadas em âmbito interno, é obrigatório o uso do atributo errorPage=“<URI da página>”.
- d) O tratamento de erros HTTP não é possível em uma JSP.
- e) Uma JSP deve tratar seus erros internamente e delegar qualquer tratamento de erro HTTP a um Servlet preparado para isso.

# Tratamento de erros em aplicações Web

Mãos à obra!



## Laboratório 1

### A – Criação de página padrão de Erro e configuração da aplicação para redirecionamento

Neste laboratório, criaremos a página de erro padrão da aplicação e a configuração necessária em Deployment Descriptor para tanto. O foco será o erro HTTP 404 – Recurso não localizado.

1. Localize o projeto chamado **LabWeb9\_Cadastro\_Erros\_Inicial**. Copie-o para o workspace atual do Eclipse e importe-o seguindo as orientações fornecidas pelo instrutor;

Este projeto é exatamente igual ao do laboratório do capítulo 8, com todas as modificações desse laboratório já realizadas.

2. Crie uma nova página JSP denominada **Erro.jsp**. Essa página deverá conter o seguinte código:

```

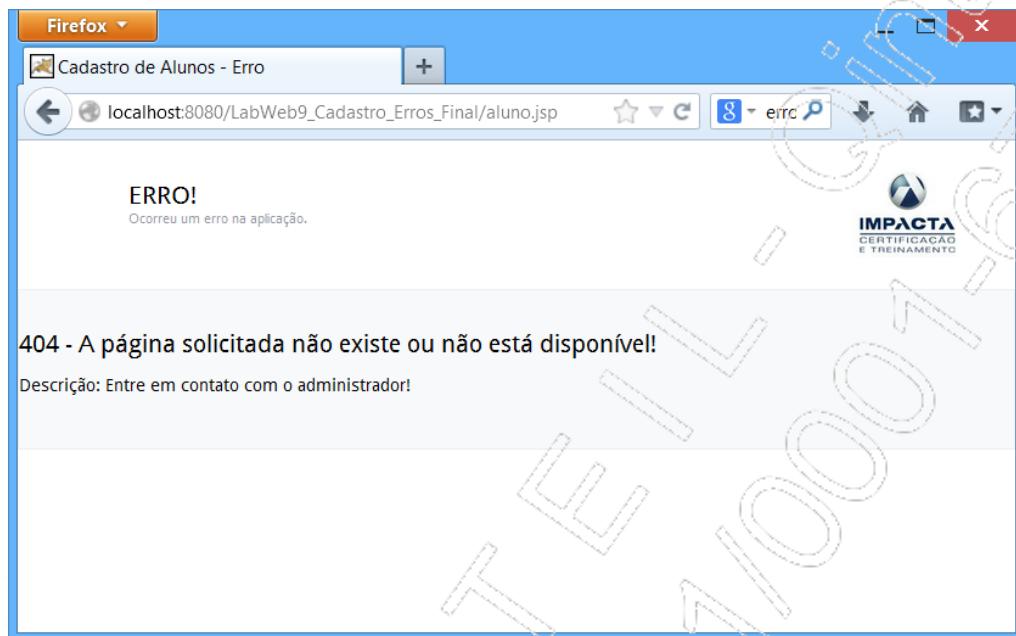
1 <%@page language="java" contentType="text/html; charset=utf-8"
2     isErrorPage="true" pageEncoding="utf-8"%>
3 <!DOCTYPE html>
4 <html lang="pt">
5 <head>
6     <jsp:include page="header.jsp">
7         <jsp:param value="Cadastro de Alunos - Erro" name="title"/>
8     </jsp:include>
9 </head>
10 <body>
11     <!-- Header -->
12     <div id="header">
13         <div class="page-full-width cf">
14             <div id="login-intro" class="fl">
15                 <h1>Erro!</h1>
16                 <h5>Ocorreu um erro na aplicação.</h5>
17             </div> <!-- login-intro -->
18
19             <!-- Logo da Aplicação -->
20             <!-- Esse logo será automaticamente redimensionado para 39px height. -->
21             <a href="#" id="company-branding" class="fr"></a>
22         </div> <!-- end full-width -->
23     </div> <!-- fim header -->
24
25     <!-- Formulário Principal -->
26     <div id="content">
27         <h1>404 - A página solicitada não existe ou não está disponível!</h1>
28         <h3>Descrição: Entre em contato com o administrador!</h3>
29     </div>
30     <!-- fim formulário principal -->
31 </body>
32 </html>
```

3. Altere o Deployment Descriptor para redirecionar os erros HTTP código 404 para a página recém-criada. O código deverá ter a seguinte conformidade:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
6     id="WebApp_ID" version="3.0">
7
8     <welcome-file-list>
9         <welcome-file>login.jsp</welcome-file>
10    </welcome-file-list>
11
12    <error-page>
13        <error-code>404</error-code>
14        <location>/Erro.jsp</location>
15    </error-page>
16 </web-app>
```

4. Execute a aplicação e faça um teste, procurando acessar uma página que não existe, por exemplo, **aluno.jsp**. A resposta de erro é imediata e, a partir de agora, personalizada:



# Capítulo 10:

## Introdução ao uso de tags

# O que são tags e para que servem as JSP Custom Tags?

O uso de tags é uma prática muito comum em aplicações Web, especialmente em locais onde são utilizadas tecnologias baseadas em linguagens de marcação, como HTML e XML. As tags estão presentes em quase todas as tecnologias de desenvolvimento atuais, especialmente em arquivos XML aplicados a uma finalidade específica, seja para a construção de interfaces gráficas ou para a configuração de aplicações em geral.

As tags vêm tomando lugar nas JSPs desde a sua concepção e, na plataforma Java, uma das técnicas mais utilizadas é a criação de tags customizadas, chamadas de **Custom Tags**.

Uma tag customizada é geralmente utilizada para encapsular a complexidade de código Java necessário para acesso a dados ou objetos de negócio, viabilizando a reutilização desse código em outras partes de forma fácil. Uma outra grande vantagem é a retirada de código Java das páginas JSP, favorecendo a interação com profissionais não dedicados à área de desenvolvimento.

As tags customizadas, ao serem criadas, podem ser agrupadas em diretórios ou bibliotecas, denominados **Tag Libraries**. Essas bibliotecas podem ser produzidas de forma personalizada e ainda é possível utilizar bibliotecas prontas.

## Criando e utilizando Custom Tags em JSPs

Veremos, a seguir, a utilização de tags simples e com atributos, além de acessar o corpo de uma tag.

## Tags simples

Tags simples são invocadas usando sintaxe XML padrão. Podem ter aparência inline (abertura e fechamento da tag na mesma declaração) ou ainda em bloco. Podem possuir corpo ou não. Supondo uma tag em sua forma mais simples, chamada neste exemplo de **MinhaTag**, hipoteticamente criada na aplicação, sua sintaxe teria a seguinte aparência:

```
<ex:MinhaTag/>
```

Apesar de simples, alguns passos são necessários para a criação da tag anterior. Na prática, a tag será substituída por conteúdo HTML no momento da renderização da página de resposta ao usuário. Vamos aos passos e o código utilizado para a criação e utilização da tag anterior em uma página JSP funcional:

1. Inicialmente, é necessária uma classe que será responsável pelo tratamento dos dados da tag, um objeto chamado de Tag Handler, dada a sua responsabilidade. Para tanto, criamos uma classe que herda da classe **javax.servlet.jsp.tagext.SimpleTagSupport**:
2. Sobrescrevemos o método **doTag()** herdado da classe referenciada anteriormente e provemos código útil para ser impresso na JSP. Vejamos este código:

```

1 package taghandlers;
2
3 import java.io.IOException;
4 import javax.servlet.jsp.JspException;
5 import javax.servlet.jsp.JspWriter;
6 import javax.servlet.jsp.tagext.SimpleTagSupport;
7
8 public class MinhaTag extends SimpleTagSupport {
9
10    @Override
11    public void doTag() throws JspException, IOException {
12        JspWriter out = getJspContext().getOut();
13        out.println("<h1>Texto vindo da Tag customizada!</h1>");
14    }
15 }
```

3. Repare na sintaxe do método **doTag()** e as exceções por ele lançadas, que devem ser mantidas. Na linha destacada, note a obtenção de um objeto de tipo **JspWriter**, que permitirá a escrita de código personalizado no stream de saída denominado **out**, um nome padrão já consagrado na API Servlet. Essa tag, quando usada, imprimirá no HTML resultante o conteúdo **Texto vindo da Tag customizada!** formatado em um header **<h1>**;

4. Agora precisamos criar o arquivo **.tld** - tag library description -, responsável por declarar nossas Custom Tags ao Container, de forma que ele possa encontrá-las sem problemas. Esse arquivo possui uma sintaxe específica que deve ser respeitada, dada a versão da especificação JSP usada. Em nosso caso, usamos a versão JSP 2.2. presente na versão Java EE 6. Esse arquivo estará localizado em **/WEB-INF**, mas essa localização não é a única possível. Ele também pode estar em algum local dentro de META-INF, especialmente para casos em que se deseja encapsular a Tag Library para distribuição por meio de JARs. Vejamos sua sintaxe:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2<taglib xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
3   web-jsptaglibrary_2_2.xsd"
4   xmlns="http://java.sun.com/xml/ns/javaee"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2">
6<tlib-version>1.0</tlib-version>
7<short-name>TLD_Exemplo</short-name>
8<tag>
9   <name>MinhaTag</name>
10  <tag-class>taghandlers.MinhaTag</tag-class>
11  <body-content>empty</body-content>
12</tag>
13</taglib>

```

5. Para utilizar a tag criada em uma JSP real, é preciso incluir uma diretiva que descreva a taglib a ser usada (neste caso, há apenas uma tag, mas poderiam ser dezenas...). Essa diretiva tem o seguinte formato:

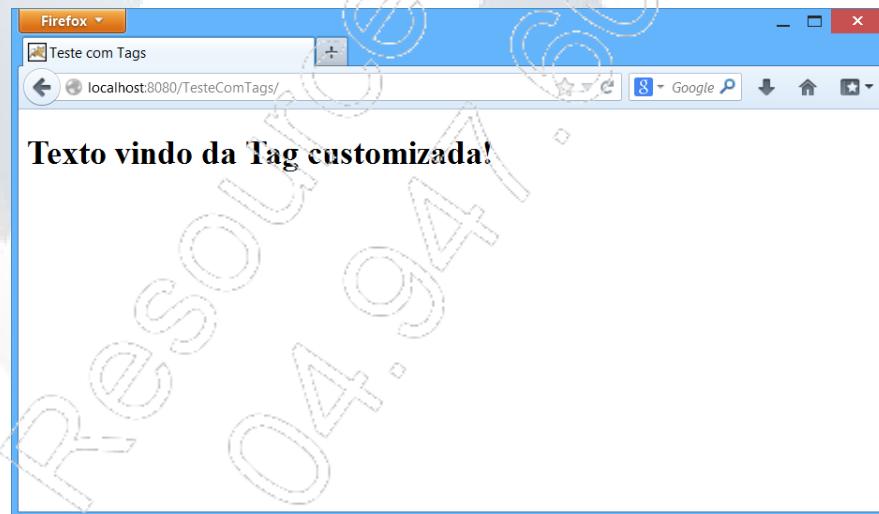
```
<%@ taglib prefix="ex" uri="WEB-INF/impacta.tld"%>
```

Nessa diretiva, definem-se duas importantes instruções para a JSP: o prefixo a ser usado para se referenciar essa tag no corpo da página e a uri, ou seu endereço no projeto, em que o arquivo **.tld** está localizado.

6. Observe a aplicação da tag e o código completo da JSP, nesse caso, denominada de **default.jsp**:

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
2     pageEncoding="utf-8"%>
3 <%@ taglib prefix="ex" uri="WEB-INF/impacta.tld"%>
4
5 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
6     "http://www.w3.org/TR/html4/loose.dtd">
7
8<html>
9<head>
10 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
11 <title>Teste com Tags</title>
12 </head>
13<body>
14     <ex:MinhaTag/>
15 </body>
16 </html>
```

7. Note o uso da tag **<ex:MinhaTag>** no corpo da página. Ao executar essa aplicação, a saída é a seguinte:



## Tags com atributos

É comum que as Custom Tags criadas diante de uma necessidade da aplicação Web demandem certa sofisticação e maleabilidade quanto à configuração de seus atributos. A criação de atributos permite a adequação da tag a diversos cenários e necessidades, possibilitando a criação de componentes reutilizáveis em aplicações Web.

Supondo que a nossa tag criada anteriormente imprima uma mensagem em um formato customizado e que essa mensagem seja recebida por meio de um atributo, vejamos as alterações que serão necessárias na classe e no arquivo **.tld** já criados:

1. Inicialmente, para poder trabalhar com atributos, precisamos criar campos e métodos assessores em nosso Tag Handler para armazenar e manusear os valores passados:

```

1 package taghandlers;
2
3 import java.io.IOException;
4 import javax.servlet.jsp.JspException;
5 import javax.servlet.jsp.JspWriter;
6 import javax.servlet.jsp.tagext.SimpleTagSupport;
7
8 public class MinhaTag extends SimpleTagSupport {
9
10    private String mensagem;
11
12    @Override
13    public void doTag() throws JspException, IOException {
14        JspWriter out = getJspContext().getOut();
15        out.println("<h1>Mensagem recebida: " + getMensagem() + "</h1>");
16    }
17
18    public String getMensagem(){
19        return mensagem;
20    }
21
22    public void setMensagem(String mensagem) {
23        this.mensagem = mensagem;
24    }
25 }
```

Getter e Setter  
necessários!

2. Perceba, no código modificado do nosso Tag Handler, que foram criados: um campo de nome **mensagem**, um getter e um setter para ele, destacados no código. Ainda, na linha grifada, note a inclusão do valor do campo no valor sendo passado para o stream que gerará o conteúdo HTML, ou seja, estamos ecoando no navegador o valor da mensagem passada por atributo em nossa Custom Tag;

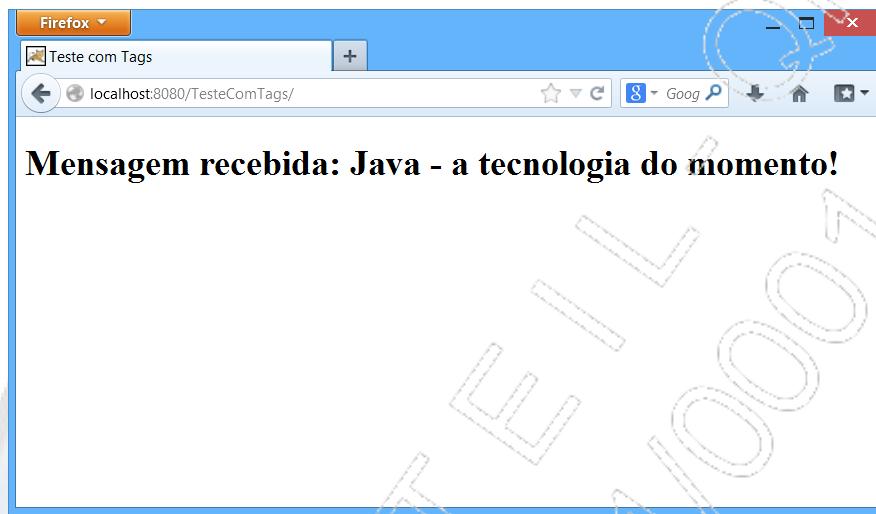
3. Vejamos agora o arquivo **.tld**, modificado para configurar o atributo criado:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <taglib xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
3   web-jsptaglibrary_2_2.xsd"
4   xmlns="http://java.sun.com/xml/ns/javaee"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2">
6   <tlib-version>1.0</tlib-version>
7   <short-name>TLD Exemplo</short-name>
8   <tag>
9     <name>MinhaTag</name>
10    <tag-class>taghandlers.MinhaTag</tag-class>
11    <body-content>empty</body-content>
12    <attribute>
13      <name>mensagem</name>
14    </attribute>
15  </tag>
16 </taglib>
```

4. Perceba que, em relação ao código inserido anteriormente, a única alteração foi a inclusão da tag **attribute**, necessária para o mapeamento de cada atributo presente na tag;

5. Observe a utilização da tag, agora com um atributo, na JSP já utilizada anteriormente e confira, adiante, a saída da página JSP após a execução no Container Java:

```
<ex:MinhaTag mensagem="Java - a tecnologia do momento!"/>
```



## Acessando o corpo de uma tag

Em diversas situações, será necessária a criação de uma tag em estilo bloco, especialmente se a tag representar um container para a inclusão de dados e conteúdo variado de maior extensão. Nesses casos, a tag divide-se em duas porções: uma de abertura, que conterá todos os atributos necessários para a sua execução, e uma outra para o fechamento do bloco de conteúdo. Para a captura do conteúdo do corpo da tag, precisaremos de novas alterações nos mesmos componentes mostrados anteriormente.

