Sistemi informativi su Web

MVC

Giovanni Grasso

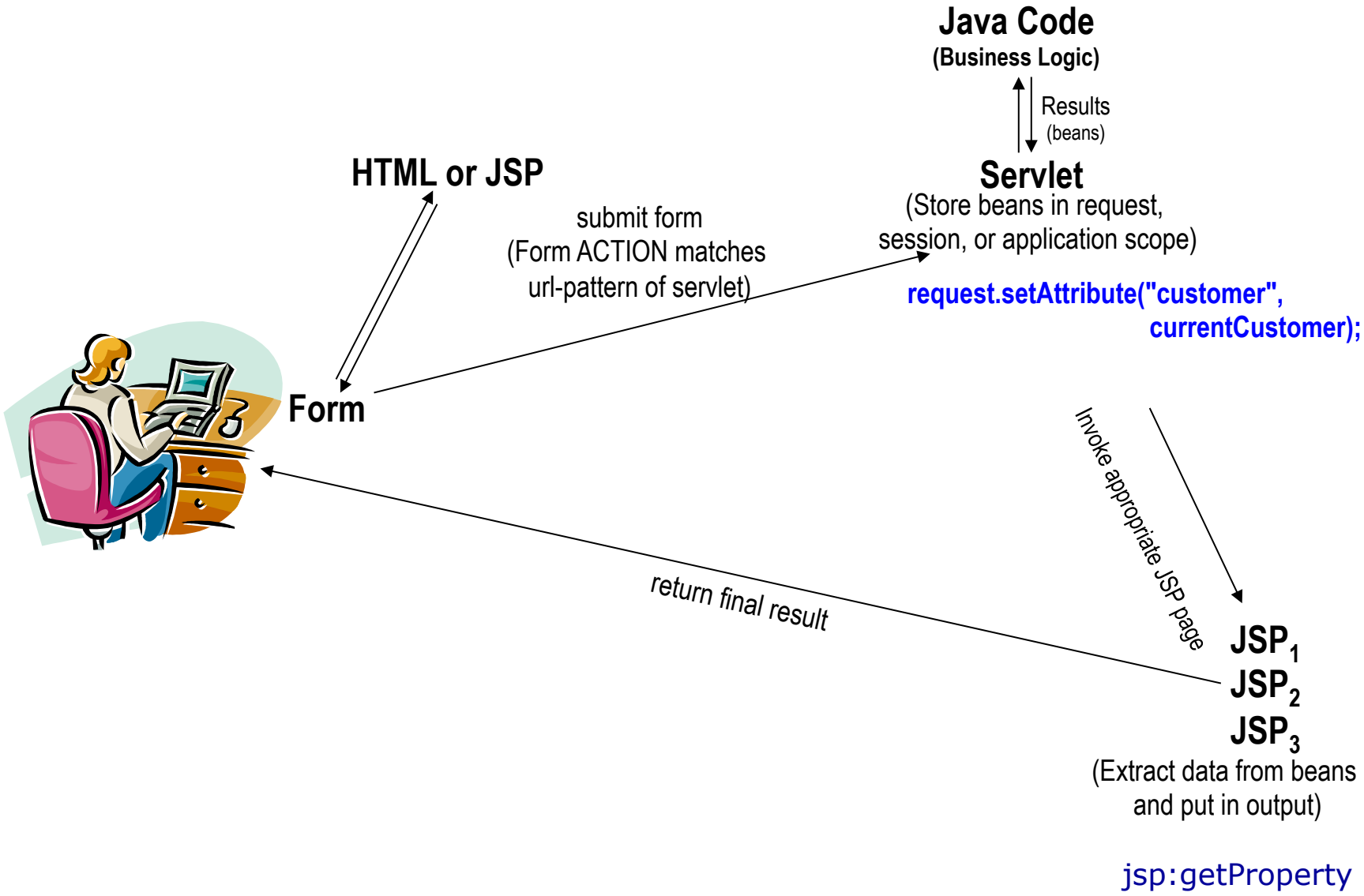
Combining Servlet and JSP

- **Servlet only**. Works well when:
  - Output is a binary type. E.g.: an image
  - There is *no* output. E.g.: you are doing forwarding or redirection as in Search Engine example
  - Format/layout of page is highly variable
- **JSP only**. Works well when:
  - Output is mostly character data. E.g.: HTML
  - Format/layout mostly fixed
- **Combination (MVC architecture/Model 2)**. Needed when:
  - A single request will result in multiple substantially different-looking results
  - You have a large development team with different team members doing the Web development and the business logic
  - You have a relatively fixed layout, but perform complicated data processing.

Model-View-Controller

- An approach where you break the response into three pieces
  - **The controller:** the part that handles the request, decides what logic to invoke, and decides what JSP page should apply
  - **The model:** the classes that represent the data being displayed
  - **The view:** the JSP pages that represent the output that the client sees
- Examples
  - **MVC** using `RequestDispatcher` – works very well for most simple and moderately complex applications
  - Struts - Spring MVC
  - JavaServer Faces JSF

MVC Flow of Control



Implementing MVC with RequestDispatcher

1. **Define beans** to represent the data
2. Use a **servlet to handle requests**
   - Servlet reads request parameters, checks for missing and malformed data, calls business logic, etc.
3. **Populate the beans**
   - The servlet invokes business logic (application-specific code) or data-access code to obtain the results. Results are placed in the beans that were defined in step 1
4. **Store the bean** in the request, session, or servlet context
   - The servlet calls `setAttribute` on the `request`, `session`, or `ServletContext` objects to store a reference to the beans that represent the results of the request.

Implementing MVC with RequestDispatcher

5. **Forward** the request to a **JSP page**
   - The servlet determines which JSP page is appropriate to the situation and uses the forward method of `RequestDispatcher` (from the `ServletRequest`) to transfer control to that page
6. **Extract the data from the beans**
   - The JSP page accesses beans with `jsp:useBean` and a scope matching the location of step 4. The page then uses `jsp:getProperty` to output the bean properties. Or use Expression Language
   - The JSP page **does not create or modify the bean**; it merely extracts and displays data that the servlet created.

Request Forwarding Example

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
  throws ServletException, IOException {
  ... // Do business logic and get data
  String operation = request.getParameter("operation");
  if (operation == null) {
    operation = "unknown";
  }
  String address;
  if (operation.equals("order")) {
    address = "/Order.jsp";
  } else if (operation.equals("cancel")) {
    address = "/Cancel.jsp";
} else {
    address = "/UnknownOperation.jsp";
  }
  RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
  dispatcher.forward(request, response);
}
```

in SPring:
return address;

Request-Based Data Sharing

- Servlet

```
ValueObject value = new ValueObject(...);
  MODEL  setAttribute("key", value);
retun "somePage"
```

- JSP (1.2 OLD)

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

- JSP (2.0 Preferred)

```
${key.someProperty}
```

Applying MVC: Bank Account Balances

- **Bean**
  - BankCustomer
- **Servlet** that populates bean and forwards to appropriate JSP page
  - Reads customer ID, calls data-access code to populate BankCustomer
  - Uses current balance to decide on appropriate result page
- **JSP** pages to display results
  - Negative balance: warning page
  - Regular balance: standard page
  - High balance: page with advertisements added
  - Unknown customer ID: error page

Bank Account Balances: Servlet Code

```
public class ShowBalance extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    BankCustomer customer =
      BankCustomer.getCustomer(request.getParameter("id"));

    String address;
    if (customer == null) {
      address =
        "/bank-account/UnknownCustomer.jsp";     }
    else if (customer.getBalance() < 0) {
      address =
        "/bank-account/NegativeBalance.jsp";
      request.setAttribute("badCustomer", customer);
    }
    …
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
```

The bean contains a small data base of customers in a static variable

BankCustomer.java

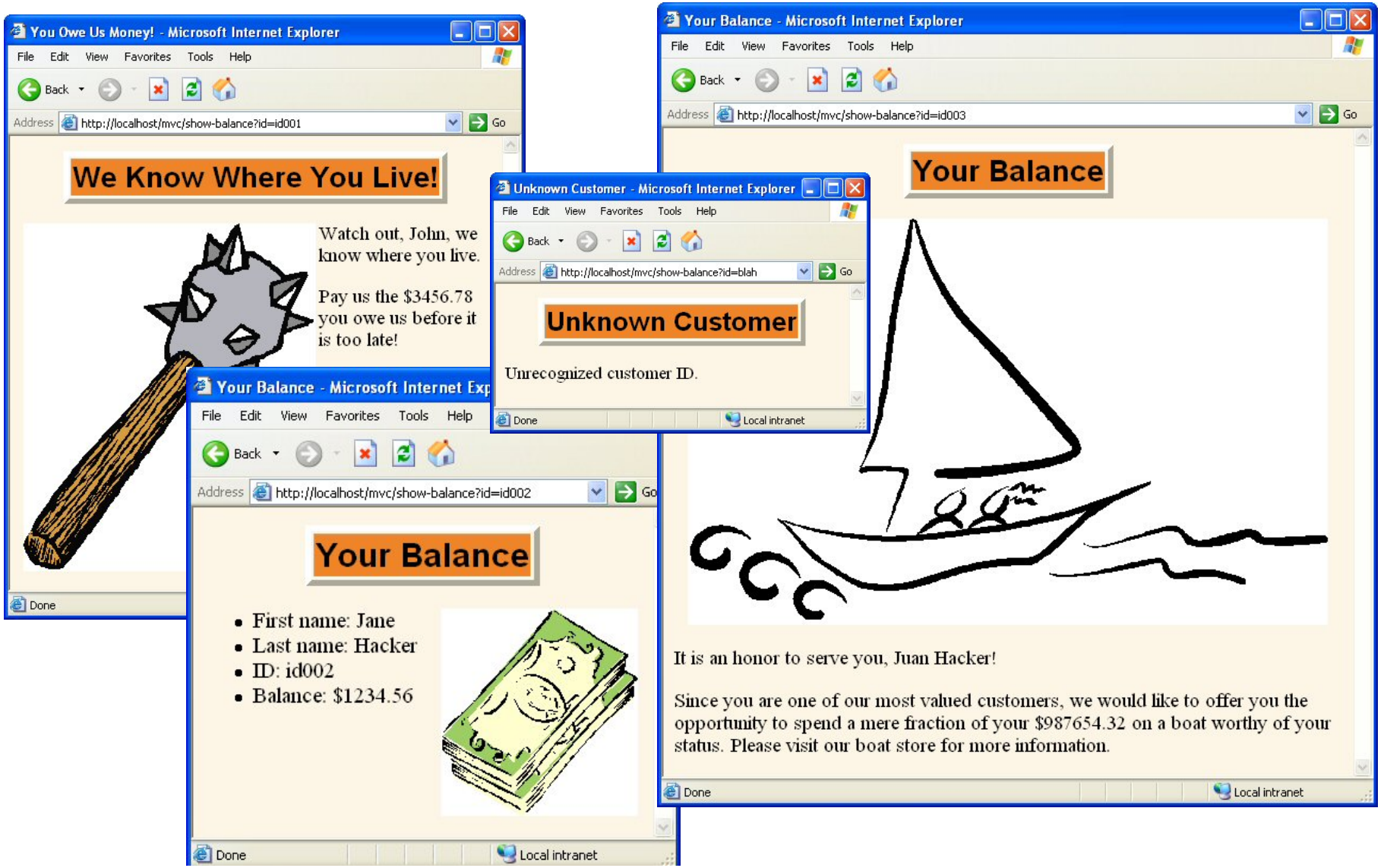
ShowBalance.java

call1 *call2* *call3*

JSP 2.0 Code (Negative Balance)

```
…
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      We Know Where You Live!</TABLE>
<P>
<IMG SRC="/bank-support/Club.gif" ALIGN="LEFT">
Watch out, ${badCustomer.firstName},
we know where you live.
<P>
Pay us the $${badCustomer.balanceNoSign}
you owe us before it is too late!
</BODY></HTML>
```

Bank Account Balances: Results



Including Pages Instead of Forwarding

- With the `forward` method of RequestDispatcher:
  - **New page generates all of the output**
  - Original page *cannot* generate any output
- With the `include` method of RequestDispatcher:
  - Output can be generated by **multiple pages**
  - Original page *can* generate output **before** and **after** the included page
  - Original servlet does not see the output of the included page
  - Applications
    - Portal-like applications (see first example)
    - Including alternative content types for output (see second example)

Using RequestDispatcher.include: portals

