argonavis
tecnologia e arte

aplicações WEB em java

servlets

war jsp

jstl cookies

jaas filtros

mvc taglib web.xml

JAVA EE 7

Helder da Rocha

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

Capítulo 2: WebServlets

1 Introdução

WebServlets são a base das aplicações Web em Java EE. Embora a principal API para criação de aplicações Web seja *JavaServer Faces*, *WebServlets* é a plataforma básica sobre a qual *JSF* é construído, e ainda serve de suporte a vários outros serviços fornecidos via Web, como *WebServices REST* e *SOAP*, *WebSockets*, *Filtros*, além de ser a base para frameworks Web de outros fabricantes (Spring MVC, Wicket, GWT, etc.)

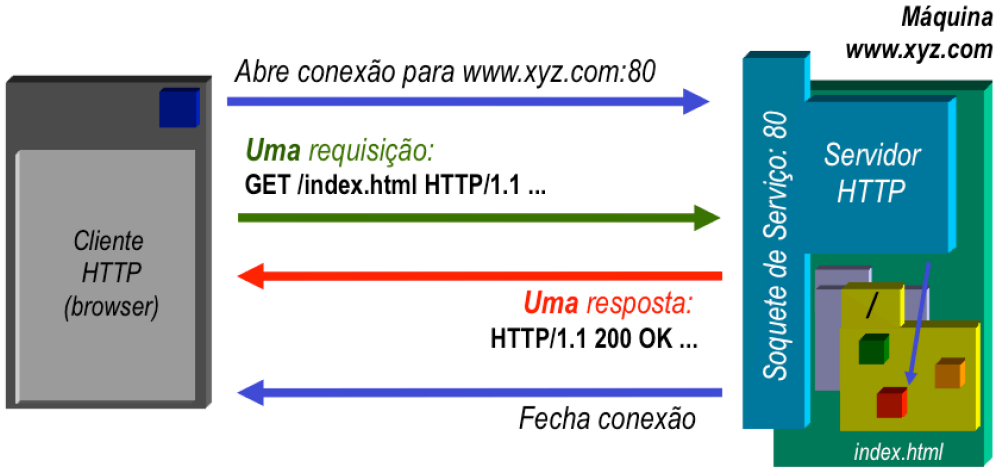
1.1 Arquitetura Web

WebServlets são classes que representam uma *requisição* e *resposta HTTP* em uma aplicação Web. A arquitetura Web é baseada em *cliente* (geralmente um browser ou um script), *protocolo HTTP* e *servidor Web* (que pode estar sob controle de um servidor de aplicações) que na maior parte das vezes serve arquivos estáticos (paginas, imagens, etc.) mas que pode ser configurado para executar comandos e servir dados dinâmicos.

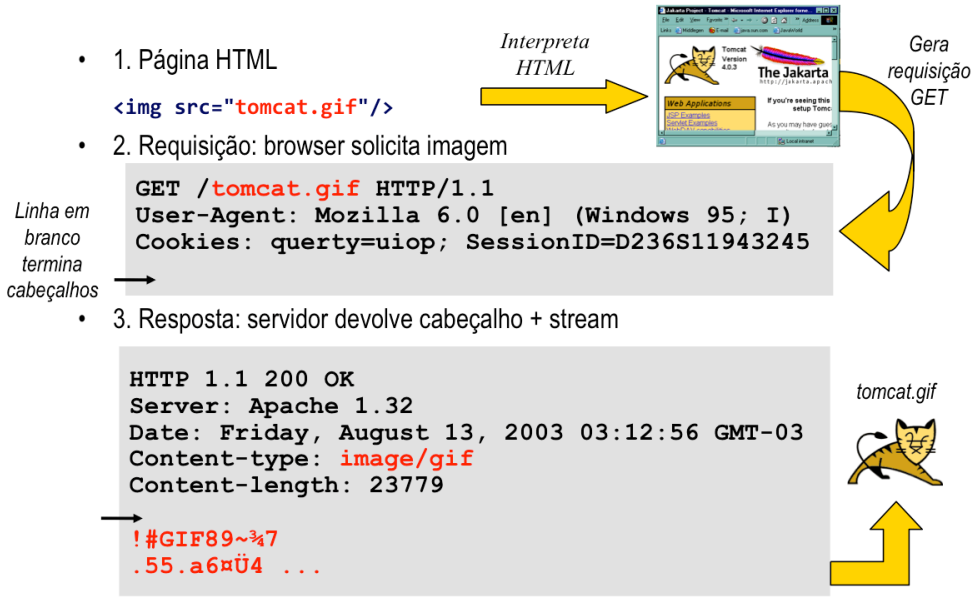
A arquitetura Web e centrada no protocolo de transferência de arquivos HTTP (padrão Internet, RFC 2068), que possui como uma das principais diferenças entre outros protocolos de transferência de arquivos (como FTP e WEBDAV) a característica de não manter o estado da sessão do cliente.

O servidor representa um sistema de arquivos virtual e responde a comandos que contém URLs de localização para cada recurso disponibilizado. Os comandos HTTP (requisições) e as respostas contém cabeçalhos com meta-informação sobre a comunicação.

O diagrama abaixo ilustra uma requisição e resposta HTTP realizada por um browser a um servidor Web:



A figura a seguir mostra detalhes dos cabeçalhos de requisição e resposta gerados quando um browser interpreta um tag HTML que requer a criação de uma requisição GET, e a correspondente resposta do servidor retornando a imagem solicitada:



1.2 Aplicaçoes Web (WAR)

Ha várias especificações Java que envolvem serviços Web. A principal é a especificação de *WebServlets*, que descreve não apenas como construir *servlets*, mas como empacotar aplicações Web para implantação em containers Web Java EE.

Uma *aplicação Web* em Java (no Java EE 7) consiste de pelo menos um arquivo HTML empacotado em um *arquivo WAR*, contendo uma pasta *WEB-INF*. Geralmente consiste de vários arquivos XHTML, JSP ou HTML, junto com imagens, arquivos JavaScript e CSS, arquivos XML de configuração, arquivo *web.xml*, arquivos *.properties*, JARs e classes Java.