Vejamos os passos necessários para a captura do corpo da tag:

1. Para a captura do corpo da tag, é necessário um campo na classe **Tag Handler** para se armazenar o conteúdo e uma chamada no método **doTag()**, de forma a se obter o valor recebido do corpo da tag. Veja o código modificado do Tag Handler:

```

1 package taghandlers;
2
3 import java.io.IOException;
4 import java.io.StringWriter;
5
6 import javax.servlet.jsp.JspException;
7 import javax.servlet.jsp.JspWriter;
8 import javax.servlet.jsp.tagext.SimpleTagSupport;
9
10 public class MinhaTag extends SimpleTagSupport {
11
12     private String mensagem;
13     private StringWriter corpoTag = new StringWriter();
14
15     @Override
16     public void doTag() throws JspException, IOException {
17         getJspBody().invoke(corpoTag);
18         JspWriter out = getJspContext().getOut();
19         out.println("<h3>" + corpoTag.toString() + "</h3>" +
20                     "<h1>" + getMensagem() + "</h1>");
21     }
22
23     public String getMensagem() {
24         return mensagem;
25     }
26
27     public void setMensagem(String mensagem) {
28         this.mensagem = mensagem;
29     }
30 }
```

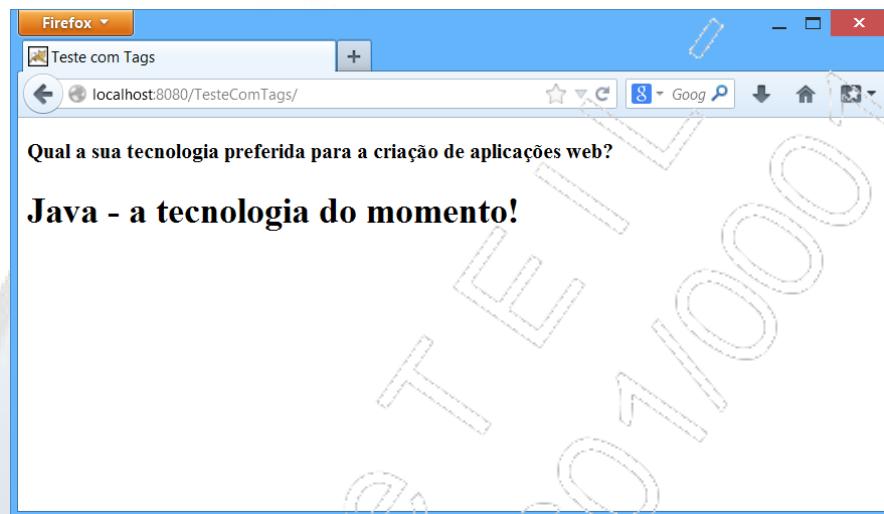
2. Note a linha de código 17, utilizada para se obter o corpo da tag e, após capturado, sua utilização na linha 19, por meio da chamada ao método **toString()** no objeto **StringWriter** (na verdade, é possível utilizar qualquer objeto do tipo Writer);
3. No arquivo **.tld** (descriptor), apenas uma linha exige modificação:

```
<body-content>scriptless</body-content>
```

4. Na página JSP, a chamada da tag, passando conteúdo no corpo, fica de acordo com o código a seguir:

```
<ex:MinhaTag mensagem="Java - a tecnologia do momento!">  
    Qual a sua tecnologia preferida para a criação de aplicações web?  
</ex:MinhaTag>
```

5. Confira a saída da aplicação, após a execução da página:



# JSTL – Java Standard Tag Library

Neste tópico, abordaremos a importância da biblioteca JSTL, destacando seu histórico e suas funções. Além disso, conheceremos as bibliotecas Core e de internacionalização.

## Histórico e definições

JSTL é uma biblioteca padrão de tags JSP que encapsula funções de uso comum, como processamento de XML, execução condicional, laços de repetição, internacionalização, entre outros. A JSTL está presente em grande parte das aplicações Web feitas com a tecnologia Java e é conhecida por toda a comunidade Java desde sua criação.

Essa biblioteca merece grande destaque, porque está presente desde as primeiras versões do Java EE e especificação JSP:

- Em sua versão JSTL 1.0., lançada em 21 de junho de 2002, pela Fundação Apache, era focada e direcionada à versão 1.2. da especificação JSP;
- Em sua versão JSTL 1.1., lançada em 30 de janeiro de 2004, pela Fundação Apache, era focada e direcionada à versão 2.0 da especificação JSP;
- Em 2006, o JCP (Java Community Process) incorpora a especificação JSTL 1.2. (dado o seu grande uso e disseminação na comunidade Java) à sua versão Java EE 5 e, desde então tornou-se um componente obrigatório em servidores Java homologados;
- Em 07 de dezembro de 2011, foi lançada a versão JSTL 1.2.1., que permanece até hoje como a versão mais atual da biblioteca.

Examinaremos em nosso curso algumas das tags de maior importância na biblioteca JSTL, mas oferecemos também o seguinte link para que o aluno possa se aprofundar no estudo da JSTL, de forma a compreender toda a variedade de tags disponíveis:

[<http://docs.oracle.com/javaee/5/tutorial/doc/index.html>](http://docs.oracle.com/javaee/5/tutorial/doc/index.html).

Esse endereço fornece acesso ao tutorial Oracle para a especificação Java EE 5, um bom ponto de partida para compreender a especificação JSTL 1.2. De qualquer forma, a página oficial do projeto contém links para download e informações sobre como participar desse projeto, caso haja interesse:  
 [<http://jstl.java.net/>](http://jstl.java.net/).

As funcionalidades oferecidas pela JSTL podem ser resumidas a algumas bibliotecas de tags conforme a tabela a seguir, extraída da documentação oficial Oracle para a plataforma Java EE 5:

Área	Subfunção	Prefixo	URI
Core	Suporte a variáveis	c	<a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>
	Controle de fluxo		
	Gerenciamento de URL		
	Diversas outras aplicações		
XML	Core	x	<a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a>
	Controle de fluxo		
	Transformação		
I18N	Localização	fmt	<a href="http://java.sun.com/jsp/jstl/fmt">http://java.sun.com/jsp/jstl/fmt</a>
	Formatação de mensagens		
	Formatação de número e datas		
Banco de Dados	SQL	sql	<a href="http://java.sun.com/jsp/jstl/sql">http://java.sun.com/jsp/jstl/sql</a>
Funções	Tamanho de coleções	fn	<a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a>
	Manipulação de Strings		

Para a utilização das referidas bibliotecas de tags em uma página JSP, utiliza-se a já vista e conhecida diretiva taglib, com o prefixo e uri indicados pela tabela anterior:

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

# Biblioteca Core

A seguir, vamos conhecer as tags mais importantes da biblioteca Core.

## < C:set >

<b>&lt;c:set&gt;</b>	Permite inserir um par “chave = valor” a algum escopo válido ou a um atributo de um JavaBean. Também permite a inserção de um objeto em um Map.
----------------------	---

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>var</b>	Não	String	Nome da variável em que será realizada a atribuição.
<b>value</b>	Sim	Object	Valor ou expressão (EL) que corresponde à atribuição.
<b>target</b>	Sim	Object	JavaBean ou Map que será alvo da atribuição.
<b>property</b>	Sim	String	Identificador do atributo do JavaBean ou chave do Map em que será feita a atribuição.
<b>scope</b>	Não	String	Escopo em que será feita a atribuição. Por padrão, esse escopo é <b>page</b> .

- Exemplo de uso:

```
<c:set var="nome" value="Um Nome Qualquer" scope="session"/>
```

- Substitui o uso de:

```
<% session.setAttribute("nome", "Um Nome Qualquer"); %>
```

## < c:remove >

**<c:remove>** Permite remover um atributo de um determinado escopo.

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>var</b>	Não	String	Nome do atributo que será removido.
<b>scope</b>	Não	String	Escopo de onde será removido o atributo. O padrão é <b>page</b> .

- Exemplo de uso:

```
<c:remove var="nome" scope="session" />
```

- Substitui o uso de:

```
<% session.removeAttribute("nome"); %>
```

## < c:if >

**<c:if>** Permite a execução de desvio de fluxo condicional simples na página. Executa o conteúdo de seu corpo somente se o teste for verdadeiro.

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>test</b>	Sim	boolean	Expressão de teste que definirá a execução do bloco da tag. Utiliza-se EL para a definição da expressão.
<b>var</b>	Não	String	Nome da variável em que será realizada a atribuição do resultado do teste.
<b>scope</b>	Não	String	Escopo da variável definida em <b>var</b> . Por padrão, esse escopo é <b>page</b> .

- Exemplo de uso (Supondo que um atributo denominado **contador** tenha sido inserido em algum escopo válido da aplicação):

```
<c:if test="#{contador == 35}">
    <h1>Teste passou - JSTL!</h1>
</c:if>
```

- Substitui o uso de:

```
<%
    Long contador = (Long)session.getAttribute("contador");
    if (contador == 35) {
%>
    <h1>Teste passou - Scriptlet!</h1>
<% } %>
```

## < c:choose >

<b>&lt;c:choose&gt;</b>	Permite a execução de desvios de fluxo condicionais mutuamente exclusivos. É parecida com a instrução switch, porém, admite para cada caso um teste lógico completo. É implementada com o auxílio das tags <c:when> (semelhante à instrução case de um bloco switch) e <c:otherwise> (semelhante à instrução default de um bloco switch).
-------------------------	---

- Atributos da tag <c:when> (única aplicável, as demais tags não têm atributos):

Nome	É dinâmico?	Tipo de dado	Função
<b>test</b>	Sim	boolean	Expressão de teste a ser avaliada.

- Exemplo de uso (Supondo que um atributo denominado **contador** tenha sido inserido em algum escopo válido da aplicação):

```
<c:choose>
  <c:when test="#{contador == 34}">
    <h1>Contador vale 34!</h1>
  </c:when>
  <c:when test="#{contador == 35}">
    <h1>Contador vale 35!</h1>
  </c:when>
  <c:otherwise>
    <h1>Contador inválido!</h1>
  </c:otherwise>
</c:choose>
```

## < c:forEach >

<b>&lt;c:forEach&gt;</b>	Representa um laço de repetição que permite a execução de seu corpo n vezes, ou ainda, a iteração sobre um array ou coleção de elementos.
--------------------------	---

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>var</b>	Não	String	Armazena o elemento atual da iteração. Atua também como contador das iterações quando forem usados limites numéricos.
<b>items</b>	Sim	Collection, Map, array, Iterator, Enumeration, String	Coleção ou conjunto de dados a serem iterados.
<b>varStatus</b>	Não	String	Objeto que guarda o status atual da iteração. É opcional nessa tag e geralmente usado como <b>contador</b> (diferentemente do usual, ele inicia a contagem em 1).

Nome	É dinâmico?	Tipo de dado	Função
<b>begin</b>	Sim	int	Limite inferior ou início da iteração.
<b>end</b>	Sim	int	Limite superior ou fim da iteração.
<b>step</b>	Sim	int	Incremento do contador na iteração. Não admite valor negativo.

- Exemplo de uso – laço iterando sobre uma List armazenada no escopo de sessão:

```
<c:forEach items="${alunos}" var="aluno">
  <tr>
    <td><input type="checkbox"></td>
    <td> ${aluno.nome} </td>
    <td> ${aluno.userName} </td>
    <td><a href="#"> ${aluno.email} </a></td>
    <td> ${aluno.idade} </td>
    <td><a href="#" class="table-actions-button ic-table-edit"></a>
      <a href="#" class="table-actions-button ic-table-delete"></a></td>
  </tr>
</c:forEach>
```

- Exemplo de uso – laço fixo:

```
<c:forEach var="cont" begin="1" end="10" step="2">
  <h1>Contagem: ${cont}</h1>
</c:forEach>
```

## Biblioteca de internacionalização e mensagens

Vejamos a seguir algumas tags de biblioteca de internacionalização e mensagens.

## <fmt:setLocale>

Em aplicações Java em geral, quando existe a necessidade de localização, um objeto específico é utilizado para representar uma localidade, o objeto `java.util.Locale`. Esse objeto é insumo para a configuração de diversos formatadores, como aqueles voltados a datas e moeda.

<code>&lt;fmt:setLocale&gt;</code>	Tag que configura o objeto Locale a ser usado no escopo definido.
------------------------------------	---

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>value</b>	Sim	String ou <code>java.util.Locale</code>	Objeto <code>Locale</code> a ser utilizado ou a String que representa a localidade, por exemplo, <code>en_US</code> (Inglês - EUA).
<b>variant</b>	Sim	String	Configuração opcional de idioma adicional.
<b>Scope</b>	Não	String	Escopo em que será armazenado o objeto <code>Locale</code> .

- Exemplo de uso (configuração de um objeto `Locale` para o idioma inglês - EUA no escopo de sessão):

```
<fmt:setLocale value="en_US" scope="session"/>
```

## <fmt:setBundle>

<code>&lt;fmt:setBundle&gt;</code>	Tag usada para adicionar um <code>ResourceBundle</code> a um determinado escopo. O escopo <code>page</code> é o padrão. Esse objeto permite extrair todas as Strings utilizadas na aplicação de um ou mais arquivos localizados, com extensão <code>.properties</code> , cuja configuração deve ser precedida por uma tag <code>&lt;fmt:setLocale&gt;</code> .
------------------------------------	--

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>basename</b>	Sim	String	Nome do arquivo principal utilizado como fonte para o ResourceBundle.
<b>var</b>	Não	String	Nome da variável em que será armazenada o ResourceBundle.
<b>scope</b>	Não	String	Escopo da variável definida no atributo <b>var</b> .

- Exemplo de uso:

```
<fmt:setBundle basename="bundles.cadastro" var="Lang" />
```

## < fmt:message >

<b>&lt;fmt:message&gt;</b>	Tag usada para inserir uma mensagem definida em um ResourceBundle por meio de uma chave declarada.
----------------------------	--

- Atributos da tag:

Nome	É dinâmico?	Tipo de dado	Função
<b>key</b>	Sim	String	Chave da mensagem para obtenção no ResourceBundle.
<b>bundle</b>	Sim	Variável a ser utilizada (usa-se EL)	Variável ResourceBundle a ser usada. Geralmente usa-se EL para referenciar uma variável declarada por meio da tag definida no item anterior.
<b>var</b>	Não	String	Nome da variável para se armazenar a mensagem (opcional).

Nome	É dinâmico?	Tipo de dado	Função
scope	Não	String	Escopo da variável definida por var.

- Exemplo de uso (Essa tag dispensa existência das duas anteriores para poder ser usada):

```
<fmt:setLocale value="pt_BR" />
<fmt:setBundle basename="bundles.cadastro" var="Lang" />
<fmt:message bundle="\${lang}" key="sistema.aba1"/>
```

Perceba que o excerto anterior declara um objeto **Locale** configurado para o idioma português do Brasil e, em seguida, configura um  **ResourceBundle** no escopo da página cujo nome base é **bundles.cadastro**. Na verdade, **bundles** é o pacote em que se encontram os arquivos e o nome do arquivo para o caso de português é **cadastro\_pt\_BR.properties** – perceba o sufixo após o nome base que equivale ao **Locale** configurado.

Onde a tag **<fmt:message>** for colocada será inserido o texto equivalente do arquivo  **ResourceBundle**. Vejamos o conteúdo desse arquivo:

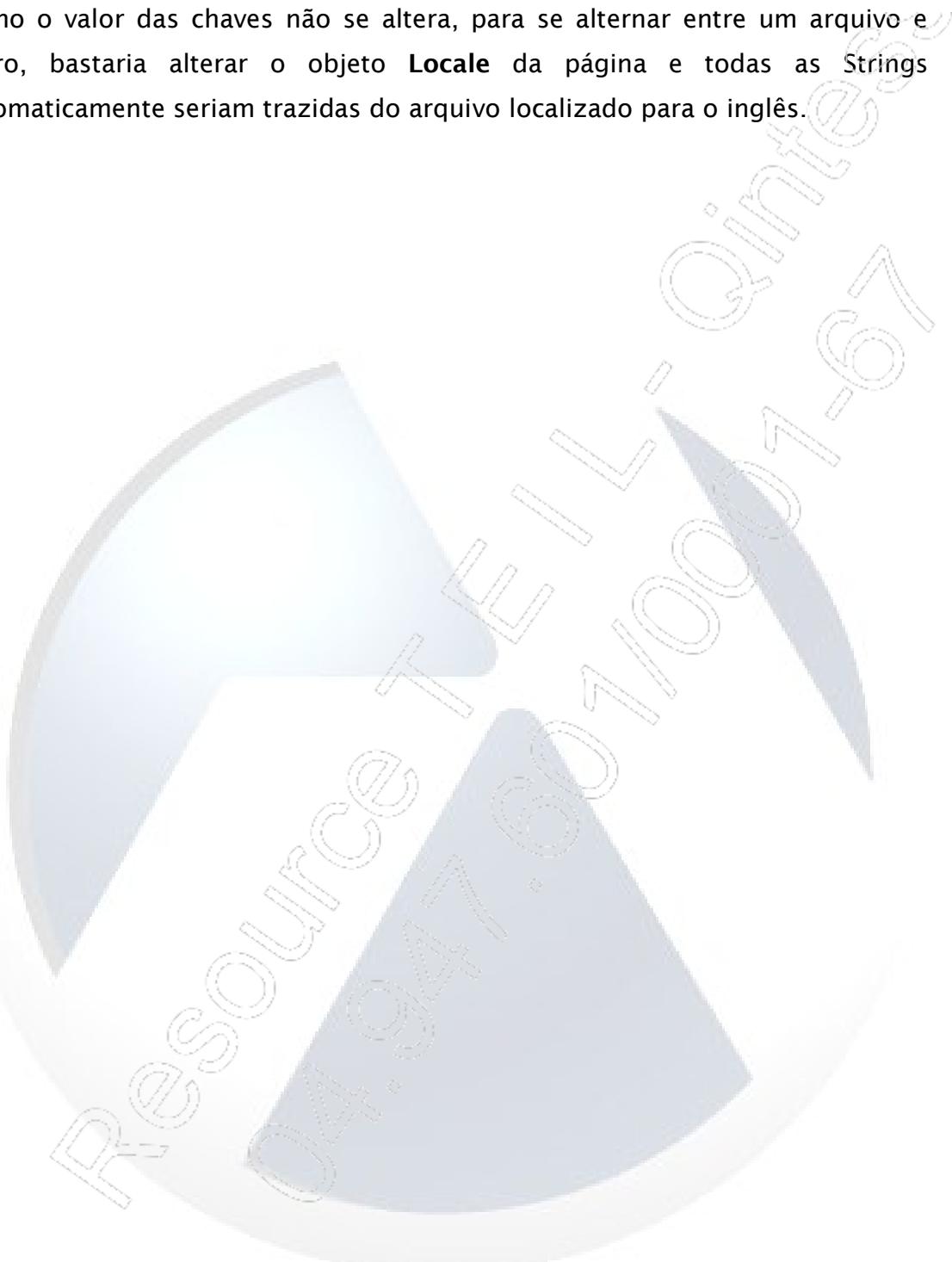
- Arquivo **cadastro\_pt\_BR.properties**:

```
sistema.aba1=Cadastro
sistema.aba2=Outra página qualquer
```

Um mesmo arquivo voltado para o idioma inglês seria denominado **cadastro\_en\_US.properties** e teria o seguinte conteúdo:

```
sistema.aba1=Register
sistema.aba2=Any other page
```

Como o valor das chaves não se altera, para se alternar entre um arquivo e outro, bastaria alterar o objeto **Locale** da página e todas as Strings automaticamente seriam trazidas do arquivo localizado para o inglês.



