

Resumo do livro Head First - JSP e Servlet

Autor: Wilson Bissi - wbissi@gmail.com

* Importante ressaltar que este material foi usado para meus estudos para obter a certificação OCWCD 1.5 e pode haver erros de ortografia.

Capítulo 03 - WebApp Architecture

O Container fornece: Suporte a Comunicação (conhece os protocolos), Gerenciamento do Ciclo de Vida (controla a vida e morte dos Servlets), Suporte a Multi-Thread (cria uma Thread automaticamente para cada requisição no servlet), Segurança Declarativa (usando o DD), Suporte a JSP (tradução das páginas em Servlets)

O Servlet pode ter 3 nomes: URL (conhecida pelo usuário), o Nome da Servlet (conhecida no DD) e o nome completo da classe Java.

DD: Minimiza a alteração no código fonte, adaptação da aplicação em diferentes ambientes, deixa a manutenção simples e dinâmica para o controle de usuários, pessoas sem conhecimentos de programação conseguem dar manutenção.

Capítulo 04 - Being a Servlet

Nunca deve ser feito o override do método **service()** da Servlet.

Deve ser implementado pelo menos um dos método **doXxx()**.

O container executa múltiplas threads para processar múltiplas requisições para uma simples Servlet.

O método **service()** de Servlet só irá executar depois que a Servlet tiver inicializada completamente.

Um objeto se torna uma Servlet quando o container adiciona o **ServletConfig** e então é chamado o método **init()**.

Os métodos **init()** e **destroy()** executam somente uma vez na vida de uma Servlet.

Servlet: Load - Instanciated - Initialised - Service - Destroy.

Não existe um método doXxx para o método HTTP CONNECT.

HTTP HEAD: Igual o GET, porém não retorna o corpo na response.

HTTP TRACE: Usado para saber o que está sendo enviado para o servidor (debug, teste etc)

HTTP PUT: Adiciona o conteúdo do body dentro da URL requisitada.

HTTP DELETE: Apaga um recurso dentro da URL requisitada.

HTTP OPTIONS: Recupera uma lista de método HTTP em que a URL requisitada poderá responder.

HTTP CONNECT: Faz a conexão para propósitos de tunneling.

Os métodos GET, PUT, HEAD são idempotente.

O método **getParameterValues()** retorna um **array** com os valores, já o método **getParameter()** retorna uma **String**.

Para escrever um caractere usa-se o `PrintWriter w = response.getWriter(); w.println("text");`

Para escrever qualquer coisa usa-se o `ServletOutputStream o = response.getOutputStream(); o.write(byteArray);`

Não é possível fazer um **sendRedirect()** depois de escrever na response.

`redirect`: faz o cliente trabalhar

`request dispatcher`: faz alguma coisa no servidor trabalhar.

O método `response.setHeader(String, String)`.

Capítulo 05 - Being a Web App

Recuperar parâmetros da Servlet: `getServletConfig().getInitParameter("")`; `<init-param>`

Recuperar parâmetros da Aplicação: `getServletContext().getInitParameter("")`; `<context-param>`

ServletConfig: um por Servlet.

ServletContext: um por web app por JVM.

Não é possível atribuir novos valores a estes parâmetros.

O **ServletConfig** possui acesso a um `getServletContext()`.

`javax.servlet.ServletContextListener` - `contextInitialized` e `contextDestroyed` (`ServletContextEvent`).

Somente os atributos de Session devem ser obrigatoriamente `Serializable`.

O `getAttribute()` sempre retorna um `Object`.

O `HttpSessionBindingListener` não é registrado no DD assim como os outros são.

Escopo dos atributos: Context (ServletContext), Request (ServletRequest), Session (HttpSession).
Atributos de Contexto não são ThreadSafe, para deixá-lo como tal, deve ser sincronizado o contexto por ex: `synchronized(getServletContext()) { }`, mas para isso o outro código que acessa os atributos do mesmo contexto também deve estar sincronizado. Sincronizar o método `service()` ou `doXXX()` não são boas práticas e não funcionam.
Atributos de Sessão não são ThreadSafe, pois o cliente pode abrir dois browsers, dessa forma é necessário sincronizar o HttpSession exemplo: `synchronized(req.getSession()) { }`.
Somente os atributos de request e variáveis locais são ThreadSafe. As variáveis de instâncias somente serão ThreadSafe se a Servlet implementar a interface `SingleThreadModel`.
Os atributos da request passados pelo RequestDispatcher são visíveis pela pagina que foi direcionada.
Há duas maneiras de se obter o RequestDispatcher pela request (`req.getRequestDispatcher("")`) podendo ou não iniciar com "/" ou contexto (`getServletContext.getRequestDispatcher("/")`) sendo NECESSÁRIO iniciar com a barra)
ServletConfig não armazena atributos.
Se a Servlet não implementar a `SingleThreadModel`, o container não irá criar mais de uma instância por JVM

