Resumo – Sun Certified Web Component Developer (SCWCD) 1.4

Daniel F. Martins – http://danielfmartins.com/

Este resumo é recomendado apenas àqueles que já leram – ou que estão terminando de ler – livros especializados para o exame CX-310-081 (SCWCD 1.4), como o HeadFirst: Servlets & JSP. As referências utlizadas para a escrita deste resumo estão na última página.

Se você for utilizar este resumo para se preparar para o exame, o faça por sua conta e risco. Sinta-se à vontade para reportar problemas através do e-mail contato@danielfmartins.com.

Capítulo 2

Serviços fornecidos por um Contêiner web:

- Suporte a comunicações;
- Gerenciamento de ciclo de vida;
- Suporte a multi-threading;
- Segurança declarativa;
- Suporte a JSPs.

Capítulo 4

Ciclo de vida de um Servlet:

- **Não existe**: O Contêiner carrega a classe, a instancia (através do construtor sem argumentos), chama o método **init()**. **Atenção**: se o construtor padrão foi definido, prestar atenção ao seu conteúdo (só se pode invocar métodos relacionados ao Servlet após sua inicialização);
- **Inicializado**: Servlet pronto para atender requisições dos usuários, o que é feito através de chamadas ao método **service()**. Quando o Contêiner é fechado (ou a aplicação é removida), o método **destroy()** é invocado nos Servlets, fazendo-os voltar ao estado de não-existência.

A inicialização do Servlet pode ocorrer quando a aplicação é implantada ou quando o Servlet é realmente necessário (como numa requisição de um cliente)... a escolha entre um e outro pode variar de acordo com a implementação do Contêiner.

Durante a inicialização, as exceções **ServletException** ou **UnavailableException** podem ser lançadas. Caso isso aconteça, o Contêiner libera o Servlet (não o coloca na lista dos Servlets ativos). O método **destroy() NÃO** é invocado.

UnavailableException

Hierarquia:

- Exception ← ServletException ← UnavailableException

Construtores:

- UnavailableException(int sec, Servlet servlet, String msg). Deprecated;
- **UnavailableException(Servlet servlet, String msg)**. Deprecated;
- **UnavailableException(String msg);**
- UnavailableException(String msg, int sec);

A exceção **UnavailableException** pode ser lançada no método **service()** do Servlet, indicando que o Servlet está indisponível temporariamente ou permanentemente (construtores em negrito). Se for permanente, o Contêiner remove o Servlet da lista de Servlets ativos, chama o método **destroy()** e libera a instância. Um erro HTTP 404 é retornado. Dependendo da implementação, a exceção **UnavailableException** pode ser tratada **sempre** como permanente (ignorando caso a indisponibilidade seja temporária). **Se** isso acontecer, o Contêiner deve retornar um erro HTTP 503 até o período especificado na exception termine (caso seja especificado um período de *timeout*).

Voltando aos Servlets...

Hierarquia: **Servlet ← GenericServlet ← HttpServlet ← YOUR_SERVLET_HERE**

- **Servlet**: interface. Declara métodos de ciclo de vida; **getServletConfig()**, **getServletInfo()**;
- **GenericServlet**: classe abstrata. Implementa a interface **Servlet** e adiciona métodos pra controle de parâmetros de inicialização/atributos, logging, obtenção do **servletContext**... adiciona um método **init()** sem parâmetros também (que, por sua vez, chama o **init(ServletConfig)**). Adiciona outros métodos, como **service(ServletRequest, ServletResponse)** e **getServletName()**;
- **HttpServlet**: adiciona a coisa do HTTP, com métodos de serviço **do*()** (menos **doConnect()**, que não é implementada pela especificação). Adiciona um método **getLastModified()** e o

service(HttpServletRequest, HttpServletResponse);
- **YOUR_SERVLET_HERE**: ahn.. você entendeu.

Atenção: considerando que a aplicação **NÃO** está distribuída em várias JVMs, existe apenas **UMA** instância de cada Servlet. **Apenas UMA para cada JVM**. O que o Contêiner faz para tratar diversas requisições **à mesma instância** do Servlet é criar várias threads onde cada thread chama o método **service()** do Servlet em questão. **SE** a aplicação estiver distribuída em mais de uma JVM, existirá **uma instância de cada Servlet em cada JVM**. A exceção para esta regra é vai para os Servlets que implementam **SingleThreadModel** (mais informações adiante).

ServletConfig e **ServletContext**

São coisas **DIFERENTES**. Um objeto **ServletConfig** representa as configurações de um determinado Servlet (cada Servlet tem seu próprio objeto ServletConfig). Este objeto **ServletConfig**, por sua vez, **PODE** ser utilizado para acessar o objeto **ServletContext**. **Caso** a aplicação seja **distribuída** em vários servidores, existirá um objeto ServletConfig **para cada JVM**.

Já o objeto **ServletContext** representa as configurações da aplicação web como um todo (existe apenas um objeto ServletContext para toda a aplicação). **Aqui também**, a aplicação seja **distribuída** em vários servidores, existirá um objeto ServletContext **para cada JVM**.

Request e **Response**

Hierarquia: **ServletRequest ← HttpServletRequest** . **ServletResponse ← HttpServletResponse**

Métodos Principais

ServletRequest
- *Principais*: getAttribute(String), getAttributeNames(), getParameter(String), getParameterMap(), getParameterNames(), getParameterValues(String), removeAttribute(String), setAttribute(String, Object). **Lembre-se**: métodos relacionados a **atributos** e **parâmetros**;
- *Outros*: getInputStream(), getReader(), getRemoteAddr(), getRemoteHost(), getServetName(), getServerPort(), isSecure(). Não precisa saber tudo, apenas **saiba que estes métodos existem**;

HttpServletRequest
- *Principais*: getCookies(), getHeader(String), getIntHeader(String), getHeaderNames(), getHeaders(String), getMethod(), getSession(), getSession(boolean), getUserPrincipal(), isUserInRole(String). **Lembre-se**: HTTP (cookies, session, header etc);
- *Outros*: getQueryString(), getRemoteUser(), getServletPath(). Não precisa saber tudo, apenas **saiba que estes métodos existem**;

ServletResponse
- *Principais*: getOutputStream(), getWriter(), setContentType(String). **Lembre-se**: Envio da resposta, nada mais (streams de saída, tipo de conteúdo);
- *Outros*: isCommited(), reset(), resetBuffer(), setContentLength(int). Métodos relacionados ao estado da response... apenas **saiba que tais métodos existem**;

HttpServletResponse
- *Principais*: addCookie(Cookie), addHeader(String, String), addIntHeader(String, int),

encodeRedirectURL(String), encodeURL(String), sendError(int), sendRedirect(String), setHeader(String, String), setIntHeader(String, int). **Lembre-se**: HTTP (header, cookies, error / redirect, URLs);

- *Outros*: addDateHeader(String, Date), containsHeader(String), sendError(int, String), setDateHeader(String, Date), setStatus(int). Métodos parecidos com os mostrados anteriormente, apenas **saiba que tais métodos existem**;

Servlets e o Protocolo HTTP

Métodos HTTP: **GET**, POST, HEAD, TRACE, PUT, DELETE, OPTIONS e ~~CONNECT~~ (não é implementado pela especificação). Nos métodos em negrito, a idempotência é **obrigatória** a nível do protocolo HTTP (no entanto, método GET **pode não ser** idempotente, dependendo da implementação).

Idempotência: Uma funcionalidade pode ser definida como idempotente quando ela puder ser invocada várias vezes, sem que nenhum efeito colateral ocorra.

Existe uma grande diferença entre métodos HTTP e métodos de serviço. Um método HTTP GET, por exemplo, é **SEMPRE** considerado idempotente, enquanto o método de serviço **doGet()**, por sua vez, **pode ser idempotente ou não**, dependendo de como o método foi implementado. (repeti só para reforçar). *:-)*

Diferença entre o GET e POST: no GET, a request query aparece na barra de navegação do browser; no POST, os dados vão direto no body do request, não aparecem na barra de navegação. Além disso, requests GET são possíveis de se fazer "bookmarking". Além disso, uma *query string* **GET** é limitada a 255 caracteres.

Formulários HTML

Em uma tag HTML <form>, o método-padrão é GET, caso o atributo method **não seja fornecido**! O método GET também é usado em links (*tag* <a>).

Um determinado parâmetro pode ter um ou vários valores:

- Para obter um valor único (String): **request.getParameter("xxx")**;
- Para obter todos os valores (String[]): **request.getParameterValues("xxx")**. **Cuidado**: não é getParameters("xxx")!!

ServerPort, **LocalPort**, **RemotePort** (relacionados a métodos de ServletRequest):

- *ServerPort*: A porta para qual a requisição foi enviada inicialmente;
- *LocalPort*: A porta na qual a requisição foi efetivamente tratada;
- *RemotePort*: O lado oposto do lado que está executando o código. Para o servidor, o *RemotePort* é a porta do cliente, enquanto que, para o cliente, o *RemotePort* é a porta do servidor.

setContentType() e **getWriter()** / **getOutputStream()**

Se os métodos **getWriter()** ou **getOuputStream()** forem invocados, certifique-se de chamar o método **setContentType()** antes de obter uma referência aos streams. (não causa erros, mas o content-type **não é usado na response!**).

Não chamar os métodos **getWriter()** ou **getOutputStream()** juntos ou um dos dois mais de uma vez.

Isso faz com que uma exceção **IllegalStateException** seja lançada.

Use **getWriter()** para enviar streams de caracteres; use **getOutputStream()** para enviar streams binários.

Exemplos:
- output.write(byte(s));
- writer.println(caractere / string);

Header
- **setHeader("xx", "xx")**: Se um header já existir, o método o substitui. Se não existir, o header é adicionado;
- **addHeader("xx", "xx")**: Se o header já existir, adiciona o novo valor ao(s) valor(es) já existente(s). Se o header não existir, adiciona um novo header com o valor dado;
- **setIntHeader("xx", 10)**: Igual ao **setHeader()**, mas para headers cujos valores são inteiros.
- setDateHeader("xxx", new Date()): Igual ao setIntHeader(), mas para headers cujos valores são datas.