# Introdução ao uso de tags

Teste seus conhecimentos



**1. Qual das alternativas a seguir não está correta sobre os benefícios oferecidos pelo uso de Custom Tags?**

- a) Facilitar a reutilização de código entre diversas páginas JSP, proporcionando maior agilidade e componentização de código.
- b) Encapsular a complexidade do código Java necessário para acesso a dados ou objetos de negócio.
- c) Retirar o código Java literal das páginas JSP, facilitando a interação entre desenvolvedores e membros não programadores do time.
- d) Prover novas funcionalidades antes não disponíveis por meio das JSP.
- e) Viabilizar a padronização de páginas por meio de reutilização de tags e, por conseguinte, a manutenção de um padrão de identidade visual.

**2. Na criação de uma tag simples, alguns passos e detalhes são importantes para se construir a infraestrutura que dará suporte ao uso dessa tag. Qual das afirmações a seguir está correta a respeito dessa infraestrutura?**

**I. É fundamental que ocorra a criação de um elemento chamado Tag Handler, que consiste em uma classe Java simples que herda de SimpleTagSupport e que deve sobrescrever o método public void doTag().**

**II. Durante a compilação da página JSP, no local em que se utilizou a Custom Tag, será feita a substituição pelo conteúdo impresso por meio do Tag Handler no stream de saída.**

**III. O arquivo descritor da Custom Tag, denominado tld, é onde serão alocadas as configurações e declarações das tags criadas, e deverá estar localizado obrigatoriamente no diretório WEB-INF.**

- a) As afirmações I e III estão corretas.
- b) Apenas a afirmação I está correta.
- c) Apenas a afirmação II está correta.
- d) As afirmações II e III estão corretas.
- e) As afirmações I e II estão corretas.

**3. Qual das alternativas a seguir não está correta sobre o uso adequado de atributos e a captura de dados colocados no corpo de uma Custom Tag?**

- a) Os atributos de uma Custom Tag devem ter campos equivalentes na classe Tag Handler e métodos assensores para sua correta utilização.
- b) A JVM inferirá os atributos existentes nas tags e os comparará com os atributos no Tag Handler, de forma que não há obrigatoriedade de declará-los no tld.
- c) Para a captura do corpo de uma tag, utilizamos o método invoke(...) passando em seu parâmetro o stream que será responsável pela captura do texto passado.
- d) Uma modificação no arquivo tld é necessária para a correta captura do corpo da tag.
- e) Um objeto comumente utilizado para capturar o texto passado no corpo de uma tag é o StringWriter.

**4. Qual das alternativas a seguir está correta sobre a JSTL?**

- a) É uma biblioteca de tags padrão que encapsula funções de uso comum no desenvolvimento de aplicações Web na plataforma Java.
- b) É uma biblioteca recente e apresenta certas incompatibilidades com as versões mais recentes da plataforma Java EE.
- c) Como foi criada pela Fundação Apache, tem sua importância reconhecida no meio Java, porém, não faz parte das especificações do Java EE.
- d) É composta por poucas bibliotecas e ainda dispensa a criação de uma biblioteca para controlar e facilitar a localização de mensagens em páginas JSP.
- e) Nenhuma das alternativas anteriores está correta.

5. Para a internacionalização de páginas JSP, a JSTL possui uma biblioteca específica, de grande valor prático. Qual das alternativas a seguir está correta sobre a utilização das tags relacionadas a esse tema?

- a) Um objeto fundamental para o uso da biblioteca é o `java.util.Locale`, utilizado para representar a localização e o idioma a ser utilizado na página JSP.
- b) A tag `<fmt:setBundle>` deve ser utilizada para adicionar um `ResourceBundle` à página ou a qualquer outro escopo desejado, que consiste em um objeto-repositório de strings localizadas.
- c) Um objeto `ResourceBundle` representa um arquivo de texto composto por um nome base, um sufixo que pode indicar idioma ou idioma e localização e uma extensão `.properties`.
- d) Para a inserção de uma mensagem string localizada em um bundle, deve-se utilizar a tag `<fmt:message>` declarando-se o bundle adequado a ser usado como parâmetro.
- e) Todas as alternativas anteriores estão corretas.

# Introdução ao uso de tags

**Mãos à obra!**



## Laboratório 1

### A – Adequação e ajustes da aplicação já construída para o uso de JSTL e localização de strings

Neste laboratório será abordado o uso da JSTL, a fim de comprovar seus benefícios. Utilizaremos a biblioteca core e a biblioteca de internacionalização para extrair algumas strings do texto hardcoded na JSP e colocá-lo em um bundle externo.

1. Observe o código da página **sistema.jsp** e perceba que já estamos usando JSTL há algum tempo. É possível observar a existência, no topo da página, de uma diretiva **@taglib**, incluindo a biblioteca **core** da JSTL para uso nessa JSP:

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

2. Insira uma tag idêntica e com os mesmos atributos na página **login.jsp** logo abaixo da diretiva **@page**, de forma a se habilitar a usar a biblioteca core da JSTL nessa página também;

3. Localize na página **login.jsp**, a partir da linha 51, o seguinte trecho de código:

```
<%
    String message = (String)request.getAttribute("message");
    if(!(message == null || message.equals("")))) {
%>
    <br /><br />
    <div class="error-box round">
        <%=message %>
    </div>
<%
}
%>
```

Esse trecho utiliza dois blocos Scriptlet. Nesse conteúdo, uma linha captura um atributo do contexto de request e o armazena em uma variável local chamada **message**. A seguir, insere um bloco **div** na página se e somente se o desvio condicional existente no bloco **if** for verdadeiro.

3.1. Substitua o trecho anterior por código JSTL, mais precisamente, por tags da biblioteca **core**, de forma a manter a mesma funcionalidade sem o uso de Scriptlets. Ao término, o código final deverá se parecer com o seguinte:

```
<c:if test="${not empty message or message == ''}">
    <br /><br />
    <div class="error-box round">
        ${message}
    </div>
</c:if>
```

4. A fim de prover internacionalização nesta aplicação, crie um pacote chamado **bundles** na pasta **src**. Dentro desse pacote, crie dois arquivos de texto, conforme indicado a seguir:

4.1. Clique com o botão direito do mouse sobre a pasta **bundles** e acesse **New / Other / General / Untitled Text File**;

4.2. Execute novamente o passo 4.1., e renomeie cada um dos arquivos para:

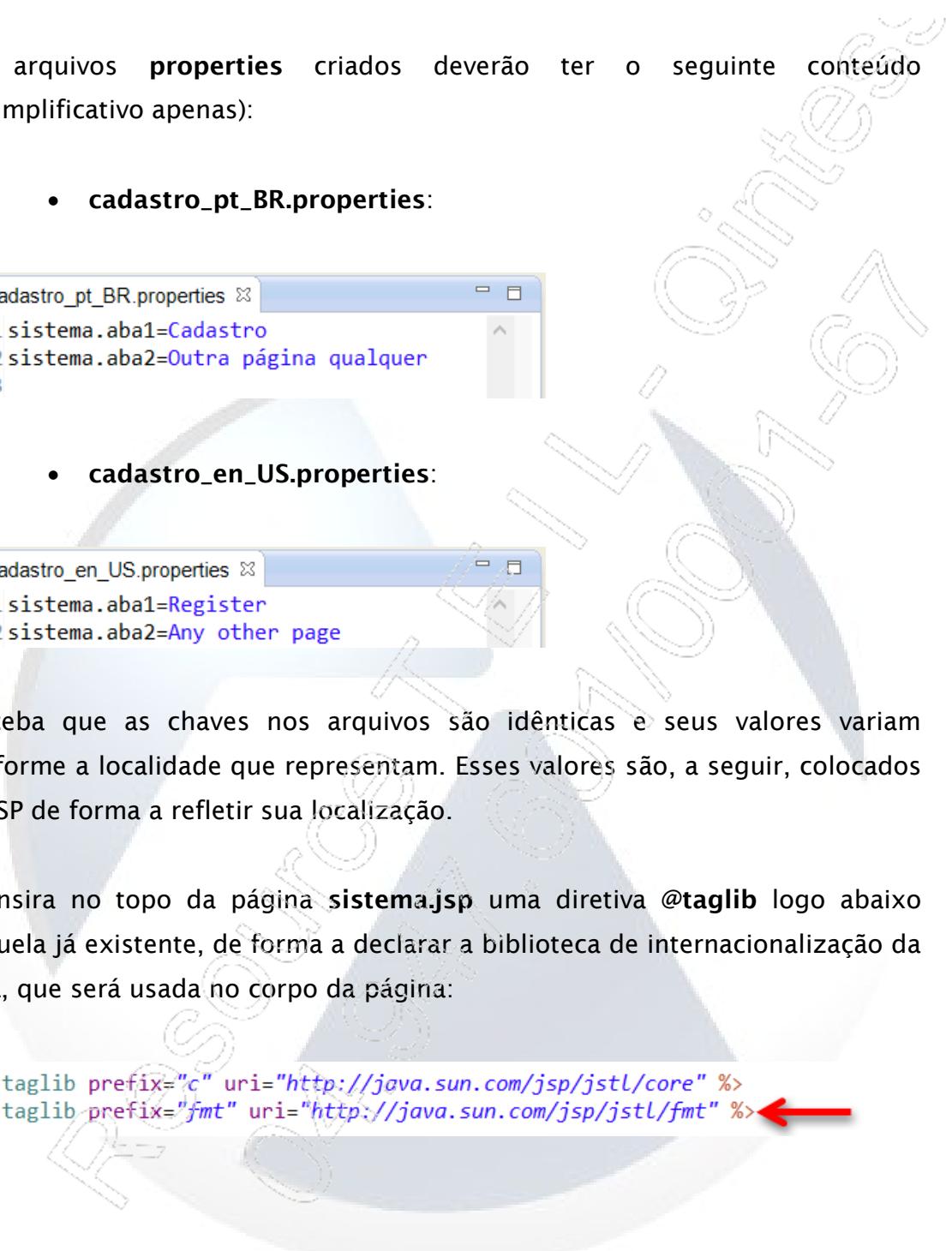
- **cadastro\_pt\_BR.properties**;
- **cadastro\_en\_US.properties**.

A estrutura de arquivos da pasta Java Resources no projeto agora deve estar da seguinte maneira:

```
▲ Java Resources
  ▲ src
    ▲ bundles
      □ cadastro_en_US.properties
      □ cadastro_pt_BR.properties
    ▷ datasource
    ▷ servlets
  ▷ Libraries
```

Os arquivos **properties** criados deverão ter o seguinte conteúdo (exemplificativo apenas):

- **cadastro\_pt\_BR.properties:**



```
cadastro_pt_BR.properties
1 sistema.aba1=Cadastro
2 sistema.aba2=Outra página qualquer
3
```

- **cadastro\_en\_US.properties:**

```
cadastro_en_US.properties
1 sistema.aba1=Register
2 sistema.aba2=Any other page
```

Perceba que as chaves nos arquivos são idênticas e seus valores variam conforme a localidade que representam. Esses valores são, a seguir, colocados na JSP de forma a refletir sua localização.

5. Insira no topo da página **sistema.jsp** uma diretiva **@taglib** logo abaixo daquela já existente, de forma a declarar a biblioteca de internacionalização da JSTL, que será usada no corpo da página:

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



6. Na página **sistema.jsp**, um pouco mais abaixo, logo após a abertura da tag **<body>**, insira duas tags JSTL, uma para inserir um objeto **Locale** e outra com o fim de inserir um bundle para ser usado na página. Em seguida, o código será parecido com este:

```
<fmt:setLocale value="pt_BR" />
<fmt:setBundle basename="bundles.cadastro" var="Lang" />
```

7. Localize na página **sistema.jsp**, a partir da linha 31, o seguinte trecho de código:

```
<ul id="tabs" class="fl">
    <li><a href="sistema.jsp" class="active-tab">Cadastro</a></li>
    <li><a href="#">Outra página qualquer</a></li>
</ul>
```

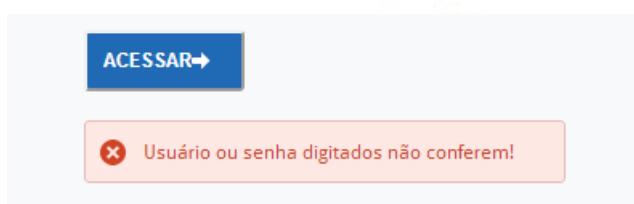
Esse trecho é o responsável pelas abas existentes no topo da página e pelo texto contido nelas. Na verdade, retirando toda a formatação CSS, trata-se apenas de uma lista não numerada e dois itens de lista inseridos. Perceba que em cada item existe um link com o texto apresentado na aba. Faremos a substituição do texto desses itens por valores existentes no bundle inserido na página no passo anterior.

7.1. Substitua o trecho apresentado pelo seguinte excerto, agora já utilizando as mensagens contidas no bundle:

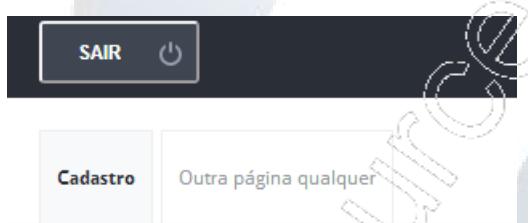
```
<ul id="tabs" class="fl">
    <li>
        <a href="sistema.jsp" class="active-tab">
            <fmt:message bundle="${lang}" key="sistema.aba1"/>
        </a>
    </li>
    <li>
        <a href="#">
            <fmt:message bundle="${lang}" key="sistema.aba2"/>
        </a>
    </li>
</ul>
```

Note, nas linhas indicadas pelas setas, que as strings, antes utilizadas nos links, agora foram substituídas por tags JSTL que utilizarão o bundle inserido na página e o locale definido para obter o valor das chaves em cada linha.

7.2. Execute o programa e, na página de login, insira as credenciais inicialmente **erradas**. Em seguida, teste se as mensagens de erro continuam aparecendo de forma satisfatória:



7.3. Agora insira as credenciais certas e verifique se o encaminhamento é feito corretamente. A seguir, note o texto nas abas da página. Eles devem permanecer como estavam antes:



7.4. Altere, na tag onde inserimos o bundle na página, o valor **pt\_BR** por **en\_US**, da seguinte forma:

- De:

```
<fmt:setLocale value="pt_BR" />
```

- Para:

```
<fmt:setLocale value="en_US" />
```

7.5. Salve o arquivo e dê um refresh na página. Verifique que as abas agora passaram para o idioma inglês. Isso ocorre porque, dada a mudança do objeto Locale da página, o bundle utilizado automaticamente passa a ser aquele criado para o idioma que coincide com o sufixo de seu nome. Nesse caso, **en\_US**. A página ao final deverá ficar desta forma:



8. A partir da linha 84, substitua o seguinte trecho no arquivo **sistema.jsp**, relativo ao código em Scriptlets, por código JSTL e Standard Actions adequadas:

- Código usando Scriptlets:

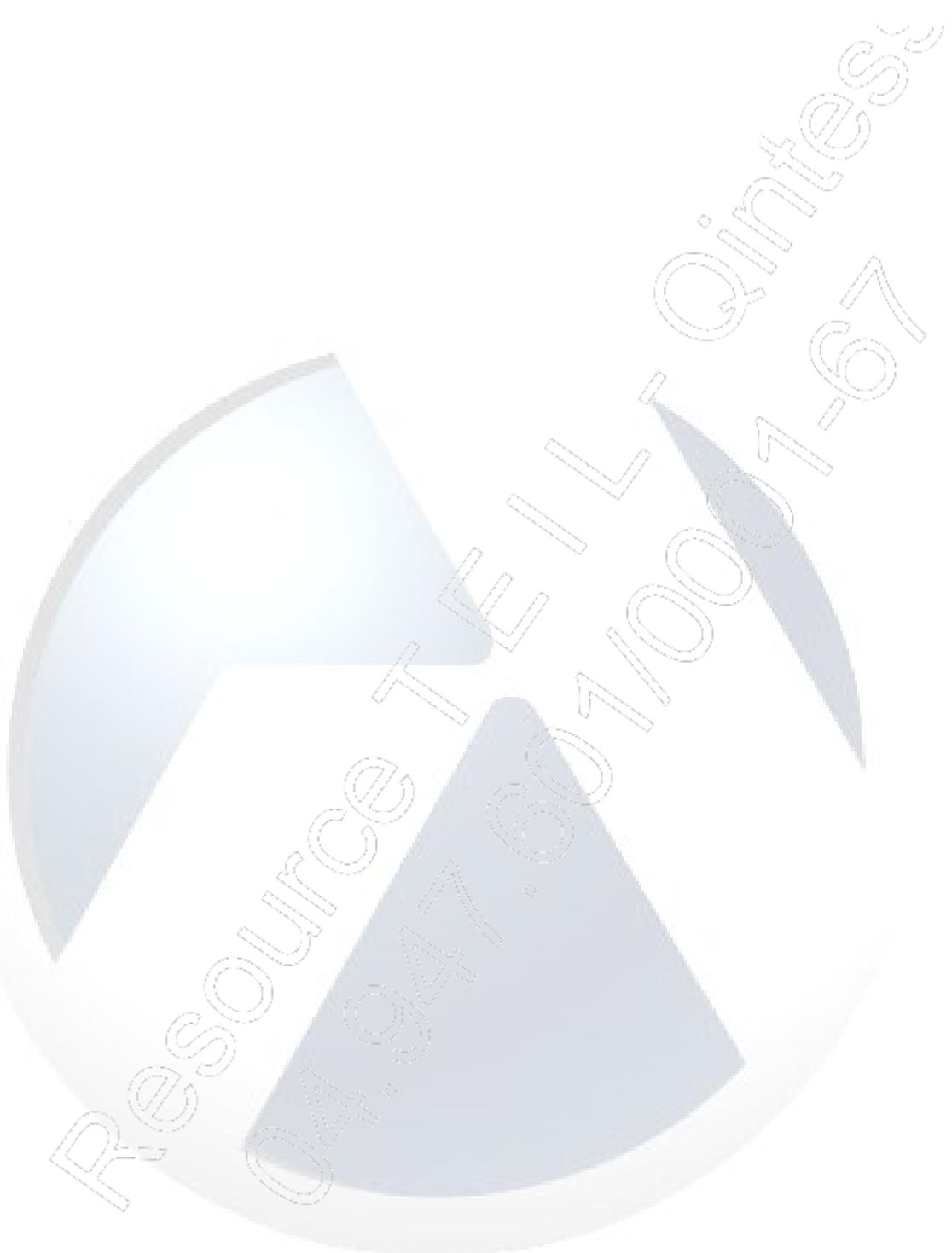
```
<%  
    Dados dados = new Dados();  
    session.setAttribute("alunos", dados.getDadosAlunos());  
    session.setAttribute("professores", dados.getDadosProfessores());  
%>
```

- Código após a substituição:

```
<jsp:useBean id="dados" class="datasource.Dados" scope="session" />  
<c:set var="alunos" value="${dados.getDadosAlunos()}" scope="session"/>  
<c:set var="professores" value="${dados.getDadosProfessores() }" scope="session" />
```

Note que a segunda versão utiliza duas tags JSTL para criar dois objetos de lista e inseri-los no escopo de sessão, porém, antes instanciamos o bean **dados** usando uma Standard Action.