Capítulo 06 - Conversational State

O objeto HttpSession mantém o estado de conversação entre várias requests do mesmo cliente.
Na primeira request do cliente, o container irá gerar um identificador único para o cliente **JSESSIONID**.
O método `HttpServletRequest.getSession(true)` é a mesma coisa que `HttpServletRequest.getSession()`, se não houver um session, esta será criada.
`HttpServletRequest.getSession(false)` será retornada uma session pré existente ou null caso não exista.
Quando o cliente desabilitar o Cookie o método `HttpSession.isNew()` sempre irá retornar **true**.
O container irá usar o Cookie primeiramente, caso não consiga será usado o URL-Rewriting.
URL-Rewriting é automática, mas somente se você encoded a URL usando `HttpServletResponse.encodeURL("")` ou `HttpServletResponse.encodeRedirectURL("")`. Na JSP pode ser usado o `<c:url/>` para realizar este trabalho.
Existem três formas em que a Session pode morrer: Time out; chamar o método `invalidate()` da session; se aplicação cair.
Ao tentar recuperar um atributo de uma Session ou chamar algum outro método como o `isNew()` após ela ter sido invalidada, será lançada uma `IllegalStateException`.
`HttpSession.setMaxInactiveInterval(0)` : Causa o timeout imediato da Session.
`HttpSession.setMaxInactiveInterval(-1)` : A Session nunca irá expirar por timeout.
Normalmente as Cookies são removidas quando o cliente fecha o navegador, porém elas podem ser persistidas chamando o `Cookie.setMaxAge(int)` definido em **segundos**, se for **-1** ela será removida quando o navegador fechar.
Cookie pode ser usado para armazenar pares de Strings name/value entre o servidor e o cliente.
Header são passados name/value String/String usando o `addHeader` ou `setHeader`.
Cookies são passados name/value String/Cookie usando somente o `addCookie`.
Enquanto o cliente não enviar o JSESSIONID pela request, a Session é considerada nova.
`HttpSessionBindingListener` notifica se a classe que o implementa, foi adicionada ou removida da session.
Somente os objetos e seus atributos da Session são movidos de uma JVM para outra na migração.
Existe um ServletContext por VM
Existe um ServletConfig por Servlet por VM
Existe somente um HttpSession para dado ID em toda aplicação
Qualquer coisa exceto HttpSession é duplicado para outra VM
O container não requer que os objetos usem serialização mas não é garantido. Se os atributos implementarem `Serializable`, seus valores serão transferidos para a outra JVM.
O `HttpSessionActivationListener` deve ser configurado no DD, o método `sessionWillPassivate` é invocado antes da migração acontecer e o `sessionDidPassivate` é executado depois para restaurar os campos.
Os atributos da Session são disponíveis para qualquer outro Servlet que pertença ao mesmo ServletContext.

Não existe um método `getCookie(name)`, eles devem ser retornados como array com o método `req.getCookies()`

Capítulo 07 - Being a JSP

A JSP é **criada** (MyJSP.jsp) em seguida **traduzida** (MyJSP_jsp.java) depois **compilada** (MyJSP_jsp.class) e por fim **carregada** como uma **Servlet** (MyJSP_jsp > Servlet).

Há três tipos de diretivas (page, include, taglib), elas fornecem informações durante a tradução da página. Scriptlets são código Java normal e por isso devem finalizar com (;), toda variável declarada nele é LOCAL.

Expression nunca há (;) no final da declaração, pois esse código vai como argumento de `out.print(...)`;

Todo código scriptlet e expression ficam dentro do método **service**.

Declaration define variáveis/métodos no escopo da classe (instância)

API	Implicit Obj	JspException	exception
JspWriter	out	ServletContext	application
HttpServletRequest	request	ServletConfig	config
HttpServletResponse	response	PageContext	pageContext
HttpSession	session	Object	page

Comments: // ...

HTML Comments: <!-- ... -->

JSP Comments: <%-- ... --%>

Não podemos sobrescrever o método `_jspService()`, somente o `jspInit()` e `jspDestroy()`.

Erros de sintaxe JSP é pego na fase de **TRANSLATE**

Erros de sintaxe Java é pego na fase de **COMPILE**

Translate e Compile acontece somente uma vez.

Os parâmetros de inicialização para um `<jsp-file>` dentro da tag `<servlet>` pode ser obtido dentro do método `jspInit()` através do `getServletConfig().getInitParameter(String)`.

	Servlet	JSP
Application	<code>getServletContext.setAttribute("foo", obj);</code>	<code>application.setAttribute("foo", obj);</code>
Request	<code>request.setAttribute("foo", obj);</code>	<code>request.setAttribute("foo", obj);</code>
Session	<code>request.getSession().setAttribute("foo", obj);</code>	<code>session.setAttribute("foo", obj);</code>
Page	---	<code>pageContext.setAttribute("foo", obj);</code>

Com o PageContext é possível obter atributos de qualquer escopo (o `getAttribute(String)` recupera o atributo do escopo de page, ao usar o `findAttribute(String)` ele recupera o primeiro atributo encontrado na sequência page, request, session e application.

diretiva page: define propriedades específicas da página. `isThreadSafe="default"` é `true` dessa forma não é necessário implementar `SingleThreadModel`. Se `erroPage=""` for definido, então o `isErroPage=""` deve ser `true`.

diretiva taglib: define as bibliotecas de tag disponíveis no JSP.

diretiva include: define o texto que será adicionado na página corrente no tempo de tradução.

O propósito da EL é deixar as chamadas para o código Java mais simples.

Scripting elements: scriptlets, expressions ou declarations

Ao utilizar `scripting-invalid`, todos os `scription elements` serão desconsiderados.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

Para ignorar as EL é usado `el-ignored`, mas o atributo de page `isELIgnored=""` tem prioridade sobre o do DD

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>
</jsp-config>
```

Learn With Questions

- Standard action: `<jsp:xxx>` e Custom action: `<c:xxx>`

- Expression `<%= xxx %>` é diferente de EL Expression `#{ xxx.ss }`
- EL Expression vão dentro do método **service()**
- Expression vão dentro do método **write()** dentro de **service()**
- Scriptlet vão dentro do método **service()**
- O método `jspInit()` tem acesso a `ServletConfig` e a `ServletContext`

Capítulo 08 - Script-free pages

Expressões de scripting (`<%= ... %>`) são adicionadas como argumento dentro do método `out.write()` e por isso não deve ter `;`

`<jsp:xx>` é uma standard action

`<jsp:useBean>` irá criar um novo objeto caso ele não seja encontrado no escopo informado.

O corpo de `<jsp:useBean>` só será invocado na criação de um novo objeto, ou seja, senão foi encontrado um atributo com objeto que tenha o `id=""` e `scope=""` definido (o escopo padrão é **page**)

Se o `type=""` for usado sem declarar um `class=""`, então o atributo deverá existir no escopo, caso contrário será lançado um `InstantiationException`.

A classe declarada em `class=""` não pode ser abstrata e deve conter um construtor sem argumentos

Pode ser adicionado scripting dentro das standard actions

O `param=""` de `<jsp:setProperty />` envia o valor para a propriedade através do valor do parâmetro de request

Se o nome do parâmetro de request bater com o da propriedade não é necessário especificar o `value=""`

Se utilizar o `property=""` de `<jsp:setProperty />` todos os parâmetros que baterem com as propriedades irão receber o valor do parâmetro.

A ação `<jsp:setProperty />` converte parâmetros de String para int e os envia ao bean, mas isso não funciona pra scripting

A ação `<jsp:getProperty />` só nos dá acesso as propriedades do bean e nada mais

EL possibilita o acesso a propriedades das propriedades. O primeiro nome na expressão deve ser um atributo (do escopo `page`, `request`, `session`, `application`) ou um objeto implícito (`pageScope`, `requestScope`, `sessionScope`, `applicationScope`, `param`, `paramValues`, `header`, `headerValues`, `cookie`, `initParam`, `pageContext`).

Ao usar o operador `(.)` conseguimos acesso a propriedades do Bean ou valores do Map: `#{person.name}` onde o nome pode ser a chave do Map ou uma propriedade do Bean.

O operador `([])` além de ser usado para Map e Bean, pode ser usado para Array ou List. Se estiver entre aspas o valor poderá ser a chave do Map, uma propriedade do Bean ou um índice do Array ou da List. `#{musicList["faixa"]}`

Se for um List ou Array o da esquerda, a String dentro de `#{musicList["1"]}` será convertida em int automaticamente

Se for um Map ou Bean o da esquerda, a String dentro de `#{musicList["name"]}` será a chave/propriedade, porém se for informado sem aspas, `#{musicList [name]}` o container irá procurar pelo valor do atributo `name` e esse valor será imputado para ser a chave/propriedade.

Pode ser usado expressões herdadas como `#{musicList[musicType[0]]}`

O objeto implícito de EL **param** recupera o valor simples do parâmetro ou o primeiro caso tenha mais de um, para recuperar todos os valores deve ser usado o **paramValues**. O mesmo acontece para o **header**.

O objeto implícito de EL `requestScope` contém todos os atributos de request e não o objeto request.

`#{requestScope["foo.person"].name}` ao invés de `#{foo.person.name}`

Pode-se criar uma função EL, adicionando um arquivo TLD dentro de `/WEB-INF` e chamando-a na JSP como `#{mine.rollIt()}`

EL são null amigáveis, somente o MOD irá lançar uma exceção se o valor a direita for zero. Em aritmética null é tratado como zero, em lógica null é tratado como false.

O operador `([])` é mais poderoso que `(.)`

A diretiva include `<%@ include file="" %>`

Copia o conteúdo de uma página e cola em outra.

Possui o atributo `file=""`

Acontece no tempo de tradução

Usado para incluir arquivos estáticos

É sensível a posição dentro da página

A standard action include <jsp:include page="">

Possui o atributo page=""

O mecanismo **<jsp:include>** pode incluir elementos dinâmicos (código JSP como EL expressions)

O código não é copiado estaticamente para a página que a declarou, e sim invocada pelo include.

Acontece no tempo de execução

É sensível a posição dentro da página

Pode receber parâmetros com a action <jsp:param name="" value="" /> essa tag informa os parâmetros de request

A action <jsp:forward> também pode receber parâmetro como a include. No forward nada irá aparecer na página depois que ele for chamado, pois ele irá limpar o buffer, ficando somente a página no qual foi redirecionado.

Não deve existir tags como <html> ou <body> dentro de uma página a ser inserida em outra.

A extensão .jspx é relacionada a um fragmento

Learn With Questions

- Ao tentar acessar uma propriedade com `#{initParam.master-mail}` somente o master será considerado, para estar correto deveria ser `#{initParam["master-mail"]}`
- Não é possível fazer soma com String `#{list['listIndex' + 1]}` a forma correta é `#{list[listIndex + 1]}`
- Ao invocar uma expressão `#{param.middle}` onde o middle não exista, não será escrito null, mas sim vazio.
- A standard action include não pode ser usada para incluir arquivos binários a página JSP
- O operador (.) é usado para acessar um propriedade do bean, porém se esta não existir uma exception será lançada.

Capítulo 09 - Custom tags are powerful

EL e Standard Actions são limitadas, neste caso temos as Custom Tags

`<c:forEach var="variável iterada" items="lista de objetos" varStatus="instância de javax.servlet.jsp.jstl.core.LoopTagStatus onde é possível obter o contador a partir do .count">`

Pode haver um forEach dentro do outro

A variável declarada em var="" é somente do escopo da Tag forEach

A Tag `<c:if test="">` é usado para blocos condicionais, porém não existe um else para ele.

A Tag `<c:choose>` e seus parceiros `<c:when>` e `<c:otherwise>` mas a `<c:otherwise>` não precisa estar dentro da `<c:choose>`

A Tag `<c:set>` pode ser usada de duas maneira, usando o var="" para setar variáveis de atributos ou usando o target="" para setar uma propriedade do Bean ou um valor de um Map. O scope="" é opcional por padrão é o **page**.

Ao invés de definir um valor em value="", este poderá ser incluído no corpo da Tag que irá representar a mesmas coisa.

Se o valor setado for null, a variável será removida.

O valor inserido em target="" deve ser um Objeto e não um id de String.

O valor pode receber uma EL expression, scripting expression (`<%= ... %>`) ou `<jsp:attribute>`.

Não pode ter var="" e target="" ao mesmo tempo

Se o target="" for nulo o Container irá lançar uma exception, e se não for um Map ou um Bean também será lançada uma exception

Se o target="" for um objeto e a property="" definida não for encontrada, será lançada uma exception

Se não for definido o scope="" para uma declaração de target="" ou var="" o container irá procurar nos escopos na ordem page, request, session, application. Usando o var="" se o atributo não for encontrado um novo será criado.

`<c:remove var="pode ser uma String e não uma expressão" scope="o default é page">` remove um atributo

A diretiva include é estático e realizada no tempo de tradução, a standard action `<jsp:include>` é dinâmica e realizada no tempo de request, `<c:import>` também é dinâmico e realizada no tempo de request.

`<c:import>` se diferencia do `<jsp:include>` por que o mesmo pode importar de uma URL fora do container no tempo de request.

Também é possível passar parâmetros assim como no include, usando a Tag `<c:param name="" value="">`. A ação `<c:url value="">` realiza a URL Rewriting automaticamente adicionando o `jsessionid` no fim do valor da url. Caso seja necessário usar uma Query String na url, o mesmo deverá passar os parâmetros por `<c:param name="" value="">` que será feito o encode automaticamente, se passar os atributos na Query diretamente, você deverá fazer o encoding antes de enviar.

`<%@ page isErrorPage="true" %>` define que a página é designada para tratar erro. Ao definir uma `erroPage` na diretiva `page` `<%@ page errorPage="error.jsp" %>` irá sobrescrever o que foi definido no DD onde pode ser definido pelo tipo de exceção `<exception-type>` ou pelo HTTP status `<error-code>` onde o `<location>` deve sempre iniciar com (/). Dessa forma, o objeto implícito `exception` do scriptlet ou do JSP através de `getContext.exception()`

`<c:catch var="">` cria um novo atributo com a `exception` dentro do escopo de `page` usado para capturar exceções, nada será executado dentro desse atributo após a exceção ser lançada.

Um simple tag handler estende de `SimpleTagSupport`

Se o `<rtexprvalue>` for `false` ou omitido poderá ser usado somente literal String como valor. Se for `true`, pode ser usado EL expression, Scripting expression ou `<jsp:attribute>`.

O `<jsp:attribute>` pode ser adicionado dentro do corpo da tag, mesmo quando ela for declarada como `empty` (é a única coisa que pode estar no corpo de uma tag com `body empty`).

empty: não tem corpo

scriptless: não deve ter scripting elements (scripting expression, scriptlets e declarations), mas pode ter template text, EL expression, Custom e Standard actions.

tagdependent: o corpo da tag é tratada como texto, onde EL não será avaliada.

JSP: o corpo pode ter qualquer coisa que possa ir dentro da JSP.

Uma tag com corpo `empty` pode ter tag como: `<my:tag />` ou `<my:tag </my:tag>` ou `<my:tag<jsp:attribute></jsp:attribute></my:tag>`

O que importa é a tabela que o container faz entre URI e o arquivo TLD automaticamente, mas se for declarada `<taglib-location>` no DD este será usada.

```
<web-app>
  <jsp-config>
    <taglib-uri>randomThings</taglib-uri>
    <taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
  </jsp-config>
</web-app>
```

O TLD deverá estar em `WEB-INF/` ou subdiretórios, ou dentro do JAR em `META-INF/` ou subdiretórios.

Learn With Questions

- O `var=""` da tag `<c:set>` não pode ter valor de runtime como EL
- O TLD descreve custom tags e funções EL principalmente mas pode declara um Tag File.

Capítulo 10 - When even JSTL is not enough

TagFiles: o arquivo JSP que usamos no `Header.jsp` será chamado de `Header.tag` que deverá ficar no diretório `WEB-INF/tags` (não precisa de TLD) ou dentro do JAR em `META-INF/tags` (precisa de TLD), para importar essa funcionalidade é usado a diretiva `taglib` com o `tagdir="WEB-INF/tags"` e o nome da Tag por padrão será o nome do tag file `<xx:Header>`

Ao invés de `<jsp:include page="Header.jsp">` teremos `<myTag:Header/>`

Ao invés de usar `<jsp:param>` 'parâmetros' como fazíamos no include, no TagFile não temos Parâmetros e sim atributos, onde todos eles são do escopo da TAG, depois que a tag fechar não temos mais acesso a eles `<myTag:Header subTitle="xxx" />`

pode ser obtido com `#{subTitle}` usando `TagFile Attribute` ou `#{param.subTitle}` para valores de parâmetros de request

Com custom tags os atributos da Tag eram definidos no TLD em:

```
<tag>
  <description>random advice</description>
  <name>advice</name>
  <tag-class>foo.Advice</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>user</name>
```

```

        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>

```

TagFiles utilizam a diretiva @attribute para defini-los: <%@attribute name="user" required="true" rtexprvalue="true"%>

Dentro do arquivo Header.tag podemos invocar o body através de <jsp:doBody/>

A definição do body-content para um TagFile é definida dentro da diretiva <%@tag body-content="tagdependent" %> o default é scriptless. As outras opções são: empty:vazio e tagdependent: trata como planilha de texto.

TagFile usa **JspContext** ao invés de **ServletContext**

Tag handlers vem em duas formas: Simple e Classic

Simple: a classe estende de SimpleTagSupport, implementa o doTag(), e cria o TLD. Para invocar o body da Tag é usado getJspBody().invoke(null), o body vai direto para a response, também pode ser passado um Writer para acessar o conteúdo do body.

Sempre deverá ser sobrescrito do método doTag()

SimpleTag handlers nunca são reutilizadas pelo container, o objeto é sempre inicializado antes de qualquer método ser chamado.

A interface JspTag não possui nenhum método ou atributo.

Caso algum atributo seja declarado, o nome definido no TLD deve bater com o do bean e com o definido na Tag

JspTag < SimpleTag < SimpleTagSupport

JspFragment é um objeto que representa o código JSP, ele pode conter template text, standard e custom tags e EL Expression, mas não **Scriptlet, Declarations ou Scripting Expressions**

Ao lançar uma exceção do tipo **SkipPageException** o resto da Tag e da Page não serão enviados para Response (o processamento da page é parado). É mostrado somente o que está acima de onde ocorreu a exceção. Isso vale somente para a página que é acessada diretamente pela tag.

O método setJspBody() só será invocado se a Tag tiver um body preenchido <foo:bar /> e <foo:bar></foo:bar> são considerados vazios

TagFiles implementam funcionalidade de Tag usando uma page, enquanto tag handlers implementam funcionalidade de Tag usando Java

Para fazer o deploy de uma Simple tag handler, deve ser criado um TLD que descreve a tag dentro do elemento <tag>

O método **doTag()** pode setar os atributos que serão usados no corpo da tag, pela chamada de getJspContext().setAttribute() seguido por getJspBody().invoke()

O método **doTag()** declara **JspException** e **IOException** e não possui retorno

Classic: a classe estende de TagSupport, implementa doStartTag(), tem acesso direto ao objeto **pageContext** que contrasta com o método **getJspContext()** de SimpleTag

O método **doStartTag()** declara somente **JspException** e possui um int de retorno que define o fluxo

Retornar um **SKIP_PAGE** no método doEndTag() é similar ao lançar um SkipPageException

O corpo da Classic tag é avaliado entre as chamadas dos métodos doStartTag() e doEndTag()

Retornar **EVAL_BODY_INCLUDE** no método doStartTag() faz com que o corpo da tag seja avaliado

O método doAfterBody() pode ser chamado mais de uma vez e sem ele não conseguiríamos realizar uma iteração na tag, ele é chamado depois que o corpo é avaliado e antes de chamar o método doEndTag()

Classic tags podem ser reutilizadas pelo Container

Retornos padrão para implementação de TagSupport: doStartTag():SKIP_BODY, doAfterBody():SKIP_BODY, doEndTag():EVAL_PAGE.

O método doStartTag() deverá sempre sobrescrito se você desejar avaliar o body da tag.

O método release() só é chamado quando o container remover a tag, não serve para limpá-la do pool.

Estender de BodyTagSupport nos provê mais dois métodos do ciclo: setBodyContent() e doInitBody(), assim o método doStartTag() pode retornar também o valor **EVAL_BODY_BUFFERED** que agora será seu valor default.

Em uma Tag vazia os métodos setBodyContent() e doInitBody() não serão chamados

Se no TLD a tag é declarada com um corpo empty, o método doStartTag() deve retornar **SKIP_BODY**

Simple tag pode ter Classic tag como pai. Usando o método getParent() uma Classic tag pode acessar as Classic tag pais e Simple tag pode acessar os dois Classic e Simple tag pais.

Se não for retornar EVAL_BODY_INCLUDE no doStartTag() as tags filhas nunca serão processadas.

Para recuperar o primeiro ancestral da tag usamos findAncestorWithClass(this, Foo.class 'a classe da tag que queremos');

Um tag handler não é uma Servlet ou JSP, dessa forma não possui acesso automático aos objetos implícitos. O método **getAttribute(String)** recupera somente o atributo do page scope, já o método findAttribute(String) varre todos os escopos (page, request, session, application) e retorna o primeiro que encontrar.

Learn With Questions

- Podemos ter static e dynamic atributos na SimpleTag;
- Não pode ser usado código de scripting no corpo de uma TagFile;
- TagFile não precisa de um arquivo TLD correspondente;

Capítulo 11 - Deploying your web app

O web.xml deve estar dentro de WEB-INF/

WAR: Web Archive

Nada dentro de WEB-INF/ ou META-INF/ tem acesso direto, se vier um request para um recurso nessa pasta, o container irá retornar um HTTP 404.

O container sempre procura por classes dentro de WEB-INF/classes antes de procurar dentro do JAR em WEB-INF/lib

Para acessar algum recurso dentro do JAR deve ser feito por métodos do classloader. O ServletContext acessa os recursos da webapp que não estão dentro de um JAR

A URL dentro de <url-pattern> no <servlet-mapping> é lógica e não precisa existir na app.

Para o batimento da URL, é executado a seguinte seqüência: **exatamente a URL, diretório, extensão.**

Não importa em qual diretório é feito a request, o container sempre irá olhar para os arquivos de <welcome-file-list> do DD.

O <error-page> pode ser sobrescrito individualmente na JSP pelo atributo errorPage="" da diretiva page; <error-code> e <exception-type> não pode aparecer juntos.

O método HttpServletResponse.sendError(int) pode emular um error-code para o container.

O atributo <load-on-startup> de <servlet> define a ordem de carregamento da servlet pelo container, caso tenha valores repetidos a ordem será a que está no DD, os valores devem ser **maiores que zero.**

Referenciando um EJB Local (que executa na mesma JVM)

```
<ejb-local-ref>
  <ejb-ref-name>ejb/Custom</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>foo.CustomHome</local-home>
  <local>foo.Custom</local>
</ejb-local-ref>
```

Referenciando um EJB Remote (que executa em JVM separadas)

```
<ejb-ref>
  <ejb-ref-name>ejb/Custom</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>foo.CustomHome</home>
  <remote>foo.Custom</remote>
</ejb-ref>
```

Env entry são como constantes em tempo de deploy usadas pela APP

```
<env-entry>
  <env-entry-name>rates/discount</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type> (não pode ser primitivo)
  <env-entry-value>10</env-entry-value> (é pego como String, ou Character qdo for o type)
</env-entry>
```

Configurando mapeamento dentre extensão e mime type

```
<mime-mapping>
  <extension>mpg</extension> (sem usar o .)
  <mime-type>video/mppeg</mime-type>
</mime-mapping>
```

O arquivo TLD deve estar em um diretório ou subdiretório de META-INF/ qdo for um JAR ou em WEB-INF/ se não for um JAR.

Learn With Questions

- O container não descobre automaticamente o TLD se ele estiver dentro de WEB-INF/classes ou em WEB-INF/lib
- Os elemento do DD <ejb-ref> e <resource-ref> fornecem acesso JNDI para componentes JEE
- Dependências das bibliotecas são definidas no arquivo MANIFEST.MF

Capítulo 12 - Keep it secret, keet it safe

Os 4 conceitos: autenticação, autorização, confidencialidade e integridade dos dados.

Pode haver mais de um <web-resource-collection> dentro de <security-constraint>

O <auth-constraint> é aplicado a todos <web-resource-collection> dentro de <security-constraint>

O <auth-constraint> define quais roles serão permitidas pra fazer requisições restritas

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name></web-resource-name>

    <url-pattern>/Beer/AddRecipe/*</url-pattern>
    <url-pattern>/Beer/ReviewRecipe/*</url-pattern>

    <http-method>GET</http-method> (Define os métodos que serão restritos as URLs, sendo liberadas somente para
                                     as roles abaixo)
    <http-method>POST</http-method> (Se nenhum método for definido, nenhum método será permitido)
  </web-resource-collection>

  <auth-constraint>
    <role-name>Admin</role-name>
    <role-name>Member</role-name>
  </auth-constraint>
</security-constraint>
```

Regras do <role-name>: se for definida como <role-name>*</role-name> todas os usuário são permitidos. Se existe <auth-constraint>, mas a <role-name> não existir, então nenhum usuário será permitido.

Regras do <auth-constraint>: se <auth-constraint> existir, o container deve fornecer autenticação para as URLs associadas, se <auth-constraint> não existir o container deve permitir acesso não autenticados para as URLs. Se <auth-constraint /> for vazia, ninguém terá acesso.

Quando dois <auth-constraint> preenchidos são aplicados para os mesmos recursos restritos, o acesso será feito a partir da união delas.

<role-name> * combinado com qualquer outra, fornecerá acesso a todos

<auth-constraint> vazia combinado com qualquer outra, bloqueará o acesso de todos

<security-constraint> com nenhum elemento <auth-constraint> combinado com qualquer outra, fornecerá acesso a todos

As roles também pode ser especificados diretamente no DD

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

O HttpServletRequest, fornece 3 métodos para realizar a segurança programática

getUserPrincipal(): principalmente usado com EJBs

getRemoteUser(): Verificar o status da autenticação

isUserInRole(): Verifica se o usuário pertence a role informada, se o usuário não estiver autenticado é retornado false, usado para testar o conteúdo de <security-role-ref>.

O <security-role-ref> sempre terá prioridade sobre <role-name>

Os quatro tipos de autenticação: **BASIC** (codificado em base64, as informações não são criptografada), **DIGEST** (não é requerido que o Container implemente esse tipo, é mais seguro que o basic), **CLIENT-CERT** (extremamente seguro, transmitindo usando Public Key Certificates PKC, o cliente precisa ter um certificado por isso é usado em negócios entre empresas), **FORM** (deixa você criar seu próprio form, é transmitido com o mínimo de segurança comparado aos todos tipo, enviado via request HTTP. necessário o **j_security_check, j_username, j_password**).

BASIC, DIGEST e CLIENT-CERT: usando um pop-up padrão do browser.

Ao usar o FORM como autenticação, você deve ter certeza que SSL ou Session Tracking (Cookie) esteja ativo no container, ou não será reconhecido essa autenticação

Data integrity define se o dado chegou da mesma forma que este foi enviado

Data confidentiality define se ninguém consegue visualizar os dados transmitidos

```
<security-constraint>
  ....
  <user-data-constraint>
    <transport-guarantee>NONE | INTEGRAL | CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

NONE: nenhuma proteção

INTEGRAL: os dados não pode ser alterado ao longo do caminho

CONFIDENTIAL: os dados não pode ser vistos por ninguém ao longo do caminho

via SSL antes de solicitar o login do usuário, o container pede para que ele envie a requisição por HTTPS

Learn With Questions

- Os mecanismos Authorization (isUserInRole()) e Authentication (getRemoteUser()) podem ser implementados usando métodos da interface HttpServletRequest

Capítulo 13 - The Power of Filters

Filter nos deixa interceptar a request

Filters são modulares e configurados no DD, que por sua vez controla a ordem em que eles serão executados

Filter na Request podem: checar a segurança; reformatar os headers ou os bodies; auditoria ou log

Filter na Response podem: comprimir o stream; adicionar ou alterar o stream; criar uma response diferente junto

Todo filtro deve implementar a interface javax.servlet.Filter

Os métodos do ciclo de vida que são implementados: init(FilterConfig), doFilter(ServletRequest, ServletResponse, FilterChain) e destroy().

Para chamar o próximo filtro, usamos chain.doFilter(ServletRequest, ServletResponse).

Filter pode ser configurado para uma <url-pattern> ou <servlet-name>

Com <url-pattern>, todos os filtros que fizerem o batimento são colocados em ordem e todos serão chamados

Para request que vem a partir de um **forward, include ou request dispatcher**, usamos o elemento <dispatcher> dentro de <filter-mapping>. Pode conter 0 ou 4 elementos, com um dos valores **REQUEST(default), INCLUDE(dispatching de include), FORWARD(dispatching de forward) ou ERROR(chamado pelo error handler)**

Classes Wrappers, ServletRequestWrapper, HttpServletRequestWrapper, ServletResponseWrapper, HttpServletResponseWrapper

No caso de uma compressão de arquivo, extendemos o HttpServletResponseWrapper e sobrescrevemos o método getOutputStream()

Learn With Questions

- A ordem da cadeia de filtros pode ser afetada se for declarado por <url-pattern> ou <servlet-name>
- Wrappers são exemplos do pattern Decorator
- Filtros são executados antes das Servlets
- Filters podem ser usados para criar request ou response wrappers
- As classes Wrappers podem ser usadas por aplicações que não suportam HTTP

Capítulo 14 - Enterprise Design Patterns

JNDI: Java Naming and Directory Interface

RMI: Remote Method Invocation

Pattern de View: MVC, Interception Filter, Front Controller

Pattern de Negócio: Service Locator, Business Delegate, Transfer Object

Business Delegate: Protege o controller do fato de alguns modelos serem remotos.

Age como um proxy; inicia comunicação com um server remoto; manipula os detalhes e exceções da comunicação; recebe a requisição do controller; traduz a request e encaminha ela para o serviço de negócio (via stub). -- Minimiza o impacto na camada Web quando tiver alteração do negócio; reduz o

acoplamento entre camadas; esconde a complexidade; codifica para interfaces; baixo acoplamento; separação de conceitos.

Service Locator: fornece um registro lookups que pode simplificar para todos os outros componentes.

Obtém o contexto inicial; fornece registro lookups; funciona com uma variedade de registros. -- Minimiza o impacto na camada Web quando tiver alteração na localização dos componentes; reduz o acoplamento entre camadas; esconde a complexidade; separação de conceitos.

Transfer Object: Minimiza o tráfego da rede fornecendo uma representação local do componente remoto.

Implementado como objetos serializados; facilmente acessado pela camada web. -- Deixar TO salvo de concorrência é complexo; reduz o tráfego da rede.

Interception Filter: Padrão para modificar a requisição que foi enviada do servlet ou modificar a response que está sendo enviada para o cliente.

Pode interceptar ou modificar a request ou response; são deployados declarativamente; o ciclo de vida é controlado pelo container; deve ser implementado os métodos de callback do container. -- Controle declarativo permite que os Filters sejam facilmente implementados; coesão; baixo acoplamento; aumenta o controle declarativo.

MVC: Cria uma estrutura lógica (Model, View, Controller) para aumentar a coesão de cada componente para prover uma reusabilidade.

View pode ser alterada independente do controller. -- Aumenta a coesão individual dos componentes; minimiza o impacto das alterações nas camadas; baixo acoplamento; separação de conceitos.

Front Controller: Reúne as requisições de processamento dentro de um componente simples, deixando a aplicação menos complexa.

Centraliza a requisição inicial manipulando as tarefas. -- Aumenta a coesão no controller da aplicação; diminui a complexidade da aplicação; esconde a complexidade; baixo acoplamento; separação de conceitos.

Learn With Questions

- Front Controller está relacionado com Intercepting Filter
- O Service Locator pode ser usado também quando você deseja encapsular as dependências do fornecedor
- O Service Locator aumenta o desempenho da rede com o caching
- O principal benefício do Business Delegate é reduzir o acoplamento dentro as camadas de apresentação e negócio