# Le *JavaServer Pages - Lezione 3*

# Le direttive delle JSP

A cura di Giuseppe De Pietro (depietro_giuseppe@yahoo.it)

## *Contenuti*

Nelle prime due lezioni abbiamo visto come funzionano le JSP e mostrato dei semplici esempi. Credo che sia ormai chiaro di come attraverso la *page translation* e *page compilation*, sia possibile ottenere delle *servlet* partendo dai *template* delle pagine JSP.
È opportuno ora illustrare nel dettaglio gli elementi che compongono una pagina JSP e i ruoli che essi assumono durante la trasformazione della pagina in una classe Java.
Un documento può contenere i seguenti elementi:

- ***direttive***: indicazioni sulle regole da utilizzare durante la trasformazione in servlet.
- ***scriptlet***: sono le istruzioni Java inserite tra i tag <% .. %>, il codice viene riportato nel metodo *jspService()* della servlet generata.
- ***dichiarazioni***: sono racchiuse tra i tag <%! ..%> e contengono metodi e variabili che devono essere visibili a tutta la servlet (il codice viene inserito al di fuori del metodo *jspService()* ).
- ***azioni***: sono dei tag XML usati per poter aggiungere delle risorse aggiuntive alla pagina JSP. Oltre alle azioni standard (standard action per l'inclusione di componenti JavaBean), esiste la possibilità di definire delle azioni personalizzate dando così la possibilità allo sviluppatore di utilizzare dei propri tag.

Questa è una classificazione di base dei componenti a cui vanno aggiunti anche altri elementi. Nelle successive lezioni quindi si parlerà anche di oggetti impliciti, di commenti JSP, di espressioni.
In questa lezione parleremo delle direttive e di come grazie ad esse sia possibile impostare delle proprietà utilizzate dal *Jsp Container* in fase di compilazione.
La sintassi di base è la seguente:

```
<%@ direttiva attributo1="valore1" attributo2="valore2" ... %>
```

Vedremo ora le possibili direttive con tutti gli attributi.

## *La direttiva page*

La direttiva page permette di impostare le proprietà della pagina JSP in fase di page translation. Essa può essere utilizzata più volte specificando però ciascun attributo una sola volta, l'unico attributo che può essere ripetuto è *import*.

| ***Attributi della direttiva Page*** | | |
|---|---|---|
| **Attributo** | **Tipo di valore** | **Valore di Default** |
| language | Nome del linguaggio di scripting | "java" |
| info | String | Dipende dal JSP Container |
| contentType | Tipo MIME o set di caratteri | "text/html;charset=ISO-8859-1" |
| extends | Nome classe | Nessuno |
| import | Nome completo della classe o del package | Nessuno |
| buffer | Grandezza del buffer in byte o *false* | 8192 |
| autoFlush | Boolean | "true" |
| session | Boolean | "true" |
| isThreadSafe | Boolean | "true" |
| errorPage | URL | None |
| isErrorPage | Boolean | "false" |

## L'attributo *language*

Inizialmente questo attributo poteva aver senso perché le JSP erano state progettate per funzionare con più linguaggi di scripting. In realtà non è mai stato implementato alcun linguaggio aggiuntivo tanto che dalla versione 1.2 delle JSP l'unico valore ammesso è Java ed è anche quello di default.

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

## L'attributo *info*

Permette di impostare un testo descrittivo per la pagina JSP. Tale testo sarà reso disponibile dal metodo getServletInfo() della Servlet corrispondente.

*Esempio 3.1*

```
<%@ page language="java" info="Esempio di direttiva page info (Es. 3_1)" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Esempio 3_1</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>

<body>
<h1 align='center'>Esempio 3.1</h1><br/>
<h3>Chiamata del metodo getServletInfo()</h3>
<%
      out.write(getServletInfo());

%>
</body>
</html>
```

## L'attributo *contentType*

Con questo attributo è possibile indicare il tipo MIME (Multipurpose Internet Mail Extension) e la codifica dei caratteri che il jsp container dovrà utilizzare nella generazione della pagina HTML.
Per esempio il seguente valore