Atenção:
- **String request.getHeader(String)** retorna o valor da header caso exista, **null** caso não exista ou o **primeiro valor** caso a header tenha vários valores. O parâmetro é **CASE-INSENSITIVE** (mas não se esqueça de que os **parameters** e **attributes SÃO CASE-SENSITIVE!!!**);
- **int request.getIntHeader(String)** retorna o valor da header caso exista, **-1** caso não exista ou o **primeiro valor** caso a header tenha vários valores;

Dica: métodos de *header* seguem a seguinte nomenclatura:
- [get | set | add | remove]**Xxx**Header("xxx", value), onde:
  - Xxx: pode ser branco, **Date** e **Int**.
  - Ex: setDateHeader(), getIntHeader(), addHeader() etc.

sendRedirect()
- *path relativo*: **sendRedirect("foo/bar.html")**. Se o diretório atual do recurso que recebeu a request é "/appcontext/test", então a URL resultante será "/appcontext/test/foo/bar.html";
- *path absoluto* (ou relativo ao Contêiner): **sendRedirect("/foo/bar.html")**. Independente do diretório atual, a URL resultante será "/foo/bar.html" (sim, **redirecionou** a request **para uma outra aplicação** cujo context-path é "/foo").

NÃO chame **sendRedirect()** (**NEM NADA** que escreva algo no response) a menos que a mesma **NÃO** tenha sido comitada (através de uma chamada ao método **flush()** dos objetos de stream de resposta ou **flushBuffer()** no objeto **response**).

Fique atento com respostas muito longas, pois elas fazem com que, em um determinado momento, o buffer de resposta estoure e a resposta seja comitada independente de se ter comitado explicitamente.

Capítulo 5

Init parameters

DD:
```
<context-param> <!-- parâmetros de inicialização do context -->
    <param-name>ctxparam</param-name>
    <param-value>1234</param-value>
</context-param>

<servlet>
   <servlet-name>MyServlet</servlet-name>
   <servlet-class>com.foo.MyServlet</servlet-class>
   <init-param>
      <param-name>init</param-name>
      <param-value>12345</param-value>
   </init-param>
</servlet>
```

No Servlet:
getServletConfig().getInitParameter("init"). Retorna **String**. // init param de um Servlet
getServletContext().getInitParameter("ctxparam"). Retorna **String**. // init param da aplicação
getServletConfig().getServletContext().getInitParameter("ctxparam") // igual acima
getServlet*().getInitParameterNames(). Retorna **Enumeration**, com os nomes dos parâmetros.

NÃO EXISTEM MÉTODOS SET PARA PARÂMETROS DE INICIALIZAÇÃO! (3x)

Attributes (lembre-se, são **DIFERENTES DOS PARÂMETROS DE INICIALIZAÇÃO**)

request.setAttribute("xxx", obj); // escopo de request, aceita **objetos**; init param só aceitam **strings**

Nomes de atributos que começam com "java.", "javax." e "sun." são reservados. Usar atributos com esses prefixos **não causam erros**, mas isso é desaconselhado, pois **podem** sobrescrever valores importantes setados pelo Contêiner.

Interface **ServletContext**
- Métodos para pegar/remover/setar atributos e pegar parâmetros de inicialização;
- Outros métodos: getRequestDispatcher(String), log(String), getMajorVersion(), getMinorVersion(), getServerInfo(), getRealPath(String), getResourceAsStream(String), getResource(String), getResourcePaths(String). Apenas **saiba que esses métodos existem**;

Métodos **URL getResource(String)** e **InputStream getResourceAsStream(String)** podem ler arquivos estáticos de arquivos WAR, arquivos locais e remotos, entre outras coisas. Lembre-se também de que tais métodos **não podem** ser utilizados para ler arquivos internos a bibliotecas (arquivos .jar dentro do diretório /WEB-INF/lib).

Thread Safety

Devemos tomar cuidado com thread-safety com os contextos **application** e **session**, pois eles **NÃO SÃO thread-safe**. Sempre sincronizar o acesso a tais objetos (**getServletContext()** e/ou **getSession()**) quando houver a possibilidade de problemas de concorrência. Atenção também aos objetos **HttpRequest** e **HttpResponse** recebidos no método de serviço, pois eles **em si NÃO SÃO** thread-safe (embora as variáveis locais ao método **service()** e os atributos de escopo de request **SEJAM**).

Muito cuidado com questões que perguntam se sincronizar o método de serviço (**doGet()**, **doPost()** etc) – ou implementar **SingleThreadModel –** resolvem o problema. Isso apenas garante que haverá só uma thread acessando este Servlet/service() num dado momento. Isso não muda o fato de que atributos em escopo session e application sejam modificados por outros Servlets.

INTERFACE **SingleThreadModel**

Não é recomendável implementar esta interface pois, para cada Servlet que a estende, existirá apenas uma thread em execução num dado momento, o que é muito ruim em termos de performance. Entretanto, uma particularidade importante é que o Contêiner pode querer criar pools de Servlets deste tipo para tentar evitar esses problemas de performance, portanto, quando se trata de Servlets SingleThreadModel, **PODE** haver mais de uma instância por Servlet! Isso inclusive é previsto pela especificação.

RequestDispatcher

O objeto **RequestDispatcher** deve ser utilizado para repassar requisições de modo a manter os atributos setados pelos Servlets/JSPs que trataram a requisição antes de passá-la adiante. **Lembre-se**: um **sendRedirect() NÃO** mantém os atributos (de fato, é como se fosse uma nova requisição feita pelo usuário).

Os métodos de **RequestDispatcher** são **include(ServletRequest, ServletResponse)** e **forward(***idem***)**

Podemos obter um RequestDispatcher das seguintes formas:

- **request.getRequestDispatcher("foo.html")**: o "foo.html" é relativo ao path para onde foi a request;
- **getServletContext().getRequestDispatcher("/foo.html")**: o "foo.html" é relativo ao diretório raíz da aplicação. **A BARRA É OBRIGATÓRIA**;
- A interface ServletContext possui um método **getNamedDispatcher(String)**, onde passamos um nome de Servlet ou JSP (configurados via DD). Provavelmente não cai no exame, mas é bom saber que **tal método existe**.

O uso de query parameters no parâmetro de **getRequestDispatcher()** (ex: test.jsp?a=2&b=3) **SÃO PERMITIDOS**. Não se engane com questões que dizem o contrário.

Método **include()**

Qualquer arquivo JSP/Servlet incluído através do **servletDispatcher.include() NÃO PODE** manipular informações de header (afinal ele está sendo incluído por outro arquivo, então esse outro arquivo é quem deve fazer tais manipulações). Um JSP/Servlet incluído também **PODE** comitar a response através do método **response.flushBuffer()** (ou chamando **flush()** no output stream). **Atenção**: comitar o fluxo da response geralmente leva a erros!!

Ao incluir um arquivo (através do método **getRequestDispatcher()**, não de **getNamedDispatcher(String servletName)**), alguns **atributos** são disponibilizados pelo Contêiner:

- javax.servlet.include.request_uri. Também pode ser obtido com **request.getRequestURI()**;
- javax.servlet.include.context_path. Também pode ser obtido com **request.getContextPath()**;
- javax.servlet.include.servlet_path. Também pode ser obtido com **request.getServletPath()**;
- javax.servlet.include.path_info. Também pode ser obtido com **request.getPathInfo()**;
- javax.servlet.include.query_string. Também pode ser obtido com **request.getQueryString()**.

Método **forward()**

Ao fazer forward para um arquivo (através do método **getRequestDispatcher()**, não de **getNamedDispatcher()**), alguns **atributos** são disponibilizados pelo Contêiner:

- javax.servlet.forward.request_uri. Ver método **include()**;
- javax.servlet.forward.context_path. Ver método **include()**;
- javax.servlet.forward.servlet_path. Ver método **include()**;
- javax.servlet.forward.path_info. Ver método **include()**;
- javax.servlet.forward.query_string. Ver método **include()**.

Como dito anteriormente, recursos acessados através de **forwards** e **includes** através do método **getNamedDispatcher() NÃO** colocam os atributos mostrados. Caso o método utilizado seja o **getRequestDispatcher()**, os atributos são criados e podem ser acessados normalmente através de chamadas ao método **getAttribute()**.

Listeners

- *Session*
  - **HttpSessionListener**: sessionCreated(**HttpSessionEvent**), sessionDestroyed(*idem*);
  - **HttpSessionBindingListener**: valueBound(**HttpSessionBindingEvent**), valueUnbound(*idem*);
  - **HttpSessionAttributeListener**: attributeAdded(**HttpSessionBindingEvent**), attributeRemoved(*idem*), attributeReplaced(*idem*);
  - **HttpSessionActivationListener**: sessionDidActivate(**HttpSessionEvent**), sessionWillPassivate(*idem*);
- *Request*
  - **ServletRequestListener**: requestInitialized(**ServletRequestEvent**), requestDestroyed(*idem*);
  - **ServletRequestAttributeListener**: attributeAdded(**ServletRequestAttributeEvent**), attributeRemoved(*idem*), attributeReplaced(*idem*);
- *Context*
  - **ServletContextListener**: contextInitialized(**ServletContextEvent**), contextDestroyed(*idem*).
  - **ServletContextAttributeListener**: attributeAdded(**ServletContextAttributeEvent**), attributeRemoved(*idem*), attributeReplaced(*idem*);

Todos os tipos, com exceção do **HttpSessionBindingListener** e **HttpSessionActivationListener**, **PRECISAM** ser declarados no DD:

```
<listener>
   <listener-class>foo.bar.MyListenerClass</listener-class>
</listener>
```

De acordo com a especificação, **A ORDEM NA QUAL OS LISTENERS FORAM REGISTRADOS IMPORTA.**

Lembre-se: Em listeners de contexto, eles são executados na ordem quando o contexto é iniciado. No entanto, quando o contexto é destruído, eles são executados na ordem inversa (simulando uma stack ou algo do tipo).

Classes de Atributos e Listeners

Classes de atributos podem implementar interfaces listeners para que tais atributos sejam notificados quando o evento é disparado pelo Contêiner. Mas **cuidado**! De todas as interfaces listener, apenas as interfaces **HttpSessionActivationListener** e **HttpSessionBindingListener** podem ser usadas com sucesso nesses casos. **Não há problemas** caso uma classe de atributo implemente outras interfaces listener, o que acontecerá é que **o atributo não será notificado** quando o evento relacionado for disparado.