O arquivo WAR é um ZIP que contém uma estrutura de arquivos e diretórios. A raiz do arquivo corresponde a raiz do contexto da aplicação, e quaisquer arquivos ou pastas lá contidas serão por default acessíveis aos clientes Web. A pasta WEB-INF é privativa e pode conter arquivos que não serão expostos na URL de acesso à aplicação. Dentro da pasta WEB-INF fica o arquivo web.xml, se existir, e se houver classes Java, como servlets, managed beans, etc. elas ou são localizadas dentro de WEB-INF/classes (que é o classpath) ou empacotadas em um JAR dentro de WEB-INF/lib:

```
biblioteca.war
    index.xhtml
    login.xhtml
    logo.png
    css/
```

```
      biblioteca.css
   js/
      jquery.js
      app.js
   WEB-INF/
      web.xml
       classes/
         br/com/unijava/biblioteca/
         CadastroServlet.class
         EmprestimoServlet.class
       lib/
         ojdbc7.jar
```

O contexto da aplicação geralmente é mapeado como um subcontexto da raiz de documentos do servidor (que geralmente é /). Por default, o nome do contexto é o nome do WAR, mas é comum que esse nome seja alterado na configuração da aplicação. Por exemplo, a aplicação *biblioteca.war*, publicada em um servidor Tomcat localizado em *http://localhost:8080* será acessível, por default, em

```
http://localhost:8080/biblioteca
```

1.3 Configuração

O arquivo *web.xml* é o *Web Deployment Descriptor*. Em aplicações Java EE 7 muito simples ele é opcional. Se estiver presente deve aparecer dentro de WEB-INF. A maior parte das configurações realizadas no web.xml podem também ser realizadas via anotações, no entanto a configuração em web.xml tem precedência e sobrepõe as configurações em anotações.

Dependendo do servidor usado, e do tipo e complexidade da aplicação, poderá ser necessário incluir outros arquivos de configuração além do web.xml em uma aplicação Web. A estrutura mínima do web.xml e:

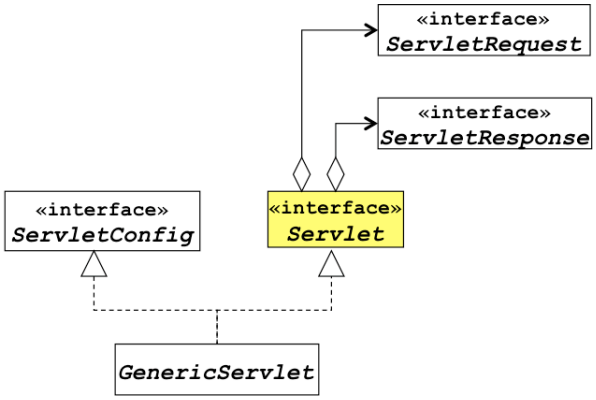
```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

</web-app>
```

2 WebServlets

WebServlets são um tipo de *javax.servlet.Servlet* – componente que encapsula um serviço em um servidor. A API de servlets contém interfaces genéricas que permite a construção

desses componentes para qualquer tipo de servidor, mas a única implementação disponível é para servlets HTTP ou WebServlets.

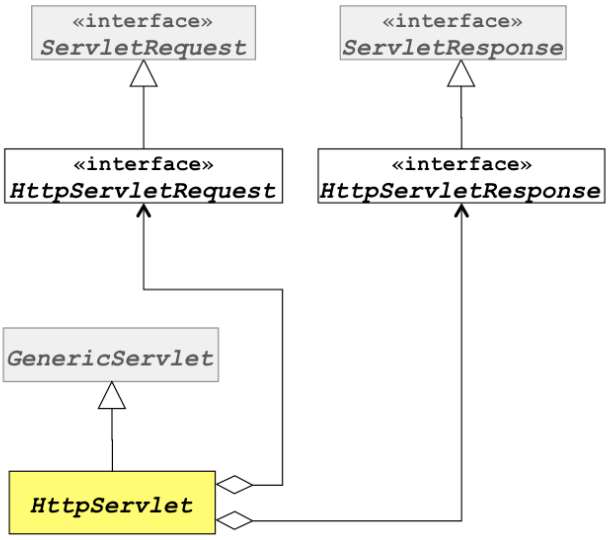
Todo servlet possui um método *service()* que representa o serviço realizado pelo servidor. Este método recebe um par de objetos que representam a requisição (*ServletRequest*), feita pelo cliente, e a resposta (*ServletResponse*), feita pelo servidor. Dentro do método é possível extrair informações do objeto da ServletRequest, e usar o ServletResponse para construir uma resposta. A hierarquia de interfaces está mostrada abaixo:



O *WebServlet* estende essas interfaces no pacote *javax.servlet.http* e implementa *GenericServlet* para lidar com os protocolos da Web. Em vez de um método *service()*, o *HttpServlet* disponibiliza métodos Java para cada um dos métodos HTTP: *GET*, *POST*, *PUT*, *HEAD*, *DELETE*, *OPTIONS*. Sua implementação de *service()* redireciona para um método *doXXX()* correspondente (*doGet()*, *doPost()*, etc.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {...}
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) {...}
...
```

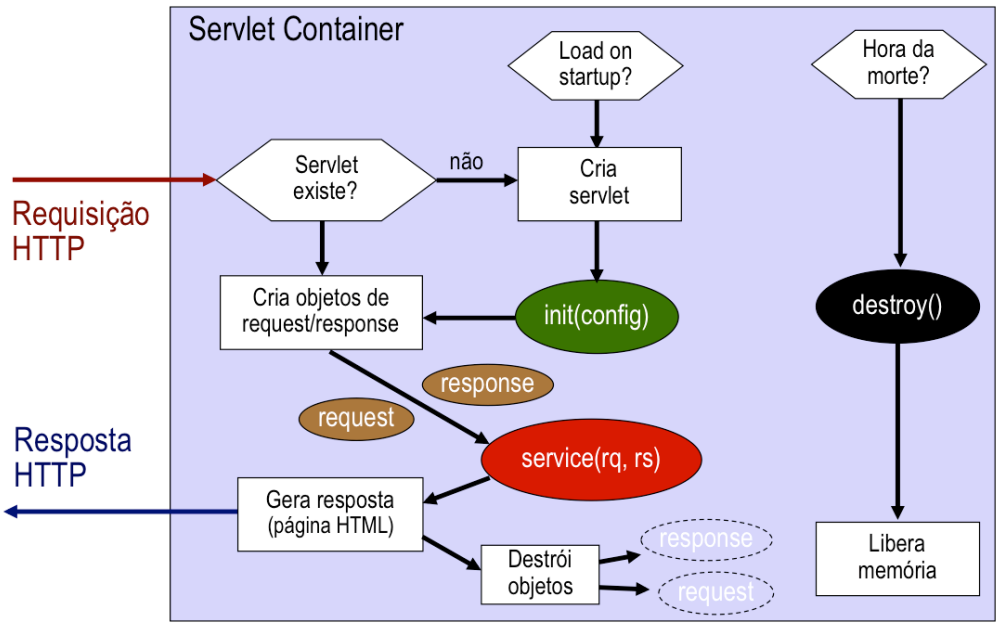
A hierarquia de *HttpServlet* está mostrada abaixo:



O container Web controla o ciclo de vida de um *WebServlet*. Após a implantação do WAR, o servlet pode ser carregado e inicializado (ou inicializado com a primeira requisição). Enquanto estiver ativo, pode receber requisições.

A cada requisição objetos request (*HttpServletRequest*) e response (*HttpServletResponse*) serão criados, configurados e passados para o método *service()*, que o repassa a um método *doGet()*, *doPost()*, ou outro de acordo com o protocolo HTTP recebido. Ao final da execução do método, os dois objetos são destruídos.

O servlet permanece ativo até que seja destruído (isto depende de política do container ou servidor). A ilustração abaixo mostra as etapas do ciclo de vida:



2.1 Criando um WebServlet

Para criar um *WebServlet* e necessário criar uma classe que estenda a classe *HttpServlet* e implementar pelo menos um dos métodos *doGet()* ou *doPost()*. Só um dos dois será chamados automaticamente quando ocorrer uma requisição HTTP. Cada um deles recebe objetos *HttpServletRequest* e *HttpServletResponse* que representam, respectivamente, a requisição e a resposta HTTP. O *request* vem preenchido com dados da requisição e cabeçalhos, e possíveis parâmetros e cookies enviados pelo cliente, que podem ser lidos dentro do método. A *resposta* deve ser configurada e preenchida, obtendo seu stream de saída para produzir uma pagina de resposta.

A classe deve ser anotada com *@WebServlet* e o caminho para executar o servlet dentro do contexto da aplicação. Depois de compilada, deve ser incluída em *WEB-INF/classes*, empacotada no WAR e instalada no servidor.

Abaixo um servlet simples que apenas gera uma resposta em HTML, acessível em *http://servidor:porta/contexto/HelloServlet*:

```
@WebServlet("/HelloServlet")
public class ServletWeb extends HttpServlet {
   public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
                                  throws IOException {

        Writer out = response.getWriter();
        response.setContentType("text/html");
        out.println("<h1>Hello</h1>");
   }
}
```

2.2 Mapeamento do servlet no contexto

Para ser acessado, o *WebServlet* precisa ser mapeado a um caminho acessível no contexto. Por exemplo, se na raiz contexto há uma página HTML, ela pode ter um link para o servlet ilustrado anteriormente através de uma URL relativa:

```
<a href="HelloServlet">Link para o servlet</a>
```

O nome escolhido pode ser qualquer um, incluindo caminhos dentro do contexto como *"/caminho/para/servlet"* ou mesmo curingas (ex: mapear a **.xyz* irá redirecionar qualquer URL terminada em *.xyz* para o servlet). Exemplos de mapeamentos possíveis incluem:

- **/nome** – mapeamento exato (relativo a contexto)
- **/nome/subnome** – mapeamento exato (relativo a contexto)

- / - servlet default (chamado se nenhum dos outros mapeamentos existentes combinar com a requisição)
- **/nome/*** - mapeamento de caminho (relativo a contexto). Aceita texto adicional (que e tratado como *path info*) após nome do servlet na requisição.
- **/nome/subnome/*** - outro exemplo de mapeamento de caminho
- ***.ext** - mapeamento de extensão. Qualquer URL terminando nesta extensão redireciona a requisição para o servlet.

O mapeamento pode ser declarado usando a anotação *@WebServlet*, contendo pelo menos um mapeamento no atributo value, que é o default:

```
@WebServlet("/listar")
public class ProdutoServlet extends HttpServlet { … }
```

Ou no atributo *urlPatterns*, que aceita vários mapeamentos:

```
@WebServlet(urlPatterns = {"/listar", "/detalhar"})
public class ProdutoServlet extends HttpServlet { … }
```

O servlet também pode ser mapeado no arquivo *web.xml*, usando dois conjuntos de tags (um para associar a classe do servlet a um nome, e o outro para associar o nome a um mapeamento de URL):

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.ServletWeb</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>umServlet</servlet-name>
    <url-pattern>/HelloServlet</url-pattern>
</servlet-mapping>
```

2.2.1 Configuração de inicialização

Parâmetros de inicialização podem ser definidos para cada instância de um servlet usando o elemento <*init-param*> dentro de <*servlet*>:

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.MeuServlet</servlet-class>
     <init-param>
       <param-name>dir-imagens</param-name>
       <param-value>c:/imagens</param-value>
     </init-param>
     <init-param> ... </init-param>
  </servlet>
```

Esses parâmetros podem ser recuperados dentro do servlet através de seu método de inicialização:

```
private java.io.File dirImagens = null;

public void init() throws ServletException {

    String dirImagensStr = getInitParameter("dir-imagens");

    if (dirImagensStr == null) {
        throw new UnavailableException("Configuração incorreta!");
    }

    dirImagens = new File(dirImagensStr);

    if (!dirImagens.exists()) {
         throw new UnavailableException("Diretorio de imagens nao existe!");
    }
}
```

Também e possível definir parâmetros de inicialização no próprio servlet usando *@WebInitParam* (embora não pareça ter tanta utilidade declará-los no mesmo arquivo onde ele é recuperado):

```
@WebServlet(
        urlPatterns = "/HelloServlet",
         initParams = @WebInitParam(name = "dir-imagens", value = "C:/imagens")
)
public class ServletWeb extends HttpServlet { … }
```

2.3 Requisição

Uma requisição HTTP feita pelo browser tipicamente contém vários cabeçalhos RFC822 (padrão de cabeçalhos de email):

```
GET /docs/index.html HTTP/1.0
Connection: Keep-Alive
Host: localhost:8080
User-Agent: Mozilla 6.0 [en] (Windows 95; I)
Accept: image/gif, image/x-bitmap, image/jpg, image/png, */*
Accept-Charset: iso-8859-1, *
Cookies: jsessionid=G3472TS9382903
```

Os métodos de *HttpServletRequest* permitem extrair informações de qualquer um deles. Pode-se também identificar o método e URL. Estas e outras informações sobre a requisição podem ser obtidas através dos métodos do objeto *HttpServletRequest*. Alguns deles estão listados abaixo:

- Enumeration **getHeaderNames**() - obtém nomes dos cabeçalhos

- String **getHeader**("nome") - obtém primeiro valor do cabeçalho
- Enumeration **getHeaders**("nome") - todos os valores do cabeçalho
- String **getParameter**(param) - obtém parâmetro HTTP
- String[] **getParameterValues**(param) - obtém parâmetros repetidos
- Enumeration **getParameterNames**() - obtém nomes dos parâmetros
- Cookie[] **getCookies**() - recebe cookies do cliente
- HttpSession **getSession**() - retorna a sessão
- void **setAttribute**("nome", obj) - define um atributo obj chamado "nome".
- Object **getAttribute**("nome") - recupera atributo chamado nome

2.3.1 Parâmetros de requisição

Parâmetros são pares nome=valor que são enviados pelo cliente concatenados em strings separados por &:

```
nome=Jo%E3o+Grand%E3o&id=agente007&acesso=3
```

Parâmetros podem ser passados na requisição de duas formas.

Se o método for GET, os parâmetros são passados em uma única linha no query string, que estende a URL após um "?"

```
GET /servlet/Teste?id=agente007&acesso=3 HTTP/1.0
```

Se o método for POST, os parâmetros são passados como um stream no corpo na mensagem (o cabeçalho *Content-length*, presente em requisições POST informa o tamanho):

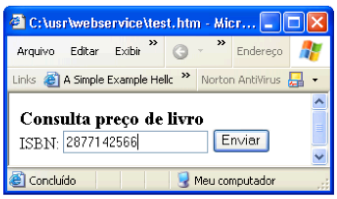
```
POST /servlet/Teste HTTP/1.0
Content-length: 21
Content-type: x-www-form-urlencoded

id=agente007&acesso=3
```

Caracteres reservados e maiores que ASCII-7bit são codificados seguindo o padrão usado em URLs: (ex: ã = %E3). Formulários HTML codificam o texto ao enviar os dados automaticamente (formato *text/x-www-form-urlencoded*).

Seja o método POST ou GET, os valores dos parâmetros podem ser recuperados pelo método *getParameter()* de *ServletRequest*, que recebe seu nome:

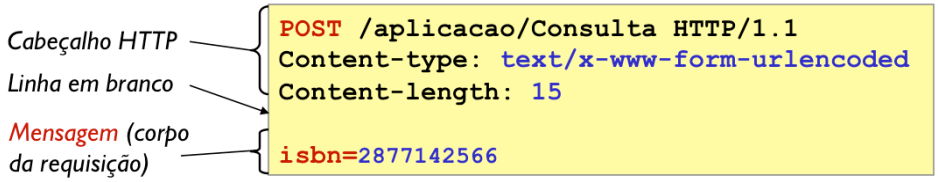
```
String parametro = request.getParameter("nome");
```

Por exemplo, o seguinte formulário HTML recebe um código em um campo de entrada:




```
<form action="/aplicacao/Consulta"
      method="post">
 <h3>Consulta preço de livro</h3>
 <p>ISBN: <input type="text" name="isbn">
 <input type="submit" VALUE="Enviar">
</form>
```

Quando o usuário clicar em enviar, o browser criará a seguinte requisição POST para o servidor:



O servlet abaixo ira receber a requisição no seu método *doPost()*, extrair a informação da requisição e enviar para um serviço (DAO) para recuperar as informações necessárias para que possa gerar uma resposta:

```
@WebServlet("/Consulta")
public class ServletWeb extends HttpServlet {
   @Inject LivroDAO dao;

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

         String isbn = request.getParameter("isbn");
         Livro livro = dao.getLivro(isbn);
       Writer out = response.getWriter();
       response.setContentType("text/html");
       out.println("<h1>"+ livro.getTitulo() +"</h1>");
   }
}
```

Parâmetros de mesmo nome podem estar repetidos. Isto é comum e pode acontecer no caso de checkboxes ou menus de seleção múltipla. Na URL gerada eles não serão sobrepostos, mas concatenados. Neste caso uma chamada a *getParameter()* retornará apenas a primeira ocorrência. Para obter *todas* e preciso usar *String[] getParameterValues()*.

Por exemplo os parâmetros da URL:

```
http://servidor/aplicacao?nome=Fulano&nome=Sicrano
```

Podem ser recuperados usando:

```
String[] params = request.getParameterValues("nome");
```

2.4 Resposta

Uma resposta HTTP é enviada pelo servidor ao browser e contém informações sobre os dados anexados. O exemplo abaixo mostra uma resposta contendo uma página HTML simples:

```
HTTP/1.0 200 OK
Content-type: text/html
Date: Mon, 7 Apr 2003 04:33:59 GMT-03
Server: Apache Tomcat/4.0.4 (HTTP/1.1 Connector)
Connection: close
Set-Cookie: jsessionid=G3472TS9382903

<HTML>
   <h1>Hello World!</h1>
</HTML>
```

Os métodos de *HttpServletResponse* permitem construir um cabeçalho. Alguns dos principais métodos estão listados abaixo:

- void **addHeader**(String nome, String valor) - adiciona cabeçalho HTTP
- void **setContentType**(tipo MIME) - define o tipo MIME que será usado para gerar a saída (text/html, image/gif, etc.)
- void **sendRedirect**(String location) - envia informação de redirecionamento para o cliente (mesmo que enviar o cabeçalho Location: url)
- Writer **getWriter**() - obtém um Writer para gerar a saída. Ideal para saída de texto.
- OutputStream **getOutputStream**() - obtém um OutputStream. Ideal para gerar formatos diferentes de texto (imagens, etc.)
- void **addCookie**(Cookie c) - adiciona um novo cookie
- void **encodeURL**(String url) - envia como anexo da URL a informação de identificador de sessão (sessionid)
- void **reset**() - limpa toda a saída inclusive os cabeçalhos
- void **resetBuffer**() - limpa toda a saída, exceto cabeçalhos

2.4.1 Geração de uma resposta

Para gerar uma resposta, primeiro é necessário obter, do objeto *HttpServletResponse*, um fluxo de saída, que pode ser de caracteres (*Writer*) ou de bytes (*OutputStream*).

```
Writer out = response.getWriter();   // se for texto
OutputStream out = response.getOutputStream();  // se outro formato
```

Apenas um deve ser usado. Os objetos correspondem ao mesmo stream de dados.

Deve-se também definir o tipo de dados a ser gerado. Isto é importante para que o cabeçalho Content-type seja gerado corretamente e o browser saiba exibir as informações

```
response.setContentType("image/png");
```

Depois, pode-se gerar os dados, imprimindo-os no objeto de saída (out):

```
byte[] image = gerarImagemPNG();
out.write(buffer);
```

2.5 Acesso a componentes da URL

A não ser que seja configurado externamente, o *nome do contexto* aparece por default na URL absoluta após o nome/porta do servidor:

```
http://serv:8080/contexto/subdir/pagina.html
http://serv:8080/contexto/servletMapeado
```

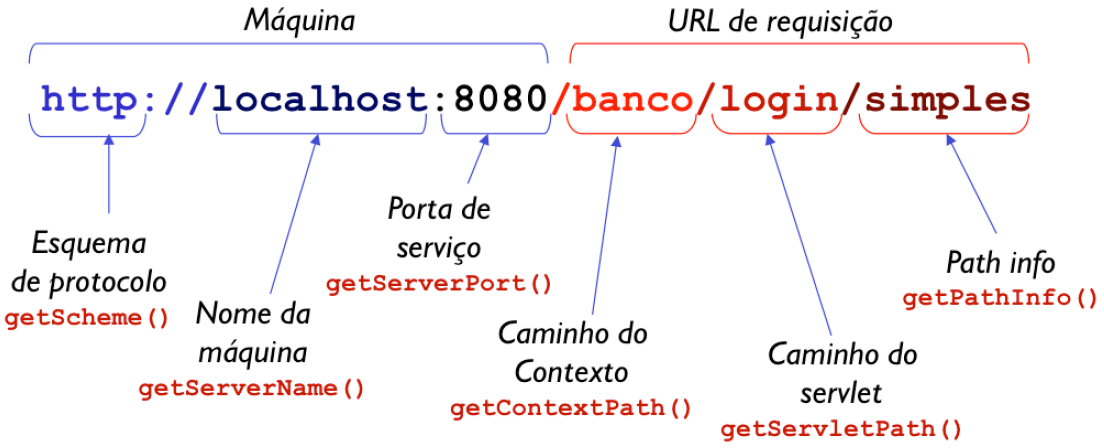
Se um contexto contiver páginas HTML estáticas, e essas páginas contiverem links para outros recursos e arquivos do contexto, elas devem usar, preferencialmente, URLs relativas. A raiz de referência para páginas estáticas não e o contexto, mas a raiz de documentos do servidor, ou *DOCUMENT_ROOT*. Por exemplo: *http://servidor:8080/*. Em links de páginas estáticas documentos podem ser achados relativos ao DOCUMENT_ROOT:

```
/contexto/subdir/pagina.html
/contexto/servletMapeado
```

Para a configuração do contexto (web.xml), a raiz de referência é a raiz de documentos do contexto (aplicação). Por exemplo: *http://servidor:8080/contexto/*. Componentes são identificados relativos ao contexto:

```
/subdir/pagina.html
/servletMapeado
```

Diferentes partes de uma URL usada na requisição podem ser extraídas usando métodos de *HttpServletRequest* listados na figura abaixo:



3 O Contexto

A interface *ServletContext* encapsula informações sobre o contexto ou aplicação. Cada servlet possui um método *getServletContext()* que devolve o contexto atual. A partir de uma referência ao contexto pode-se interagir com a aplicação inteira e compartilhar informações entre servlets.

Os principais métodos de *ServletContext* são:

- String **getInitParameter**(String): lê parâmetros de inicialização do contexto (não confunda com o método similar de ServletConfig)
- Enumeration **getInitParameterNames**(): lê lista de parâmetros
- InputStream **getResourceAsStream**(): lê recurso localizado dentro do contexto como um InputStream
- void **setAttribute**(String nome, Object): grava um atributo no contexto
- Object **getAttribute**(String nome): lê um atributo do contexto
- void **log**(String mensagem): escreve mensagem no log do contexto

3.1 Inicialização do contexto

E possível configurar parâmetros de configuração para o contexto usando o arquivo *web.xml*. Parâmetros consistem de nome e valor armazenados em <*context-param*>:

```
<context-param>
    <param-name>tempdir</param-name>
    <param-value>/tmp</param-value>
</context-param>
```

Para ler um parâmetro no servlet, é preciso obter acesso à instância de *ServletContext* usando o método *getServletContext()*:

```
ServletContext ctx = this.getServletContext();
String tempDir = ctx.getInitParameter("tempdir");

if (tempDir == null) {
   throw new UnavailableException("Configuração errada");
}
```

3.2 Resources

O método *getResourceAsStream()* permite que se localize e se carregue qualquer arquivo no contexto sem que seja necessário saber seu caminho completo. Resources podem ser arquivos de configuração (XML, properties), imagens, ou quaisquer outros arquivos necessários que estejam armazenados dentro do WAR.

O exemplo abaixo mostra como ler um arquivo XML que está dentro da pasta WEB-INF:

```
ServletContext ctx = getServletContext();
String arquivo = "/WEB-INF/usuarios.xml";
InputStream stream = ctx.getResourceAsStream(arquivo);
InputStreamReader reader =
                new InputStreamReader(stream);
BufferedReader in = new BufferedReader(reader);
String linha = "";
while ( (linha = in.readLine()) != null) {
   // Faz alguma coisa com linha de texto lida
}
```

3.3 Atributos no escopo de contexto

Objetos podem ser armazenados no contexto. Isto permite que sejam compartilhados entre servlets. O exemplo abaixo mostra como gravar um objeto no contexto:

```
String[] vetor = {"um", "dois", "tres"};
ServletContext ctx = getServletContext();
ctx.setAttribute("dados", vetor);
```

Para recuperar a chave deve ser usada em getAttribute():