9. Salve a página e faça um **Refresh** na aplicação para garantir que tudo está funcionando adequadamente. A partir de agora, é possível criar páginas distintas para idiomas diversos com pouquíssimo esforço e com facilidade para manutenção e escalabilidade futura.



# Capítulo 11:

## Conhecendo a arquitetura MVC



# Introdução à arquitetura MVC

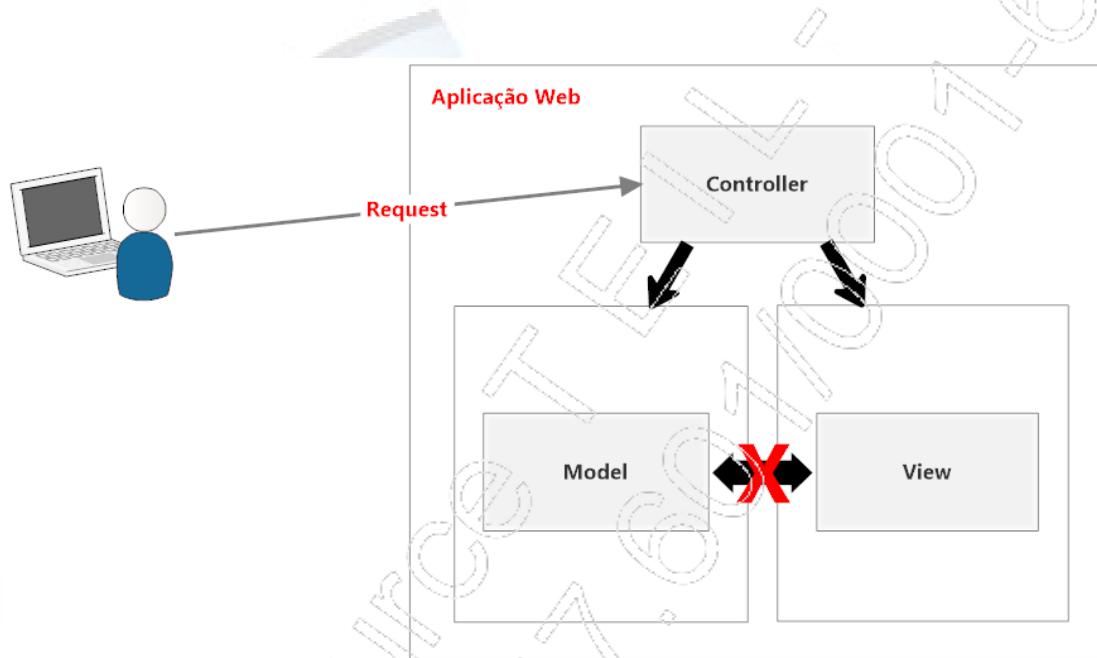
A arquitetura MVC (Model-View-Controller) é, entre outras definições possíveis, um padrão de design de software voltado a aplicar uma separação formal entre as “camadas” de um sistema ou software. Apesar de ser um padrão aplicado fundamentalmente no ambiente Web, também é utilizado em aplicações desktop que exijam a interação do usuário com uma interface gráfica.

Como o nome sugere, o padrão procura oferecer uma visualização da aplicação em três camadas fundamentais: Model, View e Controller, mais detalhadas a seguir:

- **Model:** Camada que representa o acesso a dados e lógica da aplicação. É onde residem todas as regras de negócio e toda a estrutura que representa dados no sistema, além das regras necessárias para acesso aos dados da aplicação, sejam eles provindos de um banco de dados, arquivos ou mesmo de WebServices. É totalmente controlada pela camada **Controller** e não possui permissão para acessar de qualquer forma a camada **View**;
- **View:** Camada que representa toda a interface gráfica da aplicação, as telas e componentes por onde é feita a interação com o usuário. Em uma aplicação Web, pode representar as tecnologias do lado cliente, como HTML, CSS e Javascript. Jamais deve conter lógica ou qualquer complexidade relacionada com as regras de negócio da aplicação e não deve, em hipótese alguma, possuir acesso à camada **Model**;
- **Controller:** Camada responsável por controlar e gerenciar a aplicação como um todo. É sua incumbência monitorar a camada **View** e capturar qualquer interação do usuário como eventos e dados inseridos em geral. Faz a comunicação com a camada **Model** para a obtenção ou gravação de dados e para acessar funcionalidades implementadas. Ao término, monta a tela necessária na camada **View** e retorna ao usuário.

Em uma aplicação Web, caso específico de nosso interesse, o padrão MVC é extensamente aplicado e difundido. Em Java, existem diversos frameworks que implementam a arquitetura MVC, dentre os quais um dos mais famosos e utilizados no mundo é o Struts 2.

Vejamos como esse padrão pode ser representado visualmente por meio do seguinte diagrama:



Quando um usuário faz uma requisição a uma aplicação Web por meio de seu navegador preferido, a requisição é “interceptada” e recebida por um componente da camada **Controller**. Esse membro da camada **Controller** acessa e utiliza componentes da camada **Model** para atender as funcionalidades requeridas pelo usuário, como buscar ou gravar dados, efetuar cálculos, envios, transmissões, etc. Em seguida, monta a página com conteúdo visual e a retorna ao usuário. A partir daí o processo pode se repetir quantas vezes forem necessárias.

Talvez a principal vantagem oferecida pela implementação e uso do padrão MVC seja a separação de responsabilidades em um sistema. Em geral, grandes sistemas tendem a produzir centenas e até milhares de componentes. A separação desses componentes por áreas ou camadas de aplicação com responsabilidades independentes é uma prática que facilita bastante a manutenção e o potencial crescimento do sistema em releases futuros.

## Quais as vantagens do uso de um framework?

Inicialmente, vale descrever o conceito e a ideia de um framework. O conceito de framework é, em muitas situações, confundido com o conceito de padrão de design e vice-versa, no entanto, eles representam ideias bastante distintas. Um padrão de design, ou padrão de projetos, é um modelo que idealiza uma porção de um software, geralmente por representar um problema recorrente entre os desenvolvedores em geral, e procura oferecer uma solução teórica e genérica para a melhor saída ao problema enfrentado. Essa solução é dita teórica, porque é oferecida sem implementação física, e genérica, porque procura atender todas as linguagens e plataformas de software que utilizam o paradigma em questão, geralmente a Orientação a Objetos.

Um framework é, na realidade, uma implementação física de diversos padrões que compõem uma arquitetura, voltada a uma plataforma específica. Essa implementação procura ser parte estrutural do software ou do sistema em que é usada e oferece facilidades e funcionalidades inovadoras aos desenvolvedores que a utilizam. É um grande conjunto de componentes físicos da plataforma em questão que visa oferecer uma alternativa mais rápida, segura e automatizada para determinada arquitetura.

Ainda assim, por que deveríamos usar um framework? Na verdade, nós não precisamos usar um framework. As tarefas realizadas por meio de seus componentes podem ser feitas com a mesma eficácia usando apenas os componentes nativos da plataforma. No entanto, se a eficiência for o fator mais importante, então o uso de um framework oferece uma boa vantagem.

Os benefícios do uso de um framework não param somente na eficiência oferecida. Em aplicações modernas, como as construídas nos dias atuais, com grande volume de componentes e de grande extensão, a necessidade de organização, estrutura, padronização, segurança e fragmentação torna-se fundamental. Paralelamente a esses fatores, pode-se imaginar uma equipe de desenvolvedores razoavelmente experientes e um bom investimento para manter essa equipe em pleno funcionamento.

É nesse cenário que um framework entra em ação, provendo automatização para grande parte das tarefas que demandariam atenção e trabalho extra dos desenvolvedores. Cuidando para que a padronização e a segurança sejam mantidas, uma vez que a estrutura crucial de comunicação do sistema é feita pelo framework e, como sua concepção se dá com estudo e análises, tudo é feito de forma a oferecer a melhor performance possível.

# O Struts 2 como framework MVC

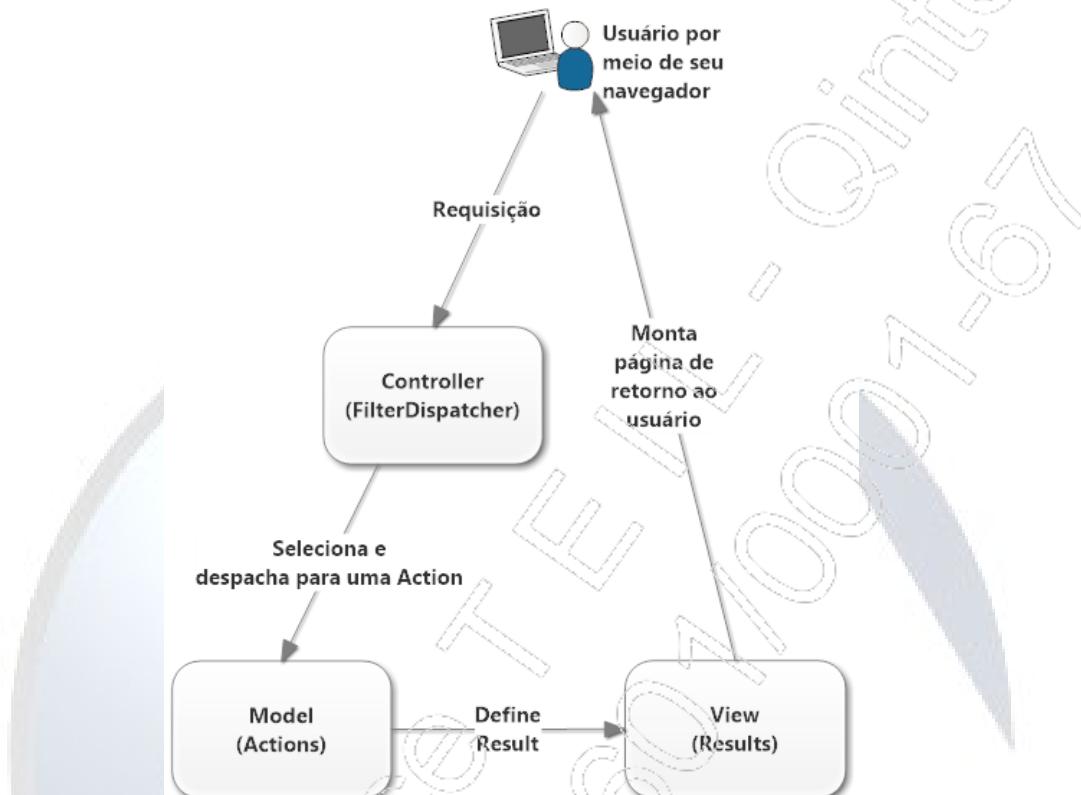
O framework Apache Struts 2 é um framework moderno e que incorpora as melhores práticas existentes no âmbito da construção de aplicações Web Java. Ele traz os padrões mais reconhecidos e aceitos pela comunidade Java ao longo dos anos, é grátis e open-source. Todos esses fatores o tornam um dos mais usados e preferidos em aplicações Java para Web em todo o mundo.

A versão 2 do Struts, ao contrário do que se poderia pensar, não é apenas uma evolução da versão 1. É um framework completamente modificado e remodelado que guarda pouca relação com seu antecessor. Foi lançado em 2007 pela Apache Foundation após a junção de sua equipe com a equipe da WebWorks.

O Struts 2 implementa todas as funcionalidades da arquitetura MVC e, neste ponto, é importante delinearmos suas características:

- Possui um conjunto de tags especializadas;
- Oferece suporte a AJAX;
- Possui componentes para realizar validação;
- É altamente testável;
- Utiliza-se de POJOs, sendo minimamente intrusivo;
- Oferece uma excelente arquitetura de plug-ins e, inclusive, já vem com alguns de grande aplicação;
- Possibilita ampla padronização e o uso de convenções como forma de se eliminar a necessidade de configurações.

Vamos analisar a arquitetura básica do framework e compreender como ele procura automatizar a arquitetura MVC por meio de componentes oferecidos. O seguinte diagrama procura demonstrar esse relacionamento:



Nesse modelo simplificado, temos os três componentes básicos da arquitetura do Struts 2, que representam, respectivamente, as três camadas do modelo MVC.

Na camada **Controller**, temos um componente de grande importância, chamado **FilterDispatcher**, que é quem recebe todas as requisições HTTP provindas de usuários e clientes em geral. Esse componente tem a incumbência de analisar a solicitação recebida (por meio do padrão da URL) e encaminhá-la ao componente **Model** adequado (chamados de **Action** no Struts 2). Nesse modelo, por si só, já é possível identificarmos um padrão existente e bem comum em frameworks voltados à Web: o Front Controller. Esse padrão prevê a existência de um único componente que receba todas as requisições e mapeie cada uma delas para o recurso adequado na aplicação. A camada em questão tem mais detalhes a serem vistos, porém, em uma visão macro do framework, esse componente a descreve bem.

Na camada **Model**, o framework oferece componentes denominados de Actions. Uma Action é um componente que recepciona, em sua camada, todas as solicitações para uma determinada URL ou funcionalidade dessa maneira mapeada. Ele pode conter chamadas a outras várias classes auxiliares, lógica de negócios, fazer uso de código de acesso a dados e, fundamentalmente, definir o resultado a ser apresentado ao usuário. Esse resultado será mapeado para uma Result (camada **View**), que poderá ser uma JSP, por exemplo. Essa comunicação não é feita diretamente entre **Model** e **View**. Nesse caso, a Action sinaliza ao Controller qual a Result a ser encaminhada e coloca os dados necessários para sua montagem em uma área de transferência. A Result, então, obterá esses dados dessa área de transferência, será montada de forma adequada e será encaminhada como retorno ao cliente.

Na camada **View**, será montada a página de retorno ao usuário. Essa montagem pode utilizar diversos mecanismos para a geração de HTML válido, sendo a JSP o mais comum deles. Existem outros, como o Velocity e o FreeMarker, porém, nosso foco permanece na JSP por ser a mais utilizada em geral e foco do nosso treinamento. A View recebe a incumbência após o término e mapeamento da Action. Ela obtém seus dados da mencionada área de transferência e monta-se visualmente de forma a ser retornada ao cliente. No framework Struts 2, os objetos que representam essa camada são chamados de Result.

# **Conhecendo a arquitetura MVC**

## **Teste seus conhecimentos**



**1. Qual a principal vantagem alcançada quando se implementa o padrão MVC em uma aplicação Web?**

- a) Redução do número de linhas de código da aplicação.
- b) Diminuição da necessidade de ter aplicações de grande porte em funcionamento.
- c) Separação das responsabilidades dos diversos componentes da aplicação em camadas.
- d) Facilitação na mudança da plataforma tecnológica de uma aplicação, pois reduz e até anula os grandes impactos existentes.
- e) Diminuição de gastos com o projeto, pois, a partir deste ponto, pode-se abrir mão de um profissional da área de arquitetura de software.

**2. Qual das alternativas a seguir está correta sobre a interação entre as camadas previstas na arquitetura MVC?**

- a) A camada Controller é a camada responsável por guardar toda a lógica da aplicação e seu devido acesso a dados.
- b) A camada Model define quando e como popular a interface gráfica prevista na camada View e, portanto, tem liberdade para manipular os componentes de interface gráfica.
- c) A camada Controller é a responsável por gerenciar toda a interface gráfica definida para comunicação com o usuário.
- d) A camada View é a responsável por montar e gerenciar as interfaces com o usuário na aplicação usando dados produzidos pela camada Model, conforme disponibilização pela camada Controller.
- e) Nenhuma das alternativas anteriores está correta.

**3. Qual das alternativas a seguir não está correta sobre o framework Apache Struts 2 e suas características?**

- a) Oferece suporte a AJAX.
- b) Está focado somente na construção de aplicações de grande porte.
- c) Oferece suporte a testes.
- d) É considerado não intrusivo por usar POJOs como componentes.
- e) É baseado e implementado sob a arquitetura MVC.

**4. Qual das alternativas a seguir está correta a respeito da camada View, sob a implementação do Struts 2?**

- a) Exige a utilização de JSP como mecanismo de construção e renderização de páginas.
- b) É representada especialmente por componentes do tipo Action.
- c) O processo de montagem da página de retorno ao cliente é feita após o recebimento da requisição do cliente, que ocorre diretamente nos componentes de resultado.
- d) Tanto os dados produzidos quanto a criação das páginas de retorno ocorrem na camada Model, sendo responsabilidade da camada View o encaminhamento das páginas ao usuário solicitante.
- e) É representada, no framework Struts 2, por objetos chamados de Result.

**5. Qual das alternativas a seguir está correta sobre a interação entre os componentes que representam as camadas no framework Struts 2?**

- a) O componente FilterDispatcher é quem recebe a requisição do usuário e a encaminha à Action correspondente, conforme mapeamento definido.
- b) As Actions, definidas no Struts 2, representam parte integrante da camada Controller, pois são elas que definem o encaminhamento da requisição.
- c) É possível afirmar que o padrão Front Controller, conhecido no mundo Web, é incompatível com a arquitetura MVC e, portanto, com o Struts 2.
- d) Um objeto Result representa uma página JSP já configurada com dados para exibição ao usuário.
- e) Nenhuma das alternativas anteriores está correta.