Distributabilidade

No arquivo web.xml, podemos indicar se a aplicação em questão deve trabalhar de forma distribuída através da declaração do elemento distributable:

<distributable/>

Em um ambiente distribuído, o Contêiner não é obrigado a replicar eventos dos tipos ServletContextEvents e ServletContextAttributeEvents para os listeners que estão em JVMs diferentes. Em outras palavras: **sua aplicação não pode depender dessas notificações**! Lembre-se: o mesmo vale para os eventos de criação/destruição da sessão em si (interface HttpSessionListener) e modificação da lista de atributos de sessão (interface HttpSessionAttributeListener).

Uma sessão apenas “vive” em uma JVM num dado momento. Por isso, todos os atributos de sessão que implementam HttpSessionActivationListener recebem notificações que indicam se a sessão foi apassivada (passivated) ou ativada (activated).

Não cai no exame, mas é bom saber

Isso costuma variar de acordo com o Contêiner utilizado, mas, normalmente, um objeto session cujos atributos foram modificados não é replicado aos outros nós do cluster se o atributo não for setado novamente após a modificação.

Por exemplo:

```
HttpSession session = request.getSession();
List names = (List) session.getAttribute(“names”);
names.add(“new name”);

// seta novamente o objeto na session após modifica-lo internamente
session.setAttribute(“names”, names);
```

Requisitos de uma aplicação distribuída

Pode cair no exame questões perguntando se a aplicação pode ser distribuída em vários servidores. Abaixo seguem algumas dicas:

- ServletContext: uma instância por JVM. Por isso, não dependa do contexto caso a aplicação precise operar de forma distribuída;
  - ServletContextListener: Os eventos ocorridos em um ServletContext (que por sua vez se encontra em uma JVM qualquer) não são propagados para outros ServletContextListeners residentes em outras JVMs, portanto, não dependa dessas notificações;
  - ServletContextAttributeListener: Ver item anterior;
- HttpSession: uma session migra entre diferentes JVMs e, portanto, nunca está em várias JVMs em um mesmo momento.;
  - HttpSessionListener: Os eventos de criação e destrução da session podem ocorrer em diferentes JVMs;
  - HttpSessionAttributeListener: Os eventos de manipulação dos atributos da session podem ocorrer em diferentes JVMs;
  - HttpSessionActivationListener: os atributos que precisam ser notificados da migração da session devem implementar esta interface.

Capítulo 6

Sessions

O objeto **HttpSession É SEMPRE** obtido através do objeto request do método de serviço (além de tal objeto também poder ser obtido através de objetos **HttpSessionEvent** recebidos em métodos de callback dos session listeners).

Uma chamada ao método **getSession()** ou **getSession(true)** faz com que a session seja criada caso ainda não exista. Quando a response for enviada, ela irá acompanhada de um COOKIE **jsessionid (**o nome é esse de acordo com a especificação.

Os clientes **NÃO SÃO OBRIGADOS** a terem suporte a COOKIES ativado! Por isso, utilize URL Rewriting, onde todos os links gerados no JSP/Servlet devem possuir o **jsessionid** no endereço. Se na próxima request, o COOKIE retornar, quer dizer que o suporte a COOKIES está ativado, e o recurso de URL Rewriting não precisará mais ser utilizado durante essa sessão. Caso contrário, o Contêiner utilizará URL Rewriting para manter o número jsessionid entre as requisições de um mesmo cliente.

Dentro do Servlet: **response.encodeURL("url.jsp");** // url rewriting

Redirect com URL Rewriting: use o resultado de **response.encodeRedirectURL("url.jsp")** como parâmetro para o método **sendRedirect()**. Se não for necessário URL rewriting, o primeiro método retorna a URL inalterada.

Atenção: o atributo/parâmetro **jsessionid NÃO** pode ser obtido através de chamadas **request.getParameter()** / **getAttribute()** / **getHeader()**!! Conexões SSL fornecem um mecanismo que é possível identificar cada cliente, possibilitando que o Contêiner faça uso deste mecanismo para manter a sessão com o usuário aberta. Detalhes de como isto é feito é pertinente à implementação do Contêiner.

Principais métodos da interface **HttpSession**:
long getCreationTime(), long getLastAccessedTime(), setMaxInactiveInterval(long), getMaxInactiveInterval(long), invalidate(), getId(), métodos para obter / setar parâmetros.

Atenção: Para evitar que a session seja criada acidentalmente, utilize o método **getSession(false)**.
Atenção: Para saber se o objeto session foi criado na requisição/evento corrente: **session.isNew()**.
Atenção: não existe um método **setId()** no objeto session!! O **ID** da sessão é gerado automaticamente pelo Contêiner.

Invalidando a sessão:

```
<session-config>
   <session-timeout>5</session-timeout> <!-- em minutos!!! -->
</session-config>
```

Lembre-se: o período padrão de timeout é definido pelo Contêiner.

Após um período de 5 minutos de inatividade, o Contêiner invalida a session. Passe < 0 para **garantir** que **todas** as sessões associadas ao Contêiner não sejam invalidadas. Nada impede que tais sessions

sejam invalidadas **programaticamente** (ver abaixo).

session.invalidate(); // arrebenta com a sessão programaticamente
session.setMaxInactiveInterval(5*60); // seta novo timeout para **esta** sessão... em segundos!!!

Passar **-1** para o método **setMaxInactiveInterval()** indica que a session nunca será invalidada pelo Contêiner (somente caso alguém chame o método **invalidate()** nesta session). Ainda, se passarmos **0** (zero) no parâmetro, estamos invalidando a session imediatamente (na prática, é o mesmo que chamar o método **invalidate()**).

Atenção: chamar qualquer método em um objeto session após ele ter sido invalidado (ou ter passado o período de timeout), uma exceção **IllegalStateException** ocorrerá.

Cookies (não confunda um Cookie com um parâmetro do header!!!)

Cookie c = new Cookie("nome", "valor"); // String/String! Não existe construtor sem parâmetros
c.setMaxAge(234); // em segundos. se = **0**, deleta o cookie; se = **<0**, expira quando o browser fechar
response.addCookie(c); // Só vai para a resposta se adicionar o cookie na resposta! :P
Cookie[] c = request.getCookies(); // cookies retornados pelo cliente (isso se o suporte a cookies do browser do cliente estiver ativo)

Migração de sessão entre JVMs

Se a aplicação estiver distribuída em várias VMs, em cada VM terá uma instância de cada Servlet, ServletConfig, ServletContext etc. **A exceção é o objeto Session**.

EXISTE APENAS UM OBJETO SESSION INDEPENDENTE DO NÚMERO DE VMS NAS QUAIS A APLICAÇÃO ESTÁ RODANDO!

O objeto session "migra" entre as VMs se necessário (e possíveis listeners **HttpSessionActivationListeners** são invocados).

Os objetos session podem migrar entre VMs **SEM QUE SEJAM SERIALIZADOS**! Embora seja recomendado que os atributos de sessão implementem **Serializable**, não há nada na especificação que obrigue o uso de serialização ao migrar o objeto entre VMs diferentes.

Ainda em relação à interface **HttpSessionActivationListener**, todos os atributos da session que implementar tal interface são notificadas pelo Contêiner antes da migração da sessão (método **sessionWillPassivate(HttpSessionEvent)**) e após a migração (método **sessionDidActivate(***idem***)**).

Capítulo 7

Expression

<%= algo_que_retorne_alguma_coisa %>

- **SEM** ";";
- Cuidado com o espaço(<% = é errado).

Exemplos:
<%="Aeee" %> .. que é igual a <% out.println("Aeee"); %> (mais detalhes abaixo)

Scriptlet

<% codigo_java %>

Exemplos:
<% out.println("Aeee"); %>

Cuidado... podemos colocar qualquer porcaria dentro de um Scriptlet, até mesmo anular objetos implícitos, portanto cuidado com questões que contenham Scriptlet... podem ser traiçoeiras:

<!-- o próximo acesso à variável **pageContext** retornará **null**, ou seja, essa atribuição é permitida -->
<% pageContext = null; %>

Declaração

<%! metodo_ou_variavel_membro_do_jsp %>

Cuidado com espaços (<% ! é errado).

Exemplos:
```
<%! int doSomething() {
      return 0;
   }
%>
```

Diretiva

<%@ alguma_coisa %>

Cuidado com espaços (<% @ é errado, não deve haver o espaço).

Alguns exemplos de diretivas:
<%@ page *import*="import1, import2" *extends*="xxx" *session*="true" *buffer*="none | **xkb**" *autoFlush*="true" *isThreadSafe*="false" *info*="" *contentType*="text/html" *pageEncoding*="UTF-8" %>
<%@ taglib [tagdir="/WEB-INF/tags" | uri="aa"] prefix="bb" %>
<%@ include file="aaa.jsp" %>

<%@ tag body-content=”empty” %> <!-- apenas para tag files -->
<%@ attribute name=”aaa” *fragment*=”false” *type*=”java.lang.Integer” *description*=”” %> <!-- tag files -->

Comentário

<%-- algo aqui --%>

JSP Implicit Objects – **MEMORIZE TODOS ELES!!!**

out, request, response, session, application, config, exception, pageContext, page.

- out – Stream de saída. **NÃO é um PrintWriter**! É um JSPWriter;
- request, response, session, application, pageContext – **Escopos e response**;
- o resto... **config, exception, page**.

pageContext também serve para que possamos acessar atributos em outros contextos, além de outros métodos importantes.

O objeto implícito **exception** só é “habilitado” caso o JSP seja uma error page.. do contrário, valerá **null**:
<%@ page isErrorPage=”true” %>

Métodos de ciclo de vida do JSP: **jspInit()**, **_jspService()**, **jspDestroy()**. O método **_jspService() NÃO DEVE SER SOBRESCRITO!!! Ah, e não se esqueça do underline (_)!**

Assinatura dos métodos:

- void jspInit();
- void jspDestroy();
- void _jspService(HttpServletRequest, HttpServletResponse) throws IOException, ServletException;

Ciclo de vida JSP

- Tradução, compilação, carregar a classe, instanciar a classe, chamar **jspInit()**, **_jspService()**..., **jspDestroy().**

Init Parameters em um JSP

No DD:
<servlet>
<servlet-name>Servlet</servlet-name>
<jsp-file>/test.jsp</jsp-file> <!-- colocou no lugar de servlet-class. E não se esqueça da barra -->
<init-param>
...
</init-param>
</servlet>

JspContext ← PageContext

JspContext, entre outras coisas, possui métodos para recuperar e setar atributos em diferentes escopos. A classe **PageContext** é quem define as constantes de escopo **(1)**, métodos para acessar objetos JSP implícitos (com exceção do objeto implícito **out**, que é definido na classe **JspContext**), entre outras coisas.

(1) APPLICATION_SCOPE, PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE etc.

Método **findAttribute("xxx")**

Pesquisa por um atributo em diferentes escopos, na seguinte ordem: **page**, **request**, **session**, **application**. O primeiro atributo a ser encontrado, independente do escopo no qual ele se encontra, é retornado.

Atenção às diretivas **page** que **caem** no exame: **import**, **isThreadSave**, **contentType**, **isELIgnored**, **isErrorPage**, **errorPage**. Lembre-se para que serve cada uma delas (não vou colocar aqui).

<%@ include file="xxx.html" %>
Adiciona um recurso em **TEMPO DE TRADUÇÃO**!!

<%@ taglib [tagdir="aa" | uri="myuri"] prefix="sa" %>
Define **taglibs** disponíveis para esta JSP. Use **tagdir** caso queira utilizar Tag Files (este parâmetro **sempre começa** com /WEB-INF/tags!). Caso queira utilizar uma Taglib normal, use o parâmetro **uri** para indicar qual taglib vai responder a qual **prefixo**. **NUNCA** indique **tagdir E uri** ao mesmo tempo.

<%@ page import="" %>
Importa classes. Similar ao comando import do Java.

Configurando propriedades pra grupos de JSPs

No DD:

```
<jsp-config>
   <taglib>
      <taglib-uri>MyTaglibUri</taglib-uri>
      <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
   </taglib>
   <jsp-property-group>
      <url-pattern>*.jsp</url-pattern> <!-- semelhante ao mapeamento de URL de um servlet -->
      <scripting-invalid>true</scripting-invalid> <!-- ativa / desativa scripting (1) -->
      <el-ignored>true</el-ignored> <!-- é possível setar a propriedade isElIgnored no JSP (2) -->
   </jsp-property-group>
</jsp-config>
```

(1) <%%>, <%!%>, <%=%> (ou as correspondentes em sintaxe Document <jsp:scriptlet>, <jsp:declaration> e <jsp:expression>). NÃO DÁ para configurar isso nos JSPs (na versão anterior da espeficicação dava, mas, nesta versão, **NÃO**).

(2) <%@ page isELIgnored="true | **false**" %> **Atenção**: configurações feitas no JSP tem prioridade sobre as configurações no DD.

StandardActions

<jsp:useBean>, <jsp:setProperty>, <jsp:getProperty>, <jsp:include>, <jsp:forward>, <jsp:param>, <jsp:plugin>, <jsp:params>, <jsp:fallback>, <jsp:attribute>, <jsp:body>, <jsp:invoke>, <jsp:doBody>, <jsp:element>, <jsp:text>, <jsp:output>

Atenção: Standard Actions: <jsp:*>, **other** actions: <my_prefix:my_tag> (JSTL se enquadra neste grupo).

Capítulo 8

<jsp:useBean> e <jsp:setProperty>

```
<jsp:useBean id=”attrid” class=”foo.class” type=”foo.class” scope=”request”> (1)
   <jsp:setProperty name=”attrid” property=”prop” value=”xxx” /> (2)
   <jsp:setProperty name=”attrid” property=”prop” param=”paramname” /> (3)
   <jsp:setProperty name=”attrid” property=”prop” /> (4)
   <jsp:setProperty name=”attrid” property=”*” /> (5)
</jsp:useBean>
```

(1) Se usar o atributo **class**, e o atributo “attrid” não existir, um novo objeto deste tipo será criado e colocado no escopo indicado. **Atenção**: o escopo padrão é **page**, então, se quer algo em um escopo mais abrangente, é melhor indicar!

Se somente o atributo **type** for usado, o objeto “attrid” deve existir no escopo indicado. Se não existir, uma exceção **InstantiationException** será lançada.

Se usar ambos os atributos **class** e **type**, o código resultante será tipo assim:

```
MyType obj = null;
obj = new MyClass();
```

Atenção: Ficar atento quanto ao uso de classes abstratas e interfaces no atributo **class**, pois isso não é permitido uma vez que não se pode instanciar classes abstratas nem interfaces.

(2) Todos os <jsp:setProperty> colocados no corpo de uma <jsp:useBean> serão interpretados **caso** o bean não exista no escopo indicado e precise ser criado. Nesta linha, temos o uso mais básico da tag <jsp:setProperty>.

(3) Para atribuir o valor de um parâmetro a uma propriedade do bean, utilizamos o parâmetro param da tag <jsp:setProperty>.

(4) Caso o bean tenha uma propriedade com o mesmo nome da parâmetro, então a atribuição pode ocorrer automaticamente, sem que seja necessário indicar os atributos **param** ou **value**.

(5) Igual ao 4, porém o código atribui todos os parâmetros cujos nomes são compatíveis com os nomes de propriedades do bean.

A converão entre tipos básicos é feito automaticamente, **a não ser que se use uma expression** para se setar um valor a uma propriedade do bean:

```
<jsp:useBean id=”xx” class=”xxx”>
   <jsp:setProperty name=”xx” property=”xx” value=”<%= algo_aqui %>” /> <!-- não converte!! -->
</jsp:useBean>
```

A tag <jsp:setProperty> pode ser usada fora da tag <jsp:useBean>, setando o valor no objeto indicado

independente se ele foi criado agora ou não.

<jsp:getProperty>

Exemplo:
<jsp:getProperty name=”xx” property=”yy” />

Expression Language (EL)

Exemplos:
${foo.bar}, ${foo[attribute]}, ${foo[“bar”]}, ${foo[0]}, ${foo[“0”]}

Objetos (**foo**):

- Um atributo em um dos escopos (page, request, session ou application); ou
- Objetos implícitos: pageScope, requestScope, sessionScope, applicationScope, param, paramValues, header, headerValues, cookie, initParam, pageContext. Com exceção do pageContext, que é do tipo PageContext, **todos são Maps**. Lembre-se que, para acessar objetos HttpRequest etc, **precisamos** o fazer através do objeto **pageContext**, e não através de objetos como requestScope (que é só um Map com os atributos lá dentro).

Atenção: Os objetos implícitos relativos aos escopos (page, request, session e application) possuem “Scope” ao final do nome do objeto! Ou seja, o objeto implícito para o escopo de request é **requestScope** e não apenas **request**!

Atenção: Muito cuidado em expressões do primeiro tipo ${foo.bar} (usam o ponto), pois o “bar” precisa seguir normas impostas pela definição de identificadores Java:

- Precisa iniciar com uma letra, _ ou $;
- Pode ter números;
- Sem espaços ou outros caracteres “especiais” (ponto, vírgula, ponto-e-vírgula etc);
- Não pode ser uma keyword Java.

Portanto, sempre que aparecer expressões ${foo.bar}, **desconfie**! Deve ter algo querendo te ferrar.

Veja o trecho abaixo:

${musicList[numbers[0] + 1]}. Supondo que musicList seja um atributo válido, e numbers[0] resulte em “1”, qual é o resultado? R: o mesmo que ${musicList[2]}, ou ${musicList[“2”]}!!!

Ou ainda: ${“1” + 1} é igual a 2 (int)!!

${a ? b : c} – Sim, **existe** operador condicional ternário em EL! (não me lembro de ter visto isso no livro Head First, **mas tem!**)

EL e Request Params

${paramValues.name[0]} é o mesmo que ${param.name}; ou ainda
se o parâmetro food possui vários valores, a expressão ${param.food} retorna o primeiro, ou seja, é o mesmo que fazer ${paramValues.food[0]}.

EL Functions

Classe com a função:
```
package foo;
public class DiceRoller {
   public static int rollDice() { // precisa ser público e estático
      return xx;
   }
}
```

Arquivo TLD:
```
<taglib ...>
   <tlib-version>x</tlib-version>
   <short-name>xxxxx</short-name>
   <uri>DiceFunctions</uri>
   <function>
      <name>rollIt</name>
      <function-class>foo.DiceRoller</function-class>
      <function-signature>int rollDice()</function-signature>
   </function>
</taglib>
```

Arquivo JSP:
```
<%@ taglib uri="DiceFunctions" prefix="mine" %>
${mine:rollIt()} <!-- Atenção! Chame o nome de método definido no TLD, não na classe -->
```

EL Functions **PODEM TER PARÂMETROS** (e estes assim como o tipo de retorno **pode** ser um objeto):

TLD:
```
<function-signature>
   int rollDice(java.util.Map) <!-- passar o full-qualified name!! -->
</function-signature>
```

JSP:
```
${uri:method_name(myMap)} <!-- myMap é um atributo que retorna um java.util.Map! -->
```

EL Operators
+, -, *, /, div, %, mod, &&, and, ||, or, !, not, ==, eq, !=, ne, <, lt, >, gt, <=, le, >=, ge

Cuidado:
${42 mod 0} ou ${42 % 0} resulta em **Exception**.
${42 div 0} ou ${42 / 0} resulta em **INFINITY**

Muita atenção: **EL is Null-Friendly!**

<jsp:include page="xxx" flush="false" /> <!-- se flush for true, bem, você já deve imaginar a bucha -->
Include é feito durante **run-time** (**e não durante translation-time**, como acontece com a diretiva <%@ include file="xx" />

Atenção: tome cuidado para não incluir elementos html do tipo <html>, <head> etc em pedaços de páginas incluídas com <%@ include %> ou <jsp:include />!! Caso isso aconteça, o documento gerado será considerado inválido e, portanto, não é possível prever como ele será renderizado no browser.

Passando parâmetro via <jsp:include>:
```
<jsp:include ...>
   <jsp:param name="x" value="y" />
</jsp:include>
```

Acessando params em um arquivo incluído: **${param.x}**

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

Preste atenção em exercícios de include/forward parecidos com o seguinte:

page1.jsp
```
<jsp:include page="page2.jsp">
  <jsp:param name="param" value="value2" />
</jsp:include>
```

page2.jsp
```
<!-- código para imprimir todos os valores do parâmetro param -->
```

O que acontecerá ocorrer uma request para page1.jsp contendo uma query string "param=value1"? Imprimirá **value2, value1**, pois quando incluímos ou redirecionamos passando parâmetros, tais parâmetros possuem precedência mais alta em relação aos parâmetros adicionados anteriormente.

```
<jsp:plugin>
<jsp:plugin type="applet | bean" code="Molecule.class" codebase="/html" ...>
   <jsp:params> <!-- esta tag só é usada com jsp:plugin -->
      <jsp:param name="xxx" value="aaa" />
   </jsp:params>
   <jsp:fallback>...</jsp:fallback>
</jsp:plugin>
```

```
<jsp:invoke>
```

Exemplo:
```
<jsp:invoke fragment="frag1" [ var="resultAsStr" | varReader="resultAsReader" ] scope="page" />
```

Atenção: caso o fragmento indicado resulte em **null**, nenhum output ocorrerá (**não dará erro**).
Atenção: só pode ser usada em **Tag Files**!!

```
<jsp:text>
```

Exemplo:

<jsp:text>Some text</jsp:text>

Atenção: é válido tanto para documentos JSP como páginas JSP!

<jsp:text>Some <jsp:text>text</jsp:text></jsp:text> - **Erro! Não se pode aninhar <jsp:text>**!

Dynamic Attributes – Se cai na prova, eu não sei.. **mas faz parte da spec**!

Podemos criar uma custom tag que aceite qualquer nome e quantidade atributos. Na declaração da tag, colocar, por último, a indicação de que esta tag trabalha com atributos dinâmicos:

<dynamic-attributes>true</dynamic-attributes> <!-- o padrão é.. erhm.. false -->

Em seguida, a classe da tag deve implementar a interface **DynamicAttributes** , que declara o método:

setDynamicAttribute(String uri, String localName, Object value) throws JspException;

Dentro deste método, coloque código que adicione o valor em um **Map**. Então, dentro do método **doTag()** (ou **doStartTag()**, caso seja uma classic tag handler), você terá acesso aos atributos indicados dinamicamente. Vale lembrar que, caso existam atributos “estáticos” declarados no TLD que sejam obrigatórios, essa feature de atributos dinâmicos **não torna a indicação de tais atributos opcional**!

<c:out>

Exemplos:
<c:out value=”qualquer coisa” *escapeXml*=”**true** | false” *default*=”...”/>
<c:out...>qualquer coisa</c:out>

Escreve o valor na response, “escapando” caracteres especiais como <, >, &, ', “.

<c:forEach>

Exemplos:
<c:forEach var=”movie” items=”${movies}” *varStatus*=”status” *begin*=”1” *end*=”10” *step*=”1” >
  ${movie}
<c:forEach>

Caso o objeto especificado no atributo **items** seja um Map, o objeto **var** retornado pela tag é do tipo Map.Entry, onde podemos obter a chave e o valor:

${movie.key} e ${movie.value}

Objeto “status” é do tipo **LoopTagStatus**, que possui alguns getters: getBegin(), getEnd(), getCount(), getCurrent(), getIndex(), getStep(), getFirst, getLast(). **Não possui métodos setter**.

NÃO tente usar a variável “movie” fora da tag forEach... o escopo dela é TAG!

O atributo **begin** deve ser maior que zero. Também deve ser maior que **end (**senão o loop não é executado). O atributo **step** deve ser maior ou igual a 1. Se begin for igual ao número de elementos do objeto especificado em **items**, o loop também não é executado.

<c:forTokens>

Exemplo:
<c:forTokens items=”tokens” var=”tk” *delims*=”,” *varStatus*=”status” *begin*=”” *end*=”” *step*=””>
${tk}
</c:forTokens>

Atributos **begin**, **end**, **step**, **varStatus** funcionam iguais à tag c:forEach.

<c:if>

Exemplo:
<c:if test=”${true}”>
  <!-- faz algo -->
</c:if>

Atenção: Se o valor do atributo avaliar para um valor **false** ou **null**, o body não será invocado (como se fosse == **false**). Se der **true**, será.

<c:choose>

Exemplo:
<c:choose>
  <c:when test="${true}"><!-- faz algo --></c:when>
  <c:otherwise><!-- faz outra coisa --></c:otherwise> <!-- deve ser o último elemento de choose!! -->
</c:choose>

Atenção: **não** deve haver um elemento <c:when> **APÓS** um <c:otherwise>!! **Não** deve haver mais de um **<c:otherwise>**.

<c:set>

Exemplo:
<c:set var="attrid" scope="session" value="xx" />
<c:set var="attrid" scope="session">xx</c:set>

Quando usado o atributo **var (**que representa um atributo), atenção ao atributo **scope** e **value**. Se o atributo **scope** não for fornecido, o valor padrão (**page**) é usado. Se o atributo **value** for diferente de **null**, o valor será setado no atributo de nome **var**; caso contrário, tal atributo será **REMOVIDO**! Igual a:

<c:remove var="attrid" scope="request" /> <!-- scope padrão é **page -->**

Atenção: o atributo var deve ser preenchido com uma String, não com uma EL que retorna um objeto, ou algo do tipo!

<c:set target="${elexpr}" property="xx" value="yy" />
<c:set target="${elexpr}" property="xx">yy</c:set>

Usado para setar valores **APENAS** em beans e maps. Nada de **lists**. O atributo **target deve** retornar um **objeto**, não uma **String**. Não é necessário utilizar o atributo **scope** aqui, uma vez que o objeto a ser modificado pela tag já está disponível na propriedade **target**.

<c:import>

Exemplos:
<c:import url="/xxx.jsp" *context*="/xxx" *var*="varName" *scope*="**page** | ..." charEncoding="""" />
<c:import url="http://xxx">
  <c:param name="aa" value="bb" /> <!-- opcional, podemos passar parâmetros -->
</c:import>

Quase igual a standard action <jsp:include page="xx" />, com a diferença de que <c:import> pode referenciar arquivos e páginas **FORA** do contexto da aplicação corrente.

Para recuperar os parâmetros enviados por <c:import/>, é igual a como se estivesse usando a tag <jsp:include/>:

${param.name}

Podemos especificar um recurso relativo a outro contexto. Caso o atributo **var** seja especificado, devemos informar o escopo (ou deixar vazio, indicando que o escopo será **page**). Informando **var**, o conteúdo do recurso importado será colocado na variável cujo nome foi especificado no parâmetro, e não *inline* (diretamente na response).

<c:url>

Exemplo:
```
<c:url value=”urllocation”>
   <c:param name=”aa” value=”bb” /> <!-- opcional, passa parâmetros através da URL -->
</c:url>
```

Faz URL Rewriting de uma determinada URL. O resultado desta tag é uma URL de resultado (com o valor **jsessionid** anexado ao endereço). **Atenção**: esta tag **NÃO RETORNA** um link (tag HTML <a>), **apenas uma String** contendo o link com URL Rewriting.

<c:url> também tem os parâmetros **var** e **scope**, e funcionam de modo idêntico à tag <c:import>.

<c:redirect>

Exemplo:
```
<c:redirect url=”http://xxx” context=”/context” />
```

JSP Error Page

```
<%@ page isErrorPage=”true” %> <!-- na página que mostra o erro -->
<%@ page errorPage=”pagina_de_erro.jsp” %> <!-- linka a página de erro nesta JSP -->
```

Error Pages (via DD)

```
<error-page>
   <exception-type>java.lang.Throwable</exception-type> <!-- full qualified name -->
   <location>/arquivo.jsp</location> <!-- não esqueça a barra -->
</error-page>

<error-page>
   <error-code>404</error-code>
   <location>/not_found.jsp</location> <!-- não esqueça a barra -->
</error-page>
```

Caso uma exceção lançada por um Servlet/JSP possa ser tratada por mais de uma página de erro, quem ganha a parada é a página de erro que declarar um tipo de exceção mais específica em relação a exceção lançada.

Toda página JSP que é uma página de erro tem acesso a um objeto implícito chamado **exception**, através

do qual podemos obter a exceção que está sendo tratada pela página.

Toda error page também recebe alguns atributos acessíveis através do objeto request:

- javax.servlet.error.status_code
- javax.servlet.error.exception_type
- javax.servlet.error.message
- javax.servlet.error.exception
- javax.servlet.error.servlet_name

<c:catch *var*=”myExceptionObj”>throw new RuntimeException</c:catch>

Esta tag simula um bloco **try/catch** dentro do JSP. Toda exceção lançada dentro deste código será “abafada”. Funciona de modo semalhante a qualquer código Java normal.

Use o atributo **var** para que a exceção lançada fique disponível em escopo de página.

CustomTags

DD:

```
<taglib...>
   <tlib-version>1.0</tlib-version>
   <short-name>mytaglib</short-name>
   <uri>MyTaglib</uri>
   <tag>
      <description>My custom tag</description>
      <name>simpletag</name>
      <tag-class>foo.bar.SimpleTag</tag-class>
      <tag-body>empty</tag-body> <!-- obrigatório para simple tag handlers. default: JSP -->
      <attribute> <!-- zero ou mais -->
         <name>xxx</name>
         <description>my tag</description>
         <required>false | true</required>
         <rtexprvalue>false | true</rtexprvalue>
      </attribute>
   </tag>
</taglib>
```

JSP:

```
<%@ taglib uri=”MyTaglib” prefix=”mtl” %>
<mtl:simpletag xxx=”aaa” />
<mtl:simpletag>
   <jsp:attribute name=”xxx”>aee</jsp:attribute> <!-- ainda é considerada uma tag empty! -->
</mtl:simpletag>
```

Tag class:

```
public class SimpleTag extends SimpleTagSupport {
   public void doTag() throws JSPException IOException {
   }
   public void setXxx(String xxx) {}
```

}

Body content padrão para cada tipo de tag:

- **Simple Tags**: empty, tagdependent, scriptless (o padrão é JSP, mas JSP **não é permitido** em Simple Tag Handlers);
- **Classic Tags**: empty, tagdependent, scriptless, **JSP**;
- **Tag Files**: empty, tagdependent, **scriptless**;

Independente do tipo de tag (Simple Tag ou Classic Tag), o valor JSP é o padrão (embora o valor **JSP** seja inválido para Simple Tags). Então, se a tag for uma **SimpleTag**, é melhor especificar um valor dentre os três valores válidos.

Declarando uma taglib no DD

```
<jsp-config>
   <taglib> <!-- zero ou mais -->
      <taglib-uri>MyURI</taglib-uri>
      </taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
   </taglib>
</jsp-config>
```

Prefixos **RESERVADOS**: jsp:, jspx:, java:, javax:, servlet:, sun:, sunw:.

Cuidado com exemplos do tipo (os dois trechos **SÃO EQUIVALENTES**):

```
<foo:bar xyz=”abc”>Body</foo:bar>

<foo:bar>
   <jsp:attribute name=”xyz”>abc</jsp:attribute>
   <jsp:body>Body</jsp:body>
</foo:bar>

<foo:bar>
   <jsp:attribute name=”xyz” value=”abc” /> <!-- inválido! jsp:attribute não tem um atributo value -->
   <jsp:body>Body</jsp:body>
</foo:bar>

<foo:bar>
   <jsp:attribute name=”xyz”>abc</jsp:attribute>
   Body
</foo:bar> <!-- INVÁLIDO! É necessário usar jsp:body se usar jsp:attribute -->

<foo:bar xyz=”abc”>
   <jsp:attribute name=”xyz”>abc</jsp:attribute>
</foo:bar> <!-- Erro! Só pode especificar um parâmetro em um lugar num dado momento -->

<foo:bar xyz=abc>Body</foo:bar> <!-- Erro! Valor deve estar entre aspas simples ou duplas -->
```

Os exemplos abaixo **também** são equivalentes:

<foo:bar><jsp:body>aee</jsp:body></foo:bar>
<foo:bar>aee</foo:bar> <!-- tag jsp:body pode aparecer -->

Capítulo 10

Tag files
Crie um arquivo com extenção .tag ou .tagx[1]* e o coloque no diretório WEB-INF/tags

No JSP:
<%@ taglib prefix=”mytags” tagdir=”/WEB-INF/tags” %>
<mytags:ARQUIVO />

Passando parâmetros para uma Tag file

No arquivo .tag:
<%@ attribute name=”subTitle required=”true” rtexprvalue=”true” %>
${subTitle} <!-- não tem o **param.** na frente! -->

No JSP:
<mytags:ARQUIVO subTitle=”abababababa” />

Pegando o body da invocação da tag

No arquivo .tag:
<jsp:doBody/>

Atenção: não se pode usar scripting dentro do body de uma invocação de **Tag file**, o que não significa que scripting possa ser usado em Tag files!! Por exemplo:

Arquivo .tag: <%=header.getHeader(“blablabla”) %> <!-- é permitido! -->
Arquivo .jsp: <mytags:ARQUIVO><%=header.getHeader(“blablabla”) %></mytags> <!-- **OOPS** -->

<%@ tag *body-content*=”**scriptless** | empty | tagdependent” *dynamic-attributes*=”**false** | true” *description*=”my desc” *language*=”java” *import*=”java.util.*” *pageEncoding*=”text/html” *isELIgnored*=”false” %>

É parecida com a diretiva **page**, mas é específica para as Tag files. Aliás, muitos dos atributos são iguais nas duas diretivas.

Atenção: as tags abaixo **NÃO** são consideradas vazias:
<foo:bar> </foo:bar> <!-- **atenção** para o espaço -->
<foo:bar><!-- a html comment --></foo:bar>

Lembre-se: Se uma Tag file for implantada dentro de um JAR, **não esqueça de que é necessário um arquivo TLD**!! Além disso, as Tag files precisam estar sob o diretório **META-INF**, e **não no WEB-INF!**

```
<taglib ...>
   <tlib-version>1.0</tlib-version>
   <uri>mytags</uri>
```

1 O sufixo .tagx é usado para tag files que usam a sintaxe XML (JSP document) em vez da sintaxe JSP “normal”.

```
  <tag-file>
    <name>Header</path>
    <path>/META-INF/tags/Header.tag</path> <!-- sempre começa com /META-INF/tags -->
  </tag-file>
</taglib>
```

Simple Tag Handler

Crie uma classe que estenda **SimpleTagSupport** e implemente o método **doTag() throws JspException, IOException**.

Declare a tag no tld:

```
<taglib ...>
  ...
  <tag>
    <description>aee</description>
    <name>aee</name>
    <tag-class>foo.TagClass</tag-class>
    <body-content>scriptless</body-content> <!-- deve ser espeficado para SimpleTag handlers -->
  </tag>
</taglib>
```

Invocar o body da tag dentro do método **doTag()**:
getJspBody().invoke(null);

Ciclo de vida de uma **Simple Tag Handler**. **Atenção**:

Carrega a classe, instancia (construtor sem argumentos), setJspContext(), setParent() (**se** a tag estiver dentro de outra), chama os setters dos atributos, setJspBody() (se o body-content for diferente de **empty E** a tag **tem um body**), doTag().

Declarando um atributo numa custom tag

TLD:

```
<tag>
  ...
  <attribute>
    <name>paramname</name>
    <required>true | false</required>
    <rtexprvalue>true | false</rtexprvalue>
  </attribute>
</tag>
```

Atenção: não esqueça de colocar um método setter no tag handler, correspondente ao atributo.

jspFragment: **NÃO** deve conter scripting. Serve apenas para ser invocado, através do método invoke(Writer). Passe **null** ou um writer caso queira tratar manualmente o body da tag.

SkipPageException: lance esta exceção caso queira que o resto da JSP (ou tag file) **NÃO SEJA**

avaliado. **Preste atenção** caso uma página incluída com <jsp:include> ou <c:import> lance tal exceção... o que acontecerá é que o resto **dessa página incluída será ignorada**, no entanto, o resto da página que chamou o include **SERÁ AVALIADA NORMALMENTE**!

Classic Tag Handlers

Hierarquia de classes

Simple tags
JspTag ← SimpleTag ← SimpleTagSupport

Classic tags
JspTag ← Tag ← IterationTag ← BodyTag
IterationTag ← TagSupport
BodyTag ← BodyTagSupport
TagSupport ← BodyTagSupport

Métodos importantes:
SimpleTag – void doTag()
SimpleTag - JspTag getParent()
SimpleTag – void setParent(JspTag)
SimpleTag - void setJspBody(JspFragment)
SimpleTag - void setJspContext(JspContext)
SimpleTagSupport – JspTag findAncestorWithClass(JspTag, Class)
SimpleTagSupport – JspFragment getJspBody()

Lembre-se: o método **findAncestorWithClass**: é um método utilitário **estático** da classe **SimpleTagSupport**!

Tag – int doStartTag()
Tag – int doEndTag()
Tag - Tag getParent()
Tag – void setParent(Tag)
Tag – void setPageContext(PageContext)
IterationTag – int doAfterBody()
BodyTag – void doInitBody()
BodyTag – void setBodyContent(BodyContent)
TagSupport – pageContext (é uma variável, não um método!)
BodyTagSupport – BodyContent getBodyContent()

A declaração TLD **NÃO** é diferente caso a tag seja Simple Tag ou Classic Tag!

Atenção: Ao contrário das SimpleTag handlers, o Contêiner **PODE** reutilizar uma instância de uma **Classic Tag**, fazendo com que a mesma instância seja utilizada várias vezes caso o JSP faça várias invocações a um mesmo tipo de tag. Isso não acontece com **SimpleTag handlers**, pois, para cada invocação de tag uma nova instância é utilizada (mesmo numa JSP onde existem várias invocações a um mesmo tipo de tag). Portanto, certifique-se de zerar as variáveis de instância aos seus valores iniciais no método doStartTag() (antes de executar o código relativo à tag, lógico).

Métodos principais

Subclasses de TagSupport
int doStartTag() throws JspException. Pode retornar: EVAL_BODY_INCLUDE, SKIP_BODY
int doAfterBody() throws JspException. Pode retornar: SKIP_BODY, EVAL_BODY_AGAIN
int doEndTag() throws JspException. Pode retornar: EVAL_PAGE, SKIP_PAGE

Subclasses de BodyTagSupport
Igual ao **doStartTag()** anterior, mas pode retornar também **EVAL_BODY_BUFFERED**. Este retorno faz com que o Contêiner chame o método **setBodyContent()** (caso haja body e o body-content seja diferente de **empty**), e, em seguida, chama o método **doInitBody()**, invocando o body da tag. Ambos os métodos podem ser sobrescritos.
Atenção: **NÃO** retorne EVAL_BODY_BUFFERED a não ser que a tag estenda BodyTagSupport! (até porque tal constante não existe na classe TagSupport e superclasses).

Importante: a relação entre o body-content e o retorno do método **doStartTag()** é **extremamente importante**. **SE** o body-content for **empty E** você retornar algo diferente de **SKIP_BODY** no método **doStartTag()**, **prepare-se para tomar um erro na cabeça**!

Ciclo de vida de uma Classic tag

Novamente, **atenção**! Muitos "se" aqui.

Carrega a tag, instancia (construtor sem parâmetros), setPageContext(PageContext), setParent(Tag) (**se** a tag estiver dentro de outra), chama os setters dos atributos, doStartTag(), setBodyContent() e doInitBody() (**se** doStartTag() retornar EVAL_BODY_BUFFERED **E** a tag **tiver um body** E body-content for **diferente** de **empty**), doAfterBody() (**se** o body foi avaliado), doEndTag().

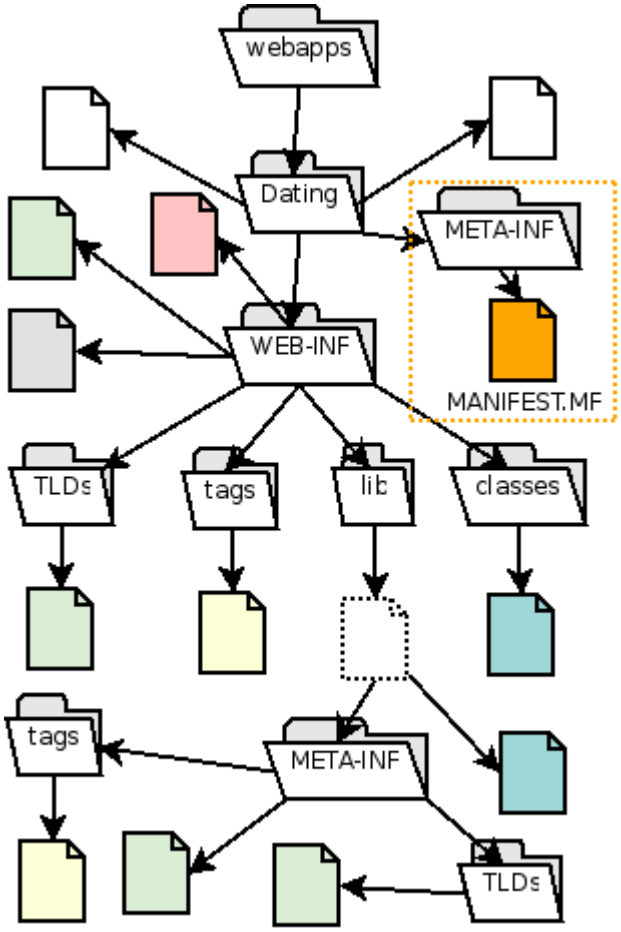
Simple tag handlers podem ter Classic tag handlers como parent tags!!

Isso é valido, pois a interface **Tag** estende **JspTag** (que é usada pelas Simple Tag handlers), bastando um casting para adaptar a referência ao tipo da tag-pai. Porém, a recíproca **NÃO É VERDADEIRA**. Pelo menos não no sentido clássico...

Para que uma Classic Tag handler possa ter uma SimpleTag como tag-pai, o Contêiner envelopa a instância da Simple Tag handler em um objeto **TagAdapter**, passando esse adapter às chamadas **setParent(Tag)**. Então, para obter uma referência à tag-pai que é uma Simple Tag, faça isso (dentro da Classic tag):

```
TagAdapter adapter = (TagAdapter) getParent();
MySimpleTag tag = (MySimpleTag) adapter.getAdaptee();
```

Cada cor de arquivo corresponde a um tipo (sim, tente se lembrar de cada um deles apenas olhando para a imagem):



Arquivos WAR

É um JAR comum, que contém toda aplicação web. **Tenha certeza** de criar o WAR cujo diretório raíz seja o diretório acima de WEB-INF (e não o diretório acima do diretório raíz).

Todo WAR **deve** ter um diretório **META-INF** e, dentro desse diretório, deve existir um arquivo MANIFEST.MF (onde declaramos possíveis dependências da nossa aplicação – **classes** e **bibliotecas**). Se, durante o deploy da aplicação, o Contêiner não conseguir achar as dependências indicadas no META-INF, a aplicação **NÃO** é implantada (ao contrário de uma aplicação 'exploded' (sem ser em formato WAR, somente os arquivos desempacotados), onde um erro de dependência aparece somente quando um cliente tenta acessar a aplicação pela primeira vez).

Assim como o WEB-INF, o diretório META-INF também **não é visível externamente**, o que implica que, quaisquer arquivos dentro destas duas pastas não serão visíveis externamente (ainda possam ser utilizados internamente pela aplicação).

Servlet mapping

<url-pattern>/Beer/Select</url-pattern>
<url-pattern>/Beer/SelectBeer.do</url-pattern>
<url-pattern>/Beer/*</url-pattern>
<url-pattern>*.do</url-pattern>
<url-pattern>/*</url-pattern>

Atenção nas barras.. elas **devem ser usadas conforme mostram os exemplos**.

Quando uma request chega, o Contêiner procura pelo pattern que casa melhor com a URL requisitada. Em outras, palavras, se mais de um pattern puder ser escolhido, **o pattern mais específico será usado**.

Welcome files

Exemplo:
<welcome-file-list><welcome-file>home.html</welcome-file></welcome-file-list>

A razão da existência dos welcome files é especificar URIs numa determinada ordem que o Contêiner pode usar caso a URI da request aponte para um diretório da aplicação web. Essas requests são conhecidas também como **partial requests**.

Quando a request mapeia para um diretório no qual não existe um <welcome-file> especificado, o comportamento é dependente de fornecedor (alguns mostram um erro 404, outros mostram uma lista com os arquivos disponíveis etc). Caso mais de um <welcome-file> possa ser usado, o primeiro <welcome-file> compatível será utilizado. (os arquivos são verificados na ordem em que estão declarados).

Servlet load-on-startup

```
<servlet>
   <servlet-name>MyServlet</servlet-name>
   <servlet-class>foo.MyServletClass</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
```

Use para definir a ordem na qual os servlets devem ser iniciados. Use valores maiores que zero para definir a ordem.

Lembre-se: O elemento <load-on-startup> **pode** ser utilizado num elemento <servlet> que mapeia para um JSP. O resultado disso é que tal JSP será pré-compilado durante o deploy da aplicação (e não quando ele for acessado pela primeira vez).

Lembre-se: Se vários Servlets/JSPs possuem um mesmo valor no elemento <load-on-startup>, o **Contêiner é quem determina a ordem** de inicialização.

JSP Document

JSP “normal” (.jsp)	Documento JSP (.jsp, .jspx[2]*)
<%@ page import=”” %>	<jsp:directive.page import=”” />
<%! int y = 3 %>	<jsp:declaration>int y = 3;</jsp:declaration>
<% a = 10; %>	<jsp:scriptlet>a = 10;</jsp:scriptlet>
<%= “test” %>	<jsp:expression>”test”</jsp:expression>
Testing	<jsp:text>Testing</jsp:text>

Para diretivas...
<jsp:directive.*****> onde ***** é **page**, **import**, **attribute** etc.

Atenção: não existe uma sintaxe especial para definir comentários. O comentário XML é o que pode ser usado: <!-- my comment -->

Estrutura básica de um Document:
<?xml version=”1.0” encoding=”utf-8”?>
<jsp:root xmlns:jsp=”http://java.sun.com/JSP/Page” version=”2.0”>
  <table>foo</table>
</jsp:root>

Atenção: **não existe** uma tag específica para definição de taglibs, já que a declaração de taglibs é feita através do uso de namespaces XML indicados na tag <jsp:root>!

Referências a EJBs

EJBs Locais:
<ejb-local-ref>
  <ejb-ref-name>ejb/Test</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>foo.TestHome</local-home>
  <local>foo.Test</local>
  *<ejb-link>*ejb/TestLink*</ejb-link>*
</ejb-local-home>

EJBs Remotos:
<ejb-ref>
  <ejb-ref-name>ejb/Test</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>foo.TestHome</home>
  <remote>foo.Test</remote>
  *<ejb-link>*ejb/TestLink*</ejb-link>*
</ejb-home>

Entradas de ambiente

<env-entry>

2 O Contêiner interpreta um arquivo .jspx como sendo um JSP document, portanto só use este sufixo em JSP documents.

```
   <env-entry-name>my/variable</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type> <!-- use OBJETOS, não primitivos -->

   <!--Algo que possa ser passado via construtor como String ou Character -->
   <env-entry-value>22</env-entry-value>
</env-entry>
```

Mapeamento MIME

```
<mime-mapping>
   <extension>mpg</extension> <!-- sem o ponto -->
   <mime-type>video/mpeg</mime-type>
</mime-mapping>
```

Capítulo 12

Requisitos de segurança:

- **Authentication**: Como o Contêiner sabe que você é você? Geralmente implica em exigir um login e uma senha;
- **Authorization**: Como o Contêiner sabe que você tem acesso às informações que requisitou? Geralmente um usuário possui uma ou mais roles (nas quais se baseiam as configurações de autorização);
- **Confidentiality**: Os dados enviados/recebidos (como números de cartão de crédito) não podem ser vistos por outras pessoas/coisas que não sejam o Contêiner e o usuário que fez a requisição;
- **Data integrity**: Os dados enviados são idênticos aos recebidos. Não se pode modificar os dados por outra coisa externa à conversação entre o Contêiner e o usuário que fez a requisição.

Definindo roles

No DD:

```
<!-- deve mapear com os usuários definidos no Contêiner (vendor-specific...) -->
<security-role> <!-- no singular!!-->
   <role-name>Admin</role-name>
</security-role>

<security-role>
   <role-name>Member</role-name>
</security-role>

<login-config> <!-- é opcional no DD, mas deve ser especificado para ativar a autenticação -->
   <auth-method>BASIC | DIGEST | CLIENT-CERT | FORM</auth-method>
   <realm-name>myRealmName</realm-name> <!-- nome de realm usado na aplicação -->
   <form-login-config> <!--se for usado FORM apenas -->
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/error.html</form-error-page> <!-- não esquecer a barra -->
   <form-login-config>
</login-config>

<security-constraint> <!-- podem haver zero ou mais elementos deste tipo -->
   <display-name>Some name</display-name>
   <web-resource-collection> <!-- um ou mais -->
      <web-resource-name>xx</web-resource-name> <!-- obrigatório -->
      <description>My bla bla bla</description>
      <url-pattern>/Beer/AddRecipe/*</url-pattern> <!-- um ou mais -->
      <http-method>POST</http-method> <!-- zero ou mais -->
   </web-resource-collection>
   <auth-constraint>
      <role-name>Admin</role-name>
      <role-name>Member</role-name>
   </auth-constraint>
</security-constraint>
```

- Se não definir <http-methods>, então **TODOS** os métodos serão protegidos. Se definir, apenas os métodos especificados serão protegidos;
- Elemento <realm-name> é opcional, ou seja, a autenticação funcionará independente de você fornecer este atributo. Este atributo é ignorado quando FORM é utilizado como método de autenticação;
- <auth-constraint> não especificado é igual a:

```
<auth-constraint>
   <role-name>*</role-name>
</auth-constraint>
```

- ... o que significa que o recurso pode ser acessado por **qualquer usuário**;
- <auth-constraint> vazio indica que nenhum usuário terá acesso aos recursos;
- role-name **é case-sensitive**.

Se um mesmo recurso é protegido por duas constraints diferentes:

- se ambas as constraints especificam role-names, então **as role-names** indicadas nas duas constraints serão usadas no controle de acesso;
- se uma das constraints aceitar qualquer usuário, então todos serão permitidos a acessar o recurso independente da outra constraint (a não ser que a outra constraint seja do tipo "proíba-tudo"... ver abaixo);
- se uma das constraints bloquear qualquer usuário, então todos os usuários serão bloqueados independente da outra constraint (inclusive se a outra constraint liberar acesso a todos, como mostrado no tópico anterior);

Segurança programática

```
boolean b = request.isUserInRole("AROLE");
request.getRemoteUser(); // não muito popular, mas pode ser usado para obter o estado da autenticação
```

Quando o método **isUserInRole()** é invocado, o usuário precisa estar autenticado. Se não estiver, o método retorna **false** de cara.

O parâmetro usado no método **isUserInRole()** normalmente precisa ser indicado no DD:

```
<servlet> <!-- atenção, a configuração é colocada dentro do servlet, e não globalmente -->
   <security-role-ref>
      <role-name>Xxxx</role-name> <!-- usado no parâmetro do método isUserInRole() -->
      <role-link>Manager</role-link> <!-- OPCIONAL, role declarada no DD -->
   </security-role-ref>
</servlet>
...
<security-role>
   <role-name>Manager</role-name> <!-- that's it! -->
</security-role>
```

Se a role indicada no argumento do método não estiver mapeada no <security-role-ref>, o Contêiner tentará localizá-la **direto** no elemento <security-role>. Se existir, tudo ok.. senão, **dá erro**.

Se a role indicada no argumento do método estiver mapeada no <security-role-ref>, o Contêiner a utilizará para obter a role real (para a qual a primeira é mapeada). Isso acontece MESMO se uma role idêntica à utilizada no método existir no elemento <security-role>.

Métodos de autenticação

- **BASIC**: Não encriptado (apenas codificado Base64, o que de fato não pode ser considerado como encriptação). Definido na especificação HTTP;
- **DIGEST**: Encriptado, mas é possível que existam Contêiners que não suportem este método (a especificação não os força a isso). Encriptação média. Definido na especificação HTTP;
- **CLIENT-CERT**: Bastante seguro. Os clientes precisam ter um certificado antes que eles possam logar no sistema. Bem difícil de se utilizar a não ser em aplicações B2B. Definido na especificação J2EE;
- **FORM**: Página customizada de login, ao contrário dos métodos anteriores. Os dados trafegam em PLAIN-TEXT, sendo este o nível mais baixo de segurança. Definido na especificação J2EE.

FORM – Atenção para os nomes dos campos e da action:

```
<form method=”POST” action=”j_security_check”>
   username: <input name=”j_username” /><br/>
   password: <input type=”password” name=”j_password” /><br/>
   <input type=”submit” />
</form>
```

Transmissão segura de dados

No DD:

```
<security-constraint>
   <display-name>name</display-name>
   <web-resource-collection>...</web-resource-collection>
   <auth-constraint>...</auth-constraint>
   <user-data-constraint>
      <description>bla bla bla</description>
      <transport-guarantee>NONE | INTEGRAL | CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

- **NONE**: nem precisa dizer o que significa;
- **INTEGRAL**: Impede mudança dos dados no meio do caminho;
- **CONFIDENTIAL**: Impede a leitura e mudança dos dados no meio do caminho.

Primeiramente, ao receber uma request, o Contêiner verifica se existe algum <transport-guarantee> para o recurso requisitado. Se não houver, continua, fazendo a checagem da autenticação/autorização.

Se houver, o Contêiner primeiramente se certifica de que tal request ocorra de modo seguro, enviando um 301 redirect para o browser, pedindo que ele se conecte de modo seguro (provavelmente usando HTTPS). Na próxima request, **SE** ela ocorreu de modo como determina o <transport-guarantee>, as checagens de autenticação/autorização são realizadas.

MUITA ATENÇÃO: para garantir que os dados sejam transmitidos de modo seguro, certifique-se de setar transport-guarantee em **TODO RECURSO** protegido que possa ativar o processo onde a

transmissão deve ocorrer com segurança. Se os dados são transmitidos de forma insegura, já era...

Capítulo 13

Filtros

Implemente a interface **Filter**:

- void init(FilterConfig) throws ServletException;
- void doFilter(ServletRequest, ServletResponse, FilterChain) throws IOException, ServletException;
- void destroy().

Atenção: o Contêiner cria apenas **UMA** instância de cada filter por JVM (**a não ser que existam duas declarações de filter no DD que usam a mesma classe filter!!**).

O filter também pode lançar uma exceção **UnavailableException** para indicar sua indisponibilidade, fazendo com que tal filtro não seja aplicado temporariamente ou permanentemente (outros filtros disponíveis e o próprio recurso web continuam funcionando).

Métodos importantes de **FilterChain**:

- void doFilter(ServletRequest, ServletResponse) throws IOException, ServletException.

Métodos importantes de **FilterConfig**:

- ServletContext getServletContext();
- String getInitParameter(String) e Enumeration getInitParameterNames()

Atenção para os ServletRequest e ServletResponse (não são HttpServletRequest e HttpServletResponse)... **é necessário fazer casting** dos parâmetros para os tipos HTTP.

Chamando o próximo filter da lista (ou o servlet/jsp, caso não existam outros filters):
chain.doFilter(req, resp); // chain é o FilterChain

Declarando um Filter

```
<filter>
   <filter-name>BeerRequest</filter-name>
   <filter-class>com.example.web.BeerRequestFilter</filter-class>
   <init-param>...</init-param> <!-- podem ter parametros de inicialização -->
</filter>

<filter-mapping>
   <filter-name>BeerRequest</filter-name>
   <url-pattern>*.do</url-pattern>
   <servlet-name>BeerServlet</servlet-name> <!-- url-pattern OU servlet-name, nunca ambos -->
   <dispatcher>REQUEST | INCLUDE | FORWARD | ERROR</dispatcher> <!-- zero até quatro -->
</filter-mapping>
```

A ordem na qual declaramos os filter-mappings **É IMPORTANTE**. Todos os filters que estiverem mapeados para uma determinada URL executarão na ordem em que são declarados. No caso,

primeiramente o Contêiner lê os filters mapeados para URLs **e concatena** a este resultado os filters mapeados diretamente a servlets (também na ordem em que são declarados).

O elemento <dispatcher> determina se os filtros são invocados em cada caso particular (request, include, forward ou error). Caso nenhum elemento dispatcher seja especificado, então o valor REQUEST é assumido como padrão.

Filters para processamento de output

Caso seja necessário processar algo após a execução do servlet/jsp/outros filters, então devemos utilizar um **(Http)ServletResponseWrapper** pois, após a chamada chain.doFilter() a response **já foi comitada para o cliente**!!! Podemos definir wrappers para a Request, através da classe **(Http)ServletRequestWrapper**.

Um wrapper é um **decorator** para o objeto **ServletRequest** / **Response** real, onde podemos fazer customizações.

Nomenclatura das classes Wrapper: Http + ServletRequest, ServletResponse + Wrapper. O objeto "envelopado" é passado por parâmetro **para o construtor** do wrapper. Sua implementação customizada do Wrapper **deve**, no construtor, receber o parâmetro e enviá-lo à superclasse através de uma chamada a **super(obj)**.

Capítulo 14

Boas práticas

- Programar para interfaces;
- Separação de responsabilidades e coesão;
- Esconder complexidade;
- Baixo acoplamento;
- Proxies remotos; (tá no livro, ué)
- Controle declarativo.

“idades”:

- Manutenibilidade, flexibilidade, modularidade, testabilidade, extensibilidade etc.

Design patterns

Intercepting Filter

- Recursos:
  - Interceptar e/ou modificar requisições/respostas antes/após que elas alcancem/deixem o servlet;
  - Deployment declarativo;
  - Podem ser executados em correntes;
  - Ciclo de vida gerenciado automaticamente;
- Princípios:
  - Coesão, baixo acoplamento, controle declarativo;
  - Permite que filtros sejam facilmente implementados (por serem genéricos);
  - Seqüência de invocação facilmente modificada;

Business Delegate

- Recursos:
  - Existem componentes diferentes para interagir com o usuário final e lidar com a lógica de negócios;
  - Tais componentes residem em locais diferentes separados por uma rede;
  - Expõe a API dos serviços de negócios a clientes, atuando como componentes “servidores” que fornecem acesso aos serviços;
  - Vários clientes precisam ter acesso a API;
  - Inicia e lida com a comunicação (e possíveis exceções);
  - Permite maior coesão nos controllers que o utilizam;
- Princípios:
  - Esconde complexidade, codificação para interfaces, baixo acoplamento, separação de responsabilidades;
  - Minimiza o impacto de mudanças em uma tier por mudanças na outra;
  - Adiciona mais uma layer, o que aumenta a complexidade;

Service Locator

- Recursos:
  - Localização e utilização de recursos e serviços através da rede;
  - Lida com detalhes/problemas de comunicação;

    - Pode melhorar performance através de caching dos objetos obtidos através do registry ou outro mecanismo que disponibiliza tais serviços;
    - Trabalha com diferentes tipos de registry: JNDI, RMI, UDDI, COS.
- Princípios:
    - Esconde complexidade, separação de responsabilidades;
    - Reduz acoplamento;
    - Minimiza impactos de mudanças na camada web quando componentes mudam de Contêiner ou local;

Transfer Object

- Recursos:
    - Representa localmente um objeto remoto;
    - Minimiza tráfego na rede;
    - Segue convenções Java;
    - Serializável;
    - Facilmente acessado por componentes da View;
- Princípios
    - Redução de tráfego na rede;
    - Reduz acoplamento entre camadas;
    - Estado do TO não necessariamente reflete o estado atual do sistema (stale data);

Model View Controller

- Recursos:
    - Pattern associado a interfaces com o usuário e em como ele interage com a aplicação;
    - View pode mudar independente de models e controllers;
    - Model esconde detalhes internos;
    - Model adere a um contrato;
    - Separação do model e controller permite fácil migração usando componentes de negócio remotos;
- Princípios:
    - Sepração de responsabilidades, baixo acoplamento;
    - Aumenta coesão dos componentes individuais;
    - Aumenta complexidade da aplicação;
    - Minimiza impacto de mudanças em outras camadas;

Front Controller

- Recursos:
    - Centraliza o tratamento de requests em um único componente;
    - Controla o fluxo de navegação da aplicação;
    - Pode conter lógica de segurança que determina se o recurso sendo acessado está disponível ao usuário;
    - Baixo acoplamento, geralmente a configuração é feita declarativamente;
    - Difícil de se escrever;
- Princípios:
    - Esconde complexidade, separação de responsabilidades, baixo acoplamento;
    - Aumenta coesão dos controllers;
    - Reduz complexidade da aplicação;
    - Melhora manutenibilidade do código de infraestrutura;

Referências

1. Head First Servlets & JSP, de Bryan Basham, Kathy Sierra e Bert Bates. Editora O'Reilly;
2. SCWCD Exam Study Kit, de Hanumant Deshmukh, Jignesh Malavia e Matthew Scarpino. Editora Manning;
3. Servlet 2.4 Specification;
4. JSP 2.0 Specification;
5. JSTL 1.1 Specification;
6. Java Ranch.