Java Server Side: JSTL

Ing. Cesare Monti - 27 aprile 2005

cosa vedremo:

- MY TAG
- JSTL
- Expression Language

JSTL

- come si fa quindi a creare dei tag personalizzati?
  - ... se è codice java associato a tag ... occorre scrivere sicuramente una classe
    - le classi per la creazione dei tag personalizzati sono due:
      - javax.servlet.jsp.tagext.SimpleTagSupport
      - javax.servlet.jsp.tagext.BodyTagSupport
    - o meglio basta estendere una di queste due classi per creare una tag personalizzato

JSTL

- javax.servlet.jsp.tagext.SimpleTagSupport
  - è il caso più semplice
  - si tratta di un TAG che non presenta un corpo (body)
    - es <myTag:faQualcosa />
  - che può presentare differenti attributi
    - es <myTag:faQualcosa attributo1="hello" attributo2="world"/>
  - o meglio che vincola i parametri necessari alla sua esecuzione nell'invocazione del tag e non nel suo corpo
    - è valida anche la notazione
      - <myTag:faQualcosa attributo1="hello" attributo2="world">
      - ...
      - </myTag:faQualcosa>

JSTL

- questi sono i tag più semplici
  - dobbiamo programmare la loro azione
    - all'apertura del tag
    - eventualmente alla chiusura del tag

JSTL

- javax.servlet.jsp.tagext.BodyTagSupport
  - al contrario del precedente permette di poter specificare alcuni parametri nel corpo del tag stesso
    - <myTag:faQualcosa2>
      - Hello World
    - </myTag:faQualcosa2>

JSTL

- in questa famiglia di tag dobbiamo invece programmare
  - cosa avviene all'apertura
  - cosa fare del corpo del tag
  - cosa fare alla chiusura del tag

JSTL

- in entrambi i casi occorre fare qualche considerazione
  - <myTag:
  - rappresenta il namespace del tag
    - vale a dire che si associa a questo prefisso l'invocazione di una determinata famiglia di tag
  - faQualcosa
  - rappresenta una determinata classe della famiglia del mio tag
    - con questo si sceglie quale classe caricare in memoria
  - attributo1
    - rappresenta un parametro necessario o meno all'esecuzione del tag

JSTL

- Q: ... e se mi trovo a voler utilizzare una classe già scaricata di cui non ho i sorgenti come faccio a sapere nameSpace e attributi?
- col tempo è diventata prerogativa di ogni container JSP quella di poter parametrizzare i nameSpaces delle classi dei tag
- Q: dove e come?
- il meccanismo scelto è stato quello di creare dei file che descrivano il tag "ai morsetti"
  - questi file vengono letti dal container in fase di startup e vengono visti a mo' di interfaccia (fino al riavvio del container) d'uso delle classi dei tag

JSTL

- vista la loro natura questi file sono stati chiamati descrittori della libreria di tag
  - Tag Library Descriptor
- sono puramente dei file XML
  - aderiscono ad un determinato schema
- siccome specificano dei parametri di esecuzione della web-application devono essere visti all'interno del file descrittore della configurazione
  - web.xml
- devo anche specificare in ogni singola pagina di importare quella determinata libreria di tag
  - con una direttiva di pagina
    - <%@taglib prefix="sql" uri="http://jakarta.apache.org/taglibs/dbtags" %>

JSTL tagLibDescriptor.tld

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd" version="2.0">

<tlib-version>1.0</tlib-version>
  <short-name>myTag</short-name>
  <uri>/WEB-INF/pathTo/myself</uri>

 <tag>
    <name>faQualcosa</name>
    <tag-class>myPackage.myClass.class</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>attributo1</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
      <name> attributo1 </name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
    </attribute>
  </tag>