# **Capítulo 12:**

## **Primeira aplicação**

### **com Struts 2**



# Introdução

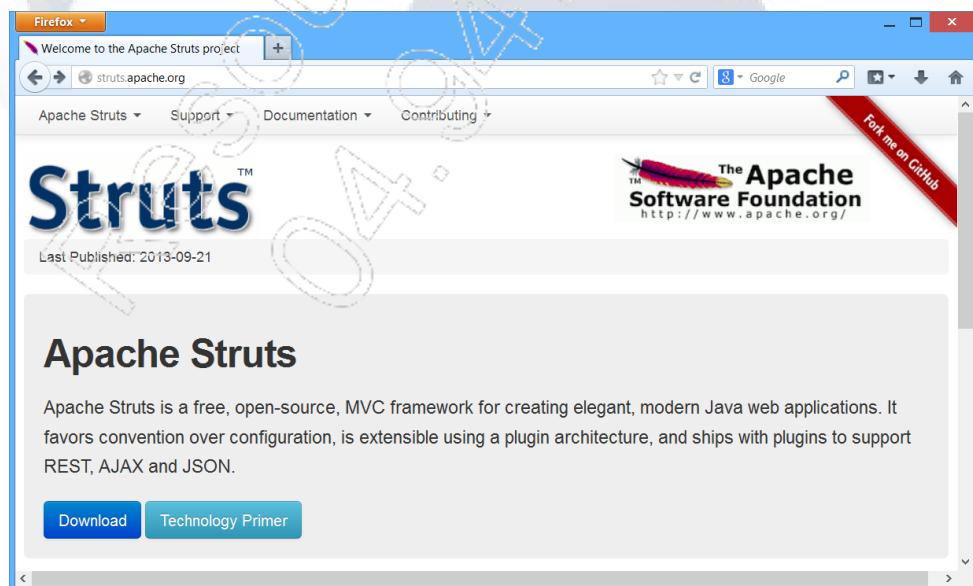
Este capítulo será inteiramente prático e com explicações ponto a ponto de cada um dos passos utilizados para a configuração do Apache Struts 2, além de propor uma primeira aplicação simples, porém funcional, utilizando sua infraestrutura. Seguindo esta proposta hands on, você deve executar cada passo, tirando possíveis dúvidas com o instrutor.

## Instalação e configuração inicial

O framework Apache Struts 2 compõe-se, como muitos outros frameworks Java, de um conjunto de arquivos **.jar**, que representam bibliotecas Java no “pacote” em que se compõe o framework.

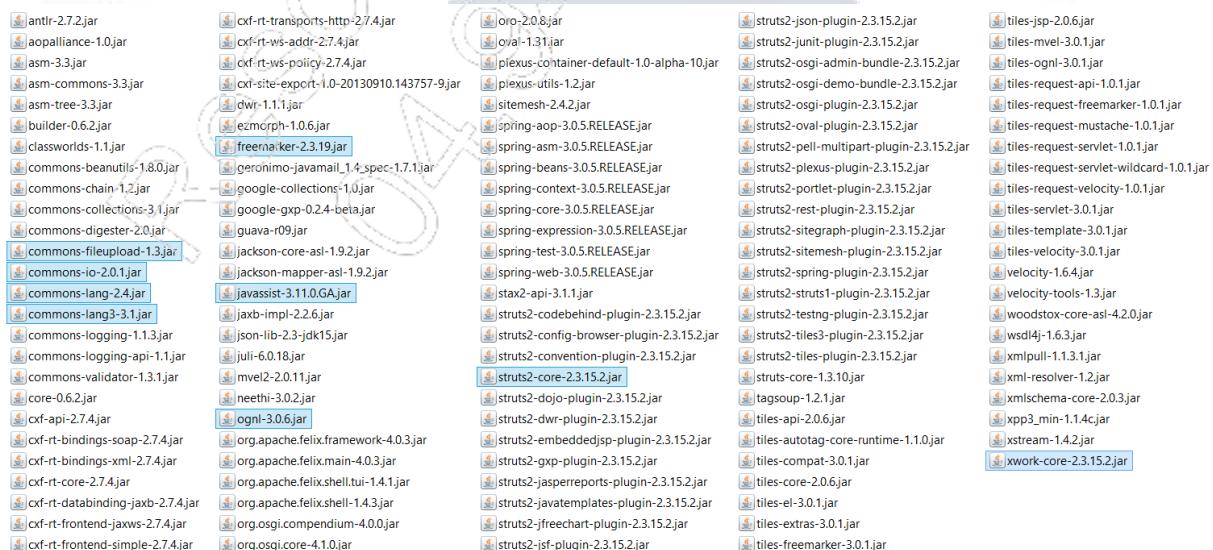
Para iniciar a configuração do ambiente para desenvolvimento Web com Apache Struts 2, precisamos realizar o download do framework em si. Para isso, siga estes passos:

1. O download pode ser realizado em sua página oficial, pelo endereço <<http://struts.apache.org>>. Em seguida, clique no link para download:



Entre todas as versões, prefira aquela completa, denominada **Full Distribution**. Usaremos apenas uma parte de todo o potencial que essa versão tem a oferecer. Nesse pacote, encontram-se os arquivos binários, que vamos utilizar, toda a documentação javadoc, código-fonte, além de aplicações de exemplo para consulta.

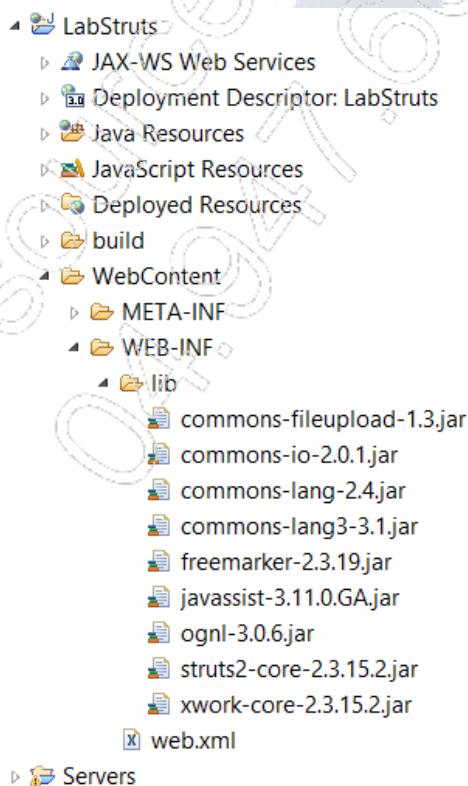
2. Depois de ter feito o download, basta descompactar a pasta existente e colocá-la em algum local em seu computador. Este procedimento já estará feito na sua máquina durante o curso;
3. Abra a **IDE Eclipse**, caso já não esteja aberta, e crie um novo projeto do tipo **Dynamic Web Project**, conforme feito em alguns dos laboratórios desta apostila. Ao projeto, dê o nome de **LabStruts** e lembre-se de marcar a opção **Generate web.xml deployment descriptor**;
4. Assim que o projeto estiver criado, abra a pasta descompactada anteriormente, resultante do download do Struts 2, e, dentro dela, abra a pasta **lib**. Nesta pasta, existe um grande número de arquivos **.jar**, dos quais usaremos apenas alguns. Na lista a seguir, estão selecionados os arquivos que serão utilizados para configurar o Struts 2 em sua forma mais básica em nossa aplicação:



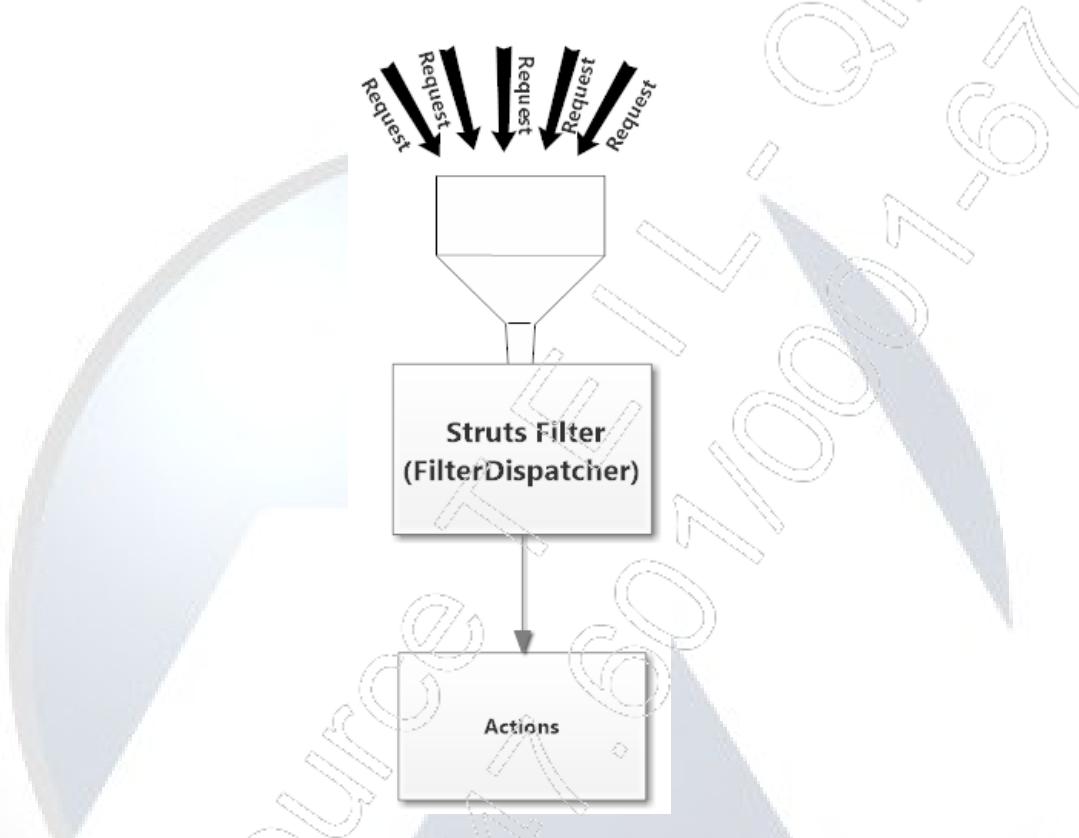
5. Copie os arquivos selecionados para a pasta **/WebContent/WEB-INF/lib** do projeto criado. Confira, a seguir, a lista dos arquivos **.jar** necessários, destacados na imagem anterior:

- commons-fileupload-1.x.jar;
- commons-io-2.x.x.jar;
- commons-lang-2.x.jar;
- commons-lang3-3.x.jar;
- freemarker-2.x.x.jar;
- javassist-3.x.x.GA.jar;
- ognl-3.x.x.jar;
- struts2-core-2.3.x.x.jar;
- xwork-core-2.3.x.x.jar.

Após a cópia dos arquivos **.jar**, a estrutura do projeto no Eclipse deverá ser semelhante à seguinte:



6. Agora que o projeto está estruturalmente montado, faremos a configuração inicial para preparar o uso do Struts 2. A primeira coisa a ser feita é declarar o filtro (**servlet filter**) do Struts 2 dentro do arquivo **web.xml**. Esse filtro é usado para interceptar todas as requisições recebidas na aplicação e direcioná-las ao framework. Essa ideia pode ser representada pelo seguinte diagrama:



7. Nesse diagrama, o filtro que direciona as requisições para o Struts pode ser entendido como sendo o **Deployment Descriptor**. Vejamos então, a sintaxe desse arquivo:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>LabStruts</display-name>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
  
```

8. Note que a declaração do filtro é uma sintaxe já conhecida. O nome completamente qualificado da classe que representa o filtro pode parecer longo, mas não se preocupe em decorá-lo, basta consultar a documentação do Struts 2 ou mesmo manter uma cópia desse arquivo para uso futuro. Esse caminho extenso está representado nas bibliotecas que importamos antes. Cabe ressaltar que o **url-pattern** utilizado no mapeamento do DD abrange todas as requisições recebidas. Ele se aplica a qualquer caminho colocado após o **Context root** da aplicação, de forma que todas as solicitações serão transferidas para o framework;

9. Agora criaremos o arquivo interno de configuração do Struts 2. Esse arquivo será utilizado para a configuração dos componentes e mapeamentos dentro do framework. Deve estar localizado na raiz da pasta **src**, no Eclipse, localizada dentro de **Java Resources**. Após empacotado e pronto para deploy, esse arquivo ficará na pasta em que residem as classes compiladas. Esse arquivo deve se chamar **struts.xml**. Vejamos a estrutura básica de um arquivo como esse:

```
<!DOCTYPE struts PUBLIC  
        "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  
        "http://struts.apache.org/dtds/struts-2_0.dtd">  
  
<struts>  
    ←—————| Toda a configuração do  
    | struts 2 vai aqui dentro!  
    </struts>
```

10. De maneira nenhuma é necessário memorizar as declarações de DTD desse arquivo. Basta obtê-las do próprio site, que contém a documentação do Struts 2 por meio do endereço

<http://struts.apache.org/release/2.3.x/docs/strutsxml-examples.html>, ou mesmo acessando a documentação do Struts 2, seguindo para **guides**, após **Configuration Files** e, por fim, escolhendo **struts.xml**:

11. Vamos configurar esse arquivo de forma a mapear um determinado padrão de URL para uma Action e, a seguir, conforme seja o retorno dessa Action, o mapeamento determinará qual Result será retornada ao usuário. Para tanto, iniciaremos configurando esse arquivo com o seguinte conteúdo, explicado detalhadamente a seguir:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="main" extends="struts-default">
        <action name="cadastro" class="br.com.impacta.actions.CadastroAction">
            <result name="success">/cadastro.jsp</result>
        </action>
    </package>
</struts>
```

A tag **package** representa a nossa aplicação em geral. Comumente não se usa mais de uma e, por esse motivo, ela recebe o nome de **main**. É uma forma de agrupar Actions, Results e outros componentes. O atributo **extends** significa que usaremos a configuração padrão do Struts 2 e essa é uma boa ideia para se obter uma aplicação funcionando rapidamente sem maiores complicações.

No interior da tag principal, declaramos uma Action, representada por uma classe que ainda criaremos dentro do pacote **br.com.impacta.actions**. Essa Action está mapeada para qualquer requisição com o padrão de URL **cadastro**. Ou seja, qualquer requisição que chame o endereço <http://localhost:8080/<Context root>/cadastro> será mapeada para essa classe.

No interior da tag **action**, declaramos uma ou mais tags **result**, que representam objetos **Result**, encaminhamentos para cada página que receberá a incumbência de ser retornada ao usuário, se **success** for o retorno da Action – caso específico do nosso exemplo.

## Criação dos componentes Struts 2 fundamentais da aplicação

Quando nos referimos aos componentes da aplicação, nessa primeira aplicação, referimo-nos à Action já mapeada e a seu respectivo Result, no caso, uma página JSP criada para ser retornada ao usuário.

1. Crie agora uma nova classe Java cujo nome deve ser **CadastroAction** e que deve estar no pacote **br.com.impacta.actions**. Essa classe deve implementar a interface **Action**, localizada no pacote **com.opensymphony.xwork2.Action**, presente nas bibliotecas importadas na aplicação logo no início desse Capítulo. Caso encontre qualquer problema para localizar a interface, revise as bibliotecas importadas e tente novamente. Essa interface exige a implementação de apenas um método, chamado **execute()**. Criaremos nessa classe, além desse método, um campo privado do tipo String denominado **saudacao**. Em conjunto, criaremos os métodos getter e setter para esse campo, padrão que deve ser seguido em todas as Actions criadas. Vejamos o código final dessa Action:

```
package br.com.impacta.actions;

import com.opensymphony.xwork2.Action;

public class CadastroAction implements Action {

    private String saudacao;

    @Override
    public String execute() throws Exception {
        setSaudacao("Bem Vindo à sua primeira aplicação feita com Struts 2!!!");
        return "success";
    }

    public String getSaudacao() {
        return saudacao;
    }

    public void setSaudacao(String saudacao) {
        this.saudacao = saudacao;
    }
}
```

2. Analisando o código da implementação do método **execute()**, perceba que ele altera o valor do campo **saudacao** por meio de seu setter e retorna uma string literal cujo conteúdo é **success**. Esse campo será capturado e exibido pela página **cadastro.jsp**;