```
ServletContext ctx = getServletContext();
String[] dados = (String[])ctx.getAttribute("dados");
```

3.4 Listeners

Não existem métodos *init()* ou *destroy()* globais para realizar operações de inicialização e destruição de um contexto. A forma de controlar o ciclo de vida global para um contexto é através da implementação de um *ServletContextListener*, que é uma interface com dois métodos:

- `public void contextInitialized(ServletContextEvent e)`
- `public void contextDestroyed(ServletContextEvent e)`

Eles são chamados respectivamente depois que um contexto é criado e antes que ele seja destruído. Para isto é preciso ou anotar a classe da implementação com *@WebListener*:

```
@WebListener
public class OuvinteDeContexto implements ServletContextListener {...}
```

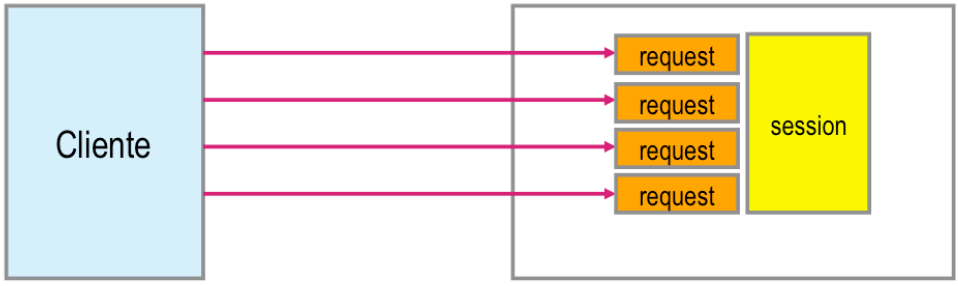
ou registrá-lo no web.xml usando o elemento *<listener>*

```
<listener>
   <listener-class>ex01.OuvinteDeContexto</listener-class>
</listener>
```

O objeto *ServletContextEvent*, recebido em ambos os métodos, possui um método *getServletContext()* que permite obter o contexto associado.

4 Sessões

Uma sessão HTTP representa o tempo que um cliente acessa uma pagina ou domínio. Uma sessão é iniciada com uma requisição, é única para cada cliente e persiste através de várias requisições.



Como o HTTP não mantém estado de sessão de forma nativa, as aplicações Web precisam cuidar de mantê-lo quando necessário. Soluções padrão incluem cookies, re-escrita de URLs, etc. A especificação WebServlet permite mais de uma solução, mas implementa por default (e recomenda) uma solução baseada em cookies.

Sessões são representadas por objetos *HttpSession* e são obtidas a partir de uma requisição. O objeto pode ser usado para armazenar objetos que serão recuperados posteriormente pelo mesmo cliente.

Para criar uma nova sessão ou para recuperar uma referência para uma sessão existente usa-se o mesmo método:

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

Para saber se uma sessão é nova, use o método *isNew()*:

```
BusinessObject myObject = null;
if (session.isNew()) {
    myObject = new BusinessObject();
    session.setAttribute("obj", myObject);
}
myObject = (BusinessObject)session.getAttribute("obj");
```

4.1 Atributos de sessão

Dois métodos da classe *HttpSession* permitem o compartilhamento de objetos na sessão.

- void **setAttribute**("nome", objeto);
- Object **getAttribute**("nome");

Por exemplo, o código abaixo irá gravar um atributo (um array) em uma requisição:

```
String[] vetor = {"um", "dois", "tres"};
```

```
HttpSession session = request.getSession();
session.setAttribute("dados", vetor);
```

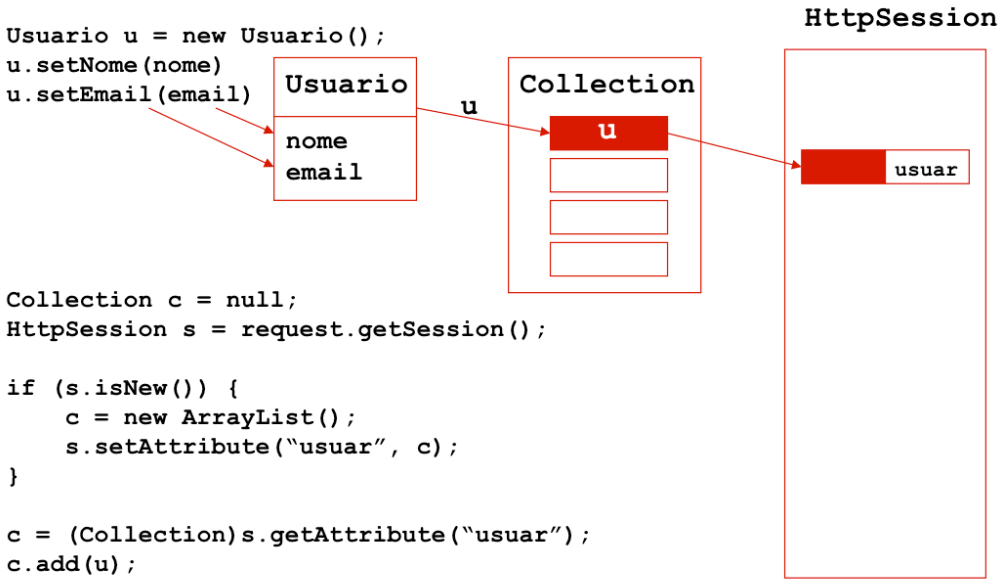
Em outro servlet que ainda executa na mesma sessão, pode-se obter o array usando:

```
HttpSession session = request.getSession();
String[] dados = (String[])session.getAttribute("dados");
```

Como a sessão pode persistir além do tempo de uma requisição, é possível que a persistência de alguns objetos não sejam desejáveis. Eles podem ser removidos usando:

```
removeAttribute("nome")
```

O exemplo abaixo ilustra o uso de sessões para guardar uma coleção de objetos. Um uso típico seria um carrinho de compras, onde cada item da coleção é um item selecionado pelo cliente. Para esvaziar o carrinho ele pode esvaziar a coleção ou invalidar a sessão (que remove não apenas a coleção da sessão atual, mas quaisquer outros objetos que tenham sido armazenados).