<tag>
    <name>faQualcosa2</name>
    <tag-class>myPackage.myClass.class</tag-class>
    <body-content>jsp</body-content>
</tag>
```

è il prefisso con cui li chiameremo nella pagina

JSTL WEB.XML

- ...
- <taglib>
  - <taglib-uri>/WEB-INF/pathTo/myself</taglib-uri>
  - <taglib-location>/WEB-INF/tld/tagLibDescriptor.tld</taglib-location>
- </taglib>
- ...

JSLT

- import javax.servlet.jsp.tagext.*;
- import javax.servlet.jsp.JspWriter;
- import javax.servlet.jsp.JspException;
- public class miaClasseFaQualcosa extends SimpleTagSupport{
  - private String attributo1;
  - private String attributo2;
  - public void doTag() throws JspException{
    - JspWriter out = getJspContext().getOut();
      - out.println(”<b>”+attributo1 + ” ” + attributo2);
  - }
    - public void setAttributo1(String value){
      - attributo1 = value;
    - }
    - public void setAttributo2(String value){
      - attributo2 = value;
    - }
- }

JSLT

- un po’ più complesso è il caso dei tag in cui occorre analizzare il corpo
  - parsing o no?
  - cosa fare all’apertura ...
  - alla chiusura ...
  - ...e se dovessi iterare sul valore del corpo del tag ??
- tutte domande valide che devono avere risposta ...
  - un passo per volta

JSTL

- public class faQualcosa2 extends BodyTagSupport{
  - public faQualcosa2(){ super();}
  - public int doStartTag(){}
  - public int doAfterBody(){}
  - public int doEndTag(){}
  - // per gli eventuali attributi il meccanismo è lo stesso del simpleTag
- }

JSTL

- Q: perchè i metodi critici ritornano degli int ?
  - ... perchè non sempre il tag si apre e chiude e basta ...
    - ... alle volte c’è bisogno di aprirsi e chiudersi n volte ...
  - es: pensate all’esempio del carrello della spesa ...
    - contesto:
      - una serie di oggetti contenuti in sessione
      - necessità di visualizzarli a video in forma tabellare
    - primo passo recuperiamo gli oggetti
      - servlet
        - l’abbiamo già visto
      - jsp
        - ora lo vediamo

JSTL

- .... //HTML ...
- <% java.util.Vector v = pageContext.getSession().getAttribute(”oggettiCarrello”);
- pageContext.setAttribute(”oggetti”, v);
- %>
- <!-- supponiamo di voler fare una cosa simile a questa -->
- <c:forEach var=”k” begin=”0” end=”<%=oggetti.size%>” step=”1” >
  - <tr><td><%= oggetti.elementAt(k).toString()%></td></tr>
- </c:forEach>

JSTL

- il tag forEach deve iterare n volte per poter permetterci di stampare a video tutti gli oggetti che sono nel vector in sessione
  - per fare questo dobbiamo avere il modo di dire al tag che alla chiusura del tag ... questo venga riaperto
  - per permetterci questo grado di libertà i progettisti SUN hanno pensato di inserire una serie di costanti intere
    - ... e lo hanno fatto per ogni metodo !

JSTL

- public int doStartTag(){}
  - può ritornare due valori:
    - EVAL_BODY_BUFFERED
      - nel caso in cui si debba valutare il corpo del tag dopo l'apertura
    - SKIP_BODY
      - nel caso si debba passare direttamente alla chiusura del tag

JSTL

- public int doAfterBody(){}
  - può ritornare due valori:
    - EVAL_BODY_AGAIN
      - per far valutare nuovamente il corpo del tag
    - SKIP_BODY
      - per saltare alla chiusura

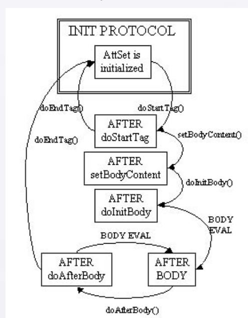
JSTL

- public int doEndTag(){}
  - può ritornare due valori:
    - EVAL_PAGE
      - per continuare nella interpretazione della pagina jsp
    - SKIP_PAGE
      - per interrompere la generazione della pagina jsp

JSTL

- nel caso del nostro for each ...
  - il do start tag
    - acquisirà il vector ... si preparerà all'elaborazione e ritornerà EVAL_BODY_BUFFERED
    - in caso di errore potremmo far ritornare SKIP_BODY
  - il doAfterBody
    - ciclerà sul Vector con il passo impostato e ritornerò EVAL_BODY_AGAIN fino a che ci sono oggetti nel Vector
    - al termine dello scorrimento ritornerà SKIP_BODY
  - il do end tag
    - ritornerà EVAL_PAGE a meno di operazioni critiche che ci potrebbero imporre di far ritornare SKIP_PAGE

JSTL



JSTL

- ora abbiamo quasi tutto
  - Q: ma i tag possono interagire tra di loro?
    - certo!
  - il che aumenta l'entropia del codice ...
    - pensiamo al caso della interrogazione ad un DataBase
    - i passi fondamentali sono:
      - connettersi
      - interrogare
      - scorrere il risultato

JSTL

- potremmo pensare alla creazione di una famiglia di tag che si occupi di gestire le connessioni con i db
  - <sql:createConnection>
    - <sql:query>select * from tabella1 </sql_query>
    - <sql:resultSet getColumn="1">
  - </sql:createConnection>

JSTL

- e magari di utilizzare il nostro forEach per scorrere il risultato
  - <sql:createConnection>
    - <sql:query>select * from tabella1 </sql_query>
    - <myTag:forEach var="x" begin="1" end="10" step="1">
      - <sql:resultSet getColumn="x">
    - </myTag:forEach>
  - </sql:createConnection>

JSTL

- il tag createConnection deve
  - verificare che il db sia raggiungibile
  - autenticarsi
  - chiedere una connessione attiva
  - rendere la connessione disponibile alla pagina fino alla chiusura del tag
    - per permettere ad altri tag di utilizzarla

JSTL

- createConnection
  - estenderà SimpleTagSupport
  - in doTag() farà tutto ciò che abbiamo detto prima

JSTL

- query
  - estenderà BodyTagSupport
  - acquisirà la stringa corrispondente all'interrogazione SQL che vogliamo effettuare al DB
    - eventualmente potrebbe effettuare test semantici sulla stringa stessa

JSTL

- resultSet
  - estenderà SimpleTagSupport
  - dovrà interrogare il db
  - farsi dare il risultato
  - stamparlo a video
- Problema ... resultSet deve vedere il contenuto di
  - query
    - per sapere quale interrogazione deve fare al db
  - createConnection
    - per sapere su che connessione effettuare l'interrogazione

JSTL

- per fare ciò le interfacce delle classi dei tag si sono arricchite con i metodi per la ricerca dei tag padri