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

visualizza la pagina in formato testo e non HTML, mostrando i tag senza interpretarli (*Esempio 3.4* non funziona con Internet Explorer 6 che adotta sempre il tipo text/html).
L'*esempio 3.5* mostra come sia possibile impostare il tipo MIME di un file su di una immagine *png*.

```
<%@ page language="java" %>
<%@ page import="java.awt.Color" %>
<%@ page import="java.awt.Graphics2D" %>
<%@ page import="java.awt.image.BufferedImage" %>
<%@ page import="java.awt.image.RenderedImage" %>
<%@ page import="java.io.OutputStream" %>
<%@ page import="javax.imageio.ImageIO" %>
<%@ page contentType="image/png" %>
<html>
<head>
<title>Esempio 3_5</title>
</head>
<body>
```

```
<%
    int width = 300;
    int height = 300;
    //Creazione dell'immagine di tipo RGB
      BufferedImage bufferedImage = new BufferedImage(width,
height,BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = bufferedImage.createGraphics();
      OutputStream out1 = response.getOutputStream();
     // disegno all'interno dell'immagine Graphics2D
        for (int j = 0; j < 3; j++) {
            int a = 0;
            int b = 0;
            int c = 0;

            if (j == 0) {
                a = 51;
                b = 0;
                c = 0;
            }

            if (j == 1) {
                a = 0;
                b = 0;
                c = 51;
            }

            if (j == 2) {
                a = 51;
                b = 0;
                c = 51;
            }

            for (int i = 0; i < 3; i++) {
                g2d.setColor(new Color(a + (a * i), b + (b * i), c + (c * i)));
                g2d.fillRect(100 * j, 100 * i, width, height);
            }
        }

        for (int j = 0; j < 3; j++) {
            for (int i = 0; i < 3; i++) {
                g2d.setColor(Color.black);
                g2d.drawRect(100 * j, 100 * i, width - 1, height - 1);
            }
        }
        g2d.dispose();

        RenderedImage rendImage = bufferedImage;
        ImageIO.write(rendImage, "png", out1);
%>
</body>
</html>
```

Inoltre e possibile specificare anche la sola codifica dei caratteri con l'attributo pageEncoding. Il valore di default è:

```
<%@ page pageEncoding="ISO-8859-1" %>
```

## L'attributo *extends*

Java come linguaggio ad oggetti supporta l'ereditarietà, ovvero quella particolare caratteristica di una classe di poter rendere propri, metodi e proprietà della classe da cui eredita (*extends*). Di conseguenza anche le Java Server Pages supportano l'erediarietà.

Esaminando le servlet generate dal JSP Container si nota come esse derivano dalla classe *org.apache.jasper.runtime.HttpJspBase* (varia a seconda del container) che è a sua volta una implementazione dell'interfaccia *javax.servlet.jsp.HttpJspPage* che fa parte di J2EE. Volendo si potrebbe far derivare la pagina JSP da una nostra implementazione della stessa interfaccia, basta definirla con l'attributo *extends*. Per esempio:

```
<%@ page extends="it.depietro.jsp.NewHttpJspPage" %>
```

Si sconsiglia tuttavia l'utilizzo di tale attributo perchè potrebbe limitare le funzionalità del *container*.

## L'attributo *import*

Questo attributo ha le stesse funzioni dell'istruzione import di una classe Java. Ci permette di poter utilizzare i nomi delle classi senza dover indicare il package completo. È l'unico attributo che può essere ripetuto, perché ovviamente potremmo aver bisogno di importare più package e supporta varie sintassi:

```
<%@ page import="java.lang.*,java.util.*" %>
```

Oppure

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

Oppure

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

Utilizzarli significa comunque aumentare la leggibilità del codice. Per esempio per poter utilizzare l'oggetto Calendar (per prelevare la data) dovremmo scrivere:

```
java.util.Calendar data = java.util.Calendar.getInstance();
```

mentre con un

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

potremmo scrivere

```
Calendar data = Calendar.getInstance();
```

rendendo il codice più leggibile.

## Gli attributi *buffer* e *autoFlush*

Questi due attributi permettono di impostare il tipo di bufferizzazione della pagina JSP. Di default una pagina, in fase di elaborazione, viene temporaneamente conservata nel buffer. Al client non viene inviato nulla finché il processo di elaborazione non è terminato o il buffer è pieno (raggiungimento delle dimensioni di default di 8kb).
La bufferizzazione migliora le prestazioni dell'applicativo Web perché il JSP Container impiegherà meno tempo ad inviare l'output una sola volta che non ogni volta che trova le istruzioni di scrittura sul client.
Il buffer può essere disabilitato con il comando:

```
<%@ page buffer="none" %>
```

O si può modificare la sua dimensione impostandola per esempio a 32kb con:

```
<%@ page buffer="32kb" %>
```

Inoltre la funzione di scaricamento automatico del buffer al suo riempimento può essere gestita con l'attributo *autoFlush* (default settato a *true*), se si cambia il valore in *false* bisognerà provvedere manualmente allo svuotamento del buffer perché se quest'ultimo raggiunge le dimensioni massime, viene generata un'eccezione che interrompe l'esecuzione della pagina.
Si consiglia di lasciare inalterate queste impostazioni perché di solito sono quelle che ottengono le migliori prestazioni dal server.