```
Collection c = null;
HttpSession s = request.getSession();

if (s.isNew()) {
    c = new ArrayList();
    s.setAttribute("usuar", c);
}

c = (Collection)s.getAttribute("usuar");
c.add(u);
```

4.2 Validade e duração de uma sessão

Não há como saber que cliente não precisa mais da sessão. Normalmente estabelece-se um tempo de validade contado pela inatividade do cliente e assim definir um timeout em minutos para a duração de uma sessão desde a última requisição. Há quatro métodos em *HttpSession* para lidar com duração de uma sessão:

- void **setMaxInactiveInterval**(int) – define novo valor para timeout
- int **getMaxInactiveInterval**() – recupera valor de timeout
- long **getLastAccessedTime**() – recupera data em UTC
- long **getCreationTime**() – recupera data em UTC

Timeout default pode ser definido no web.xml para todas as sessões. Por exemplo, para que dure ate 15 minutos:

```
<session-config>
   <session-timeout>15</session-timeout>
</session-config>
```

Para destruir imediatamente uma sessão usa-se:

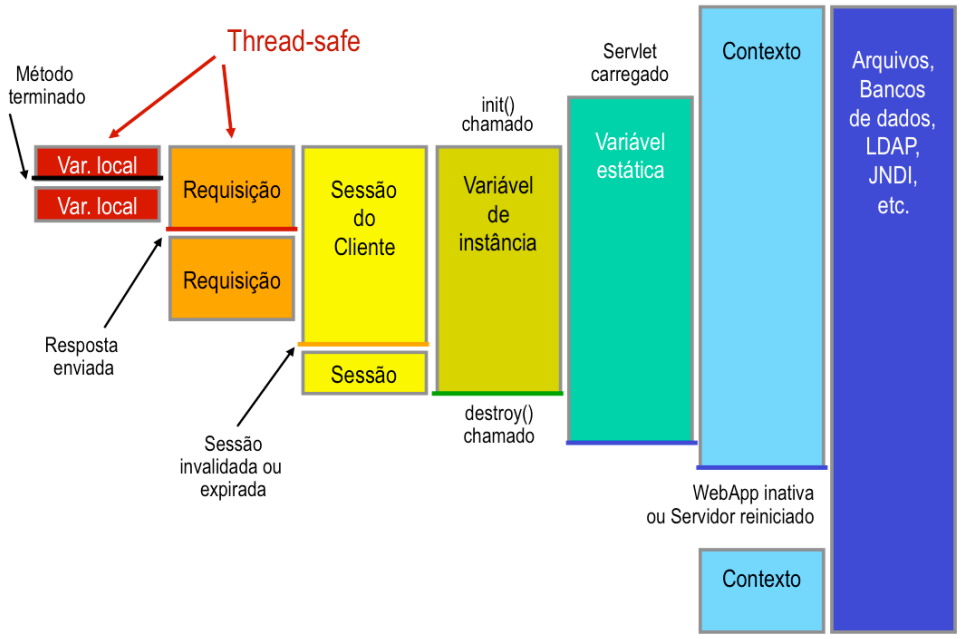
```
session.invalidate();
```

5 Escopos

Servlets podem compartilhar informações de várias maneiras

- Usando meios persistentes (bancos de dados, arquivos, etc)
- Usando objetos na memória por escopo (requisição, sessão, contexto)
- Usando variáveis estáticas ou de instância

Esses escopos estão ilustrados na figura abaixo:



Devido à natureza dos servlets e sua forma de execução, não é recomendado o compartilhamento usando variáveis estáticas e de instancia. A forma recomendada consiste em usar os métodos get/setAttribute contidos nos três objetos de escopo:

- javax.servlet.**ServletContext** – que representa o contexto da aplicação e existe enquanto a aplicação estiver executando.
- javax.servlet.http.**HttpSession** – que representa o contexto da sessão do cliente e existe enquanto o cliente estiver conectado.

- javax.servlet.**ServletRequest** – que representa o contexto da requisição e existe enquanto o método service() nao terminar.

Portanto, para gravar dados em um objeto de persistência na memória, deve-se usar:

```
objeto.setAttribute("nome", dados);
```

E para recuperar ou remover os dados:

```
Object dados = objeto.getAttribute("nome");
objeto.removeAttribute("nome");
```

6 Cookies

Sessões geralmente são implementados com cookies, mas são cookies gravados na memória que deixam de existir quando o browser e fechado. A especificação de cookies define outro tipo de cookie que é gravado em disco, no cliente, e persiste por tempo indeterminado. Eles são usados principalmente para definir "preferências" (nem sempre o usuário preferiu receber um cookie desse tipo) que irão durar além do tempo da sessão.

O cookie é sempre gerado no cliente. O servidor cria um cabeçalho HTTP que instrui o browser a criar um arquivo guardando as informações do cookie. Para criar um cookie persistente é preciso:

- Criar um novo objeto Cookie
- Definir a duração do cookie com o método setMaxAge()
- Definir outros métodos se necessário
- Adicionar o cookie à resposta

Por exemplo, para definir um cookie que contenha o nome do usuário recebido como parâmetro na requisição, com duração de 60 dias:

```
String nome = request.getParameter("nome");
Cookie c = new Cookie("usuario", nome);
c.setMaxAge(1000 * 24 * 3600 * 60); // 60 dias
```

O cookie é adicionado à resposta:

```
response.addCookie(c);
```

Para recuperar o cookie na requisição, usa-se

```
Cookie[] cookies = request.getCookies();
```

Um HttpCookie tem name e value. Para extrair cookie para um objeto local pode-se usar:

```
for (int i = 0; i < cookies.length; i++) {
   if (cookies[i].getName().equals("nome") {
        usuario = cookies[i].getValue();
```

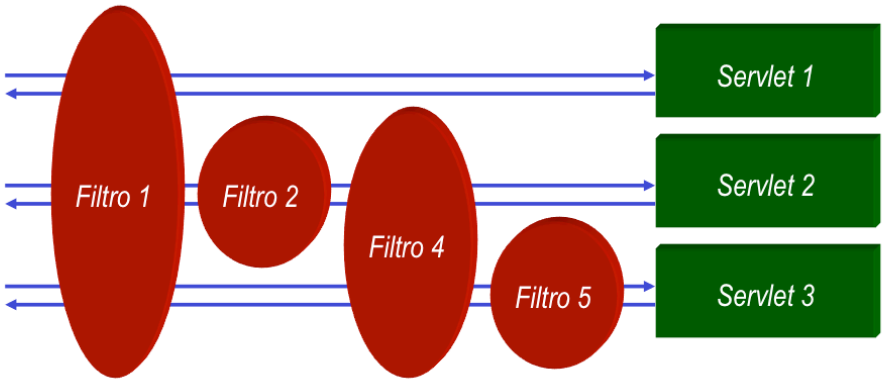
```
    }
}
```

7 Filtros

Filtros não são abordados neste tutorial. Esta seção contém apenas uma visão geral sobre o assunto.

Um filtro é um componente Web que reside no servidor e intercepta as requisições e respostas no seu caminho até um servlet, e de volta ao cliente. Sua existência é ignorada por ambos. É totalmente transparente tanto para o cliente quanto para o servlet.

Filtros podem ser concatenados em uma corrente. Neste cenário, as requisições são interceptadas em uma ordem e as respostas em ordem inversa. Filtros são independentes dos servlets e podem ser reutilizados.



Um filtro pode realizar diversas transformações, tanto na resposta como na requisição antes de passar esses objetos adiante (se o fizer). Aplicações típicas envolvem

- Tomada de decisões: podem decidir se repassam uma requisição adiante, se redirecionam ou se enviam uma resposta interrompendo o caminho normal da requisição
- Tratamento de requisições e respostas: podem empacotar uma requisição (ou resposta) em outra, alterando os dados e o conteúdo dos cabeçalhos

Aplicações típicas envolvem autenticação, tradução de textos, conversão de caracteres, MIME types, tokenizing, conversão de imagens, compressão e decompressão e criptografia.

Quando o container recebe uma requisição, ele verifica se há um filtro associado ao recurso solicitado. Se houver, a requisição é roteada ao filtro

O filtro, então, pode gerar sua própria resposta para o cliente, repassar a requisição, modificada ou não, ao próximo filtro da corrente, se houver, ou ao recurso final, se ele for o último filtro, ou rotear a requisição para outro recurso. Na volta para o cliente, a resposta passa pelo mesmo conjunto de filtros em ordem inversa.

Em aplicações JSF, que são interceptadas por um único servlet, o filtro, que é mapeado a um servlet, afeta todas as requisições, portanto é necessário que ele use outras informações da URL (subcaminhos do contexto) para decidir se deve ou não processar a requisição ou resposta.

Filtros sao criados implementando a interface *javax.servlet.Filter* e mapeados a servlets através de anotações *@WebFilter* ou via *web.xml*.

8 JSP (JavaServer Pages) e Taglibs

JSP e Taglibs não serão abordadas neste tutorial. Esta seção contém apenas uma visão geral sobre o assunto.

JSP é uma tecnologia padrão, baseada em templates para servlets. Em JSF 1.x JSP era usada para construir templates de páginas dinâmicas, mas desde JSF 2.x o seu uso não e mais recomendado e deve ser substituído por XHTML. Pode ainda ser usada para construir páginas simples que não precisam da arquitetura de componentes proporcionada pelo JSF. Mas uma arquitetura MVC é recomendada, evitando usar scriplets <%...%> que podem ser embutidos em JSP.

Uma página JSP equivale a e efetivamente é um servlet que gera o HTML da página. A forma mais simples de criar documentos JSP, é

1. Mudar a extensão de um arquivo HTML para .jsp
2. Colocar o documento em um servidor que suporte JSP

Fazendo isto, a página será transformada em um servlet. A compilação é feita no primeiro acesso. Nos acessos seguintes, a requisição é redirecionada ao servlet que foi gerado a partir da página.

Transformado em um JSP, um arquivo HTML pode conter blocos de código (scriptlets): <% ... %> e expressões <%= ... %> que são os elementos mais frequentemente usados (mas que hoje são desaconselhados em favor de JSTL e Expression Language, usados em arquiteturas MVC:

```
<p>Texto repetido:
```

```
<% for (int i = 0; i < 10; i++) { %>
  <p>Esta é a linha <%=i %>
<% }%>
```

8.1 JSTL

JSTL é uma biblioteca de tags que permite que o autor de páginas controle a geração de código HTML usando laços e blocos condicionais, transformações e comunicação com objetos externos, sem a necessidade de usar scripts. JSTL (junto com EL) são a forma recomendada de usar JSP.

A adoção de JSTL estimula a separação da apresentação e lógica e o investimento em soluções MVC.

Para usar JSTL em uma página JSP é preciso declarar taglib em cada página (não é necessário declarar em web.xml ou incluir TLDs em WEB-INF, como ocorre com outras bibliotecas, porque a distribuição faz parte do container Java EE):

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head><title>Simple Example</title></head>
  <body>
  <c:set var="browser"
         value="${header['User-Agent']}"/>
  <c:out value="${browser}"/>
  </body>
</html>
```

8.2 EL (Expression Language)

Expression Language (EL) e uma linguagem declarativa criada para viabilizar a comunicação entre views (páginas JSP) e controllers (beans e outros objetos Java) em aplicações Web que usam arquitetura MVC. Usando EL é possível embutir expressões simples dentro de delimitadores ${...} diretamente na pagina ou em atributos de tags JSTL.

Através de EL e JSTL é possível eliminar completamente os scriptlets das páginas JSP. Com EL e possível ter acesso a objetos implícitos e beans. Por exemplo, em vez de usar, para ler o atributo de uma requisição, o scriptlet

```
<% request.getAttribute("nome")        %>
```

pode-se usar simplesmente:

```
${nome}
```

E em vez de

```
<% bean.getPessoa().getNome() %>
```

usa-se

```
${bean.pessoa.nome}.
```

EL suporta também expressões simples com operadores aritméticos, relacionais e binários:

```
${ !(empty biblioteca.livros) }
${ livro.preco * pedido.quantidade }
```

Tambem converte tipos automaticamente e suporta valores default.

```
<tag item="${request.valorNumerico}" />
<tag value="${abc.def}" default="todos" />
```

Exemplos usando JSTL e EL:

```
<c:out value='${param.emailAddress}'/>
<c:if test='${not empty param.email}'>
    ...
</c:if>
```

9 Referências

[1] Shing Wai Chan, Rajiv Mordani. Java™ Servlet Specification. Version 3.1. Oracle. 2013. http://download.oracle.com/otndocs/jcp/servlet-3_1-fr-eval-spec/index.html

[2] Eric Jendrock et al. *The Java EE 7 Tutorial*. Oracle. 2014. https://docs.oracle.com/javaee/7/JEETT.pdf

[3] Linda deMichiel and Bill Shannon. *JSR 342. The Java Enterprise Edition Specification. Version 7*. Oracle, 2013. http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-spec/JavaEE_Platform_Spec.pdf

[4] Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014.