  - getParent()
  - ci permette di avere a disposizione tutti i campi della classe "padre" del tag
    - della classe il cui tag racchiude il corrente
  - con questo meccanismo possiamo propagare le informazioni/oggetti necessari al corretto funzionamento del sistema

JSTL

- abbiamo visto che esiste una famiglia di tag che inizia con <jsp: inclusi già nel linguaggio
  - ... col tempo ne sono nati altri
- o meglio ... visto che alla progettazione del sistema c'era il requisito di potersi creare Tag da soli, diciamo che alcuni di quelli di utilità più comune sono stati sviluppati sotto forma di progetti open source
- col tempo il progetto per lo sviluppo di queste famiglie di tag è diventato Java Standard TAG Library
  - Repository dei vari progetti open source è la stessa fondazione che ha portato avanti lo sviluppo di Tomcat
    - http://jakarta.apache.org/taglibs/index.html

JSTL

- all'interno del progetto Jakarta troviamo quindi una serie di famiglie di tag per le utilità più comuni
  - DB
  - Sessione
  - String
  - Mail
  - DateTime
  - Random
- ne sono alcuni esempi

JSTL

- la famiglia dei tag l'interazione con i db si articola sul prefisso <sql:
  - i tag che possiamo utilizzare sono relativi a:
    - connessioni
    - statement
    - result set

JSTL

- **Connection Tags**
  connection
  Get a java.sql.Connection object from the DriverManager or a DataSource.
  url
  Sets the database URL of the enclosing connection tag.
  jndiName
  Sets the JNDI named JDBC DataSource of the enclosing connection tag.
  driver
  Sets the driver class name for the connection tag.
  userId
  Sets the user id for the connection tag.
  password
  Sets the password for the connection tag.
  closeConnection
  Close the specified connection. The "conn" attribute is the name of a connection object in the page context.

JSTL

Statement Only Tags
statement
Create and execute a database query.
escapeSql
Replaces each single quote in the tag body with a pair of single quotes.

Statement/PreparedStatement Tags
query
Set a query for a statement or preparedStatement tag
execute
Executes an insert, update or delete for a statement or preparedStatement tag

PreparedStatement Only Tags
preparedStatement
Create and execute a tokenized database query
setColumn
Set a field in a preparedStatement. Set the value as a String inside the tag body.

ResultSet Tags

resultSet

JSP tag resulset, executes the query and loops through the results for the enclosing statement or preparedstatement tag. The body of this tag is executed once per row in the resultset. The optional "loop" attribute, which default to true, specifies whether to execute the tag body once per row "true", or to simply assign the ResultSet to the page attribute specified by "id".

wasNull

Executes its body if the last getColumn tag received a null value from the database. You must be inside a resultset tag and there must be a previous getColumn tag, or an error will be generated.

wasNotNull

Executes its body if the last getColumn tag did not encounter a null value from the database.

getColumn

Gets the value, as a String, of a coulmn in the enclosing resultset. The column number is set via the "position" attribute. You can optionally set the value, as a String, to a serlvet attribute instead of the tag body with the "to" attribute. The scope of the servlet attribute is specified by the "scope" XML attribute (default = page).

getNumber

Similar to getColumn, but provides more precise control over number formatting. The "format" attribute can be either a pattern as accepted by the DecimalFormat constructor or a style: "CURRENCY", "PERCENT" or "NUMBER". The "locale" attribute can have one to three components as accepted by the Locale constructor: language, country and variant. They are separated by "_". If neither the format nor locale attribute is set, output should be identical to getColumn.

getTime

Similar to getColumn, but provides more precise control over java.sql.Time formatting. The "format" attribute can be either a pattern as accepted by SimpleDateFormat or a style: "FULL", "LONG", "MEDIUM" or "SHORT". The "locale" attribute can have one to three components as accepted by the Locale constructor: language, country and variant. They are separated by "_". If neither the format nor locale attribute is set, output should be identical to getColumn.

getTimestamp

Similar to getColumn, but provides more precise control over java.sql.Timestamp formatting. The "format" attribute can be either a pattern as accepted by SimpleDateFormat or a style: "FULL", "LONG", "MEDIUM" or "SHORT". The "locale" attribute can have one to three components as accepted by the Locale constructor: language, country and variant. They are separated by "_". If neither the format nor locale attribute is set, output should be identical to getColumn.

getDate

Similar to getColumn, but provides more precise control over java.sql.Date formatting. The "format" attribute can be either a pattern as accepted by SimpleDateFormat or a style: "FULL", "LONG", "MEDIUM" or "SHORT". It is required. The "locale" attribute can have one to three components as accepted by the Locale constructor: language, country and variant. They are separated by "_".

wasEmpty

Executes its body if the last ResultSet tag received 0 rows from the database. You must be after a ResultSet tag, or an error will be generated.

wasNotEmpty

Executes its body if the last ResultSet tag received more than 0 rows from the database. You must be after a ResultSet tag, or an error will be generated.

rowCount

Prints out the number of rows retrieved from the database. It can be used inside a ResultSet tag to provide a running count of rows retreived, or after the ResultSet tag to display the total number. Using the tag before the ResultSet will produce an error.

JSTL