## L'attributo *session*

Quando un utente accede ad un sito viene creata una sessione relativa a quell'utente. In pratica una sessione può essere vista come un contenitore che tiene traccia di tutti gli spostamenti e tutti gli oggetti relativi solo a quell'utente specifico. Ogni sessione è separata e distinta dalle altre, quindi un utente non potrà mai accedere agli oggetti di un altro utente.
Il protocollo HTTP non permette la gestione delle sessioni perché è un protocollo senza memoria, quindi ogni richiesta fatta da uno stesso client viene visto sempre come nuovo utente. In tanti applicativi web è fondamentale riconoscere le richieste effettuate da uno stesso utente, per esempio i siti di e-commerce si servono delle sessioni per memorizzare gli articoli selezionati dall'utente.
Tale meccanismo dovrà quindi essere gestito dal Web Server tenendo conto delle limitazioni del protocollo HTTP.
Di solito tutti i Web Server adottano la soluzione dei cookie per riconoscere l'utente. Alla prima connessione, il Web Server genera un numero univoco e identificativo (*Session ID*) che verrà salvato sul client come cookie. Alle successive richieste verrà letto l'Id sul client ed associato alla rispettiva sessione utente. Il problema per questo meccanismo potrebbe essere la disabilitazione dei cookie sul client, in questo caso Tomcat (e qualche altro Web Server) adotta la soluzione dell'identificativo associato all'URL riconoscendo così l'utente ad ogni richiesta.
Tutte queste operazioni non dovranno essere gestite dallo sviluppatore che potrà solo abilitare o disabilitare le sessioni utente (di default sono abilitate). Si consiglia di disabilitare le sessioni nelle pagine in cui non è richiesta questa funzionalità, alleggerisce il carico del server migliorando i tempi di risposta delle pagine:

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

## L'attributo *isThreadSafe*

Ogni volta che una pagina JSP viene richiesta, il container esegue il metodo *_jspService()* in processi distinti, quindi più istanze della stessa pagina potrebbero essere in esecuzione contemporaneamente. Ci sono situazioni particolari in cui più accessi simultanei ad una stessa risorsa potrebbero creare dei problemi, per esempio una pagina che esegue operazioni di scrittura file. In questi casi è opportuno autorizzare solo accessi singoli alla pagina impostando la direttiva:

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

Se la pagina è già in esecuzione, le successive richieste saranno poste in attesa del loro turno, in questo modo è possibile gestire senza problemi le risorse condivise. L'inconveniente di tale gestione è che andrebbe utilizzato con cautela in siti con un alto numero di accessi perché causerebbe una maggiore latenza di risposta.

## Gli attributi *errorPage* e *isErrorPage*

Nella lezione 2 abbiamo parlato della robustezza del linguaggio Java che, grazie alla sofisticata gestione degli errori (costrutto *try..catch*), è in grado di tenere sempre sotto controllo l'applicazione senza causare blocchi indesiderati.
Purtroppo sappiamo benissimo che per quanto riguarda gli applicativi Web i fattori in gioco che potrebbero minare la stabilità delle pagine sono numerosi. Non basta gestire gli errori che il codice potrebbe generare o testare la nostra applicazione per essere sicuri che tutto funzioni sempre perfettamente. Dobbiamo dare per scontato che prima o poi si verifichi qualche situazione anomala e prepararci a gestirla nel migliore dei modi.
In questo caso, visto che l'errore non si può evitare, si devono prevedere delle pagine di errore alternative, più amichevoli per l'utente ignaro della tecnologia che c'è dietro. Preparare messaggi che avvisino che il problema è momentaneo e il tutto spiegato con un linguaggio semplice e non con codici incomprensibili. In conclusione è importante dimostrare ai nostri utenti che tutto sia sempre sotto controllo.