3. Adiante, será criada a página JSP que representará o Result (View) de nossa pequena aplicação. Para tanto, clique na pasta **WebContent** com o botão direito e escolha **New / JSP File**. Nomeie a página como **cadastro.jsp**. Insira o seguinte conteúdo, delineado e detalhado a seguir:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!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=ISO-8859-1">
    <title>Meu cadastro com Struts 2</title>
</head>
<body>
    <h2>
        <s:property value="saudacao" />
    </h2>
</body>
</html>
```

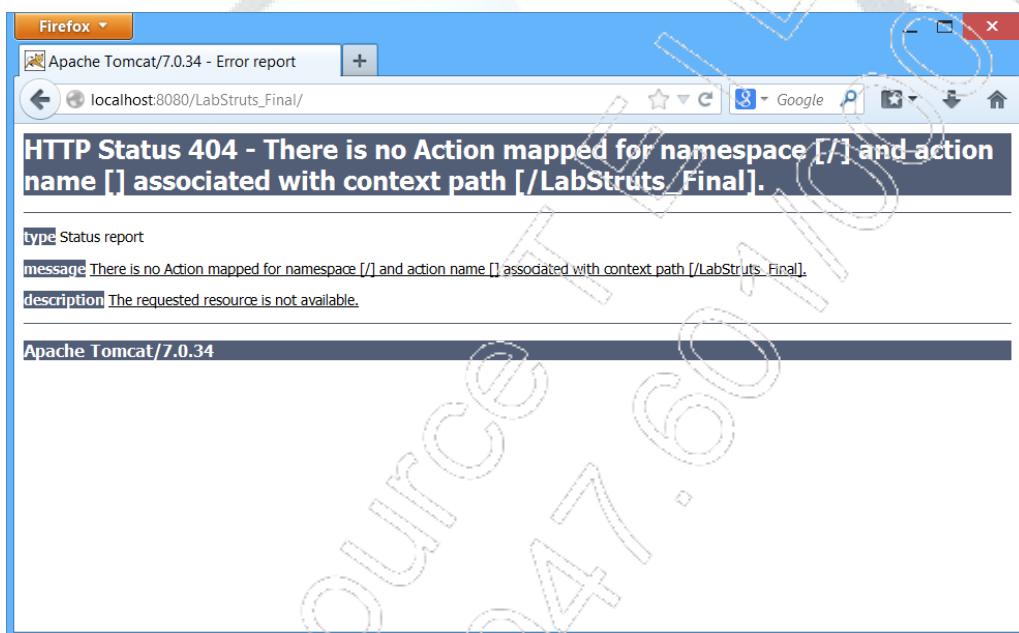
A página anterior é uma JSP simples e comum como as tantas que já foram vistas e estudadas ao longo desta apostila, com dois detalhes apenas, que merecem destaque. O primeiro deles é a diretiva **@taglib**, que insere a biblioteca de tags do struts 2. A uri usada é padrão e comumente o prefixo que se usa é o próprio **s**. Existem diversas tags do Struts 2 com funcionalidades diversas e todas elas devem ser estudadas com maior atenção, pois são fundamentais para a implementação de diversas funcionalidades do framework na camada View.

O segundo detalhe é a própria tag **<s:property value="saudacao">**, que insere, na página onde é colocada, o conteúdo de um atributo denominado **saudacao** localizado na Action relativa a essa Result.

4. Precisamos do servidor Apache Tomcat 7 funcionando para poder executar a aplicação criada. Caso ainda não tenha sido feita a configuração desse servidor, recorra ao Apêndice 1 para realizá-la passo a passo.

## Executando a aplicação e analisando os resultados

Ao executar a aplicação no Eclipse, obtemos a seguinte saída no navegador, para uma requisição na raiz de nossa aplicação:

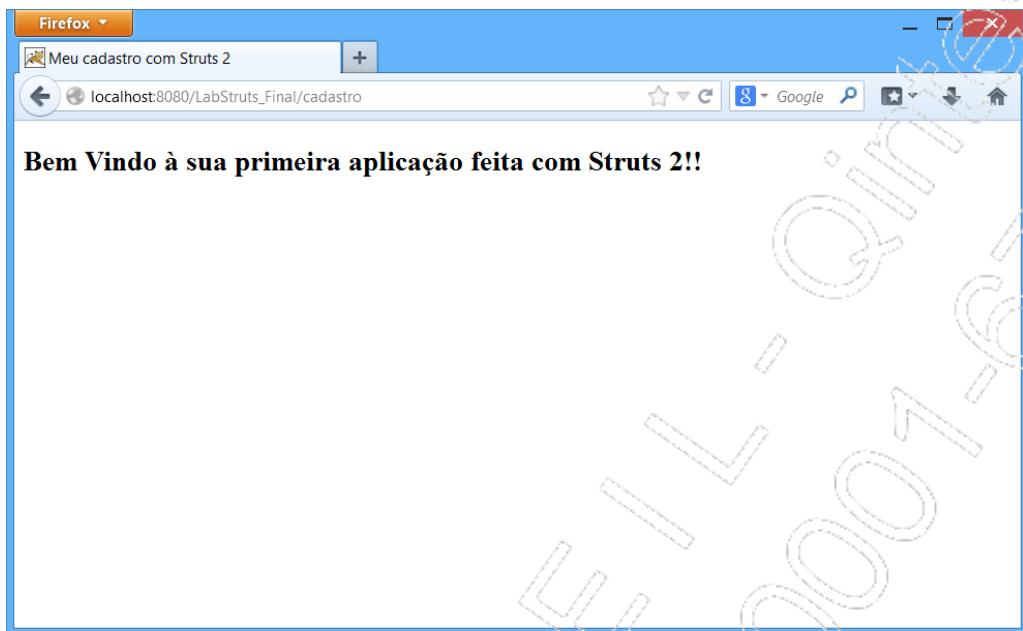


Note o erro apresentado e veja que ele ocorre porque não há nenhuma **Action** mapeada para atender requisições feitas na raiz de nossa aplicação. A única Action existente está mapeada para o caminho **/cadastro**.

Façamos agora uma requisição nesse caminho e vejamos a saída. A URL completa a ser inserida agora é

[http://localhost:8080/LabStruts\\_Final/cadastro](http://localhost:8080/LabStruts_Final/cadastro).

A saída, conforme esperado, é a seguinte:



## Para onde seguir a partir deste ponto?

Para o domínio completo de um framework poderoso e sofisticado como o Struts 2, seria necessário focar somente o aprendizado dessa ferramenta. Contudo, você já deu os passos iniciais para a configuração e utilização do Struts 2 e já é capaz de compreender a arquitetura do framework e de montar uma pequena aplicação de exemplo.

É importante destacar que toda a tecnologia Java apresentada e oferecida por meio deste curso compõe matéria-prima para todos os frameworks oferecidos e conhecidos no ambiente Web Java. Nada impede que você procure, além de conhecer e trabalhar com diversos frameworks, criar o seu próprio. Diversas iniciativas surgem de ideias de um ou alguns desenvolvedores que, depois de algum tempo, atraem diversos seguidores e colaboradores na comunidade Java.

Como indicação final, para aprofundamento e profissionalização em todo o framework Struts 2, segue como ponto principal e maior referência a página de documentação do Struts 2 na Web, que possui tutoriais e guias muito úteis ao aperfeiçoamento no uso da ferramenta:

<<http://struts.apache.org/release/2.3.x/docs/home.html>>.



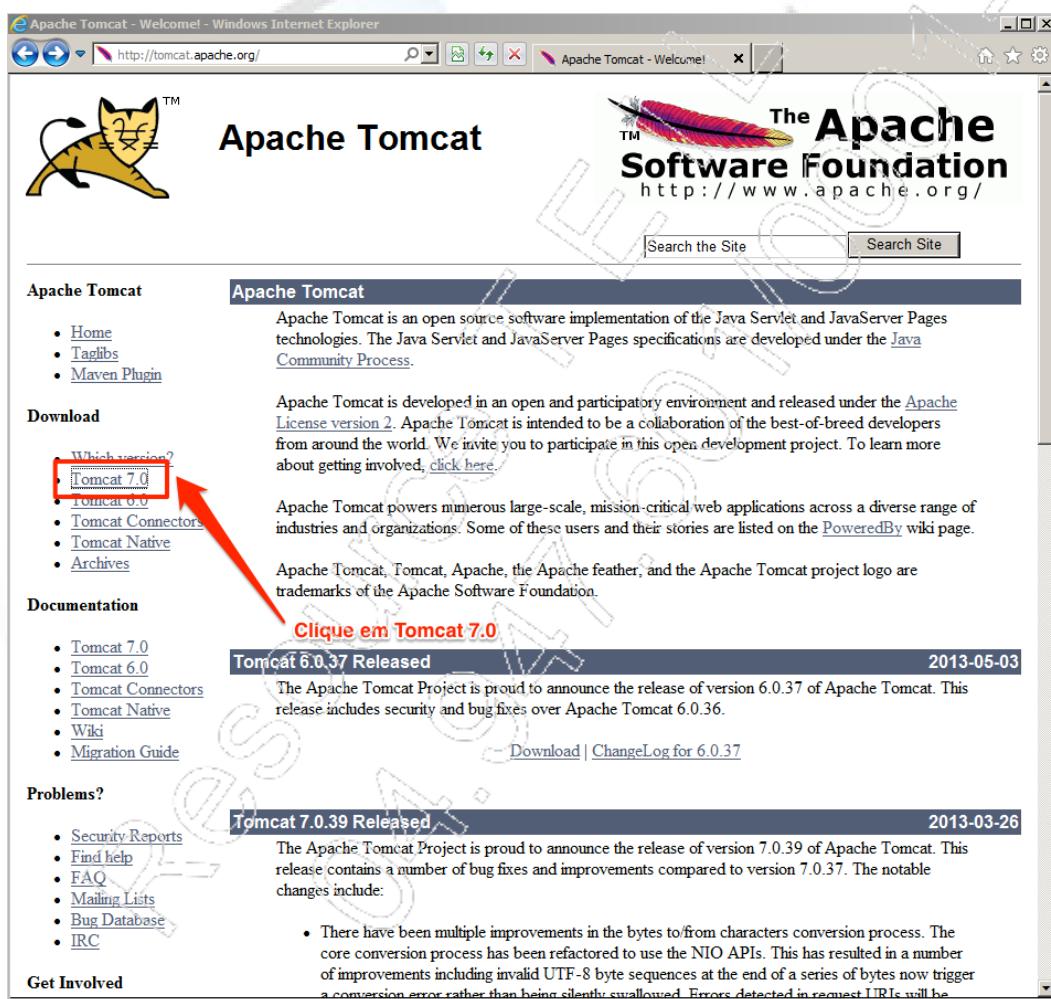
# Apêndice 1: Instalação e configuração do Apache Tomcat 7



# Download do Apache Tomcat

Inicialmente, para a instalação do servidor, acessaremos a página da Apache Foundation - projeto Tomcat, no seguinte endereço:  
[<http://tomcat.apache.org>](http://tomcat.apache.org).

Uma vez nessa página, clique no link **Tomcat 7.0** à esquerda, dentro do menu **Download**. Veja a página e o link a seguir:



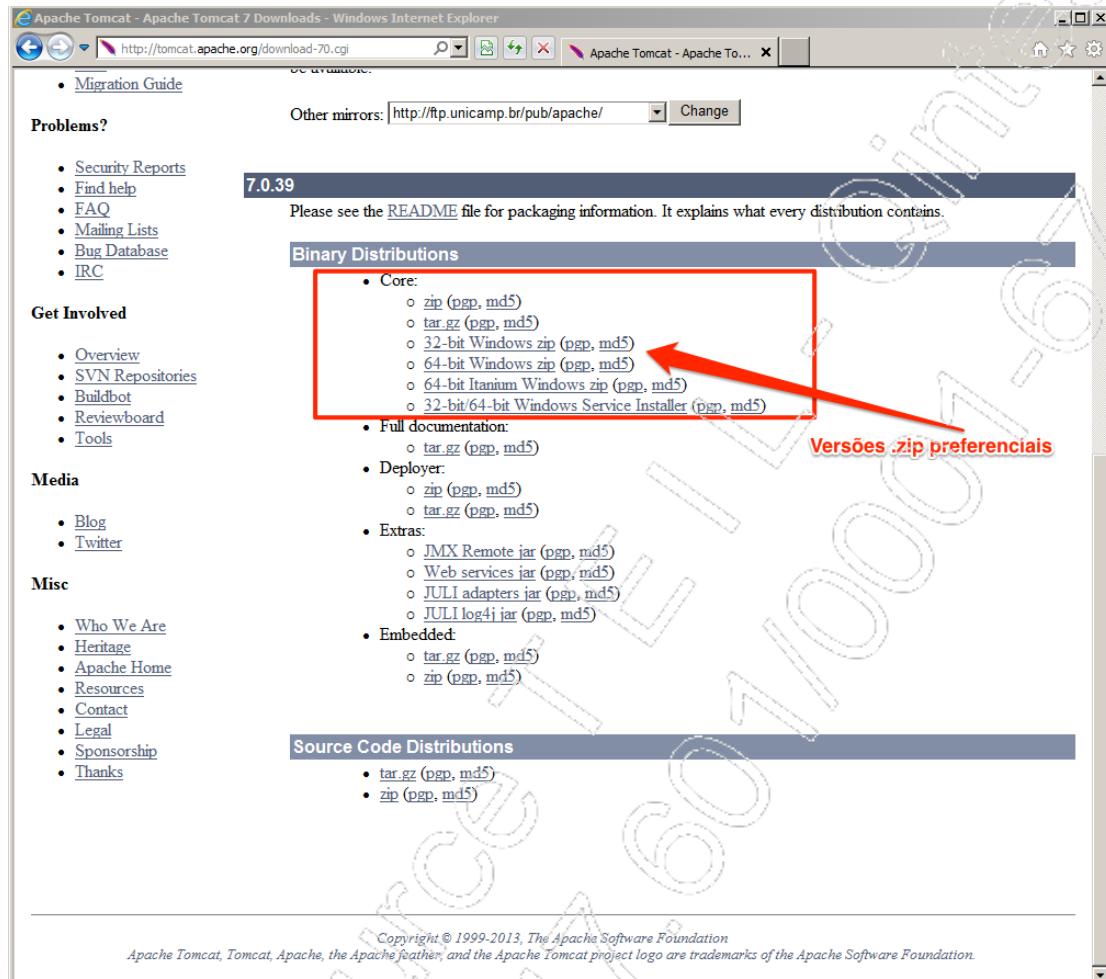
Já na página de downloads, mova o botão de rolagem da página para baixo até encontrar a versão adequada ao seu sistema operacional.

Sugerimos que você utilize a versão 7.0.39 ou mais atual. Caso não encontre essa versão na página principal, é possível acessá-la por meio do endereço <<http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.39/bin/apache-tomcat-7.0.39-windows-i64.zip>> (Acesso em 23 out. 2013).

Faça o download para sua máquina. No caso de download para o sistema operacional Windows, existem várias opções:

- Arquivo ZIP (Genérico);
- Arquivo ZIP para Windows 32 bits;
- Arquivo ZIP para Windows 64 bits;
- Instalador de Serviço Windows (executável).

Veja a tela com as opções citadas:

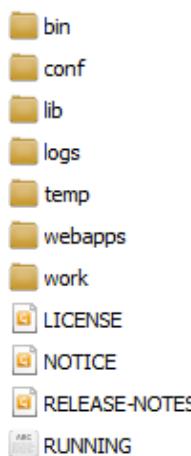


Dê preferência à versão ZIP referente ao seu sistema operacional. Utilize o instalador de serviço para o Apache Tomcat somente se a máquina em que o servidor estiver sendo instalado for utilizada para um ambiente de produção, onde as aplicações realmente serão acessíveis à Web.

Descompacte o arquivo zip e coloque a pasta descompactada em um local de fácil acesso, preferencialmente na raiz dos arquivos da máquina (C:\).

# A estrutura de arquivos do Apache Tomcat

Após ter descompactado a pasta do Apache Tomcat, a estrutura de arquivos deverá se parecer com a seguinte:



É importante conhecer a função de algumas pastas:

- **bin**: Contém todos os arquivos binários e executáveis do servidor. É através desta pasta que é possível iniciar e parar o servidor;
- **conf**: Contém arquivos de configuração do servidor. Nos passos seguintes, vamos alterar o conteúdo de um deles para inserir um usuário com privilégios suficientes para operar o aplicativo Manager do Tomcat;
- **lib**: Contém todos os arquivos **.jar** utilizados pelo servidor para seu funcionamento. Podem ser acrescentados outros jars dentro dessa pasta de forma que sejam compartilhados por todas as aplicações a serem executadas no servidor;
- **logs**: Pasta que conterá os logs do servidor. Apesar de utilizarmos a IDE Eclipse para inspecionar os logs, sempre que for preciso consultá-los individualmente, é nessa pasta que eles estarão;

- **webapps**: Pasta mais importante, pois conterá todas as nossas aplicações depois de terem sido implantadas no servidor (após o deploy).

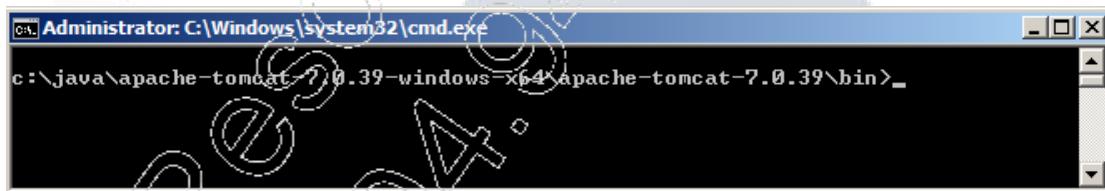
## Execução do Apache Tomcat via prompt de comando

Para executar o servidor, é preciso que as variáveis de ambiente **JAVA\_HOME** e / ou **JRE\_HOME** estejam configuradas. Se não estiverem, crie ao menos a variável **JAVA\_HOME** e preencha-a com o endereço físico da pasta onde está instalado o JDK em sua máquina. Esse caminho geralmente tem um padrão no sistema operacional Windows:

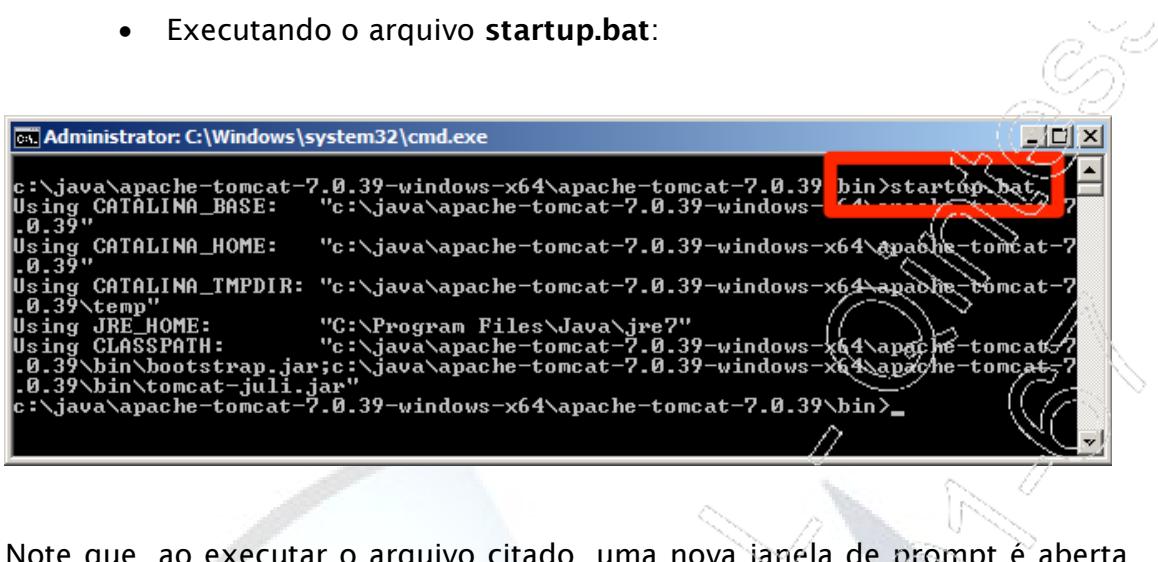
C:\Program Files\Java\jdk1.7.0\_XX

Feita a configuração inicial com a variável de ambiente, para executar o servidor, basta navegar via prompt de comando até o diretório **/bin** do Apache Tomcat e executar o arquivo **startup.bat**, conforme indicado a seguir:

- Navegando até a pasta adequada:

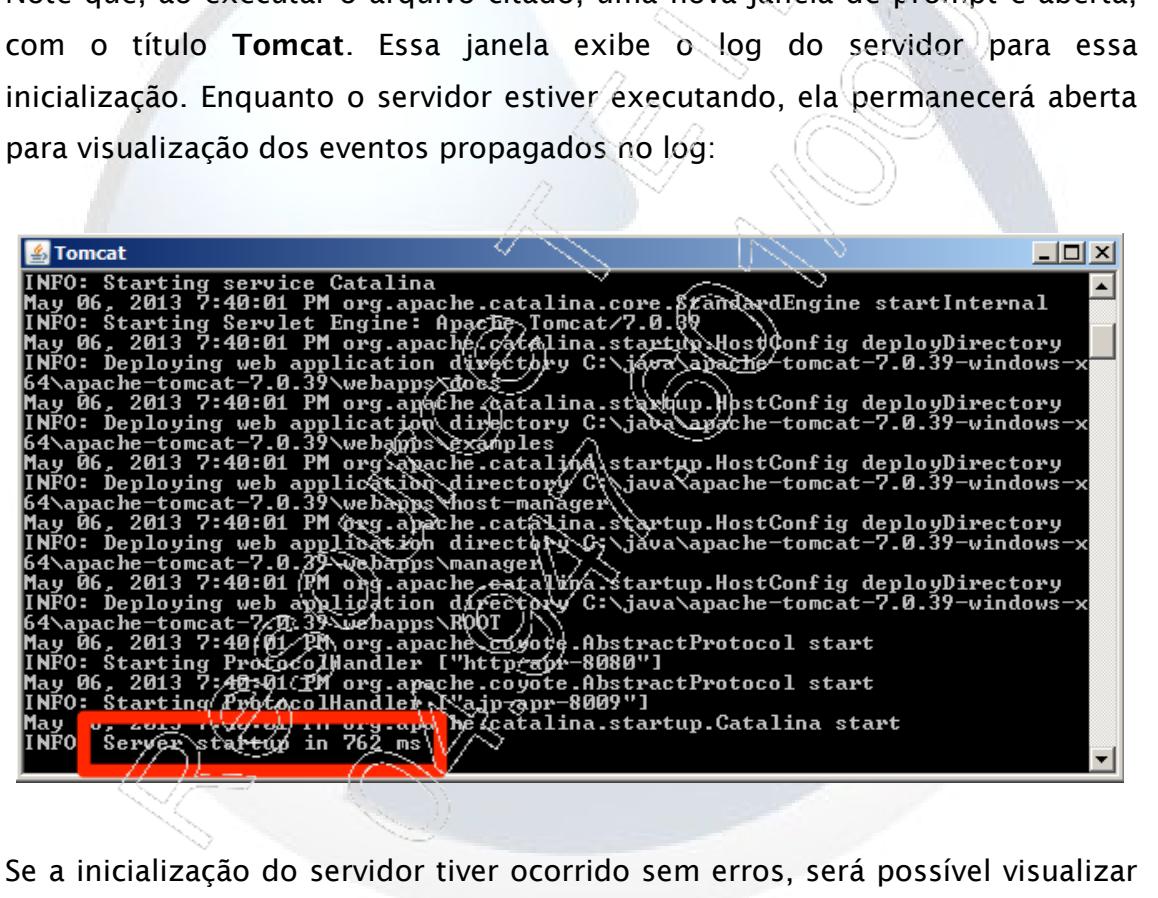


- Executando o arquivo **startup.bat**:



```
c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\bin>startup.bat  
Using CATALINA_BASE: "c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39"  
Using CATALINA_HOME: "c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39"  
Using CATALINA_TMPDIR: "c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\temp"  
Using JRE_HOME: "C:\Program Files\Java\jre7"  
Using CLASSPATH: "c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\bin\bootstrap.jar;c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\bin\tomcat-juli.jar"  
c:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\bin>
```

Note que, ao executar o arquivo citado, uma nova janela de prompt é aberta, com o título **Tomcat**. Essa janela exibe o log do servidor para essa inicialização. Enquanto o servidor estiver executando, ela permanecerá aberta para visualização dos eventos propagados no log:



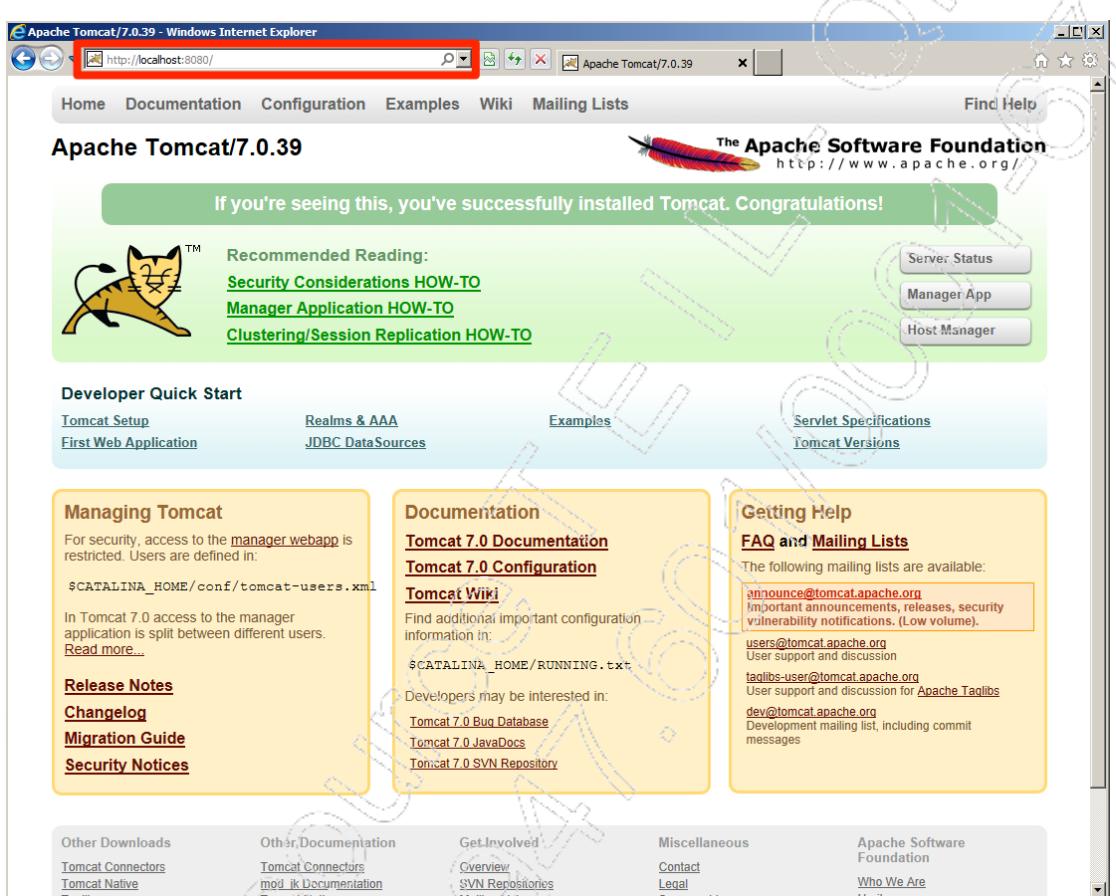
```
Tomcat  
INFO: Starting service Catalina  
May 06, 2013 7:40:01 PM org.apache.catalina.core.StandardEngine startInternal  
INFO: Starting Servlet Engine: Apache Tomcat/7.0.39  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.HostConfig deployDirectory  
INFO: Deploying web application directory C:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\webapps\docs  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.HostConfig deployDirectory  
INFO: Deploying web application directory C:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\webapps\examples  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.HostConfig deployDirectory  
INFO: Deploying web application directory C:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\webapps\host-manager  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.HostConfig deployDirectory  
INFO: Deploying web application directory C:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\webapps\manager  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.HostConfig deployDirectory  
INFO: Deploying web application directory C:\java\apache-tomcat-7.0.39-windows-x64\apache-tomcat-7.0.39\webapps\ROOT  
May 06, 2013 7:40:01 PM org.apache.coyote.AbstractProtocol start  
INFO: Starting ProtocolHandler ["http-apr-8080"]  
May 06, 2013 7:40:01 PM org.apache.coyote.AbstractProtocol start  
INFO: Starting ProtocolHandler ["ajp-apr-8009"]  
May 06, 2013 7:40:01 PM org.apache.catalina.startup.Catalina start  
INFO: Server startup in 762 ms
```

Se a inicialização do servidor tiver ocorrido sem erros, será possível visualizar ao final do log: **Server startup in 762 ms**. Isso significa que o servidor agora está executando.

Para conferir que sua execução está correta, abra uma janela de seu navegador e insira o seguinte endereço: <http://localhost:8080>.

Esse endereço é o padrão para execução na máquina local e a porta 8080 é a utilizada por padrão pelo Apache Tomcat para acesso HTTP às suas aplicações.

A seguinte página deverá ser visualizada no navegador caso a execução do servidor esteja correta:



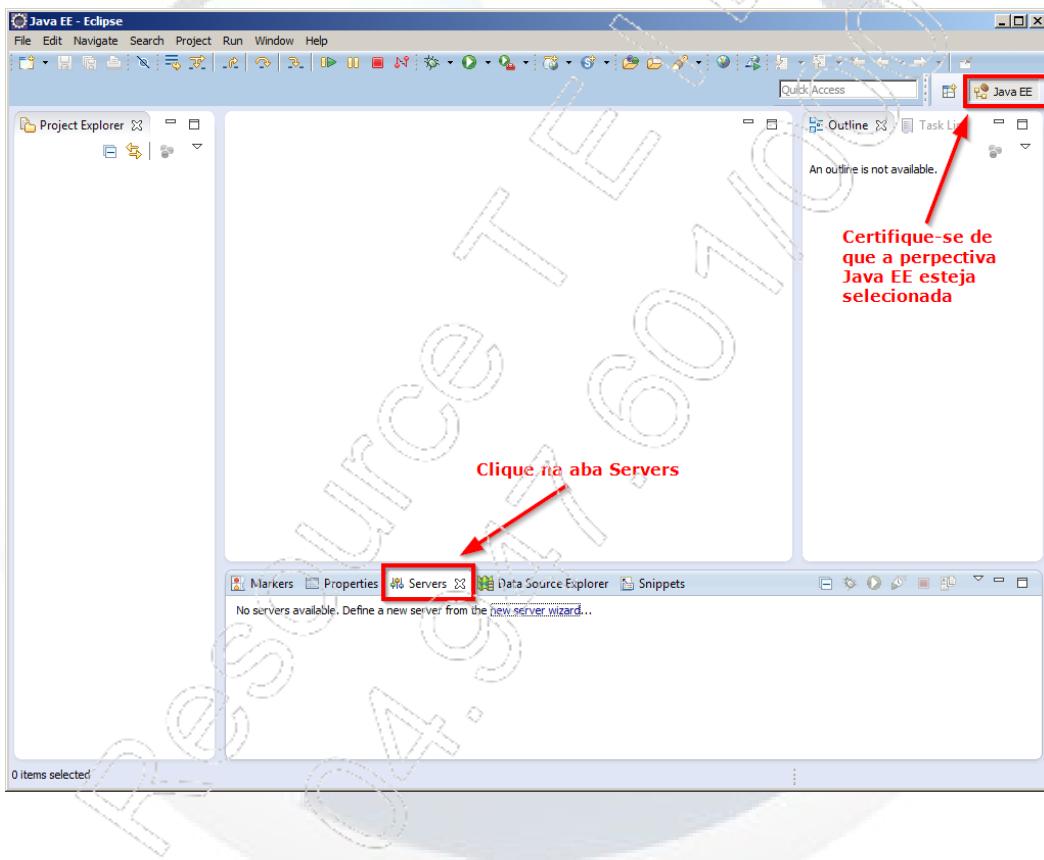
Muito bem! Agora o Apache Tomcat está executando com sucesso.

Para interromper seu funcionamento, basta executar o comando **shutdown.bat** no prompt de comando ou simplesmente fechar a janela do prompt que exibe o log do servidor.

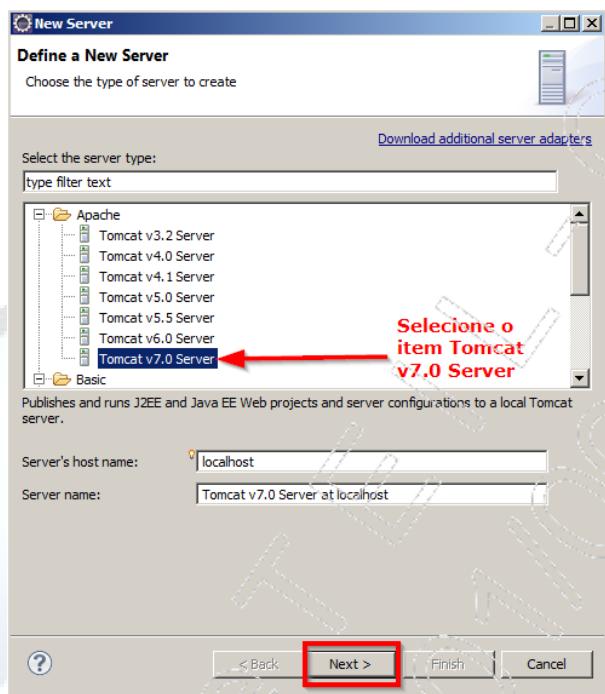
# Execução do Apache Tomcat via IDE Eclipse

Abra o IDE Eclipse e já selecione a perspectiva Java EE como padrão, de forma a iniciar a configuração e execução do Apache Tomcat usando os assistentes fornecidos com a instalação do IDE.

Verifique a existência da aba **Servers** no quadrante inferior da tela do Eclipse e selecione-a, de acordo com a imagem a seguir:

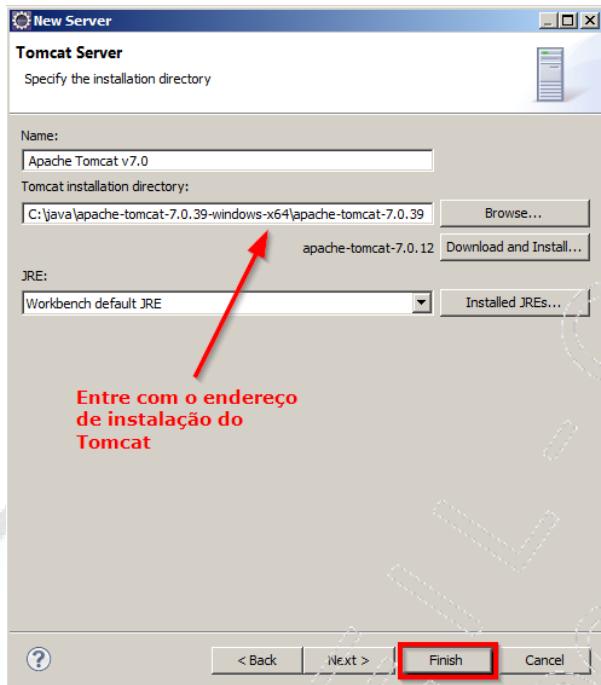


Uma vez escolhida a aba **Servers**, clique no link **new server wizard**, que vai guiá-lo para a instalação de um novo servidor no Eclipse. A seguinte caixa de diálogo será exibida:



Nessa caixa de diálogo, abra a opção **Apache**, escolha o item **Tomcat v7.0 Server** e clique em **Next**.

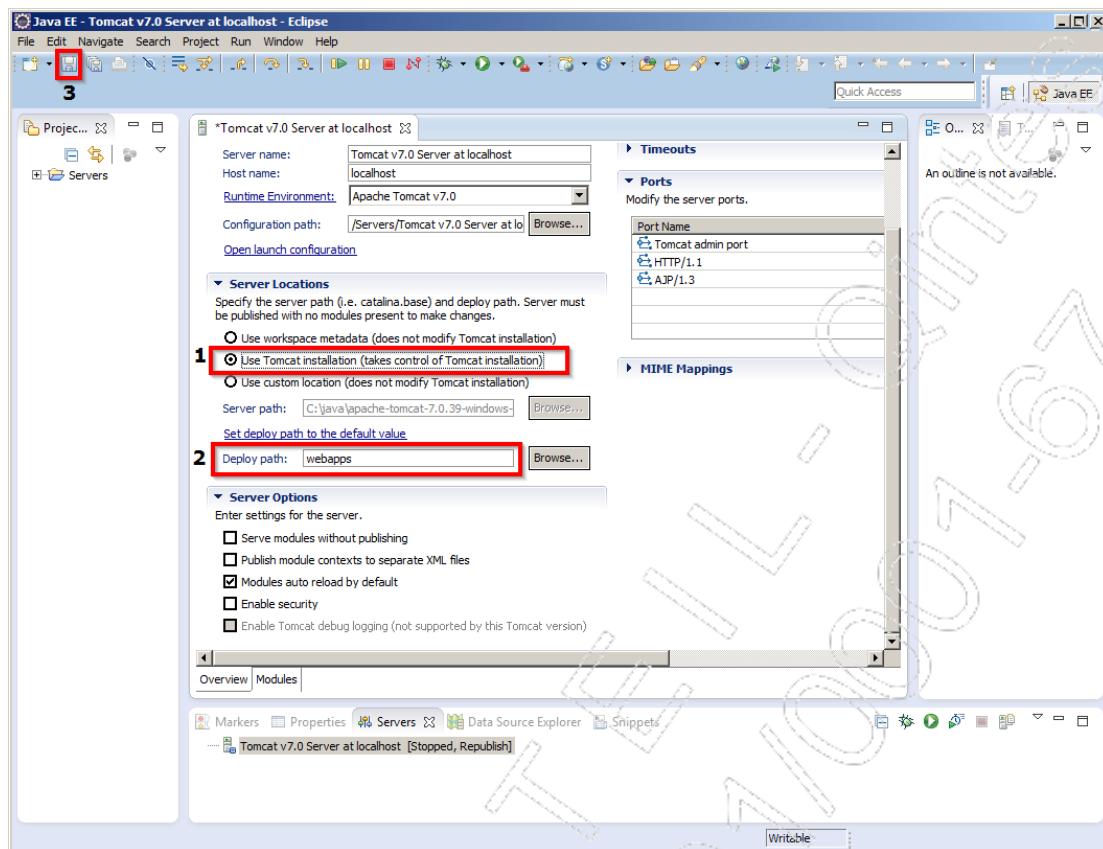
Em seguida, escolha um nome para identificar o servidor na aba **Servers** (é recomendado manter o padrão). Clique em **Browse...** ou digite o endereço completo para a pasta raiz do Apache Tomcat. Clique em **Finish** em seguida e a instalação estará completa.



Dentro do Eclipse, aplique um duplo-clique sobre o nome do servidor recém-instalado. Uma janela com vários itens de configuração do Apache Tomcat será aberta no editor principal do Eclipse. Nessa janela, dois itens deverão ser alterados, ambos no campo **Server Locations**.

Para isso, altere a seleção feita clicando no radio button **Use Tomcat installation (takes control of Tomcat installation)** (1). No campo de texto **Deploy path**, altere o valor de **wtpwebapps** para somente **webapps** (2). Em seguida, salve o arquivo (3). A imagem a seguir apresenta esse procedimento:

## Apêndice 1: Instalação e configuração do Apache Tomcat 7



O servidor, anteriormente instalado, agora está pronto para ser utilizado via Eclipse. Para testá-lo imediatamente, clique com o botão direito do mouse sobre o servidor e selecione **Start**. Perceba que a aba **Console** será preenchida com todo o log do servidor. O status apresentado na aba **Servers**, entre parênteses à frente do servidor, passa de **Stopped** para **Started**.

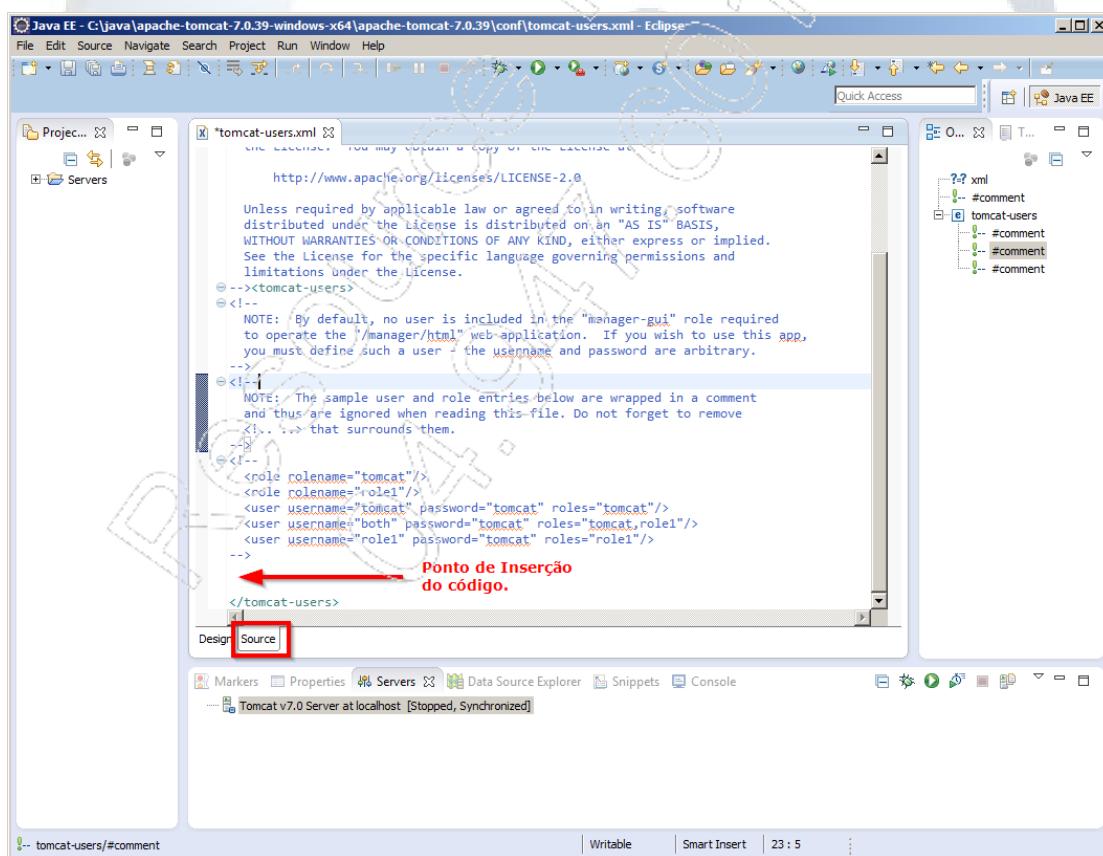
Para testar se o servidor está realmente executando, abra um navegador e insira a URL: <http://localhost:8080>. A mesma tela apresentada para a execução via prompt de comando deverá ser exibida.

# Configuração de usuários no Apache Tomcat

Para obter acesso ao aplicativo manager do Apache Tomcat, precisamos configurar ao menos um usuário com esse tipo de acesso. Interrompa o servidor, clicando com o botão direito do mouse sobre ele na aba **Servers** e selecionando **stop**.

Em seguida, dentro do Eclipse, clique em **File / Open File...** e navegue até encontrar o arquivo **tomcat-users.xml**, dentro da pasta **conf**, na raiz de instalação do seu Apache Tomcat.

Abra o arquivo e selecione, no editor do Eclipse, a aba **Source**, para ver o conteúdo do arquivo:



Exatamente como foi apontado na imagem anterior, insira o seguinte código no ponto de inserção do arquivo:

```
<role rolename="manager-gui" />  
<user username="impacta" password="1234" roles="manager-gui" />
```

Perceba que o trecho anterior define um usuário de nome **impacta** e uma senha **1234**, mas você poderá definir o usuário e senha que preferir.

Essa configuração permite acessar o **manager-gui**, interface gráfica do Apache Tomcat que permite visualizar, testar, implantar e “desimplantar” aplicações via navegador.

Para testar, inicie novamente o seu servidor. Se já estava iniciado, é necessário reiniciá-lo para que possa refletir as mudanças feitas no arquivo de usuários.

Abra um navegador e insira o endereço <http://localhost:8080>. Na página inicial do tomcat, clique no botão **Manager App**, conforme indicado na imagem a seguir:

## Apêndice 1: Instalação e configuração do Apache Tomcat 7

The Apache Software Foundation <http://www.apache.org/>

If you're seeing this, you've successfully installed Tomcat. Congratulations!

**Developer Quick Start**

[Tomcat Setup](#) [Realms & AAA](#) [Examples](#)

[First Web Application](#) [JDBC DataSources](#)

**Managing Tomcat**  
For security, access to the [manager](#) webapp is restricted. Users are defined in:  
`$CATALINA_HOME/conf/tomcat-users.xml`  
In Tomcat 7.0 access to the manager application is split between different users.  
[Read more...](#)

**Release Notes**  
**Changelog**  
**Migration Guide**  
**Security Notices**

**Documentation**  
[Tomcat 7.0 Documentation](#)  
[Tomcat 7.0 Configuration](#)  
[Tomcat Wiki](#)  
Find additional important configuration information in:  
`$CATALINA_HOME RUNNING.txt`  
Developers may be interested in:  
[Tomcat 7.0 Bug Database](#)  
[Tomcat 7.0 JavaDocs](#)  
[Tomcat 7.0 SVN Repository](#)

**Getting Help**  
[FAQ and Mailing Lists](#)  
The following mailing lists are available:  
[announce@tomcat.apache.org](mailto:announce@tomcat.apache.org)  
Important announcements, releases, security vulnerability notifications. (Low volume).  
[users@tomcat.apache.org](mailto:users@tomcat.apache.org)  
User support and discussion  
[taglibs-user@tomcat.apache.org](mailto>taglibs-user@tomcat.apache.org)  
User support and discussion for [Apache Taglibs](#)  
[dev@tomcat.apache.org](mailto:dev@tomcat.apache.org)  
Development mailing list, including commit messages

**Other Downloads**  
[Tomcat Connectors](#)  
[Tomcat Native](#)  
[Taglibs](#)  
[Deployer](#)

**Other Documentation**  
[Tomcat Connectors](#)  
[mod\\_ajp Documentation](#)  
[Tomcat Native](#)  
[Deployer](#)

**Get Involved**  
[Overview](#)  
[SVN Repositories](#)  
[Mailing Lists](#)  
[Wiki](#)

**Miscellaneous**  
[Contact](#)  
[Legal](#)  
[Sponsorship](#)  
[Thanks](#)

**Apache Software Foundation**  
[Who We Are](#)  
[Heritage](#)  
[Apache Home](#)  
[Resources](#)

Copyright ©1999-2013 Apache Software Foundation. All Rights Reserved

Será solicitada a entrada de usuário e senha. Insira o usuário e a senha configurados no arquivo `tomcat-users.xml`. Caso ocorra algum erro, verifique novamente o arquivo, salve-o, reinicie o servidor e verifique novamente.

## Apêndice 1: Instalação e configuração do Apache Tomcat 7

Com o acesso concedido, deverá ser apresentada a seguinte página:

The screenshot shows the Apache Tomcat 7 Manager application running in a Windows Internet Explorer browser window. The title bar reads "e /manager - Windows Internet Explorer" and the address bar shows "http://localhost:8080/manager/html". The main content area is titled "Tomcat Web Application Manager". At the top left, there's the Apache Software Foundation logo and a cartoon cat logo. A message box says "Message: OK". Below that is a navigation bar with tabs: "Manager", "List Applications", "HTML Manager Help", "Manager Help", and "Server Status". The main table is titled "Applications" and lists the following entries:

Path	Version	Display Name	Running	Sessions	Commands
/	None specified	Welcome to Tomcat	true	0	<input type="button" value="Start"/> <input type="button" value="Stop"/> <input type="button" value="Reload"/> <input type="button" value="Undeploy"/> <input type="button" value="Expire sessions"/> with idle ≥ 30 minutes
/docs	None specified	Tomcat Documentation	true	2	<input type="button" value="Start"/> <input type="button" value="Stop"/> <input type="button" value="Reload"/> <input type="button" value="Undeploy"/> <input type="button" value="Expire sessions"/> with idle ≥ 30 minutes
/examples	None specified	Servlet and JSP Examples	true	0	<input type="button" value="Start"/> <input type="button" value="Stop"/> <input type="button" value="Reload"/> <input type="button" value="Undeploy"/> <input type="button" value="Expire sessions"/> with idle ≥ 30 minutes
/host-manager	None specified	Tomcat Host Manager Application	true	0	<input type="button" value="Start"/> <input type="button" value="Stop"/> <input type="button" value="Reload"/> <input type="button" value="Undeploy"/> <input type="button" value="Expire sessions"/> with idle ≥ 30 minutes
/manager	None specified	Tomcat Manager Application	true	2	<input type="button" value="Start"/> <input type="button" value="Stop"/> <input type="button" value="Reload"/> <input type="button" value="Undeploy"/> <input type="button" value="Expire sessions"/> with idle ≥ 30 minutes

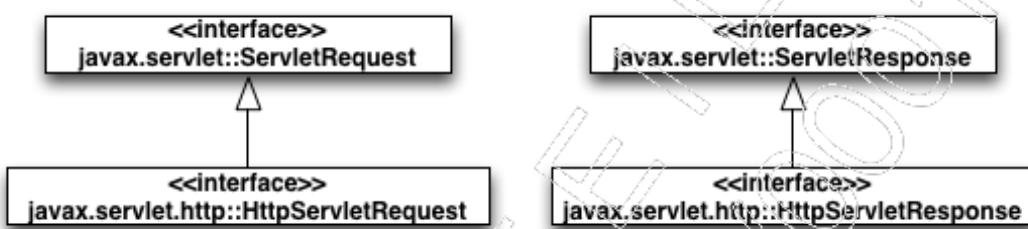
Após realizar todos os passos anteriores, o Apache Tomcat estará pronto para utilização e teste das suas aplicações!

# Apêndice 2: Os objetos Request e Response com detalhes

# Introdução

Em toda requisição HTTP feita e direcionada a um Servlet ou a uma JSP, dois objetos são criados e disponibilizados pelo Container ao desenvolvedor por meio dos métodos **doGet(...)** e **doPost(...)**: um objeto de tipo **HttpServletRequest** e outro de tipo **HttpServletResponse**.

Esses objetos são, na verdade, passados por meio de uma referência polimórfica e podem ser expressos pela seguinte árvore de herança:



A seguir, listaremos os métodos disponíveis em cada uma dessas interfaces, para utilização pelo desenvolvedor, bem como um exemplo prático dessa aplicação e a saída correspondente.

# A interface HttpServletRequest

Métodos	Descrição
<b>getAttribute(name:String):Object</b>	Retorna um Object que representa um atributo gravado na Request.
<b>getAttributeNames():Enumeration</b>	Retorna uma Enumeration com todos os nomes dos atributos contidos na request.
<b>setAttribute(name:String, value:Object):void</b>	Armazena um atributo na Request, recebendo seu nome e valor.
<b>removeAttribute(name:String):void</b>	Remove o atributo da Request cujo nome é passado por parâmetro.
<b>getParameter(name:String):String</b>	Retorna o valor String de um parâmetro cujo nome é o passado para o método, ou null.
<b>getParameterMap():Map</b>	Retorna um mapa com todos os parâmetros dessa request.
<b>getParameterValues(name: String):String[]</b>	Retorna um array de Strings contendo o valor de cada um dos parâmetros passados a essa Request, ou null.
<b>getParameterNames():Enumeration</b>	Retorna uma Enumeration de objetos String com o nome de cada um dos parâmetros passados à Request.
<b>getMethod():String</b>	Retorna o nome do método HTTP usado para essa requisição.
<b>getHeader(name:String):String</b>	Retorna o valor em String do header especificado por parâmetro.
<b>getCookies():Cookie[]</b>	Retorna um array com os objetos Cookie provindos do cliente na Request.
<b>getInputStream():InputStream</b>	Retorna um InputStream - os dados constantes do corpo da Request como dados binários.
<b>getContextPath():String</b>	Retorna a porção da URI que indica o contexto da Request.
<b>getQueryString():String</b>	Retorna a porção da URI que representa os parâmetros passados por GET, se

---

## Apêndice 2: Os objetos Request e Response com detalhes

Métodos	Descrição
	existirem.
<b>getRequestURI():String</b>	Retorna a porção desta URL que corresponde ao caminho iniciado após a definição.
<b>getRequestURL():StringBuffer</b>	Retorna a URL completa usada para realizar a Request.
<b>getServletPath():String</b>	Retorna a porção da URL usada para chamar o Servlet.



# A interface HttpServletResponse

Métodos	Descrição
<b>setContentType(len:String):void</b>	Envia o MIME Type ao navegador do solicitante de forma a alertá-lo sobre o tipo de conteúdo que está sendo enviado.
<b>getWriter():PrintWriter</b>	Retorna um objeto <b>PrintWriter</b> , um stream pronto para enviar texto ao cliente.
<b>getOutputStream(): ServletOutputStream</b>	Retorna um objeto <b>ServletOutputStream</b> , um stream necessário para envio de dados binários ao cliente.
<b>encodeURL(url:String):String</b>	Recebe uma URL e a retorna codificada com o ID da sessão, se necessário.
<b>sendRedirect(location:String):void</b>	Redireciona a response para o cliente, encarninhando-o para a URL passada.
<b>addCookie(cookie:Cookie):void</b>	Adiciona um cookie na Response do cliente.
<b>addHeader(name:String, value:String):void</b>	Adiciona um header na Response com nome e valor especificados.
<b>containsHeader(name:String): boolean</b>	Verifica se existe na Response um header com o nome passado e retorna um boolean.
<b>sendError(sc:int [, msg:String]):void</b>	Envia um erro para a Response com o status especificado.
<b>setStatus(sc:int):void</b>	Define o código de Status para essa Response.

# Exemplo da utilização dos métodos da interface HttpServletRequest

Supondo a utilização do projeto criado no Capítulo 4 no Eclipse, chamado de **PassagemDeParametrosGET**, criamos um novo Serviet denominado **TesteMetodos**, com o seguinte código em seu método **doGet(...)**:

```

20  protected void doGet(HttpServletRequest request,
21      HttpServletResponse response) throws ServletException, IOException {
22      response.setContentType("text/html");
23      PrintWriter out = response.getWriter();
24      out.println("<HTML>");
25      out.println("<HEAD><TITLE>Teste Métodos Request"
26          + "</TITLE></HEAD>");
27      out.println("<BODY>");
28      out.println("<h1>Teste Métodos Request</h1>");
29      out.println("<hr><br />");
30      out.println("getMethod(): " + request.getMethod() + "<br />");
31      out.println("getContextPath(): " + request.getContextPath() + "<br />");
32      out.println("getQueryString(): " + request.getQueryString() + "<br />");
33      out.println("getRequestURI(): " + request.getRequestURI() + "<br />");
34      out.println("getRequestURL(): " + request.getRequestURL() + "<br />");
35      out.println("getServletPath(): " + request.getServletPath() + "<br />");
36      out.println("getServerName(): " + request.getServerName() + "<br />");
37      out.println("getServerPort(): " + request.getServerPort() + "<br />");
38      out.println("</BODY>");
39      out.println("</HTML>");
40
41  }

```

Garantindo que esse Servlet está declarado e mapeado no **web.xml** de forma adequada para a URL **/TesteMetodos**, executamos a aplicação e acessamos a seguinte URL no browser:

**http://localhost:8080/PassagemDeParametrosGET/TesteMetodos?id=2&categoria=5**

Note que estamos passando parâmetros propositalmente para analisar o comportamento dos métodos do objeto request.

Obteremos, então, a seguinte página de saída, produto do Servlet criado:

