Aplicații Integrate pentru Întreprinderi

Laborator 9

02.12.2013

Realizarea de aplicații web folosind JavaServer Pages

Scopul laboratorului îl reprezintă folosirea mecanismelor oferite de tehnologia JavaServer Pages pentru realizarea de aplicații (cu conținut dinamic) care să ofere utilizatorilor aceeaşi funcționalitate de care ar beneficia prin folosirea unor aplicații care să solicite resurse (programe instalate) pe maşina unde sunt rulate. Aceste cerințe se vor transfera maşinii pe care este găzduită aplicația, funcționalitatea fiind accesibilă printr-un client universal – browser-ul.

1. Tehnologia JavaServer Pages

JSP este o tehnologie pentru realizarea de pagini web generate dinamic şi bazate pe HTML, XML sau alte tipuri de documente. A fost lansată pe piață de compania Sun Microsystems în anul 1999 ca răspuns la tehnologiile PHP şi ASP[1], demonstrând faptul că Java este un limbaj de programare suficient de robust pentru a răspunde la provocările web-ului.

Tehnologia JavaServer Pages e parte integrantă Java Enterprise Edition implementând de regulă intefața cu utilizatorul dintr-o aplicație web Java.

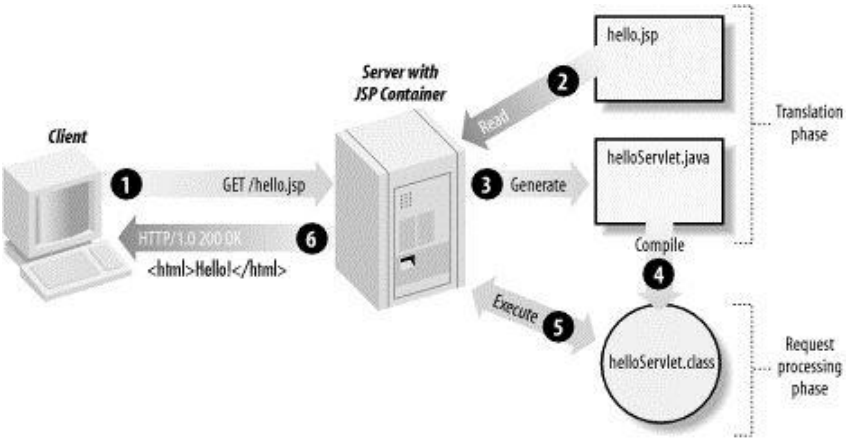
O pagină JSP este un document text care cuprinde două tipuri de date: statice care pot fi descrise în orice format (HTML, XML, SVG, WML) – denumite şi elemente de adnotare (*eng.* markup) şi dinamice, care pot fi directive JSP şi *scripleți*, adică blocuri de cod sursă Java (de obicei cuprinse între adnotările `<%` şi `%>`) folosite pentru implementarea unor funcționalități complexe, cum ar fi, de exemplu, comunicația cu o bază de date, reținerea preferințelor utilizatorilor, accesarea componentelor Java Beans, transferul controlului între pagini şi partajarea informației între cereri, răspunsuri şi pagini.

[1] Dacă tehnologia JavaServer Pages este comparabilă cu PHP (cu excepția eficienței), mai ales de când limbajul a fost dezvoltat în direcția orientării pe obiecte, în cazul celorlalte tehnologii, JSP prezintă o serie de avantaje. Față de ASP .NET, avantajul tehnologiei JSP este derivat din faptul că aceasta este construită peste limbajul de programare Java, mai puternic şi mai uşor de utilizat decât limbajele Microsoft, dispunând de o portabilitate mai mare atât în privința sistemelor de operare cât şi a serverelor de aplicații. JavaScript este util în generarea de cod HTML dinamic la nivelul clientului dar nu şi la nivelul serverului, mai ales când vine vorba de accesarea de resurse existente pe acesta. De asemenea, SSI (Server Side Includes) a fost proiectat pentru incluziuni destul de simple, nu pentru programe cu funcționalități complexe. Utilizarea JSP este recomandată şi față de folosirea exclusivă a Java Servlets întrucât codul devine mult mai uşor de întreținut, realizându-se totodată o delimitare netă între logica aplicației şi nivelul de prezentare.

Funcționalitatea pe care o pune la dispoziție tehnologia JSP este aceeaşi cu a aplicațiilor implementate folosind CGI (Common Gateway Interface), distingându-se de acestea prin performanță îmbunătățită (posibilitatea integrării de elemente dinamice în pagina HTML în loc să se folosească fişiere separate), compilarea înainte de dezvoltarea în contextul serverului[2], posibilitatea accesării tuturor funcționalităților oferite de interfețele de programare Java (JDBC, JNDI, JAXP, EJB), utilizarea unui model care implică delegarea logicii aplicației către Java Servlets.

Avantajele utilizării JavaServer Pages includ: separarea logicii aplicației de prezentare[3] (întreținerea este mai uşoară atât din partea programatorilor, cât şi a dezvoltatorilor web), portabilitate (asigurată de limbajul de programare Java – aplicația web poate fi utilizată pe mai multe platforme cu arhitectură diferită fără a fi necesară modificarea lor), conținutul dinamic care poate fi generat precum şi faptul că se bazează pe tehnologia Java Servlets[4], astfel încât preia şi avantajele pe care le oferă aceasta.

Deşi din punctul de vedere al programatorului un Java Servlet reprezintă o clasă Java (unde generarea elementelor de adnotare se face prin apelarea unor metode) iar o pagină JSP se aseamănă mai mult cu un document (unde conținutul static este separat de conținutul dinamic – generat de componente Java Beans sau etichete JSP), tehnologiile sunt echivalente. O pagină JSP este transformată într-un Java Servlet (care este compilat) responsabil de comunicația cu clientul şi de toate celelalte prelucrări. Modificările realizate asupra unei pagini JSP sunt detectate în mod automat, astfel încât atât procesul de transformare într-un Java Servlet cât şi procesul de compilare corespunzător sunt reluate atunci când se face o cerere pentru aceasta.



Interacțiunea dintre client şi server pentru prelucrarea unei pagini Internet folosind JavaServer Pages

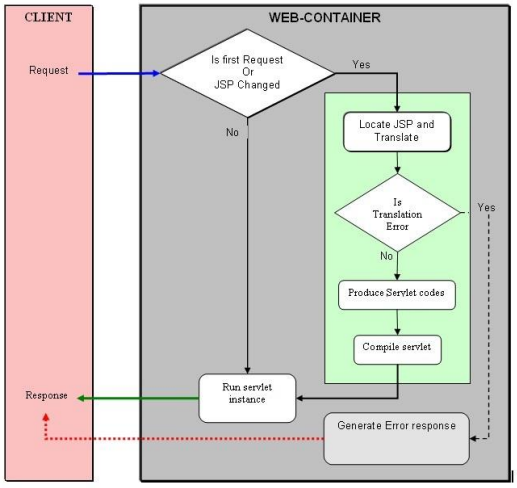
Atunci când ❶ un client transmite (din browser) o cerere HTTP către server pentru o pagină .jsp, ❷ aceasta este transmisă mai departe motorului JSP care încarcă de pe disc resursa care a fost indicată prin intermediul URL-ului.

[2] În cazul CGI/Perl este necesar ca serverul să încarce un interpretor şi un script de fiecare dată când o pagină este solicitată în browser.

[3] Logica aplicației este folosită pentru a genera conținutul dinamic şi poate fi realizată prin componente Java Beans, în timp ce interfața cu utilizatorul este formată prin etichete JSP.

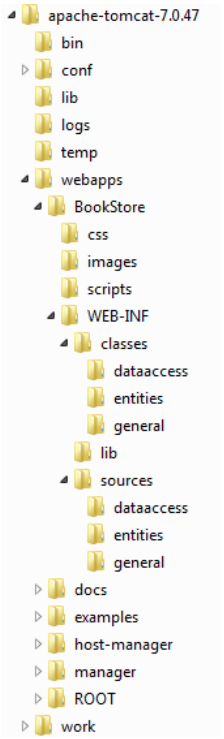
[4] Tehnologia JavaServer Pages reprezintă o abstractizare de nivel înalt a Java Servlets. Totodată, în pagina JSP poate fi reutilizat codul care a fost dezvoltat într-un Java Servlet.

Dacă pe server nu există un Java Servlet corespunzător celei mai recente versiuni a paginii respective, ❸ acesta este generat (prin instrucțiuni de tip println ce includ etichetele respective şi convertirea elementelor JSP în cod Java care implementează comportamentul dinamic al paginii) şi ❹ apoi compilat într-o clasă executabilă. Acest cod, căruia i se pasează cererea originală ❺ este rulat de motorul Java Servlet producând un rezultat în format HTML care este transmis serverului ca răspuns HTTP, acesta ❻ fiind pasat clientului care îl afişază în browser, ca şi când ar fi conținut static.



Mecanismul de generare al unui Java Servlet dintr-o pagină JSP

2. Integrarea JavaServer Pages cu serverul web Apache Tomcat 7.x



Structura de directoare în serverul web Apache Tomcat 7.x

În cazul unei aplicații web folosind tehnologia JSP, aceasta va fi plasată în directorul `webapps`.

Paginile `.jsp` se vor găsi în rădăcina directorului unde a fost dezvoltată aplicația web. Pagina care va fi afişată automat atunci când se cere adresa: `http://localhost:8080/<locatie>/` este `index.jsp` sau cea precizată prin elementul `<welcome-file>` din secțiunea `<welcome-file-list>` a fişierului de configurare `web.xml`. Acesta va fi plasat în `WEB-INF`, specificându-se diferiți parametrii ai aplicației web (inclusiv despre clasele Java Servlet ce implementează logica aplicației). De asemenea, în condițiile în care sunt folosite metode din clase Java, acestea vor fi plasate într-un director `WEB-INF/classes`. Este recomandat ca acestea să fie modularizate pe pachete pentru ca referirea lor să se poată realiza cu uşurință. Dacă se folosesc biblioteci speciale[5] (în arhive `.jar`), acestea trebuie incluse în `WEB-INF/lib`. Deşi sursele aplicației se pot găsi în orice fişier, este recomandat ca acestea să se găsească tot aici, într-un subdirector denumit `src` sau `sources`, astfel încât să poată fi identificat cu uşurință, iar procesul de compilare al surselor şi de transfer al claselor (realizat înainte de operația de configurare – *eng.* deploy a aplicației web) să nu implice un efort deosebit.

[5] Astfel de biblioteci speciale pot fi „driver”-ul de conectare la baza de date precum şi biblioteca pentru accesarea funcționalităților oferite de JSTL (JavaServer Pages Standard Tag Library).

Încărcarea tuturor acestor clase se face atunci când este pornit serverul:

```
Dec 2, 2013 8:00:00 AM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory
ApacheTomcat\apache-tomcat-7.0.47\webapps\BookStore
```

Spre diferență de Java Servlets, nu este necesară realizarea configurării aplicației web atunci când sunt realizate modificări asupra paginilor JSP, întrucât ele sunt detectate automat, determinând generarea unor Java Servlets corespunzătoare atunci când paginile sunt accesate de către client. Repornirea serverului Apache Tomcat va fi necesară doar în cazul operării unor modificări pentru clasele Java ce sunt utilizate de aplicație, fiind necesară reîncărcarea lor (după ce compilarea lor este realizată de către programator).

Clasele Java Servlet generate pentru fiecare aplicație pot fi vizualizate în `%CATALINA_HOME%\work\Catalina\<adresaserver>\<locatie>\org\apache\jsp`.

Corespondentul unei pagini `fisier.jsp` va fi un Java Servlet `fisier_jsp.java`. Tot aici vor fi plasate şi clasele obținute prin compilarea acestor Java Servlets. Atât operația de transformare din pagină JSP în Java Servlet cât şi compilarea claselor Java Servlet generate este realizată automat de către serverul Apache Tomcat.

Datorită faptului că o cerere pentru o pagină JSP implică (în unele situații) transformarea într-un Java Servlet precum şi compilarea acestuia, vizualizarea acesteia se va face după o perioadă mai mare decât în situația când aceste prelucrări nu mai trebuie realizate. De asemenea, trebuie avută în vedere situația în care o astfel de operație eşuează, astfel încât în browser sunt afişate excepțiile generate în loc de conținutul paginii solicitate.

Versiunea Apache Tomcat 7.0.47 suportă specificația JSP 2.2 şi EL 2.2.

3. Ciclul de viață al unei pagini JSP

O pagină JSP tratează cererile asemenea unui servlet. Acesta este motivul pentru care ciclul de viață (ca de altfel şi multe alte capabilități) al paginilor JSP (în special aspectele dinamice) este asemănător cu cel al tehnologiei Java Servlet.

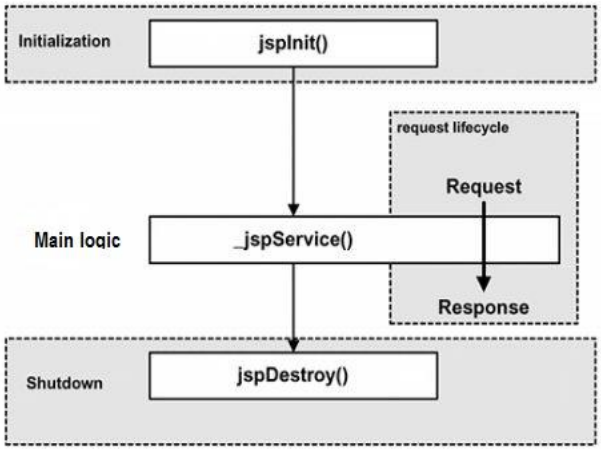
În plus față de etapele caracteristice Java Servlets (creare, inițializare, gestiune cereri şi răspunsuri, distrugere), mai există şi etapa de compilare. Atunci când o cerere este asociată unei pagini JSP, aceasta este tratată de un servlet special care verifică dacă servlet-ul asociat paginii JSP este mai vechi decât pagina în cauză, situație în care îl generează şi îl compilează[6].

Odată ce pagina JSP a fost transformată şi compilată, servlet-ul corespunzător va urma ciclul de viață corespunzător (încărcarea claselor, instanțierea lor, apelul metodei `init()` pentru inițializarea resurselor (partajate) şi încărcarea unor fişiere de configurare, execuția (interacțiunea cu clientul şi realizarea procesărilor prin metoda `service()`), precum şi apelul metodei `destroy()` dacă servlet-ul nu mai este necesar). Metodele (generate) se vor numi, în acest caz, `_jspInit`, `_jspService`, `_jspDestroy`[7].

[6] Un avantaj pe care îl au paginile JSP față de Java Servlets este că procesul de compilare este realizat automat.

[7] Metodele `jspInit` şi `jspDestroy` (apelate o singură dată în ciclul de viață al unei pagini JSP) pot fi suprascrise ca declarații JSP (cuprinse între `<%!` şi `%>`). Nu se recomandă însă suprascrierea metodei `_jspService` întrucât aceasta este generată în mod automat prin transformarea elementelor JSP în codul Java corespunzător, respectiv prin apelul metodei `println` (pe obiectul `PrintWriter` asociat) pentru generarea etichetelor.

```
public void _jspInit() {
    // ...
}
public void _jspDestroy() {
    // ...
}
public void _jspService
            (final javax.servlet.http.HttpServletRequest request,
             final javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException, javax.servlet.ServletException {
    // ...
}
```



Ciclul de viață al unei pagini JSP

Toate aceste metode fac parte din interfața `javax.servlet.HttpJspPage`.

În cadrul procesului de **compilare**, motorul JSP verifică dacă este necesar să compileze pagina respectivă (în situația în care aceasta nu a fost niciodată compilată, sau eticheta de timp a servlet-ului corespunzător nu corespunde celei mai recente modificări), acestă operație fiind precedată de transformarea paginii, ce include şi parsarea acesteia.

Atât procesul de transformare şi cât şi procesul de compilare pot genera erori care sunt scoase în evidență doar atunci când pagina este accesată pentru prima dată.

- dacă eroarea apare în timpul transformării, serverul va întoarce excepția `ParseException` iar clasa servlet va fi incompletă, astfel că ultima linie incompletă va fi un indicator către elementul JSP incorect;
- dacă eroarea se produce atunci când pagina JSP este compilată (dacă avem o eroare de sintaxă în scriptlet), serverul va întoarce excepția `JasperException` precum şi un mesaj care include numele servlet-ului asociat paginii JSP şi linia la care s-a constatat eroarea.

Atunci când se execută o pagină JSP pot apărea excepții, acestea trebuind tratate separat, în cadrul unei pagini dedicate unei astfel de acțiuni:

```
<%@ page errorPage="errorpage.jsp" %>
```

Pentru ca excepția (într-un obiect de tip `javax.servlet.jsp.JspException`) să fie disponibilă în cadrul paginii de eroare pentru a o trata, trebuie specificată, la începutul paginii de eroare, directiva:

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

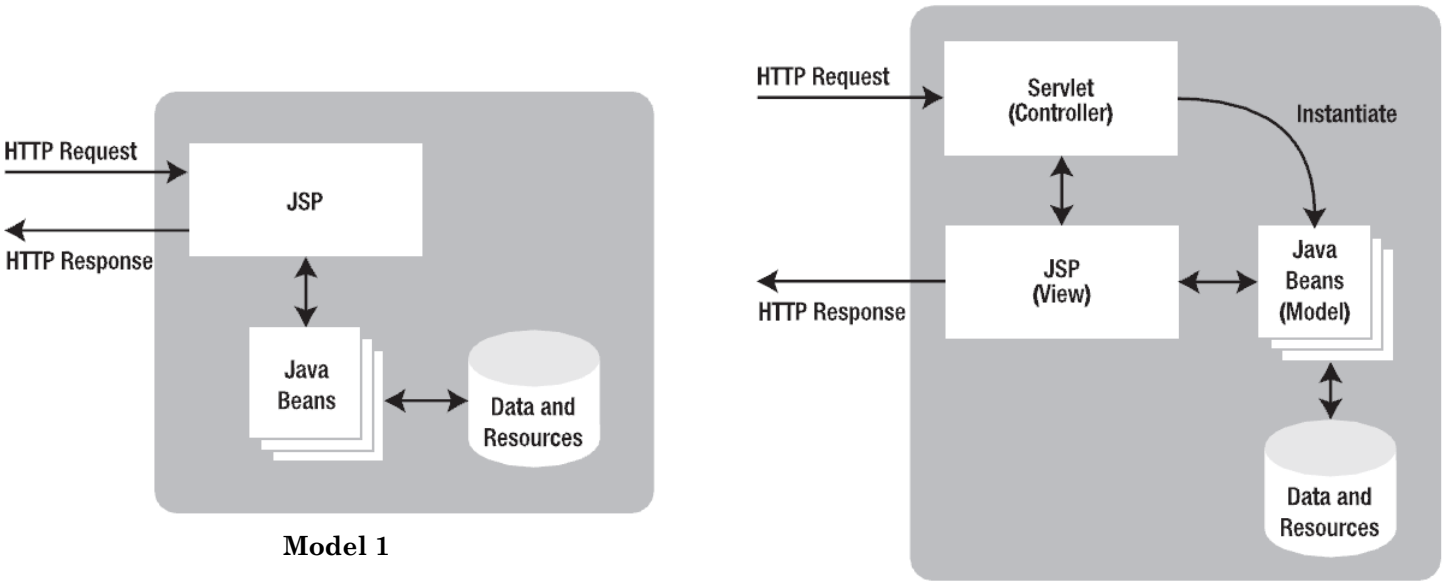
În etapa de **inițializare**, care se execută o singură dată (după crearea servletului asociat paginii JSP), este apelată metoda `_jspInit` unde se realizează conexiunea la baza de date, sunt încărcate diferite resurse şi se fac alte operații ce țin de configurarea aplicației web. În general, aici sunt inițializate obiectele pentru accesarea funcționalităților puse la dispoziție de diferite biblioteci (frecvent, Java Server Pages Standard Tag Library), însă programatorul dispune de posibilitatea de a realiza propriile operații, prin suprascrierea metodei `jspInit` care va fi inclusă în cadrul acestei funcții.

Etapa de **execuție** constă în generarea unui răspuns pentru fiecare cerere prin apelul metodei `_jspService`. Aceasta va trata fiecare dintre metodele HTTP prin care clientul comunică cu serverul (`GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `TRACE`).

În etapa de **distrugere**, corespunzătoare procesului de înlăturare a paginii JSP din container, este apelată metoda `_jspDestroy` unde sunt eliberate resursele utilizate de aplicație, astfel încât acestea să fie procesate de modulul de gestiune a memoriei. În situația în care se dorește eliberarea manuală a acelor resurse care au fost inițializate (conexiunea la baza de date, diverse fişiere de configurare), poate fi suprascrisă metoda `jspDestroy`, conținutul ei fiind copiat în cuprinsul funcției generate.

4. Structura unei pagini JSP

Combinarea de cod Java cu etichete HTML oferă posibilitatea realizării de pagini cu conținut dinamic, însă în cazul aplicațiilor cu funcționalitate complexă, procesul de întreținere poate fi dificil, problema provenind în special din faptul că partea de logică a aplicației (care este dezvoltată de programatori – implicând algoritmi şi interacțiune cu baza de date) nu este separată de partea de prezentare (realizată de regulă de dezvoltatori web – concentrându-se mai ales pe elemente de grafică). Arhitectura aplicației JSP va reflecta raportul dintre aceste aspecte.



Model 1 **Model 2**

Arhitectura aplicațiilor JSP [1]

În modelul 1, logica aplicației este implementată în clase Java (componente Java Beans) şi poate fi apelată din partea de prezentare realizată prin pagini JSP. Această soluție este adecvată pentru cazul în care aplicația nu este foarte complexă, o problemă fiind constituită de faptul că toată comunicația cu clientul trebuie realizată din pagina JSP. O astfel de abordare se mai numeşte client-server, are avantajul simplității şi dezavantajul lipsei de scalabilitate.

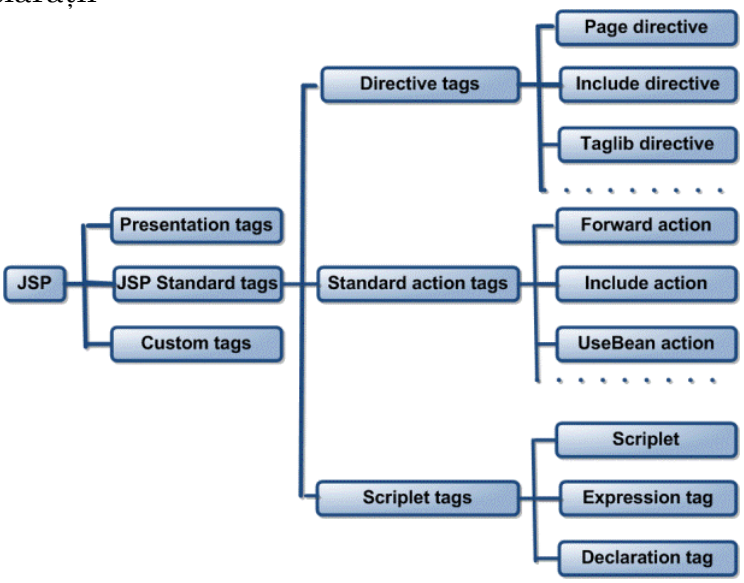
În modelul 2 (denumit şi abordare pe N-niveluri datorită separării arhitecturii serverului pe mai multe niveluri), se respectă şablonul de proiectare Model View Controller unde servlet-ul se ocupă de cererea de la client, implementează logica aplicației, realizând totodată şi instanțierea componentelor Java Beans. Pagina JSP obține date de la componentele Java Beans şi transmite un răspuns către client, prelucrările fiind realizate în mod transparent. Acest tip de abordare e adoptat atunci când aplicația este mai complexă, caracterizându-se prin uşurința de întreținere şi eficiență.

Aşadar, în cadrul unei pagini JSP se utilizează o combinație între etichete HTML (sau XML) şi blocuri de cod sursă Java pentru generarea de elemente dinamice. Fiecare pagină JSP este compilată într-un servlet, acesta primind cererea şi transmițând mai departe răspunsul.

O pagină JSP poate conține etichete de prezentare (în limbajul HTML), etichete JSP standard şi etichete definite de utilizator.

JSP foloseşte ca etichete standard:

- directive
- acțiuni
- scripleți
  - expresii
  - declarații



Structura unei pagini JSP

De asemenea, pot fi folosite *comentarii* care pot fi:

- comentarii JSP: acestea sunt ingorate de motorul JSP;

```
<%-- comentariu; --%>
```

- comentarii HTML: acestea sunt ignorate de browser;

```
<!-- comentariu -->
```

O **directivă** este o instrucțiune care indică informații generale despre pagina respectivă. Aceasta afectează întreaga structură a clasei servlet. Directiva oferă container-ului diverse informații cu privire la modul în care va gestiona anumite aspecte ale procesării paginii JSP.

```
<%@ directive attribute="value" %>
```

Directivele pot avea un număr variabil de perechi atribut-valoare, acestea fiind separate prin virgulă. Spațiile dintre `<%@` şi numele directivei, respectiv dintre ultima pereche atribut-valoare şi `%>` sunt opționale.

Există trei tipuri de directive:

- `<%@ page ... %>` – defineşte atribute specifice paginii JSP, cum ar fi limbajul de programare pentru scripturi, pagina corespunzătoare erorilor şi cerințe legate de stocarea temporară;
- `<%@ include ... %>` – include un fişier în cadrul etapei de transformare a paginii JSP în servletul corespunzător;
- `<%@ taglib ... %>` – defineşte o bibliotecă de etichete, conținând acțiuni definite de utilizator, care vor fi utilizate în cadrul paginii.

Directiva `page` este utilizată pentru a oferi containerului informații despre pagina JSP curentă. Cu toate că această directivă poate fi plasată oriunde în cadrul paginii, se recomandă ca ea să fie inclusă la început.

Sintaxa acestei directive poate lua următoarele forme:

```
<%@ page attribute="value" %>
<jsp:directive.page attribute="value" />
```

Atributele suportate de directiva `page` sunt:

autoFlush	controlează comportamentul buffer-ului asociat servletului dacă se umple (conținutul său va fi golit automat – valoarea `true` (implicită) sau se generează o eroare – valoarea `false`); se foloseşte de regulă în asociere cu atributul `buffer`:
	`<%@ page buffer="8kb" autoFlush="true" %>`
	`<%@ page autoFlush="false" %>`

buffer	specifică un model pentru fluxul de ieşire: valoarea none indică faptul că răspunsul va fi transmis imediat clientului; alte valori specifică dimensiunea maximă (în octeți) a buffer-ului, răspunsul fiind construit în această zonă tampon înainte de a fi transmis clientului
	`<%@ page buffer="none" %>`
	`<%@ page buffer="8kb" %>`

contentType	specifică schema de codificare a caracterelor pentru pagina JSP şi pentru răspunsul generat
	`<%@ page contentType="text/html" %>`
	`<%@ page contentType="text/xml" %>`
	`<%@ page contentType="application/msword" %>`
	`<%@ page contentType="text/html:charset=ISO-8859-1" %>`

errorPage	defineşte un URL către pagina care raportează excepțiile Java netratate, generate la rulare `<%@ page errorPage="errorpage.jsp"%>`

isErrorPage	specifică dacă pagina JSP curentă este indicată de proprietatea `errorPage` a altei pagini JSP
	`<%@ page isErrorPage="false" %>`
	`<%@ page isErrorPage="true" %>`

extends	specifică o superclasă pe care servlet-ul generat trebuie să o extindă `<%@ page extends="myPackage.MyClass" %>`
import	indică o listă de pachete sau de clase Java care vor fi utilizate în pagina JSP; în mod implicit, sunt importate automat `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `javax.servlet.http.*` dacă se doreşte importarea mai multor pachete sau clase (inclusiv definite de utilizator), acestea vor fi separate prin virgulă: `<%@ page import="java.sql.*, java.util.*, java.io.*" %>`
info	defineşte un şir de caractere care poate fi accesat prin metoda `getServletInfo()` a servlet-ului asociat `<%@ page info="Some info on my JSP Page" %>`

isELIgnored	specifică dacă expresiile EL (Expression Languge) din pagina JSP de forma `${...}` vor fi evaluate (valoarea true, implicită) sau vor fi tratate ca text static
	`<%@ page isELIgnored="true" %>`
	`<%@ page isELIgnored="false" %>`

isScriptingEnabled	determină dacă script-urile (scripleți, expresii non-EL şi declarații) sunt permise pentru a fi utilizate (valoarea `true` – implicită) sau nu (valoarea `false`)
	`<%@ page isScriptingEnabled="true" %>`
	`<%@ page isScriptingEnabled="false" %>`

isThreadSafe	defineşte modelul firului de execuție pentru servlet-ul generat; implicit, paginile JSP sunt socotite ca fiind sigure (valoarea `true`) pentru a fi folosite într-un context concurent, iar în caz contrar (valoarea `false`), container-ul se asigură ca pagina să fie accesată de un singur client la un moment dat
	`<%@ page isThreadSafe="true" %>`
	`<%@ page isThreadSafe="false" %>`

language	defineşte limbajul de programare utilizat în pagina JSP
	`<%@ page language="java" %>`
	`<%@ page language="javascript" %>`

session	specifică dacă pagina JSP participă (valoarea `true`) sau nu (valoarea `false`) la sesiuni HTTP, adică dacă poate accesa sau nu obiectul implicit session şi metodele asociate acestuia
	`<%@ page session="true" %>`
	`<%@ page session="false" %>`

Directiva `include` este folosită pentru a include resursa specificată în fişierul curent, în cadrul etapei de transformare a paginii JSP în servlet. Practic, se concatenează conținutul acestor fişiere externe, la locația unde a fost specificată (oriunde în pagina JSP):

```
<%@ include file="myJSPPage.jsp" %>
<jsp:directive.include file="myJSPPage.jsp" />
```

Se consideră că resursa specificată este indicată prin calea sa relativă, astfel încât atunci când ea lipseşte, se consideră că fişierul se găseşte în acelaşi director cu pagina JSP din care este invocat.

O astfel de funcționalitate este folosită mai ales pentru includerea de resurse comune pentru toate paginile Internet cum ar fi: header, footer sau meniuri ale aplicației web.

Directiva `taglib` permite definirea de către utilizator a unor etichete JSP care implementează o anumită funcționalitate, acestea fiind grupate în cadrul unor biblioteci. Se specifică pentru fiecare set de etichete locația la care se află biblioteca respectivă (atributul `uri`), identificând un mecanism (atributul `prefix`) prin care acestea vor fi identificate în cadrul paginii JSP:

```
<%@ taglib uri="uri" prefix="prefixOfTag" %>
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

Pot fi folosite biblioteci standard (care implementează controlul fluxului într-un program, care pun la dispoziție mecanisme pentru gestiunea informațiilor dintr-o bază de date sau care implementează diferite metode uzitate mai frecvent în cadrul oricărei aplicații) sau se pot crea biblioteci definite de utilizator:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="my" uri="http://www.mysite.com/mylibrary" %>
```

Utilizarea unei astfel de etichete se face cu `<prefixName:tagName ...>` unde `prefixName` este valoarea specificată pentru atributul prefix de la definirea bibliotecii respective, iar `tagName` numele unei etichete implementate.

O **acțiune** este un marcaj care specifică comportamentul motorului Java Servlets de la nivelul serverului web, în momentul compilării marcajele fiind înlocuite cu codul sursă Java corespunzător acțiunii.

Sintaxa unei acțiuni este:

```
<jsp:actionName attribute="value" />
```

Atributele comune tututor tipurilor de acțiuni sunt:

- `id`: identifică în mod unic acțiunea respectivă permițând ca aceasta să poată fi referită în cadrul paginii JSP; dacă acțiunea este folosită pentru crearea unei instanțe a unui obiect, prin intermediul identificatorului aceasta poate fi utilizată pe tot parcursul obiectului implicit `PageContext`;
- `scope`: defineşte ciclul de viață al acțiunii, adică vizibilitatea elementului concretizată în posibilitatea de utilizare a identificatorului acestuia; valorile pe care le poate sunt: `page`, `request`, `session` şi `application`.

O acțiune este de fapt o funcție predefinite, tipurile de acțiuni definite de API-ul JSP fiind:

<jsp:attribute>	defineşte atributul unui element XML definit dinamic
<jsp:body>	defineşte corpul unui element XML definit dinamic
<jsp:element>	defineşte elemente XML în mod dinamic

Prin acțiunile **<jsp:attribute>**, **<jsp:body>** şi **<jsp:element>** elementele XML pot fi generate în momentul rulării, ca răspuns la o cerere şi nu atunci când sunt compilate sursele.

```
<jsp:element name="xmlElement">
    <jsp:attribute
     name="xmlAttribute">
         Attribute Value
    </jsp:attribute>
    <jsp:body>
        Body for XML element
    </jsp:body>
</jsp:element>
```

→

```
<xmlElement
 xmlAttribute="Attribute Value">
   Body for XML element
</xmlElement>
```

<jsp:useBean>	găsirea sau instanțierea unei componente Java Bean; se caută dacă este vizibil vreun obiect având identificatorul specificat şi în cazul în care acesta nu este găsit, se încearcă instanțierea clasei indicate; ulterior, se pot folosi acțiunile `setProperty` şi `getProperty` pentru a stabili şi pentru a obține proprietățile componentei JavaBean Atributele asociate acțiunii `useBean` sunt: • `class` – desemnează numele componentei Java Bean, incluzând şi pachetul în care se găseşte aceasta; • `type` – indică tipul variabilei ce va reda obiectul în cauză • `beanName` – specifică numele componentei Java Bean, aşa cum este indicat de metoda `instantiate()` a clasei `java.beans.Beans`. `<jsp:useBean id="myObject" class="myPackage.myClass" />`

<jsp:getProperty>	obținerea valorii proprietății unei componente pe care o converteşte apoi la tipul şir de caractere şi apoi o integrează în fluxul de ieşire
	Atributele obligatorii asociate acțiunii `getProperty` sunt:
	• `name` – desemnează componenta Java Bean a cărei proprietate va fi stabilită; trebuie să aibă aceeaşi valoare cu a identificatorului precizat în acțiunea `useBean`;
	• `property` – indică proprietatea care se doreşte a fi stabilită;
	`<jsp:useBean id="myObject" class="myPackage.myClass" />`
	`...`
	`<jsp:getProperty name="myObject" property="myProperty"/>`

<jsp:setProperty>	stabilirea valorii proprietății unei componente Java Bean; anterior utilizării acestei proprietăți, componenta Java Bean referită trebuie să fi fost definită; poate fi utilizată în cadrul acțiunii `useBean` (situație în care se execută numai dacă a fost instanțiat un obiect nou) sau independent de ea (caz în care se execută atât atunci când a fost găsit un element cu numele respectiv atunci când a fost instanțiat un obiect nou)
	Atributele asociate acțiunii `setProperty` sunt:
	• `name` – desemnează componenta Java Bean a cărei proprietate va fi stabilită; trebuie să aibă aceeaşi valoare cu a identificatorului precizat în acțiunea `useBean`;
	• `property` – indică proprietatea care se doreşte a fi stabilită; valoarea specifică faptul că parametrii cererii ale căror nume se potrivesc cu numele proprietăților componentei Java Beans vor fi transmişi către metodele setter respective;
	• `value` – specifică valoarea care se doreşte asignată proprietății în cauză; dacă valoarea parametrului e `null` sau parametrul nu există, acțiunea este ignorată;
	• `param` – numele parametrului cererii a cărei valoare trebuie să o primească proprietatea; nu se poate folosi concomitent cu `value` (dar cei doi parametrii pot fi omişi);
	`<jsp:useBean id="myObject" class="myPackage.myClass">`
	`  <jsp:setProperty name="myObject"`
	`  property="myProperty" value="myValue" ... />`
	`</jsp:useBean>`
	`<jsp:useBean id="myObject" class="myPackage.myClass" />`
	`...`
	`<jsp:setProperty name="myObject"`
	`property="myProperty" value="myValue" ... />`

<jsp:forward>	pagina JSP curentă îşi încheie activitatea, cererea fiind transmisă către o altă resursă (pagină statică, pagină JSP sau un Java Servlet) Atributul asociat acțiunii `forward` este `page`, prin care se indică un URL relativ al resursei spre care se doreşte a se realiza redirectarea. `<jsp:forward page="myJSPPage.jsp" />`
<jsp:include>	includerea resursei specificate în momentul când pagina JSP este solicitată (spre diferență de directiva cu acelaşi nume, în care includerea avea loc în cadrul procesului de transformare în servlet-ul asociat) Atributele asociate acțiunii `include` sunt: • `page` – un URL (relativ) spre pagina ce se doreşte inclusă; • `flush` – determină dacă buffer-ul resursei va fi golit înainte de operația de includere. `<jsp:include page="myJSPPage.jsp" flush="true" />`
<jsp:plugin>	generarea codului specific browser-ului (o etichetă <OBJECT> sau <EMBED>) pentru o componentă Java (applet sau clasă Java Bean); dacă plugin-ul necesar nu este prezent, acesta este descărcat şi apoi se poate executa componenta Java; parametrii sunt transmişi prin acțiunea `<jsp:param>`, în caz de producere a unei erori, se poate specifica un şir de caractere care să fie afişat prin acțiunea `<jsp:fallback>` `<jsp:plugin type="applet" codebase="mydir" code="MyClass.class" width="100" height="100"> <jsp:param name="paramname" value="paramvalue" /> <jsp:fallback> Unable to load Java Plugin </jsp:fallback> </jsp:plugin>`
<jsp:text>	transcrierea unui text folosind un anumit format în pagini şi documente JSP; corpul acțiunii nu poate conține alte elemente în afară de text şi expresii EL `<jsp:text>My text template</jsp:text>`

Un **scriplet** conține cod sursă Java (declarații de variabile şi metode, expresii) care va fi plasat în metoda de tip `service()` ce se va genera în servlet-ul paginii JSP respective.

```
<% Java code fragment %>
<jsp:scriptlet>
   Java code fragment
</jsp:scriptlet>
```

Acest cod va fi executat pe server atunci când se realizează o cerere pentru pagina JSP care conține scriplet-ul, iar rezultatul va fi plasat în răspunsul care este transmis clientului.

De regulă, utilizarea scripleților este de evitat în paginile JSP întrucât separarea dintre logica aplicației şi prezentare nu mai este atât de evidentă, codul fiind şi mai dificil de întreținut. Recursul la această tehnologie se face numai pentru a beneficia de facilitățile pe care le pune la dispoziție limbajul de programare Java şi numai în situația în care un comportament similar nu poate fi obținut prin folosirea EL (Expression Language) împreună cu metodele din bibliotecile JavaServer Pages Standard Tag Library.

O **expresie** este un scriplet, cuprins între `<%=` şi `%>`, evaluat în momentul în care este rulat servlet-ul şi convertit la tipul `String`, fiind inserat în pagina JSP acolo unde este apelat, putând fi inclusă oriunde, fie că se găseşte sau nu în cadrul unei etichete HTML. Ea poate conține orice expresie care este validă conform specificației Java, însă nu este încheiată prin ;, ca de obicei.

```
<%= expression %>
<jsp:expression>
    expression
</jsp:expression>
```

O **declarație** permite precizarea de variabile sau de metode, similar cu modul în care s-ar face într-o clasă, domeniul de vizibilitate fiind pe tot parcursul paginii respective. În servlet-ul corespunzător paginii JSP, declarațiile vor fi incluse astfel încât să poată fi accesate din toate metodele acesteia.

```
<%! declaration; [declaration; ]+ %>
<jsp:declaration>
    declaration;
    [declaration; ]+
</jsp:declaration>
```

În JSP pot fi referite **obiecte implicite**[8], disponibile la nivelul fiecărei pagini, putând fi utilizate fără a fi declarate în prealabil.

Obiect Implicit	Tip	Descriere
request	subclasă a `javax.servlet.ServletRequest`	cererea care a invocat pagina JSP
response	subclasă a `javax.servlet.ServletResponse`	răspunsul pe care îl generează pagina JSP
out	`javax.servlet.jsp.JspWriter`	obiect care scrie în fluxul de ieşire
session	`javax.servlet.http.HttpSession`	obiect sesiune asociat cererii care a invocat pagina JSP
application	`javax.servlet.ServletContext`	context pagină JSP
config	`javax.servlet.ServletConfig`	informații de configurare referitoare la servlet
pageContext	`javax.servlet.jsp.PageContext`	contextul paginii JSP
page	`java.lang.Object`	referință către pagina JSP prin care poate fi accesat servlet-ul asociat
exception	`java.lang.Throwable`	acces la detalii cu privire la eroare, în paginile de tratare a excepției

Astfel de obiecte vor fi disponibile doar în cadrul metodei `_jspService`, astfel încât referirea acestora în cadrul unei declarații JSP nu are sens.

Este important ca programatorii să cunoască denumirile acestor obiecte implicite precum şi domeniul lor de vizibilitate, astfel încât să le poată accesa funcționalitatea în acelaşi mod în care ar fi făcut-o în cadrul clasei Java Servlet corespunzătoare, în care acesteau erau declarate fie ca parametrii ai metodelor respective, fie se putea obține o instanță a lor pornind de la obiectele cerere şi răspuns. Prin acestea, tehnologiile Java Servlet şi JavaServer Pages sunt echivalente, oferind aceeaşi funcționalitate şi aceeaşi uşurință în utilizare.

[8] Obiectele implicite mai sunt numite şi variabile predefinite.

Un obiect **`request`** este creat de fiecare dată când un client solicită o pagină JSP, acesta oferind metode spre a obține informații din antetele HTTP (`Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Language`, `Authorization`, `Cache-Control`, `Connection`, `Content-Length`, `Cookie`, `Host`, `If-Modified-Since`, `If-Unmodified-Since`, `Referer`, `User-Agent`). Fiind derivat din `javax.servlet.http.HttpServletRequest`, metodele pe care le pune la dispoziția programatorilor acest obiect sunt aceleaşi ca în cazul utilizării ca parametru al metodei `service` din cadrul Java Servlets.

Astfel, cele mai frecvent folosite funcții sunt:

- `Object getAttribute(String)` – întoarce valoarea obiectului identificat prin numele său (transmis ca parametru) – convertit la `Object` sau `null` dacă obiectul solicitat nu există;
- `Enumeration getAttributeNames()` – întoarce o enumerare conținând numele tuturor atributelor disponibile în cadrul cererii;
- `String getAuthType()` – întoarce numele mecanismului de autentificare utilizat pentru securizarea servlet-ului sau `null` dacă nu se foloseşte un astfel de mecanism;
- `String getCharacterEncoding()` – întoarce denumirea schemei de codificare utilizată pentru corpul cererii;
- `int getContentLength()`– întoarce dimensiunea (în octeți) a corpului cererii disponibilă în fluxul de intrare, sau -1 dacă această valoare nu e cunoscută
- `String getContentType()` – întoarce tipul `MIME` al corpului cererii sau `null` dacă tipul nu este cunoscut;
- `String getContextPath()` – întoarce porțiunea din URI-ul cererii care indică contextul acesteia;
- `Cookie[] getCookies()` – întoarce toate obiectele `Cookie` care au fost transmise de client împreună cu cererea sa;
- `String getHeader(String)` – întoarce valoarea antetului specificat din cadrul cererii sub formă de şir de caractere;
- `Enumeration getHeaderNames()` – întoarce o enumerare ce cuprinde numele tuturor antetelor conținute de cerere;
- `int getIntHeader(String)` – întoarce valoarea antetului specificat din cadrul cererii sub formă de întreg;
- `ServletInputStream getInputStream()` – întoarce corpul cererii (ca date binare) folosind un `ServletInputStream`;
- `Locale getLocale()` – întoarce localizarea preferată a conținutului, în funcție de valoarea conținută de antetul `Accept-Language`;
- `String getMethod()` – întoarce numele metodei HTTP prin care a fost transmisă cererea (`GET`, `POST`, `PUT`, `OPTIONS`, `TRACE`, `DELETE`);
- `String getParameter(String)` – întoarce valoarea parametrului specificat (prin nume) în cadrul cererii ca şir de caractere sau `null` dacă nu există;
- `Enumeration getParameterNames()` – întoarce o enumerare incluzând denumirile tuturor parametrilor din cadrul cererii;
- `String[] getParameterValues(String)` – întoarce un vector de obiecte de tip şir de caractere conținând toate valoarile pe care le deține parametrul identificat (prin nume) în cadrul cererii sau `null` dacă nu există;
- `String getPathInfo()` – întoarce informații suplimentare referitoare la cale asociate cu URL-ul transmis împreună cu cererea;

- `String getProtocol()` – întoarce numele şi versiunea protocolului prin care a fost transmisă cererea;
- `String getQueryString()` – întoarce un şir de caractere reprezentând interogarea cuprinsă în URL după cale;
- `String getRemoteAddr()` – întoarce adresa IP (Internet Protocol) a clientului care a transmis cererea;
- `String getRemoteHost()` – întoarce numele complet al maşinii de pe care clientul a transmis cererea;
- `String getRemoteUser()` – întoarce numele utilizatorului autentificat care a transmis cererea sau `null` dacă utilizatorul nu este autentificat;
- `String getRequestURI()` – întoarce partea din URL-ul asociat cererii începând de la numele protocolului până la şirul ce reprezintă interogarea din cadrul cererii HTTP;
- `String getRequestedSessionId()` – întoarce identificatorul sesiunii specificat de client în cerere;
- `int getServerPort()` – întoarce portul pe care a fost primită cererea;
- `String getServletPath()` – întoarce partea din URL-ul asociat cererii care invocă pagina JSP;
- `HttpSession getSession([boolean])` – întoarce sesiunea asociată cererii sau, dacă aceasta nu există (iar parametrul metodei, în caz că este utilizat, are valoarea `true`), asociază cererii o sesiune nouă
- `boolean isSecure()` – indică dacă cererea a fost transmisă folosind un canal sigur (cum este HTTPS).

O listă cu antetele HTTP transmise prin intermediul cererii poate fi obținută astfel:

```
...
<table>
    <tr><th>Header Name</th><th>Header Value</th></tr>
    <%
        Enumeration headers = request.getHeaderNames();
        while (headers.hasMoreElements()) {
            String headerName = (String)headers.nextElement();
            String headerValue = request.getHeader(headerName);
    %>
        <tr><td><%= headerName %></td><td><%= headerValue %></td></tr>
    <%
        }
    %>
</table>
...
```

Un obiect **`response`** este de asemenea creat de fiecare dată de către server, conținând statutul[9], antetele HTTP (`Allow`, `Cache-Control`, `Connection`, `Content-Disposition`, `Content-Encoding`, `Content-Language`, `Content-Length`, `Content-Type`, `Expires`, `Last-Modified`, `Location`, `Refresh`, `Retry-After`, `Set-Cookie`) şi documentul reprezentând pagina Internet care va fi afişată în browser. Fiind derivat din `javax.servlet.http.HttpServletResponse`, metodele pe care le pune la dispoziția programatorilor acest obiect sunt aceleaşi ca în cazul utilizării ca parametru al metodei `service` din cadrul Java Servlets.

[9] Statutul constă în numele protocolului şi versiunea acestuia, codul care indică statutul precum şi un mesaj de dimensiuni mici care reprezintă o descriere a statutului.

Astfel, cele mai frecvent folosite funcții sunt:

- `void addCookie(Cookie)` – adaugă la răspuns obiectul Cookie specificat;
- `void addDateHeader(String, long)` – adaugă la răspuns un antet reprezentând o dată calendaristică, având numele şi valoarea specificate ca parametrii ai metodei;
- `void addHeader(String, String)` – adaugă la răspuns un antet având numele şi valoarea (şir de caractere) specificate ca parametrii ai metodei;
- `void addIntHeader(String, int)` – adaugă la răspuns un antet având numele şi valoarea (număr întreg) specificate ca parametrii ai metodei;
- `boolean containsHeader(String)` – verifică dacă un antet (indicat prin nume, dat ca parametru) a fost specificat în cadrul răspunsului;
- `String encodeRedirectURL(String)` – codifică URL-ul specificat ca parametru pentru a fi utilizat de metoda `sendRedirect` sau dacă nu este necesară codificarea, îl lasă neschimbat;
- `String encodeURL(String)` – codifică URL-ul specificat ca parametru (incluzând identificatorul sesiunii în cadrul său) sau dacă nu este necesară codificarea, îl lasă neschimbat;
- `void flushBuffer()` – forțează afişarea conținutului zonei de memorie tampon în browser;
- `boolean isCommitted()` – verifică dacă răspunsul a fost transmis către client;
- `void reset()` – elimină toate informațiile din zona de memorie tampon, inclusiv coduri de statut şi antete;
- `void resetBuffer()`– elimină conținutul zonei de memorie tampon, menținând însă codurile de statut şi antetele intacte;
- `void sendError(int[, String])` – transmite un răspuns către client reprezentând o eroare indicată prin codul său (şi eventual printr-un mesaj) eliminând şi conținutul zonei de memorie tampon;
- `void sendRedirect(String)` – transmite clientului un răspuns temporar de redirectare către o locație indicată prin URL-ul dat ca parametru;
- `void setBufferSize(int)` – stabileşte dimensiunea preferată (în octeți) pentru zona de memorie ce reține conținutul răspunsului;
- `void setCharacterEncoding(String)` – stabileşte mecanismul de codificare a caracterelor (setul de caractere MIME) corespunzând răspunsul transmis către client;
- `void setContentLength(int)` – stabileşte dimensiunea conținutului transmis către client ca răspuns[10];
- `void setContentType(String)` – stabileşte tipul de conținut corespunzător răspunsului transmis către client, dacă această operație nu s-a produs deja;
- `void setDateHeader(String, long)` – stabileşte un antet reprezentând o dată calendaristică având numele şi valoarea specificate ca parametrii ai metodei;
- `void setHeader(String, String)` – stabileşte un antet cu numele şi valoarea (şir de caractere) specificate ca parametrii ai metodei;
- `void setIntHeader(String, int)` – stabileşte un antet cu numele şi valoarea (număr întreg) specificate ca parametrii ai metodei;

[10] Pentru protocolul HTTP, se stabileşte valoarea pe care o are antetul `Content-Length`.

- `void setLocale(Locale)` – stabileşte localizarea răspunsului, dacă acesta nu a fost transmis către client;
- `void setStatus(int)` – stabileşte codul de statut pentru răspuns.

Stabilirea unor valori corespunzătoare antetelor HTTP din răspuns poate fi realizat astfel:

```
...
<%
    response.setDateHeader("Last-Modified",System.currentTimeMillis());
    response.setHeader("Content-Type","text/html");
    response.setIntHeader("Refresh", 60);
%>
...
```

În exemplu, se stabileşte valoarea antetului `Last-Modified` ca fiind momentul curent, tipul conținutului pe care îl are transmis către client (indicat de antetul `Content-Type`) ca fiind un document HTML, forțând reîncărcarea paginii în mod automat (prin antetul `Refresh`) la fiecare 60 de secunde.

Obiectul **`out`** este echivalentul obiectului `PrintWriter` din Java Servlets[11], fiind utilizat la afişarea conținutului dinamic în cadrul paginii JSP, atunci când aceasta nu se poate face prin intermediul unor etichete HTML (eventual combinate cu expresii JSP). Inițial, obiectul este instanțiat diferit în funcție de activarea mecanismului de utilizare a unei zone de memorie tampon (aşa cum este specificat de atributul `buffered` al directivei `page`). Metodele `print` şi `println` pot fi utilizate spre a afişa conținutul unor obiecte cu tipurile `boolean`, `char`, `String`, `int`, `long`, `float`, `double`, `Object` şi altele. De asemenea, metoda `flush` este folosită pentru a transmite conținutul zonei de memorie tampon către client, golind conținutul fluxului de ieşire.

Obiectul **`session`** este creat în mod automat (nu mai este necesară inițializarea sau obținerea dintr-un obiect de tip `request` prin metoda `getSession`) pentru fiecare client ce accesează pagina în cauză. Dezactivarea acestui tip de comportament poate fi realizată doar prin intermediul atributului cu acelaşi nume din directiva `page`.

Metodele pe care le pune la dispoziție acest obiect sunt cele specificate de interfața `javax.servlet.http.HttpSession`:

- `Object getAttribute(String)` – întoarce obiectul care este asociat numelui transmis ca parametru al metodei în cadrul sesiunii sau `null` dacă nu există nici un obiect asociat;
- `Enumeration getAtrributeNames()` – întoarce o enumerare ce conține denumirile tuturor obiectelor asociate sesiunii;
- `long getCreationTime()` – întoarce momentul la care a fost creată sesiunea, exprimată în milisecunde raportate la data 1 ianuarie 1970 (GMT);
- `String getId()` – întoarce un şir de caractere care identifică în mod unic sesiunea;
- `long getLastAccessedTime()` – întoarce momentul la care clientul a transmis ultima dată o cerere asociată sesiunii, exprimată în milisecunde raportate la data 1 ianuarie 1970 (GMT);

[11] Spre diferență de obiectul `PrintWriter`, un obiect `JspWriter` va genera excepții de tipul `java.io.IOException`. De asemenea, pune la dispoziția utilizatorilor metode suplimentare pentru gestiunea zonei de memorie tampon.

- `int getMaxInactiveInterval()` – întoarce perioada de timp maximă, exprimată în secunde, în care containerul servletului va menține sesiunea, între diferite accesări provenite de la clienți;
- `void invalidate()` – invalidează sesiunea, eliminând orice asociere a unui obiect la aceasta;
- `boolean isNew()` – stabileşte dacă clientul nu are cunoştință de sesiune sau dacă alege să nu participe la aceasta;
- `void removeAttribute(String)` – elimină asocierea obiectului identificat prin numele transmis ca parametru al metodei la sesiune;
- `void setAttribute(String, Object)` – asociază la sesiune un obiect ale cărui nume şi valoare sunt specificate ca parametrii ai metodei;
- `void setMaxInactiveInterval(int)` – stabileşte perioada de timp maximă, exprimată în secunde, care se poate scurge între diferite accesări provenite de la clienți până când sesiunea va fi invalidată de containerul servletului; valoarea poate fi specificată şi în fişierul de configurare `web.xml`[12]:

```
<session-config>
    <session-timeout>60</session-timeout>
</session-config>
```

În afară de sesiuni, pentru gestiunea comunicației dintre client şi server, pot fi utilizate **cookie**-uri, fişiere text reținute pe maşina client (dacă aceasta permite stocarea lor) şi citite de server pentru a identifica în mod unic conexiunea respectivă. Acestea sunt transmise de regulă prin antete HTTP, câmpul `Set-Cookie` reținând perechi de tipul cheie-valoare pentru atributele `name`, `expires`, `path` şi `domain`. De fiecare dată când clientul va accesa o pagină care corespunde cu valorile reținute de atributele `path` şi `domain`, în cazul în care nu a fost depăşită perioada specificată de atributul `expires`, cookie-ul va fi inclus în antetele HTTP transmise către server.

Metodele pe care le pune la dispoziție clasa `Cookie` sunt:

- `String getComment()` – întoarce comentariul care descrie scopul obiectului cookie sau `null` dacă nu există nici un comentariu asociat;
- `String getDomain()` – returnează domeniul pentru care se aplică obiectul cookie;
- `int getMaxAge()` – pune la dispoziție durata de viață (exprimată în secunde) pentru obiectul cookie; valoarea -1 exprimă faptul că obiectul cookie va persista până când se va părăsi browserul;
- `String getName()` – obține denumirea obiectului cookie[13];
- `String getPath()` – reflectă calea pentru care se aplică obiectul cookie;
- `String getValue()` – redă valoarea asociată obiectului cookie;
- `void setComment(String)` – fixează un comentariu care indică scopul obiectului cookie, acesta fiind util în situația în care browserul îl afişează utilizatorului pentru a lua o decizie în privința sa;
- `void setDomain(String)` – indică domeniul pentru care se aplică obiectul cookie;

[12] Această valoare va suprascrie valoarea implicită specificată de serverul Apache Tomcat 7.x, care este de 30 de minute. În fişierul de configurare `web.xml` timpul de expirare va fi specificat, spre diferență de metoda `setMaxInactiveInterval` nu în secunde, ci în minute.
[13] Denumirea unui obiect cookie nu mai poate fi modificată după crearea acestuia.

- `void setMaxAge(int)` – stabileşte timpul (exprimat în secunde) care ar trebui să se scurgă înainte de expirarea obiectului cookie; în cazul în care atributul nu este precizat, obiectul cookie va exista doar pe parcursul sesiunii curente;
- `void setPath(String)` – setează calea pentru care se aplică obiectul cookie; dacă acest atribut nu este specificat, obiectul cookie este transmis pentru toate URL-urile care se găsesc în acelaşi director sau subdirector cu pagina curentă;
- `void setSecure(boolean)` – determină dacă obiectul cookie ar trebui transmis numai prin legături criptate;
- `void setValue(String)` – precizează valoarea asociată cu obiectul cookie.

Utilizarea unui obiect `Cookie` presupune crearea sa[14], precizarea duratei (maxime) de viață şi transmiterea sa în cadrul obiectului `response` la client prin metoda `addCookie`. Totodată, ştergerea presupune stabilirea duratei (maxime) de viață nule înainte de a fi inclus în antetele HTTP către browserul ce se ocupă cu reținerea sa.

Obiectul **`application`** este de fapt un wrapper pentru obiectul `ServletContext` al servletului asociat paginii JSP, fiind o reprezentare a acesteia pe parcursul întregului său ciclu de viață, fiind inițializat în metoda `_jspInit()` şi distrus în metoda `_jspDestroy()`. Este folosit împreună cu metodele `getAttribute(String)` şi `setAttribute(String, Object)` atunci când se doreşte ca proprietățile respective să aibă un domeniu de vizibilitate corespunzător întregii aplicații, adică tuturor paginilor care o alcătuiesc. O funcționalitate ce se pretează folosirii acestui obiect este determinarea numărului de accesări al aplicației respective[15].

Obiectul **`config`** este şi el un wrapper pentru obiectul `ServletConfig`, permițând programatorilor să acceseze parametrii de inițializare ai motorului Java Servlet sau JSP, obținuți prin metodele `getInitParameter(String)`, respectiv `getInitParameterNames()`. Numele servletului asociat, aşa cum este specificat prin elementul `<servlet-name>` din fişierul de configurare `web.xml`, poate fi accesat prin metoda `getServletName()`.

Obiectul **`pageContext`** este folosit ca o reprezentare a întregii pagini JSP, având scopul de a accesa informația cu privire la aceasta, evitând majoritatea detaliilor cu privire la implementare. Acest obiect reține referințe către obiectele `request` şi `response`, iar obiectele `config`, `session` şi `out` pot fi obținute prin atribute ale sale. De asemenea, el conține şi informații cu privire la directivele transmise paginii JSP. În cadrul său sunt definite mai multe câmpuri, printre care şi domeniile de vizibilitate `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, `APPLICATION_SCOPE`. Implementează mai multe metode, majoritatea fiind moştenite din clasa `javax.servlet.jsp.JspContext`. Se pot realiza operații pe atribute (stabilire, obținere sau ştergere) de fiecare dată specificându-se şi domeniul său de vizibilitate.

Obiectul **`page`** este o referință către instanța paginii JSP curente, în fapt un sinonim pentru obiectul `this`.

---

[14] Crearea unui obiect de tip `Cookie` implică specificarea unei denumiri şi a valorii asociate. Nici denumirea şi nici valoarea asociată nu trebuie să conțină în cadrul lor caracterele `[`, `]`, `(`, `)`, `=`, `,`, `"`, `/`, `?`, `@`, `:`, `;`.

[15] Într-o astfel de situație, pentru a se evita situația pierderii datelor în cazul repornirii aplicației (serverului web care o găzduieşte), trebuie asigurată şi persistența acestora prin stocarea lor într-o bază de date.

Obiectul **`exception`** este un wrapper ce conține excepția generată[16] de către pagina anterioară, fiind folosit pentru a construi un răspuns adecvat la eroarea în cauză. Acesta este disponibil numai în paginile JSP de eroare, indicate prin atributul `errorPage` al directivei `page` spre a fi invocate în momentul în care se produc anumite erori.

Fiind un obiect de tip `java.lang.Throwable`, pentru un astfel de obiect pot fi apelate metodele:

- `String getMessage()` – oferă un mesaj[17] detaliat cu privire la excepția care s-a produs;
- `Throwable getCause()` – întoarce cauza excepție;
- `String toString()` – întoarce numele clasei concatenat cu mesajul detaliat referitor la excepția care s-a produs;
- `void printStackTrace()` – afişează rezultatul metodei `toString()` împreună cu stiva de erori la fluxul de ieşire pentru erori `System.err`;
- `StackTraceElement[] getStackTrace()` – întoarce un vector conținând fiecare element al stivei de erori[18];
- `Throwable fillInStackTrace()` – completează stiva de erori a obiectului `Throwable` cu stiva de erori curentă.

Aceste metode pot fi obținute şi prin intermediul obiectului `pageContext` care dispune de atributele `exception` (prin intermediul căruia poate fi vizualizată şi stiva de erori – atributul `stackTrace`) şi `errorData` (ce oferă acces la calea relativă a paginii JSP care a generat eroarea – atributul `URI` şi codul de statut corespunzător excepției – atributul `statusCode`).

Alternativ, în cadrul unui scriplet se poate folosi un bloc `try { ... } catch`, tratarea erorii realizându-se în cadrul aceleiaşi pagini JSP. Totuşi, o astfel de abordare trebuie evitată, recurgându-se la ea numai în situația în care se vrea ca recuperarea din eroare să se realizeze cât mai rapid.

Filtrele[19] sunt folosite pentru a intercepta cererile de la un client înainte ca acestea să acceseze anumite resurse sau pentru a gestiona răspunsurile generate de server înainte ca acestea să fie transmise înapoi. Funcționalitatea unui filtru trebuie specificată în metoda `doFilter`, care va fi suprascrisă de către orice clasă ce implementează interfața `javax.servlet.Filter`:

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws java.io.IOException, javax.servlet.ServletException
```

În afară de această metodă, mai trebuie implementate şi metodele `init` şi `destroy`.

[16] O excepție poate fi **verificată** (o eroare de utilizator care nu poate fi prevăzută de programator, generată în momentul compilării), **generată la rulare** (ignorată la momentul compilării, dar care ar fi putut fi evitată prin anumite mecanisme de prevenție) sau **eroare**, în privința căreia nu se poate face mare lucru, depăşind posibilitățile de intervenție atât din partea utilizatorului cât şi din partea programatorului).

[17] Acest mesaj detaliat cu privire la excepția care s-a produs poate fi inițializat în constructorul obiectului `Throwable`.

[18] Elementul de pe poziția 0 reprezintă vârful stivei de erori, iar ultimul element reprezintă baza stivei de erori (metoda de la apelul căreia a pornit eroarea).

[19] Pot fi implementate filtre pentru autentificare, filtre pentru compresia datelor, filtre de criptare filtre care declanşează evenimente de acces a unor resurse, filtre de conversie a imaginilor, filtre pentru jurnalizare şi auditare, lanțuri de filtre MIME-TYPE, filtre pentru parsare şi filtre XSLT care transformă conținutul XML.

Asocierea dintre un filtru şi un Java Servlet sau pagină JSP precum şi ordinea execuției acestora este determinată din fişierul de configurare `web.xml` în care elementul `<filter>` indică numele filtrului, clasa asociată precum şi parametrii folosiți la inițializarea acestuia, iar elementul `<filter-mapping>` specifică asocierile (şi ordinea[20]) dintre un filtru şi clasele Java Servlet respectiv paginile JSP.

Încărcarea unui fişier pe server (la locația specificată de parametrul `file-upload` din fişierul de configurare `web.xml`) se face prin intermediul unui formular care specifică ca tip de criptare (`enctype`) `multipart/form-data` în care controlul pentru intrare (`input`) este de tip (`type`) `file`:

```
<context-param>
    <param-name>file-upload</param-name>
    <param-value>
        My Upload Location
    </param-value>
</context-param>
```

```
<form action="myJSPPage.jsp" method="POST" enctype="multipart/form-data">
    <input type="file" name="file" size="100" />
    <br />
    <input type="submit" value="Upload File" />
</form>
```

Pentru procesarea în cadrul paginii JSP a încărcării fişierului pe server, poate fi folosită biblioteca commons-fileupload dezvoltată de Apache[21] ce oferă posibilitatea încărcării mai multor fişiere simultan, implementând metode pentru obținerea diferitelor proprietăți ale resursei respective (cale absolută, denumire, dimensiune) precum şi funcționalități precum citirea din memorie şi scrierea pe disc.

Frecvent, **invocarea altor resurse web** se referă la **redirectare**, realizată atunci când documentul s-a mutat la o altă adresă sau atunci când se doreşte echilibrarea încărcării. În acest sens se foloseşte metoda `sendRedirect` apelată pe obiectul `response`, care primeşte ca parametru URL-ul resursei către care se doreşte a se realiza redirectarea. Alternativ, acest comportament se poate obține prin stabilirea unor valori pentru antetele HTTP:

```
response.setIntHeader("Status", response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", myLocation);
```

În cadrul aplicațiilor web, se lucrează destul de des cu obiecte de tipul **dată calendaristică**, integrarea lor în cadrul tehnologiei Java Servlets sau JSP făcându-se prin obiecte de tip `java.util.Date`, ce pot fi create fie prin intermediul unui constructor vid (caz în care obiectul va conține data curentă) sau al unui constructor ce primeşte un parametru de tip `long`, ce indică durata, exprimată în milisecunde, care s-a scurs de la 1 ianuarie 1970 (GMT). Operațiile pe care le suportă un astfel de obiect sunt `after(Date)`, `before(Date)`, `clone()`, `compareTo(Date[|Object])`, `equals (Object)`, `getTime()`, `hasCode()`, `setTime(long)`, `toString()`. Formatarea unui astfel de obiect se poate realiza folosind clasa `SimpleDateFormat` în care anumite caractere codifică atributele obiectului `Date`.

[20] Ordinea în care sunt executate filtrele pentru un Java Servlet respectiv pagină JSP este aceeaşi cu cea în care acestea sunt mapate în fişierul de configurare `web.xml`. Dacă se doreşte ca un filtru să fie asociat la mai multe resurse, se poate folosi masca `/*` în cadrul elementului `<url-pattern>`.

[21] Versiunea 1.3 a bibliotecii commons-fileupload ce implementează funcționalitatea pentru încărcarea pe server a unui fişier poate fi descărcată de la http://commons.apache.org/fileupload.

De asemenea, se întâlneşte frecvent necesitatea de a transmite mesaje folosind **poşta electronică**, o astfel de funcționalitate putând fi integrată prin API-ul JavaMail şi framework-ul Java Activation Framework (JAF)[22].

Un obiect `Session` necesar construirii unui mesaj ce poate fi transmis prin poşta electronică va fi construit dintr-o serie de proprietăți între care obligatorie este `mail.smtp.host`. În situația în care serverul de poştă electronică necesită autentificare, vor fi precizate şi atributele `mail.user` / `mail.password`. Obiectul reprezentând mesajul propriu-zis va avea tipul `MimeMessage`, precizarea câmpurilor corespunzătoare (`From:`, `To:`, `Subject:` şi `Text:`) realizându-se prin metodele:

- `void setFrom(InternetAddress)` – parametrul metodei construindu-se pornind de la adresa de poştă electronică a destinatarului;
- `void addRecipients(Message.RecipientType, Address[])`, unde
  - tipul poate avea valorile
    - `Message.RecipientType.TO`;
    - `Message.RecipientType.CC` (CarbonCopy);
    - `Message.RecipientType.BCC` (Blind Carbon Copy).
  - vectorul de adrese se creează folosind constructorul `InternetAddress` căruia i se transmite ca parametru adresa de poştă electronică a expeditorului.
- `void setSubject(String)` – se stabileşte subiectul mesajului;
- `void setText(String)` – se precizează corpul mesajului;
- `void setContent(String, String)` – se precizează corpul mesajului, având atât conținutul, cât şi formatul precizate ca parametri[23].

Transmiterea propriu-zisă a mesajului se face prin metoda statică `send` definită în clasa `Transport`, aceasta generând o excepție `MessagingException` dacă mesajul nu a putut fi transmis din diverse motive.

De asemenea, pot fi incluse ataşamente prin construirea unui obiect `MimeMultiPart` la care se adaugă (folosind metoda `addBodyPart`) fragmente ale ataşamentului respectiv, de tipul `MimeBodyPart`. Un astfel de obiect suportă metodele `setText(String)` pentru a indica o corpul mesajului, respectiv `setDataHandler(DataHandler`[24]`)` şi `setFileName(String)` pentru a specifica fişierul care conține ataşamentul. Indicarea conținutului ca fiind un obiect de tipul `MimeMultiPart` se face tot prin metoda `setContent`.

Informațiile cu privire la destinatar, expeditor, subiect şi conținut pot fi incluse într-un formular, apoi preluate din obiectul `request` astfel încât transmiterea mesajului să se facă pe baza acestora.

---

[22] Bibliotecile corespunzătoare JavaMail API 1.5.1 şi Java Activation Framework 1.1.1 pot fi descărcate de la adresele https://java.net/projects/javamail/ (fişierul javax.mail.jar) respectiv http://www.oracle.com/technetwork/java/javase/index-jsp-136939.html (fişierul activation.jar).

[23] În acest mod se pot transmite documente care au alt format în afară de plain-text, de exemplu HTML / XML. În acest caz, dimensiunea pe care o poate avea corpul mesajului nu este limitată decât de serverul de poştă electronică.

[24] Un obiect de tip `DataHandler` se obține dintr-un obiect `FileDataSource`, care la rândul său este construit pornind de la denumirea fişierului în cauză, la care se adaugă şi calea către el, relativă sau absolută. În situația în care nu se specifică nici o cale, fişierul va fi căutat în acelaşi director în care este plasată clasa Java Servlet sau pagina JSP care încearcă transmiterea sa prin intermediul poştei electronice.

```
<%@ page import="javax.mail.*", "javax.mail.internet.*", "javax.activation.*" %>
...
<%
    Properties properties = System.getProperties();
    properties.setProperty("mail.smtp.host","localhost");
    properties.setProperty("mail.user",myUser);
    properties.setProperty("mail.password",myPassword);
    Session mailSession = Session.getDefaultInstance(properties);
    try {
        MimeMessage mailMimeMessage = new MimeMessage(maiSession);
        mailMimeMessage.setFrom(new InternetAddress(from));
        mailMimeMessage.addRecipient(
                                    Message.RecipientType.TO,
                                    new InternetAddress(to)
                                   );
        mailMimeMessage.setSubject("My Subject");
        Multipart mailMultipart = new MimeMultipart();
        BodyPart mailBodyPart;
        mailBodyPart = new MimeBodyPart();
        mailBodyPart.setText("My Mail Body");
        mailMultipart.addBodyPart(mailBodyPart);
        mailBodyPart = new MimeBodyPart();
        String filename = "myFileName.myExtension";
        mailBodyPart.setDataHandler(new DataHandler(new FileDataSource(filename)));
        mailBodyPart.setFileName(filename);
        mailMultiPart.addBodyPart(mailBodyPart);
        mailMimeMessage.setContent(mailMultiPart);
        Transport.send(mailMimeMessage);
    } catch (MessagingException exception) {
        System.out.println("exceptie: "+exception.getMessage();
        exception.printStackTrace();
    }
%>
```

5. Folosirea EL (Expression Language) în pagini JSP

EL (Expression Language) pune la dispoziția utilizatorilor un mecanism prin care nivelul de prezentare poate comunica cu nivelul de logică al aplicației (mai ales componente Java Beans), implementând operații aritmetice şi logice folosind tipuri de date întregi, reale, şiruri de caractere, valori adevărat / fals şi null.

Tehnologia Java Server Pages foloseşte Expression Language pentru:

- evaluarea imediată şi leneşă a expresiilor prin realizarea de operații aritmetice şi logice în mod dinamic;
- posibilitatea de a stabili şi de a obține informații (având structuri de date variate), în special proprietăți ale componentelor Java Beans, pe baza interacțiunii cu utilizatorul;
- folosirea de obiecte implicite;
- abilitatea de a invoca metode (publice şi statice).

În Expression Language pot fi folosite următoarele tipuri de expresii:

- **expresii cu evaluare imediată** (a căror valoare este stabilită instant de către tehnologia împreună cu care este utilizată, în cazul de față Java Server Pages) şi **expresii cu evaluare leneşă** (a căror valoare poate fi evaluată ulterior de către tehnologia împreună cu care este utilizată);
- **expresii valoare** (ce referă date) şi **expresii metodă** (ce invocă metode);
- **expresii rvalue** (care pot doar să citească date dintr-un obiect extern) şi **expresii lvalue** (care pot atât să citească cât şi să scrie date din / într-un obiect extern).

Expresiile cu evaluare imediată, desemnate prin sintaxa `${expression}`, sunt calculate înainte ca pagina Internet să fie afişată. Aceste expresii pot fi folosite numai ca valori ale unei etichete care acceptă expresii evaluate în momentul rulării şi sunt întotdeauna expresii de tip rvalue.

```
<input type="submit" name="${inputName}" value="${inputValue}">
```

Pentru expresiile cu evaluare leneşă, pentru care se foloseşte sintaxa `#{expression}`, valoarea poate fi determinată mai târziu, prin mecanisme ce țin de tehnologia împreună cu care e folosită Expression Language, pe parcursul ciclului de viață al paginii Internet[25]. Aceste expresii pot fi expresii valoare utilizate atât pentru a citi cât şi pentru a scrie informații dar şi expresii metodă.

```
<table><tr><td>#{inputName}</td><td>#{inputValue}</td></tr></table>
```

Expresiile de tip valoare pot fi clasificate în expresii rvalue şi lvalue după cum dispun sau nu de posibilitatea de a scrie informații. Expresiile cu evaluare imediată sunt întotdeauna expresii rvalue, în timp ce expresiile cu evaluare leneşă sunt de regulă expresii lvalue.

Expresia `${entitybean.attribute}` obține valoarea unui atribut din cadrul unei componente Java Bean, evaluându-l instant şi transmițând mai departe rezultatul obținut.

Expresia `#{entitybean.attribute}` poate avea acelaşi comportament ca expresia de mai sus, însă eticheta în cadrul căreia este folosită poate decide evaluarea sa la un moment de timp ulterior. Mai mult, atunci când pagina este solicitată, expresia va fi de tipul rvalue (obținând valoarea atributului), în timp ce în cadrul etapei postback ea poate fi de tipul lvalue (putând stabili valoarea atributului).

Toate expresiile pot referi următoarele tipuri de obiecte, împreună cu atributele lor: componente Java Beans, colecții, structuri de date Java de tip enumerare[26], obiecte implicite. Pentru a le utiliza, se foloseşte o expresie în care variabila este reprezentată de denumirea obiectului:

```
${myObject}
#{myObject}
```

Container-ul paginii Internet evaluează variabila ce apare în expresie[27] căutându-i valoarea ce este oferită de `PageContext.findAttribute("myObject")`, domeniile de vizibilitate investigate fiind `page`, `request`, `session` şi `application`. Dacă obiectul nu este găsit, se returnează o valoare `null`.

Pentru a referi proprietățile unei componente Java Bean sau a instanței unei enumerări, elementele unei colecții sau atributele unui obiect implicit pot fi folosite notațiile . sau `[]`, care sunt echivalente. Pentru şirurile de caractere se pot folosi atât apostroafe, cât şi ghilimele.

---

[25] De regulă, expresiile cu evaluare leneşă sunt folosite mai des împreună cu JavaServer Faces pentru că această tehnologie are un ciclu de viață împărțit în mai multe faze, astfel încât determinarea valorii unei expresii nu se poate realiza decât la un anumit moment, după tratarea evenimentelor cu privire la componente şi validarea datelor.

[26] În cazul structurilor de date Java de tip enumerare, elementele pot fi referite folosind tipul String. Pentru tipul de dată `public enum EnumType {attribute1, atrribute2, ..., attributen}`, dacă este întâlnit un şir de caractere având valoarea `${"atrributek"}`, acesta va fi convertit automat la tipul `EnumType.atrributek`.

[27] Pot fi utilizare motoare EL particularizate pentru a se modifica modul în care sunt evaluate expresiile.

Următoarele expresii sunt echivalente:

```
${myObject.myAttribute}
${myObject['myAttribute']}
${myObject["myAttribute"]}
```

Expresii similare pot fi formate şi pentru asigurarea evaluării leneşe.

De asemenea, pot fi folosite şi combinații ale operatorilor . şi [] în cadrul aceleiaşi expresii.

În cazul componentelor JavaBeans, un atribut `myAttribute` poate fi referit în cadrul unei expresii EL doar în situația în care este implementată o metodă getter de acces la acesta, având numele `getMyAttribute`.

Pentru accesarea elementelor unui vector sau al unei liste, trebuie folosite valori literale (fără apostroafe sau ghilimele) ce pot fi convertite la un întreg sau notația [] ce primeşte de asemenea un întreg, în timp ce pentru un obiect de tip Map accesarea implică folosirea unei chei de tip şir de caractere:

```
// array or list: firstComponent can be narrowed to 0
${myObject.myAttribute[0]}
${ myObject.myAttribute[firstComponent]}
// map
${myObject.myAttribute["firstComponent"]}
```

În Expression Language pot fi folosite următoarele tipuri de date:

- `Boolean`: poate avea valorile `true` sau `false`;
- numere întregi, numere reale: utilizate la fel ca în Java;
- `String` (şir de caractere), referite cu apostroafe sau ghilimele, \ fiind folosit pe post de caracter escape (`" → \", ' → \', \ → \\`);
- `null`.

Expresiile de tip valoare pot fi utilizate în text static sau în etichete (standard sau definite de utilizator) care acceptă o expresie[28].

Valoarea unei expresii în text static este evaluată şi ulterior inclusă în pagina Internet:

```
<my:tag>my text1 ${expression} my text2</my:tag>
```

Atributul valoare al unei etichete poate fi inițializat folosind o expresie rvalue sau lvalue astfel:

- folosind o singură expresie; expresiile sunt evaluate şi rezultatul convertit la tipul atributului respectiv:

  ```
  <my:tag value="${expression}" />
  <my:tag value="#{expression}" />
  ```

- folosind una sau mai multe expresii, separate sau înconjurate de text; se numesc **expresii compuse** şi sunt evaluate de la stânga la dreapta, fiecare expresie fiind convertită la tipul `String` şi concatenată cu alte texte, şirul de caractere rezultat fiind convertit la tipul atributului respectiv:

  ```
  <my:tag value="my text1 ${expression1} my text2 ${expression2}" />
  <my:tag value="my text1 #{expression1} my text2 #{expression2}" />
  ```

  În EL 3.0 operatorul + e folosit pentru concatenarea şirurilor de caractere.

- folosind doar text; se numesc **expresii literale** şi valoarea `String` a atributului este converită la tipul atributului respectiv

  ```
  <my:tag value="my text" />
  ```

[28] Unele etichete acceptă doar expresii de tip rvalue în timp ce alte expresii pot accepta şi expresii de tip lvalue.

Toate expresiile utilizate pentru a stabili valori sunt evaluate în contextul unui tip de date aşteptat. Dacă rezultatul obținut în urma evaluării expresiei nu corespunde exact tipului de date aşteptat, se va realiza o conversie de tip.

Expresiile de tip metodă permit invocarea unei metode implementate în cadrul unei componente Java Bean, care întoarce un rezultat. De regulă, acestea sunt folosite pentru a realiza anumite procesări asociate cu controlul din contextul căruia sunt apelate, fiind invocate în momentul în care este generat un eveniment în cadrul acelui control. Întrucât o metodă poate fi invocată pe parcursul diferitelor etape ale ciclului de viață, expresiile de tip metodă trebuie să fie întotdeauna cu evaluare leneşă.

```
<my:tag id="myTagId" action="#{myObject.myValue}" />
<my:tag id="myTagId" action="#{myObject['myValue']}" />
```

Expresiile de tip metodă trebuie folosite întotdeauna numai în cadrul unei etichete, fie folosind o singură expresie care este evaluată şi transmisă modulului de gestiune a etichetei, putând fi invocată mai târziu sau folosind doar text. La execuția metodei asociată expresiei în cauză, se întoarce un rezultat de tip `String` care este convertit la tipul aşteptat, aşa cum este definit de descriptorul bibliotecii de etichete.

Expresiile de tip metodă pot fi utilizate pentru a invoca funcții care primesc un set de parametrii (0 sau mai mulți, separați prin virgulă). Se pot folosi atât operatorul . cât şi [], comportamentul acestora fiind similar:

```
myObject["myMethod"](myparameters)
myObject.myMethod(myParameters)
```

În Expression Language pot fi folosite şi expresii lambda, acestea fiind expresii de tip valoare ce primesc parametrii. Utilizarea acestora este comparabilă cu cazul limbajului de programare Java, cu excepția faptului că şi corpul expresiei lambda este tot o expresie EL. O expresie lambda foloseşte operatorul `->`, identificatorii din stânga expresiei fiind parametrii lambda[29], în dreapta expresiei aflându-se o expresie EL.

O expresie lambda are un comportament asemănător cu al unei funcții, aceasta putând fi apelată imediat sau valoarea ei poate fi asociată unei variabile ce va fi utilizată ulterior pentru invocarea metodei respective primind parametrii care vor fi disponibili la momentul respectiv.

O expresie lambda poate fi transmisă ca argument al unei metode, fiind executată în cadrul metodei respective sau poate fi inclusă la rândul ei, în altă expresie lamda.