Gli attributi *errorPage* e *isErrorPage* ci aiutano in questo. Il primo ci permette di specificare la pagina da caricare in caso di errore non gestito, mentre il secondo impostato a *true* ci permette di poter avere a disposizione nella pagina di errore, l'oggetto implicito *exception* (vedremo meglio cosa sono nelle lezioni successive) che conterrà tutti i messaggi e i codici dell'errore verificatosi.
Capiremo meglio facendo un esempio. Creiamo la pagina *esempio3_7.jsp* dove volutamente è stato inserito un errore non gestito, al momento del verificarsi dell'eccezione verrà caricata la pagina *mioErrore.jsp* indicata tramite l'attributo *errorPage*. Nella pagina *mioErrore.jsp* imposteremo l'attributo *isErrorPage* a *true* così potremo visualizzare oltre ai messaggi destinati all'utente comune, anche i messaggi tecnici destinati al gestore del sito.
*Esempio3.7*

```
<%@ page language="java" errorPage="mioErrore.jsp" %>
<%@ page import="java.io.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Esempio 3_7</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1 align='center'>Esempio 3.7</h1><br/>
<h3>Gestione degli errori tramite direttiva page</h3>
<%
//si tenta di aprire un file inesistente
FileInputStream fileIn=new FileInputStream( "inesistente.txt");
%>
</body>
</html>
```

La pagina *mioErrore.jsp*

```
<%@ page language="java" isErrorPage="true" %>
<%@ page import="java.io.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Esempio 3_7 Pagina di errore</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1 align='center'>Esempio 3.7</h1><br/>
<h3>Pagina di errore personalizzata</h3>
<br/>
<h4>La pagina richiesta ha generato un errore.</h4>
<h4>Il problema sarà corretto il pi&ugrave; presto possibile.</h4>
<br/><br/><br/><br/><hr/>
Per il webmaster:
<p>Messaggio di errore:</p>
<p><%= exception.getMessage()%></p>
<p>Stack Trace:</p>
<pre><%
 exception.printStackTrace(new PrintWriter(out));
 %></pre>
</body>
</html>
```

## *La direttiva include*

Ogni sito in genere rispetta un proprio modello con sezioni predefinite, intestazione, menu laterali, piè di pagina ed altro. Spesso tanti elementi sono comuni alla maggior parte delle pagine e in una tecnologia statica (solo pagine HTML), l'unica soluzione a questo problema è copiare lo stesso codice in tutte le pagine che ne fanno uso. I problemi di tale tecnica (anche se risolti in parte da alcuni ambienti di sviluppo) sono evidenti: ridondanza del codice e difficoltà di manutenzione.
Con le tecnologie lato server si hanno a disposizione strumenti di inclusione di file che risolvono egregiamente il problema.
JSP essendo una tecnologia molto evoluta mette a disposizione due tipi di inclusione, in questa lezione ne vedremo solo uno evidenziandone i pregi e i difetti e i contesti di applicazione.
La direttiva include ha la seguente sintassi:

```
<%@ include file="file_da_includere" %>
```

L'URL del file è relativo ed è possibile includere qualsiasi file testuale, quindi sia codice statico HTML che scriptlet Java.
Quello che avviene con questo tipo di applicazione è che il codice incluso viene copiato staticamente nella servlet in fase di *page translation*.
Vediamo un primo esempio in cui potrebbe essere utile utilizzare la direttiva include, la creazione di template da utilizzare per il nostro sito. Creiamo dei file HTML che conterranno le parti comuni a tutte le pagine della nostra applicazione e che saranno incluse staticamente con la direttiva *include*.
La pagina *index.jsp* includerà i file *header.html* e *menu.html*:

*index.jsp*

```
<%@ page language="java" session="false" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Esempio di direttiva include</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<link rel="stylesheet" type="text/css" href="stili.css">
</head>
<body>
<%@ include file="header.html" %>
<div id="corpo">
      <%@ include file="menu.html" %>
      <div id="main">
            <div class="box">
                  <h3>Corso JSP - Lezione 3</h3>
                  <h3>Home page</h3>
                  <p>Esempio di inclusione di file HTML</p>
            </div>
      </div>
</div>
</body>
</html>
```

*header.html*

```
<div id="testata">
      <h1>Intestazione Sito</h1>
      <h3>(Esempio di direttiva include)</h3>
</div>
```

*menu.html*

```
<div id="menu">
	<div class="box">
		<ul>
			<li><a title="pagina 1" href="pagina1.jsp">Pagina 1</a></li>
			<li><a href="#">Pagina 2</a></li>
		</ul>
	</div>
</div>
```

Infine il foglio di stile *stili.css* per la gestione dei div:

```
BODY
{
	margin:		0 1px 0 0;
	padding:	10px;
}
#testata
{
	background-color: #dcb2ff;
	padding:10px;
}
#menu
{	float: left;
	width: 20%;}
#main
{	float: left;
	width: 80%;
	background-color: gainsboro;  }
#main1
{	float: left;
	width: 80%;
	background-color: #A6C4CE;	}
.box
{
	padding:10px;
}
```