```
• <sql:connection id="conn1">
    <%-- required --%>
    <sql:url>jdbc:mysql://localhost/test</sql:url>
    <%-- optional --%>
    <sql:driver>org.gjt.mm.mysql.Driver</sql:driver>
    <%-- optional --%>
    <sql:userId>root</sql:userId>
    <%-- optional --%>
    <sql:password>notVerySecure</sql:password>
  </sql:connection>

• <%-- showing the contents of the table --%>
  <table>
  <tr><th>id</th><th>name</th><th>description</th></tr>
  <sql:preparedStatement id="stmt6" conn="conn1">
    <sql:query>
      select * from tabella1
    </sql:query>
    <sql:resultSet id="rset4">
      <tr>
        <td><sql:getColumn position="1"/></td>
        <td><sql:getColumn position="2"/></td>
        <td><sql:getColumn position="3" to="description"/></td>
      </tr>
    </sql:resultSet>
    <tr>
      <td colspan="3">
      <%-- show different text, depending on whether or not
      any rows were retrieved --%>
      <sql:wasEmpty>No rows retrieved.</sql:wasEmpty>
      <sql:wasNotEmpty><sql:rowCount/> rows retrieved.</sql:wasNotEmpty>
      </td>
    </tr>
  </sql:preparedStatement>