```
() -> null
(x,y) -> (x+y)/2
((x,y) -> Math.sqrt(x*y))(4,9)
distance = (x1,y1,x2,y2) -> Math.sqrt(Math.sqr(x1-x2)+Math.sqr(y1-y2));
distance(1,1,2,2)
```

O expresie literală[30] este evaluată la textul expresiei, care are tipul `String`, fiind utilizată atunci când este necesară folosirea sintaxei rezervate `${}` sau `#{}`. Dacă se doreşte folosirea şirului de caractere `${expression}` se pot folosi alternativele `${'${'}expression}` sau `\${expression}` (idem şi pentru expresii `#{}`).

[29] De obicei, se folosesc parantezele () pentru a include parametrii lambda, acestea putând fi omise dacă există un singur parametru.
[30] O expresie literală nu foloseşte delimitatorii `${}` sau `#{}`.

Expresiile literale pot fi cu evaluare imediată sau cu evaluare leneşă[31] şi pot fi atât expresii de tip valoare cât şi expresii de tip metodă.

În EL sunt suportate operații pe colecții cum ar fi:

- mulțimi
  ```
  {1, 2, ..., n}
  ```
- liste – acestea pot conține elemente având tipuri diferite
  ```
  {1, two, "3", ...}
  ```
- obiecte de tip map
  ```
  {"one":1, "two":2, ..., "n":n}
  ```

Prin Expression Language, se permite crearea dinamică a acestora, exploatarea lor implicând utilizarea unor obiecte de tip flux, respectiv benzi de asamblare (*eng.* pipeline). Astfel, metodele vor fi apelate pe fluxul de elemente obținut din colecția respectivă folosind metoda `stream()`[32]. Întrucât unele metode întorc tot un flux de elemente, acestea pot fi înlănțuite împreună în cadrul unei benzi de asamblare[33]. Operațiile realizate pe fluxul de elemente nu modifică colecția originală.

```
collection.stream().filter(a->a.attribute1=='value1')
                   .map(a->a.attribute2)
                   .sorted()
                   .toList()
```

Operațiile suportate de fluxul de elemente sunt: `allMatch`, `anyMatch`, `average`, `count`, `distinct`, `filter`, `findFirst`, `flatMap`, `forEach`, `iterator`, `limit`, `map`, `max`, `min`, `noneMatch`, `peek`, `reduce`, `sorted`, `substream`, `sum`, `toArray`, `toList`.

În afară de operatorii . şi [], Expression Language pune la dispoziția programatorilor şi alți operatori, ce pot fi folosiți doar în expresii de tip rvalue:

Categorie Operator	Operator	Precedență	Observații
	`. []`	1	
	`()`		folosit pentru a schimba precedența operatorilor
aritmetici	`-` (unar)	2	
logici	`not !`	2	
empty	`empty`	2	utilizat pentru a verifica dacă o valoare este `null` sau vidă
aritmetici	`* / div % mod`	3	
	`+ -` (binar)	4	
concatenare şiruri de caractere	`+=`	5	
relaționali	`<> < <= > >= lt le gt ge`	6	comparația se poate face cu alte valori sau literali
	`== != eq ne`	7	
logici	`&& and`	8	
	`|| or`	9	
condiționali	`? :`	10	`X ? Y : Z`
lambda	`->`	11	
atribuire	`=`	12	
	`;`	13	

[31] Momentul la care este evaluată o expresie depinde de contextul în care aceasta este utilizată.

[32] Metoda `stream` obține un obiect `Stream` pornind de la un obiect `java.util.Collection` sau un vector Java.

[33] Astfel, o bandă de asamblare constă dintr-o sursă (obiect de tip `Stream`), orice număr de operații intermediare care întorc un flux (spre exemplu, `filter` şi `map`) şi o operație terminală care nu întoarce un flux (spre exemplu, `toList`).

În Expression Language există şi termeni rezervați, care nu pot fi utilizați ca identificatori: `and`, `or`, `not`, `eq`, `ne`, `lt`, `gt`, `le`, `ge`, `true`, `false`, `null`, `instanceof`, `empty`, `div`, `mod`.

Există posibilitatea ca expresiile EL să nu fie evaluate în cazul în care atributul `isELIgnored` al directivei `page` are valoarea `true` (implicit, valoarea sa este `false`, deci în mod obişnuit, expresiile EL vor fi evaluate într-o pagină JSP).

6. Utilizarea JSTL (JavaServer Pages Standard Tag Library) pentru implementarea unor funcționalități complexe

JSTL (JavaServer Pages Standard Tag Library) este o colecție de etichete JSP, care implementează funcționalitatea de bază specifică pentru numeroase aplicații JSP.

Etichetele JSTL pot fi clasificate, în funcție de comportamentul pe care îl oferă utilizatorilor în mai multe grupuri: ❶ etichete JSTL de bază, ❷ etichete JSTL pentru formatare, ❸ etichete JSTL destinate gestiunii informațiilor dintr-o bază de date SQL, ❹ etichete JSTL pentru gestiunea documentelor XML, ❺ funcții JSTL.

Pentru ca aceste funcționalități să fie accesibile în pagina JSP a unei aplicații web, biblioteca JSTL trebuie plasată în directorul `WEB-INF\lib` corespunzător acesteia.

Etichetele JSTL de bază sunt cele mai frecvent utilizate, implementând funcționalitate legată de controlul fluxului.

Pentru a putea referi aceste etichete, trebuie specificată locația de unde acestea pot fi încărcate.

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

Cele mai frecvent utilizate etichete JSTL de bază sunt:

<c:out>	evaluează valoarea unei expresii pe care o afişează Atributele asociate etichetei sunt: • `value` (obligatoriu) – informația care trebuie evaluată; • `default` – informație asociată în cazul în care evaluarea eşuează; • `escapeXml` – utilizată pentru a ignora caractere XML speciale. `<c:out value="${myObject.myAttribute}" />`

<c:set>	asociază rezultatul evaluării unei expresii la o variabilă, disponibilă pentru un anumit domeniu de vizibilitate.
	Atributele asociate etichetei sunt:
	• `value` – informația care trebuie evaluată şi asociată obiectului;
	• `target` – denumirea obiectului a cărui proprietate trebuie stabilită;
	• `property` – proprietate a obiectului `target` a cărui valoare va fi stabilită la valoarea indicată (trebuie specificat dacă se foloseşte `target`);
	• `var` – denumirea variabilei care va stoca informația;
	• `scope` – scopul variabilei care va stoca informația.
	`<c:set value="${expression}" target="myObject" property="myAttribute" scope="page" />`
	`<c:set value="${expression}" var="myVariabile" scope="application" />`

<c:remove>	disociază o variabilă de contextul unui anumit domeniu de vizibilitate Atributele asociate etichetei sunt: • `var` (obligatoriu) – denumirea variabilei care va fi eliminată; • `scope` – scopul variabilei care va fi eliminată. `<c:remove var="myVariable" scope="application" />`

<c:catch>	gestionează orice obiect de tip `Throwable` întâlnit în corpul său, eventual expunându-l utilizatorului
	Atributul asociat etichetei este `var` ce indică denumirea variabilei care va stoca excepția generată, în cazul în care va fi produsă de instrucțiunile din corpul său
	`<c:catch var="${exception}">`
	`  <% some expression(s) that may potentially throw an error %>`
	`</c:catch>`
	`<c:out value="exception: ${exception.message}" />`

<c:if>	etichetă de tip condițional, al cărui corp este evaluat în cazul când condiția asociată este îndeplinită
	Atributele asociate etichetei sunt:
	• `test` (obligatoriu) – condiția care se dorește a fi evaluată;
	• `var` – denumirea variabilei care va stoca rezultatul condiției;
	• `scope` – scopul variabilei care va stoca rezultatul condiției.
	`<c:if test="${expression}" />`
	`  <% do some operations %>`
	`</c:if>`

<c:choose>	etichetă de tip condițional, indicând condiții ce se exclud mutual, specificate prin `<when>` şi `<otherwise>`; funcționează ca instrucțiunea switch din Java, permițând alegerea dintre mai multe alternative: ramurile `case` sunt specificate prin `<when>` în timp ce ramura `default` este specificată prin `<otherwise>`.
	Eticheta nu are atribute.
	`<c:choose>`
	`  <c:when test="${expression1}"><% do some operations %></c:when>`
	`  <c:when test="${expression2}"><% do some operations %></c:when>`
	`  ...`
	`  <c:when test="${expressionn}">`
	`  <% do some operations %>`
	`  </c:when>`
	`  <c:otherwise><% do some other operations %></c:otherwise>`
	`</c:choose />`

<c:when>	subetichetă a `<choose>`, al cărui corp este evaluat în cazul când condiția asociată este îndeplinită Atributul asociat etichetei este `test` ce indică condiția care se dorește a fi evaluată.
<c:otherwise>	subetichetă a `<choose>`, ce urmează uneia sau mai multor etichete `<when>`, determinând evaluarea corpului său, dacă nici una dintre condițiile care o preced nu este îndeplinită. Eticheta nu are atribute.
<c:import>	primește un URL relativ sau absolut şi expune conținutul său întregii pagini, unui şir de caractere specificat în atributul `var` sau unui obiect `Reader` specificat în atributul `varReader` Atributele asociate etichetei sunt: • `url` (obligatoriu) – URL-ul ce trebuie obținut şi inclus în pagină; • `context` – caracterul / urmat de numele aplicației web; • `charEncoding` – setul de caractere utilizat pentru datele preluate; • `var` – denumirea variabilei utilizată pentru a stoca textul preluat; • `scope` – domeniul de vizibilitate al variabilei utilizată pentru a stoca textul preluat; • `varReader` – denumirea unei variabile alternative pentru a expune conținutul sub forma unui obiect `java.io.Reader`; `<c:import var="myVariabile" url="http://www.mysite.com" />`

<c:forEach>	etichetă de tip iterație, acceptând mai multe tipuri de colecții, suportând parcurgerea submulțimilor; este o alternativă pentru instrucțiunile de iterare `for`, `while` şi `do-while` din cadrul Java, care pot fi utilizate în cadrul unui scriplet Atributele asociate etichetei sunt: • `items` – informația (colecția) pe care se realizează iterația; • `begin` – elementul colecției de la care se începe iterația; • `end` – elementul colecției la care se termină iterația; • `step` – pasul cu care se parcurg elementele colecției; • `var` – denumirea variabilei care expune elementul curent; • `varStatus` – denumirea variabilei care expune statutul iterației; `<c:forEach var="myVariabile" items="${myCollection}">` `  <c:out value="${myVariabile}" /> ` `</c:forEach>`
<c:forTokens>	iterează asupra unor particule, separate prin delimitatori Suportă aceleaşi atribute ca şi `<foreach>`, la care se adaugă `delim`, care indică caracterele utilizate ca delimitatori `<c:forTokens var="myVariabile" items="${myString}" delim=",">` `  <c:out value="${myVariabile}" /> ` `</c:forTokens>`

<c:param>	adaugă un parametru la URL-ul specificat în cadrul unei etichete `<import>`, realizând şi codificarea acestuia
	Atributele asociate etichetei sunt:
	• `name` – denumirea parametrului care va fi inclus în URL;
	• `value` – valoarea parametrului care va fi inclus în URL;
	`<c:import var="myVariabile" url="http://www.mysite.com">`
	`  <c:param name="param1Name" value="param1Value" />`
	`  <c:param name="param2Name" value="param2Value" />`
	`  ...`
	`  <c:param name="paramnName" value="paramnValue" />`
	`</c:import>`

<c:redirect>	redirectează controlul la un alt URL, realizând suprascrierea sa în browser; suportă URL-uri relative la context şi eticheta `<param>` Atributele asociate etichetei sunt: • `url` (obligatoriu) – URL-ul la care se realizează redirectarea controlului; • `context` – caracterul / urmat de numele aplicației web. `<c:redirect url="http://www.mysite.com">`
<c:url>	creează un URL cu parametrii de interogare opționali, stocând valoarea acestuia într-o variabilă şi realizând rescrierea URL-ului dacă este necesar; este o alternativă la invocarea metodei `response.encodeURL()`, suportând şi eticheta `<param>` Atributele asociate etichetei sunt: • `value` (obligatoriu) – URL-ul de bază; • `context` – caracterul / urmat de numele aplicației web; • `var` – denumirea variabilei ce reține URL-ul procesat; • `scope` – domeniul de vizibilitate al variabilei ce reține URL-ul procesat; `<c:url value="http://www.mysite.com" />`

Etichetele JSTL pentru formatare sunt utilizate pentru a formata şi afişa text, informații cu privire la dată calendaristică şi timp, numere precum şi site-uri Internet într-un format localizat.

Pentru a putea referi aceste etichete, trebuie specificată locația de unde acestea pot fi încărcate.

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

Cele mai frecvent utilizate etichete JSTL pentru formatare sunt:

<fmt:formatNumber>	utilizat pentru redarea unei valori numerice cu o anumită precizie sau format Atributele asociate etichetei sunt: • `value` (obligatoriu) – valoarea numerică ce se doreşte afişată; • `type` – poate lua valorile `NUMBER`, `CURRENCY`, `PERCENT`; • `pattern` – modelul de formatare pentru valoarea numerică ce se doreşte afişată; se folosesc codurile: o 0 – reprezintă o cifră; o E – reprezintă o formă exponențială; o # – reprezintă o cifră; se afişează 0 dacă lipseşte; o . – separator pentru valorile zecimale o , – separator pentru grupurile reprezentând miile; o ; – separator pentru formate; o - – prefix pentru valori negative; o % – afişează procentajul; o ? – afişează miimile; o * – înlocuit de simbolul valutei curente; o X – indică faptul că orice alt caracter poate fi folosit ca prefix sau sufix; o ' – utilizat pentru a cita caractere speciale în prefix sau sufix. • `currencyCode` – codul valutei (pentru tipul `CURRENCY`), preluat din localizarea curentă; • `currencySymbol` – simbolul valutei (pentru tipul `CURRENCY`), preluat din localizarea curentă; • `groupingUsed` – dacă se folosesc grupuri de numere; se foloseşte pentru a introduce , între grupurile care reprezintă miile; • `maxIntegerDigits` – numărul maxim de cifre întregi care pot fi afişate; dacă valoarea este depăşită, atunci rezultatul se trunchează; • `minIntegerDigits` – numărul minim de cifre întregi care pot fi afişate; • `maxFractionDigits` – numărul maxim de zecimale care pot fi afişate; dacă valoarea este depăşită, atunci rezultatul se rotunjeşte; • `minFractionDigits` – numărul minim de zecimale care pot fi afişate; • `var` – denumirea variabilei care reține valoarea formatată; • `scope` – domeniul de vizibilitate al variabilei care reține valoarea formatată.
<fmt:parseNumber>	parsează reprezentarea sub formă de şir de caractere pentru o valoare numerică, o valută sau un procentaj Atributele asociate etichetei sunt: • `value` – valoarea numerică ce se doreşte a fi parsată; • `type` – poate lua valorile `NUMBER`, `CURRENCY`, `PERCENT`; • `parseLocale` – localizarea folosită la parsarea numărului; • `integerOnly` – indică parsarea de valori întregi sau reale; • `pattern` – modelul de parsare; • `timeZone` – zona de timp a datei calendaristice parsate; • `var` – denumirea variabilei care reține valoarea parsată; • `scope` – domeniul de vizibilitate al variabilei care reține valoarea parsată.

<fmt:formatDate>	utilizat pentru formatarea unei date calendaristice / timp folosind stilurile şi modelele oferite Atributele asociate etichetei sunt: • `value` (obligatoriu) – data calendaristică de afişat; • `type` – poate lua valorile `DATE`, `TIME`, `BOTH`; • `dateSytle` – poate lua valorile `FULL`, `LONG`, `MEDIUM`, `SHORT`, `DEFAULT`; • `timeStyle` – poate lua valorile `FULL`, `LONG`, `MEDIUM`, `SHORT`, `DEFAULT`; • `pattern` – modelul de formatare pentru valoarea de tip dată calendaristică / timp ce se doreşte afişată; se folosesc codurile: o G – era; o y – anul; o M – luna; o d – ziua din lună; o h – ora (format 12 ore); o H – ora (format 24 ore); o m – minutul; o s – secunda; o S – milisecunda; o E – ziua din săptămână; o D – ziua din an; o F – ziua din săptămână în cadrul lunii; o w – săptămâna din an; o W – săptămâna din lună; o a – indicatorul AM / PM; o k – ora (format 12 ore); o K – ora (format 24 ore); o z – zona de timp; o ' – caracter escape pentru informații de tip text; o " – utilizat pentru citări; • `timeZone` – zona de timp a datei calendaristice afişate; • `var` – denumirea variabilei care reține valoarea formatată; • `scope` – domeniul de vizibilitate al variabilei care reține valoarea formatată.
<fmt:parseDate>	parsează reprezentarea sub formă de şir de caractere pentru o valoare de tip dată calendaristică / timp Are aceleaşi atribute ca `<formatDate>` la care se adaugă `parseLocale`, indicând localizarea care trebuie utilizată atunci când se parsează informația de tip dată calendaristică.

<fmt:bundle>	încarcă o resursă ce urmează să fie utilizată în corpul său, de eticheta `<message>`
	Atributele asociate etichetei sunt:
	• `basename` (obligatoriu) – specifică numele de bază pentru resursa care se doreşte a fi încărcată;
	• `prefix` – valoarea care va preceda fiecare denumire de cheie în subeticheta `<message>`.
	`<fmt:bundle basename="myBaseName">`
	`  <fmt:message key="myPrefix.myKey" />`
	`</fmt:bundle>`
	echivalent cu
	`<fmt:bundle basename="myBaseName" prefix="myPrefix">`
	`  <fmt:message key="myKey" />`
	`</fmt:bundle>`

<fmt:setLocale>	stochează localizarea dată în variabila de configurare referitoare la localizare Atributele asociate etichetei sunt: • value (obligatoriu) – specifică un cod format din două părți indicând codul de limbă (ISO-639) şi codul de țară (ISO-3166) • variant – variantă specifică browser-ului; • scope – domeniul de vizibilitate al variabilei de configurare referitoare la localizare.
<fmt:setBundle>	încarcă o resursă pe care o stochează într-o variabilă având un domeniu de vizibilitate, respectiv în variabila de configurare referitoare la resurse Atributele asociate etichetei sunt: • basename (obligatoriu) – specifică numele de bază pentru familia de resurse pentru a fi expusă ca o variabilă având un anumit domeniu de vizibilitate, respectiv ca o variabilă de configurare; • var – denumirea variabilei care reține valoarea resursei; • scope – domeniul de vizibilitate al variabilei care reține valoarea resursei.
<fmt:timeZone>	specifică zona de timp pentru orice tip de acțiune vizând formatarea sau parsarea din corpul său Atributul asociat etichetei este value ce indică zona de timp care va fi aplicată corpului său.
<fmt:setTimeZone>	stochează zona de timp dată în variabila de configurare referitoare la zona de timp Atributele asociate etichetei sunt: • value (obligatoriu) – zona de timp care va fi expusă ca o variabilă având un anumit domeniu de vizibilitate, respectiv ca o variabilă de configurare; • var – denumirea variabilei care reține zona de timp; • scope – domeniul de vizibilitate al variabilei care reține zona de timp.
<fmt:message>	afişează un mesaj folosind un format localizat Atributele asociate etichetei sunt: • key – cheia mesajului care se doreşte a fi obținut; • bundle – identificatorul resursei care se doreşte a fi folosit; • var – denumirea variabilei care reține valoarea mesajului localizat; • scope – domeniul de vizibilitate al variabilei care reține valoarea mesajului localizat.
<fmt:requestEncoding>	stabileşte mecanismul de configurare al caracterelor pentru cerere Atributul asociat etichetei este key prin care se indică mecanismul de configurare al caracterelor care va fi aplicat la decodificarea parametrilor din obiectul request.

Etichetele JSTL pentru gestiunea informațiilor într-o bază de date SQL oferă funcționalitatea pentru interacțiunea cu baze de date relaționale ca Oracle, MySQL sau Microsoft SQL Server.

Pentru a putea referi aceste etichete, trebuie specificată locația de unde acestea pot fi încărcate.

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

Cele mai frecvent utilizate etichete JSTL pentru gestiunea informațiilor dintr-o bază de date sunt:

<sql:setDataSource>	creează un obiect `DataSource` adecvat numai operațiilor de prototipare Atributele asociate etichetei sunt: • `driver` – numele clasei corespunzătoare „driver”-ului JDBC care trebuie înregistrată; • `url` – URL-ul pentru realizarea conexiunii la baza de date; • `user` – numele de utilizator folosit în mecanismul de autentificare pentru baza de date; • `password` – parola folosită în mecanismul de autentificare pentru baza de date; • `dataSource` – baza de date pregătită anterior; • `var` – denumirea variabilei care va reprezenta baza de date; • `scope` – domeniul de vizibilitate al variabile care va reprezenta baza de date. `<sql:setDataSource var="connection" url="${dataBaseConnection}" user="${dataBaseUser}" password="${dataBasePassword}" />`
<sql:query>	execută interogarea SQL de tip `SELECT` definită în cadrul corpului folosind atributul `sql`, reținând rezultatul într-o variabilă având un anumit domeniu de vizibilitate Atributele asociate etichetei sunt: • `sql` – comanda SQL care va fi executată (ar trebui să întoarcă un obiect de tip `ResultSet`); • `dataSource` – conexiunea la baza de date care va fi utilizată (suprascrie valoarea implicită); • `maxRows` – numărul maxim de rezultate care vor fi stocate în cadrul variabilei; • `startRow` – numărul înregistrării (din cadrul rezultatului) de la care va începe stocarea; • `var` – denumirea variabilei care va reprezenta baza de date; • `scope` – domeniul de vizibilitate al variabilei care va reprezenta baza de date. `<sql:query sql="${myQuery}" var="result" />`

<sql:update>	execută instrucțiunea SQL de actualizare a informațiilor (de tip `INSERT`, `UPDATE`, `DELETE`) – care nu întoarce date, definită în cadrul corpului folosind atributul `sql`
	Atributele asociate etichetei sunt:
	• `sql` – comanda SQL care va fi executată (ar trebui să NU întoarcă un obiect de tip `ResultSet`);
	• `dataSource` – conexiunea la baza de date care va fi utilizată (suprascrie valoarea implicită);
	• `var` – denumirea variabilei care va stoca numărul de înregistrări afectate de instrucțiune;
	• `scope` – domeniul de vizibilitate al variabilei care va stoca numărul de înregistrări afectate de instrucțiune.
	`<sql:update dataSource="${connection}" var="count">`
	`  <%-- some INSERT, UPDATE or DELETE query -->`
	`</sql:update>`

<sql:param>	stabileşte valoarea unui parametru al unei instrucțiuni SQL, fiind folosit de obicei împreună cu etichetele `<query>` sau `<update>` Atributul asociat etichetei este `value` prin care se indică valoarea care se doreşte a fi asociată parametrului.

	`<sql:query dataSource="${connection}" var="result">`
	`  SELECT FROM table WHERE attribute = ?`
	`  <sql:param value="${missingTableAttributeValue}" />`
	`</sql:query>`
	`<sql:update dataSource="${connection}" var="count">`
	`  <%-- some INSERT, UPDATE or DELETE query with one or more missing attribute values -->`
	`  <sql:param value="${missingTableAttributeValue}" />`
	`</sql:update>`

<sql:dateParam>	stabileşte valoarea unui parametru al unei instrucțiuni SQL având tipul `java.util.Date` Atributele asociate etichetei sunt: • `value` – valoarea parametrului având tipul `java.util.Date`, care trebuie stocat; • `type` – poate avea valorile `DATE` (doar date caledaristice), `TIME` (doar timp) sau `TIMESTAMP` (dată calendaristică şi timp). Utilizarea sa este similară cu a etichete `<param>`.

<sql:transaction>	oferă elemente care realizează acțiuni folosind un obiect `Connection` partajat, executate în cadrul unei tranzacții; grupează interogări exprimate de etichetele `<query>` şi `<update>` într-o tranzacție, asigurând că efectul acestora este fie unitar asupra bazei de date, menținându-se starea sa inițială în cazul producerii unei erori.
	Atributele asociate etichetei sunt:
	• `dataSource` – conexiunea la baza de date care va fi utilizată (suprascrie valoarea implicită);
	• `isolation` – poate avea valorile `READ_COMMITED`, `READ_UNCOMMMITTED`, `REPEATABLE_READ` sau `SERIALIZABLE`.
	`<sql:transaction dataSource="${connection}">`
	`  <sql:update var="count1">`
	`  <%-- some INSERT queries -->`
	`  </sql:update>`
	`  <sql:update var="count2">`
	`  <%-- some UPDATE queries -->`
	`  </sql:update>`
	`  <sql:update var="count3">`
	`  <%-- some DELETE queries -->`
	`  </sql:update>`
	`</sql:transaction>`

Etichetele JSTL pentru gestiunea documentelor XML oferă posibilitatea de a manipula acest tip de resurse, implementând operații precum parsarea şi transformarea acestora precum şi controlul fluxului bazat pe expresii XPath.

Pentru a putea referi aceste etichete, trebuie specificată locația de unde acestea pot fi încărcate.

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

Cele mai frecvent utilizate etichete JSTL pentru gestiunea documentelor XML sunt:

<x:out>	evaluează valoarea unei expresii XPath pe care o afişează Atributele asociate etichetei sunt: • `select` (obligatoriu) – expresia XPath care trebuie evaluată, adeseori folosind variabile XPath; • `escapeXml` – indică dacă caracterele speciale XML ar trebui ignorate

<x:parse>	utilizat pentru a parsa date XML specificate fie prin intermediul unui atribut, fie în corpul etichetei
	Atributele asociate etichetei sunt:
	• `var` – variabila care conține datele XML parsate;
	• `xml` – textul documentului care se dorește a fi parsat (de tip `String` sau `Reader`);
	• `systemId` – URI-ul identificatorului de sistem pentru a parsa documentul;
	• `filter` – filtrul ce trebuie aplicat documentului sursă;
	• `doc` – documentul XML ce se dorește a fi parsat;
	• `scope` – domeniul de vizibilitate al variabilei specificate de atributul `var`;
	• `varDom` – variabila care conține datele XML parsate;
	• `scopeDom` – domeniul de vizibilitate al variabilei specificate de atributul `varDom`.
	`<x:parse xml="${myXmlDocument}" var="variable" />`
	`<x:out select="$variable/root/element[index]/attribute" />`

<x:set>	atribuie unei variabile valoarea unei expresii XPath; în funcție de tipul rezultat la evaluarea expresiei XPath, vor fi create obiecte având tipul `java.lang.Boolean`, `java.lang.String`, `java.lang.Number`.
	Atributele asociate etichetei sunt:
	• `var` (obligatoriu) – variabila care va reține valoarea expresiei XPath;
	• `select` – expresia XPath care se dorește evaluată;
	• `scope` – domeniul de vizibilitate al variabilei specificate de atributul `var`.
	`<x:parse xml="${myXmlDocument}" var="variable1" />`
	`<x:set var="variable2" select="$variable1/root/element" />`

<x:if>	evaluează o condiție ce are forma unei expresii XPath şi dacă aceasta se verifică se procesează corpul etichetei
	Atributele asociate etichetei sunt:
	• `select` (obligatoriu) – expresia XPath exprimând condiția ce se dorește evaluată;
	• `var` – variabila care va reține rezultatul evaluării condiției;
	• `scope` – domeniul de vizibilitate al variabilei specificate de atributul `var`.
	`<x:parse xml="${myXmlDocument}" var="variable" />`
	`<x:if select="$variable/root/element[index]/attribute != 0" />`
	`The attribute of the element at index is not null!`
	`</x:if>`

<x:forEach>	iterează asupra nodurilor unui document XML
	Atributele asociate etichetei sunt:
	• `select` (obligatoriu) – expresia XPath care va fi evaluată;
	• `var` – denumirea variabilei care va reține valoarea elementului curent pentru fiecare iterație;
	• `begin` – indexul de început pentru iterație;
	• `end` – indexul de sfârşit pentru iterație;
	• `step` – valoarea cu care va fi incrementat indexul pe parcursul incrementării colecției;
	• `varStatus` – denumirea variabilei în care se va reține statutul iterației
	`<x:parse xml="${myXmlDocument}" var="variable" />`
	`<x:forEach select="$variable/root/element/attribute" var="item" />`
	`<x:out select="$item" />`
	`</x:forEach>`

<x:choose>	etichetă de tip condițional care stabileşte un context pentru operații condiționale care se exclud mutual, marcate prin `<when>` şi `<otherwise>`
	Eticheta nu are atribute.
	`<x:parse xml="${myXmlDocument}" var="variable" />`
	`<x:choose>`
	`  <x:when select="$variable//element/attribute eq value1">`
	`  Attribute has value 1!`
	`  </x:when>`
	`  <x:when select="$variable//element/attribute eq value2">`
	`  Attribute has value 2!`
	`  </x:when>`
	`  ...`
	`  <x:when select="$variable//element/attribute eq valuen">`
	`  Attribute has value n!`
	`  </x:when>`
	`  <x:otherwise>`
	`  Attribute has unknown value!`
	`  </x:otherwise>`
	`</x:choose>`

<x:when>	subetichetă a etichetei `<choose>` al cărei corp este inclus dacă se verifică condiția asociată Eticheta are un singur atribut, select, care conține condiția ce se dorește a fi evaluată.
<x:otherwise>	subetichetă a etichetei `<choose>` ce urmează uneia sau mai multor etichete `<when>`, determinând evaluarea corpului său, dacă nici una dintre condițiile care o preced nu este îndeplinită. Eticheta nu are atribute.

<x:transform>	se aplică la transformare XSL a unui document XML
	Atributele asociate etichetei sunt:
	• `doc` – documentul XML sursă pentru transformarea XSLT;
	• `docSystemId` – URI-ul documentului XML original;
	• `xslt` (obligatoriu) – foaia de stil XSLT care conține instrucțiunile pentru transformare;
	• `xsltSystemId` – URI-ul documentului XSLT original;
	• `result` – obiectul rezultat care va accepta rezultatul transformării;
	• `var` – denumirea variabilei în care se va reține rezultatul transformării
	• `scope` – domeniul de vizibilitate ce va expune rezultatul transformării.
	`<x:parse xml="${myXmlDocument}" var="variable" />`
	`<c:import url="http://www.mysite.com/document.xsl" var="xslt" />`
	`<x:transform doc="${variable}" xslt="${xslt}" />`

<x:param>	utilizat împreună cu eticheta de transformare pentru a stabili un parametru în foaia de stil XSLT
	Atributele asociate etichetei sunt:
	• `name` – denumirea parametrului XSLT ce se doreşte a fi stabilit;
	• `value` – valoarea parametrului XSLT ce se doreşte a fi stabilit.
	`<x:transform doc="${variable}" xslt="${xslt}" />`
	`  <x:param name="attributeName" value="attributeValue" />`
	`</x:transform>`

Expresion Language permite folosirea de funcții în cadrul expresiilor, acestea trebuind să fie definite în cadrul unor biblioteci de etichete predefinite, apelul său făcându-se folosind sintaxa:

```
${ns:function(param1, param2, ..., paramn)}
```

JSTL include un număr de **funcții standard**, dintre care numeroase implică manipularea şirurilor de caractere.

Pentru a putea referi aceste etichete, trebuie specificată locația de unde acestea pot fi încărcate.

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

Lista de funcții standard JSTL este redată mai jos:

fn:contains()	verifică dacă şirul de caractere conține un alt subşir; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere, respectiv a subşirului; rezultatul întors este de tip `boolean`.
	`<c:set var="variable"`
	` value="${fn:contains(stringToBeSearched,`
	`substringToSearch)}" />`

fn:containsIgnoreCase()	verifică dacă şirul de caractere conține un alt subşir, ignorând capitalizarea; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere, respectiv a subşirului; rezultatul întors este de tip `boolean`.
	`<c:set var="variable"`
	` value="${fn:containsIgnoreCase(stringToBeSearched,`
	`substringToSearch)}" />`

fn:endsWith()	verifică dacă şirul de caractere are un anumit sufix; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere, respectiv a sufixului; rezultatul întors este de tip `boolean`.
	`<c:set var="variable"`
	` value="${fn:endsWith(stringToBeSearched, sufix)}" />`

fn:escapeXml()	transformă caracterele care ar putea fi interpretate ca adnotări XML la valoarea corespunzătoare; primeşte un parametru de tip `java.lang.String` având semnificația şirului de caractere care se doreşte a fi transformat; rezultatul întors este de tip `java.lang.String`.
	`<c:set var="variable"`
	` value="${fn:escapeXml(stringToBeXmlEscaped)}" />`

fn:indexOf()	întoarce prima apariție a unui subşir specificat în cadrul şirului de caractere; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere în care se realizează căutarea, respectiv a şirului care este căutat; rezultatul întors este de tip `int` (dacă şirul nu este găsit se întoarce valoarea -1)
	`<c:set var="variable"`
	` value="${fn:indexOf(stringToBeSearched,`
	`substringToSearch)}" />`

fn:join()	concatenează toate elementele unui vector într-un singur şir de caractere; primeşte doi parametrii, unul de tip `java.lang.String[]` conținând elementele vectorului şi unul de tip `java.lang.String`, reprezentând separatorul; întoarce un rezultat de tip `java.lang.String`
	`<c:set var="variable"`
	` value="${fn:join(arrayToBeJoined, separator)}" />`

fn:length()	întoarce numărul de elemente al colecției sau dimensiunea şirului de caractere; primeşte un parametru de tip `java.lang.Object` (de tip colecție sau şir de caractere); întoarce un rezultat de tip `int`.
	`<c:set var="variable1" value="${fn:length(myList)}" />`
	`<c:set var="variable2" value="${fn:length(myString)}" />`

fn:replace()	întoarce un şir de caractere obținut prin înlocuirea unei secvențe din cadrul său cu o altă secvență; primeşte trei parametrii de tip `java.lang.String`, reprezentând şirul de caractere în care se realizează înlocuirea, secvența care se doreşte înlocuită şi valoarea cu care aceasta este înlocuită; întoarce un rezultat de tip `boolean` `<c:set var="variable" value="${fn:length(myString, sequenceToReplace, sequenceToReplaceWith)}" />`
fn:split()	împarte şirul de caractere într-un vector de subşiruri primeşte doi parametrii de tip `java.lang.String`, reprezentând şirul de caractere care va fi împărțit, respectiv delimitatorii pe baza cărora se va realiza această operație; întoarce un rezultat de tip `java.lang.String[]`. `<c:set var="variable" value="${fn:split(myString, delimiter)}" />`
fn:startsWith()	verifică dacă şirul de caractere are un anumit prefix; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere, respectiv a prefixului; rezultatul întors este de tip `boolean`. `<c:set var="variable" value="${fn:startsWith(stringToBeSearched, prefix)}" />`
fn:substring()	întoarce un subşir al şirului de caractere; primeşte un parametru de tip `java.lang.String` reprezentând şirul din care se determină subşirul şi doi parametrii de tip `int` reprezentând pozițiile de început şi de sfârşit; rezultatul întors este de tip `java.lang.String`. `<c:set var="variable" value="${fn:substring(myString, startIndex, endIndex)}" />`
fn:substringAfter()	întoarce un subşir al şirului de caractere ce succede unei anumite secvențe; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere din care se determină subşirul, respectiv secvența după care urmează subşirul respectiv; rezultatul întors este de tip `java.lang.String`. `<c:set var="variable" value="${fn:substringAfter(myString, subSequence)}" />`
fn:substringBefore()	întoarce un subşir al şirului de caractere ce precede o anumită secvență; primeşte doi parametrii de tip `java.lang.String` având semnificația şirului de caractere din care se determină subşirul, respectiv secvența înainte de care se găseşte subşirul respectiv; rezultatul întors este de tip `java.lang.String`. `<c:set var="variable" value="${fn:substringBefore(myString, subSequence)}" />`

fn:toLowerCase()	transformă toate caracterele şirului în minuscule; primeşte un caracter de tip `java.lang.String` reprezentând şirul de caractere ce se doreşte a fi transformat; rezultatul întors este de tip `java.lang.String` `<c:set var="variable" value="${fn:toLowerCase(myString)}" />`
fo:toUpperCase()	transformă toate caracterele şirului în majuscule; primeşte un caracter de tip `java.lang.String` reprezentând şirul de caractere ce se doreşte a fi transformat; rezultatul întors este de tip `java.lang.String` `<c:set var="variable" value="${fn:toUpperCase(myString)}" />`
fn:trim()	elimină spațiile care se găsesc la începutul şi sfârşitul şirului de caractere; primeşte un caracter de tip `java.lang.String` reprezentând şirul de caractere care se doreşte a fi transformat; rezultatul întors este de tip `java.lang.String` `<c:set var="variable" value="${fn:trim(myString)}" />`

În JavaServer Pages pot fi implementate etichete definite de utilizator, care în momentul transformării servlet-ului corespunzător sunt convertite în operațiile specificate de clasa care îi este asociată, pentru ca atunci când se accesează pagina respectivă, acestea să fie executate. O clasă care defineşte funcționalitatea unei anumite etichete trebuie să fie derivată din `javax.servlet.jsp.tagtext.SimpleTagSupport`, implementând metoda `doTag()`. Aceasta poate accesa obiectele din cadrul paginii JSP prin intermediul metodei `getJSPContext()`. De asemenea, în cazul în care în corpul etichetei se specifică anumită valoare, aceasta poate fi obținută într-un obiect de tip `StringWriter` prin metoda `getJspBody().invoke()`.

```
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class CustomTag extends SimpleTagSupport {

    private String customAttribute;
    public void setCustomAttribute(String customAttribute) {
        this. customAttribute = customAttribute;
    }
    public String getCustomAttribute() {
        return customAttribute;
    }

    StringWriter stringWriter = new StringWriter();
    public void doTag() throws JspException, IOException {
        if (customAttribute != null) {
            // process tag attribute
            getJspContext().getOut().println(customAttribute);
        } else {
            // get the content from the tag body
            getJspBody().invoke(stringWriter);
            // process tag body's content
            getJspContext().getOut().println(stringWriter.toString());
        }
    }

}
```

Clasa asociată etichetei definite de utilizator va trebui plasată la o locație vizibilă prin variabila de mediu `CLASSPATH`.

Asocierea dintre etichetă şi clasa asociată se face într-un fişier `custom.tld` care va fi plasat în `webapps/root/WEB-INF` al serverului web Apache Tomcat 7.x. Astfel, se va specifica numele etichetei (aşa cum va fi specificat în pagina JSP – elementul `<name>`, denumirea clasei care îi implementează funcționalitatea (elementul `<tag-class>`) precum şi tipul de conținut al corpului etichetei (elementul `<body-content>`, având ca valori posibile `empty`, dacă trebuie să fie vid, `scriptless` dacă este acceptat numai text static, expresii EL sau alte etichete şi `tagdependent` în cazul în care conținutul este scris în alt limbaj, fiind interpretat de implementarea etichetei). De asemenea, pentru fiecare atribut trebuie indicate proprietățile `name` (identificator unic al atributului în cadrul etichetei), `required` (care distinge între atributele obligatorii şi cele opționale), `rtexprvalue` (care stabileşte validitatea valorii unei expresii definită în momentul rulării pentru atributul etichetei), `type` (care specifică tipul Java al atributului respectiv, implicit fiind `java.lang.String`), `description` şi `fragment` (care indică dacă valoarea asociată atributului va fi tratată ca un `JspFragment`).

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>A short description of my custom tag</short-name>
  <tag>
    <name>CustomTag</name>
    <tag-class>mypackage.CustomTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
        <name>customAttribute</name>
        <required>true</true>
        <type>java.lang.Integer</type>
        <fragment>false</fragment>
    </attribute>
  </tag>
</taglib>
```

Pentru a putea fi folosită în cadrul unei pagini JSP, eticheta definită de utilizator trebuie declarată în acelaşi mod în care se procedează şi cu etichetele standard JSTL:

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

În momentul în care este utilizată, eticheta va avea numele definit în fişierul `custom.tld`, fiind prefixată cu denumirea indicată la declararea ei, precizând valori pentru toate atributele care au proprietatea `required`:

```
...
<customPrefix:CustomTag customAttribute="0">
    Some custom tag body
</customPrefix:CustomTag>
...
```

Potrivit implementării clasei asociate etichetei, efectul utilizării sale va fi afişarea valorii atributului specificat, în situația în care nu se decide realizarea altor tipuri de procesări asupra ei.

Se pot defini mai multe biblioteci de etichete definite de utilizator, grupate în funcție de tipul de funcționalitate pe care o pun la dispoziția programatorilor, fiecare dintre acestea regăsindu-se într-un alt fişier şi utilizându-se un prefix diferit.

Activitate de Laborator

[0p] 1. Să se instaleze baza de date prin rularea script-ului `Laborator09.sql`.

Conținutul scriptului `Laborator09.sql` este identic cu cel din `Laborator04.sql` astfel că această etapă nu mai este necesară în cazul în care aceasta a fost deja realizată.

[0p] 2. În cazul Unix, dați drepturi de execuție tuturor fișierelor .sh din directorul `bin`, întrucât acestea sunt apelate la rândul lor de `startup.sh`, respectiv `shutdown.sh`.

```
chmod +x *.sh
```

[0p] 3. Să se completeze în clasa `Constants` din pachetul `general` valorile pentru utilizatorul şi parola pentru accesarea bazei de date.

[0p] 4. Să se „publice" aplicația BookStore în contextul serverului Apache Tomcat

Detalii cu privire la „publicarea" unei aplicații în contextul serverului Apache Tomcat sunt redate în Laboratorul 8 (pentru mediile de dezvoltare NetBeans şi Eclipse EE, cât şi pentru cazul în care nu se foloseşte un mediu de dezvoltare);

[0p] 5. Să se testeze aplicația prin accesarea adresei http://localhost:8080/BookStore/.

În script-ul `Laborator09.sql` există exemple de utilizatori care pot fi folosite pentru accesarea paginilor de administrator, respectiv de client.

[0p] 6. Să se acceseze pagina de administrator şi să se testeze funcționalitățile implementate în cadrul acesteia (adăugare, editare, ştergere).

[1p] 7. În pagina `client.jsp`, să se construiască clauza `WHERE` a interogării corespunzătoare tabelei `carti`, astfel încât conținutul tabelei să fie filtrat după valoarea selectată din lista de colecții / domenii.

Filtrarea se va realiza numai în cazul valorile selectate sunt diferite de şirul de caractere 'toate'. În cazul în care vor fi specificate ambele valori, condiția de filtrare va presupune îndeplinirea ambelor criterii.

Pentru fiecare colecție / domeniu există mai mulți identificatori în tabelele aferente (colecții / domenii) care au denumirea respectivă, astfel încât interogarea va avea forma:

```
SELECT ... FROM carti
WHERE
    idcolectie IN (SELECT idcolectie FROM colectii WHERE denumire='... ')
    AND
    iddomeniu IN (SELECT iddomeniu FROM domenii WHERE denumire= '... ');
```

În specificația EL 2.2 concatenarea a două şiruri de caractere se face prin alăturarea lor (operatorul + pentru şiruri de caractere este definit în EL 3.0):

```
<c:set scope= "request" var= "myvar" value="${var1} some text ${var2}" />
```

[3,50p] 8. În clasa `ClientServlet` să se completeze metoda `doPost` pentru a crea conținutul coşului de cumpărături, ținând cont de situația în care pentru un produs care există deja în coşul de cumpărături se poate actualiza cantitatea.

În situația în care solicitarea pentru un produs depăşeşte stocul existent, aceasta nu va fi luată în considerare, afişându-se un mesaj de eroare.

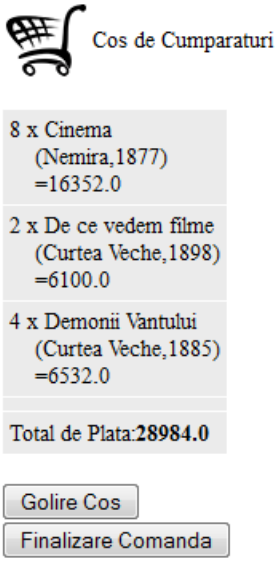
Dacă un produs există deja în coşul de cumpărături, valoarea respectivă va fi suprascrisă. Dacă valoarea este 0, produsul va fi şters din coşul de cumpărături.

Sugestii de Rezolvare. Parametrul care conține identificatorul cărții care se doreşte adăugată în coş are forma exemplare_idCarte (de exemplu, exemplare_1, pentru idCarte=1). Se verifică dacă parametrul are această formă şi în acest caz, identificatorul cărții se obține prin parsarea acestui câmp, iar numărul de exemplare prin apelul `request.getParameter(parameter)`. Dacă în coşul de cumpărături există o intrare cu acelaşi idCarte, aceasta este actualizată (suprascrisă / ştearsă în cazul în care numărul de exemplare este 0), altfel se introduce o nouă intrare, nu înainte de a se verifica faptul că operația este posibilă (numărul de exemplare solicitat nu depăşeşte stocul). În cazul în care sunt solicitate mai multe exemplare decât sunt disponibile, se va afişa un mesaj de eroare.

Coşul de cumpărături este un obiect `ArrayList<Record>` unde ca atribut se reține identificatorul cărții, iar ca valoare numărul de exemplare.

[1,50p] 9. În pagina `client.jsp`, să se afişeze conținutul coşului de cumpărături.

În coşul de cumpărături se va afişa numărul de exemplare comandate, titlul cărții, editura şi anul, precum şi suma pentru fiecare produs în parte, respectiv prețul total pentru întreg coşul de cumpărături.



În cazul în care rezultatul unei interogări se află într-o variabilă result, elementele sale pot fi obținute prin indexare, începând cu 0: `result.rows[0]`, `result.rows[1]`, ..., `result.rows[n-1]`. Acestea sunt nişte tablouri care conțin toate atributele interogării, acestea putând fi accesate tot prin indexare, prin numele lor: `result.rows[k]['attribute1']`, `result.rows[k]['attribute2']`, ..., `result.rows[k]['attributem']`.

[1p] 10. În pagina `client.jsp` să se să se adauge două butoane prin care se poate anula, respectiv finaliza o comandă (o comandă conține produsele selectate în coşul de cumpărături).

Aceste butoane vor fi afişate numai în condițiile în care coşul de cumpărături nu este gol.

[4p] 11. În clasa `ClientServlet`, metoda `doPost`, să se implementeze operația pentru anularea, respectiv finalizarea unei comenzi.

Sugestii de Rezolvare. În cazul anulării unei comenzi, se şterge conținutul obiectului shoppingCart.

În cazul finalizării unei comenzi, trebuie realizate următoarele operații:

❶ adăugarea unei înregistrări în tabela facturi; pentru generarea aleatoare a câmpului serienumar se poate folosi metoda `Utilities.generateBillNumber()`; ca dată se va completa data curentă (prin funcția MySQL `CURDATE()`), starea este emisă, iar CNP-ul utilizatorului va fi dedus din `userDisplayName` care este concatenarea dintre prenumele şi numele reținute în tabela utilizatori;

❷ pentru fiecare intrare din coşul de cumpărături:

a) se compară numărul de exemplare din coşul de cumpărături cu stocul din tabela cărți şi dacă relația este de tipul <=, se actualizează stocul (scăzând exemplarele comercializate);
b) se adaugă o înregistrare în tabela detalii_factura;

❸ se goleşte coşul de cumpărături.

[1p] 12. Să se implementeze operația de deautentificare printr-un buton plasat sub mesajul de întâmpinare pentru fiecare utilizator în pagina de tip administrator, respectiv client. De asemenea, utilizatorul se va întoarce în pagina de autentificare (`login.jsp`).

Se va lucra în clasele `AdministratorServlet` şi `ClientServlet`, plasarea butoanelor realizându-se în paginile `administrator.jsp` şi `client.jsp`.

După apelul metodei `forward` pentru obiectul `requestDispatcher`, este recomandat să se apeleze metoda `return` pentru că apelul ulterior al altor metode poate genera o excepție de tipul `IllegalStateException`.

Bibliografie

[1] Giulio Zambon with Michael Sekler – *Beginning JSP™, JSF™ and Tomcat Web Development: From Novice to Professional*, *Apress*, 2007
[2] Irina Athanasiu – *Java ca limbaj pentru programarea distribuită*, editura Matrix Rom, Bucureşti, 2002
[3] *JSP Tutorial* - http://www.tutorialspoint.com/jsp/index.htm
[4] *JSP Tutorial* - http://www.jsptutorial.net/
[5] *Servlet World* - http://www.servletworld.com/jsp-tutorials/index.html