Ovviamente potremo includere non solo del codice statico ma anche del codice Java. La direttiva include diventa indispensabile quando vogliamo creare dei metodi da utilizzare in più pagine.
Facciamo un esempio, creiamo un metodo che dato un testo ci restituirà una stringa compatibile nel linguaggio SQL (quindi racchiusa tra singoli apici e apici raddoppiati in caso di occorrenza all'interno del testo) e poi un secondo metodo che dato un oggetto ci restituirà la sua trasformazione in stringa o nel caso di oggetto nullo una stringa vuota (sono metodi che poi ci serviranno quando lavoreremo con i database). Il file da includere sarà *funzioni.jsp*:

```
<%!
public String fc(String campo){
	campo="'"+ campo.replaceAll("'","''") + "'";
	return campo;
}
public String nn(Object oggetto){
	if (oggetto == null) return "";
	return oggetto.toString();
}
%>
```

Notate il punto esclamativo nella prima riga, esso indica che si tratta di una *declaration* (dichiarazione) e non di uno *scriptlet*. Per ora non interessa conoscere la differenza tra questi concetti, comunque li affronteremo nella lezione successiva. Mentre la pagina che includerà il codice la chiameremo *pagina2.jsp*:

```
<%@ page language="java" session="false" %>
<%@ include file="funzioni.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Esempio di inclusione di codice</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>

<body>
<h1 align='center'>Esempio di inclusione di codice JSP</h1><br/>
<p>La pagina invierà a se stessa i parametri e visualizzer&agrave; gli stessi
in formato compatibile in SQL</p>

<form method="post">
  <fieldset>
      <legend>Inserisci i dati</legend>
      <label for="nome">Nome:</label>
      <input type="text" id="nome" name="nome"><br>
      <label for="cognome">Cognome:</label>
      <input type="text" id="cognome" name="cognome"><br>
      <input type="submit" value="Invia valori" name="invia">
   </fieldset>
 </form>
<%
if (request.getParameter("invia")!=null){
      out.write("Nome :");
      out.write(nn(fc(request.getParameter("nome"))));
      out.write("<br/>Cognome :");
      out.write(nn(fc(request.getParameter("cognome"))));
 }
%>
</body>
</html>
```

Riassumendo possiamo dire che con la direttiva include, è come copiare il testo (sia statico che codice JSP) all'interno della pagina stessa. Il metodo è molto comodo evitando così la ridondanza di codice senza avere alcun effetto sulle prestazioni, però ci sono degli inconvenienti.
Il primo è che bisogna fare attenzione al container che dovrà processare le pagine. Abbiamo detto che una pagina JSP viene tradotta in una servlet, se la servlet è già stata creata il container non effettuerà nuovamente la conversione ma caricherà la classe java esistente. La riconversione avverrà solo quando la pagina JSP sarà modificata, ma prendiamo in esame l'esempio precedente. Al primo caricamento Tomcat copierà il file *funzioni.jsp* nel file *pagina2.jsp* e la tradurrà in una servlet. Ma cosa succede se noi modifichiamo il file *funzioni.jsp*? La *pagina2.jsp* non risulterà modificata al container e quindi potrebbe non aggiornare la servlet corrispondente anche perché le specifiche J2EE di Sun Microsystems non prevedono queste verifiche. Il che si traduce in un lavoro di compilazione manuale delle pagine.
In realtà le ultime versioni degli Application Server cominciano a prevedere delle annotazioni delle dipendenze di file. Per esempio la versione 5.x di Tomcat aggiorna sempre i contenuti, quindi non avrete questo problema, ma fate attenzione se doveste pubblicare il vostro sito su di un server che utilizza un altro container, verificate sempre che il tutto funzioni.

Vi riporto una parte del codice della servlet generata nell'esempio precedente, in cui Tomcat annota le dipendenze dei file inclusi:

```
private static java.util.Vector _jspx_dependants;

  static {
    _jspx_dependants = new java.util.Vector(1);
    _jspx_dependants.add("/Lezione3/Include/funzioni.jsp");
  }

  public java.util.List getDependants() {
    return _jspx_dependants;
  }
```

Un altro problema all'inclusione statica è quello che in presenza di tante righe di codice all'interno del metodo _jspService() potrebbe causare problemi alla JVM (Java Virtual Machine) che ha un limite nelle dimensioni dei metodi di una classe. Più avanti vedremo un altro metodo di inclusione che risolve tali problemi (pur avendo altre limitazioni).

## *La direttiva taglib*

Accenneremo solo per completezza a questa direttiva, vedremo degli esempi nelle lezioni successive quando si parlerà dei custom tag.
La direttiva *taglib* ci permette di estendere i tag JSP utilizzando funzionalità aggiuntive.