• <sql:closeConnection conn="conn1"/>
```

JSTL

- Q: ... e se cambio il db devo riscrivere il codice delle pagine ?
  - assolutamente no!
  - al massimo cambio il parametro che mi specifica il driver JDBC
    - se sono stato furbo l'ho messo come parametro dentro al web.xml
  - in quanto i tag sono stati pensati proprio per aver qualcosa che funzionasse con una logica comune a tutti i db

JSTL

- per la sessione troviamo tag che permettono di leggere, scrivere e modificare gli oggetti in sessione
- **Session Tags**
  session
  Access general information about session.
  isNew
  Determine if a session is new.
  invalidate
  Invalidate a user session and remove it.
  maxInactiveInterval
  Set the maximum inactive interval before a session times out.

  Session Attribute Tags
  attribute
  Get the value of a single session attribute.
  attributes
  Loop through all session attributes.
  equalsAttribute
  See if a session attribute equals some value.
  existsAttribute
  See if a session attribute exists.
  removeAttribute
  Removes an attribute from a session.
  setAttribute
  Sets the value of a session attribute.

JSTL

```
• <sess:setAttribute name="nome">Valore</sess:setAttribute>

• ...

• <sess:existsAttribute name="test1">
   The session attribute with name test1 exists.
  </sess:existsAttribute>
  <sess:existsAttribute name="test1" value="false">
   The session attribute with name test1 does not exist.
  </sess:existsAttribute>

• ...
```

JSTL

- alla stessa maniera possiamo avere tutte le informazioni che ci servono sugli altri tag su:
  - http://jakarta.apache.org/taglibs/index.html
  - di cui alcuni:
    - alcuni stabili
    - altri in development
    - altri deprecated

JSTL

- c'è poi una famiglia di tag che appare in ogni distribuzione di un container
  - i tag di controllo o di "core"
  - <c:
    - forEach
    - choose
    - if
    - ...e altri comodi per controllare le iterazioni e via dicendo

JSTL

- ok, ora in jsp riesco a fare qualcosa ...
  - Q:ma per recuperare un parametro deve sempre passare per java?
    - <%=request.getParameter("nomeParametro")%>
    - <%=pageContext.getSession().getAttribute("nomeAttributo")%>
  - Q:ma per effettuare una valutazione logica deve sempre passare per java?
    - <c:if test="<%=StringA.equals(StringB)%>"%>...
  - Q: non riesco a fare in modo di non usare mai Java e restare sempre a livello di tag?

JSTL

- per rispondere a questa domanda la SUN a rilasciato a partire dalla release 2.3 delle API JSP l'Espression language
  - l'obiettivo era quello di
  - completare il compito iniziato con JSTL e le taglib
  - di creare mezzi per non far trasparire java a chi programma in jsp
- l'espression language è un linguaggio vero e proprio che permette la rapida valutazione di espressioni da JSP

JSTL

- l'Espression Language (EL) si invoca con un marker
  - ${}
  - tutto ciò che sta dentro alle parentesi graffe viene valutato come EL
  - ovviamente deve essere abilitato sulla pagina su cui lo utilizziamo
    - per abilitarlo in realtà basta abilitare i tag di controllo (core)

JSTL

- l'espressione più semplice del linguaggio è quella di stampare il valore di una variabile
  - si ottiene con
    - ${nomeVariabile}
  - con questo statement riusciamo a recuperare sia le variabili in page che le variabili in session
  - per recuperare le variabili in request occorre aggiungere "param" davanti al nome
    - ${param.nomeVariabile}

JSTL

- abbiamo detto che è un linguaggio per realizzare espressioni
  - ergo deve avere degli operatori di confronto
    - eq , ne , gt , lt , ge ,le
  - e permettere le operazioni elementari
    - + - * /

JSTL

- così il controllo che prima facevamo così:
  - <c:if test="<%=StringA.equals(StringB)%>"%>...
- diventa
  - <c:if test="${StringA eq StringB}"%>
- il recupero di parametri diventa
  - ${param.nomeParametro}
- le operazioni tra due variabili diventano
  - ${a + b}
  - ${(a+c) *B}