```
response.setContentType("text/html");
String firstTable, secondTable, thirdTable;
if (someCondition) {
  firstTable = "/WEB-INF/Sports-Scores.jsp";
  secondTable = "/WEB-INF/Stock-Prices.jsp";
  thirdTable = "/WEB-INF/Weather.jsp";
} else if (...) { ... }
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/Header.jsp");
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher("/WEB-INF/Footer.jsp");
dispatcher.include(request, response);
```

Sistemi informativi su Web

JSP

http://docs.oracle.com/javaee/5/tutorial/doc/bnadp.html

Giovanni Grasso

Java Server Pages

- Le JSP sono uno dei due componenti di base della tecnologia J2EE, relativamente alla parte Web:
  - template per la generazione di contenuto dinamico
  - estendono HTML con codice Java custom
- Quando viene effettuata una richiesta ad una JSP:
  - parte HTML viene direttamente trascritta sullo stream di output
  - codice Java viene eseguito sul server per la generazione del contenuto HTML dinamico
  - pagina HTML così formata (parte statica + parte generata dinamicamente) viene restituita al client
- Sono assimilabili a linguaggio di script: in realtà vengono trasformate in servlet dal container

Come funzionano le JSP

- Ogni volta che arriva una request, server compone dinamicamente il contenuto della pagina
- Ogni volta che incontra un tag <%...%>
  - valuta l'espressione Java contenuta al suo interno
  - inserisce al suo posto il risultato dell'espressione
- Questo meccanismo permette di generare pagine dinamicamente

Esempio: Hello world

JSP, denominata helloWorld.jsp, che realizza il classico esempio “Hello World!” in modo parametrico:

```
<html>
  <body>
  <% String visitor=request.getParameter("name");
     if (visitor == null) visitor = "World"; %>
     Hello, <%= visitor %>!
  </body>
</html>
```

http://myHost/myWebApp/helloWord.jsp

```
<html>
  <body>
     Hello, World!
  </body>
</html>
```

http://myHost/myWebApp/helloWord.jsp?name=Mario

```
<html>
  <body>
     Hello, Mario!
  </body>
</html>
```

Tag

- La parti variabili della pagina sono contenute all'interno di tag speciali
- Sono possibili due tipi di sintassi per questi tag:
  - Scripting-oriented tag
  - XML-Oriented tag
- Le **scripting-oriented tag** sono definite da delimitatori entro cui è presente lo scripting (self-contained)
- Sono di quattro tipi:
  - <%! %> Dichiarazione
  - <%= %> Espressione
  - <% %> Scriptlet
  - <%@ %> Direttiva

Dichiarazioni

- Si usano i delimitatori <%! e %> per dichiarare variabili e metodi
- Variabili e metodi dichiarati possono poi essere referenziati in qualsiasi punto del codice JSP
- I metodi diventano metodi della servlet quando la pagina viene tradotta

```
<%! String name = "Paolo Rossi";
    double[] prices = {1.5, 76.8, 21.5};

    double getTotal() {
       double total = 0.0;
       for (int i=0; i<prices.length; i++)
         total += prices[i];
       return total;
    }
%>
```

Espressioni

- Si usano i delimitatori `<%=` e `%>` per valutare espressioni Java
- Risultato dell'espressione viene convertito in stringa inserito nella pagina al posto del tag

Continuando l'esempio della pagina precedente:

JSP

```
<p>Sig. <%=name%>,</p>
<p>l'ammontare del suo acquisto è: <%=getTotal()%> euro.</p>
<p>La data di oggi è: <%=new Date()%></p>
```

Pagina HTML risultante

```
<p>Sig. Paolo Rossi,</p>
<p>l'ammontare del suo acquisto è: 99.8 euro.</p>
<p>La data di oggi è: Tue Feb 20 11:23:02 2010</p>
```

Scriptlet

- Si usano `<%` e `%>` per aggiungere un frammento di codice Java eseguibile alla JSP (**scriptlet**)
- Lo scriptlet consente tipicamente di inserire logiche di controllo di flusso nella produzione della pagina
- La combinazione di tutti gli scriptlet in una determinata JSP deve definire un blocco logico completo di codice Java

```
<% if (userIsLogged) { %>
  <h1>Benvenuto Sig. <%=name%></h1>
<% } else { %>
  <h1>Per accedere al sito devi fare il login</h1>
<% } %>
```

Direttive

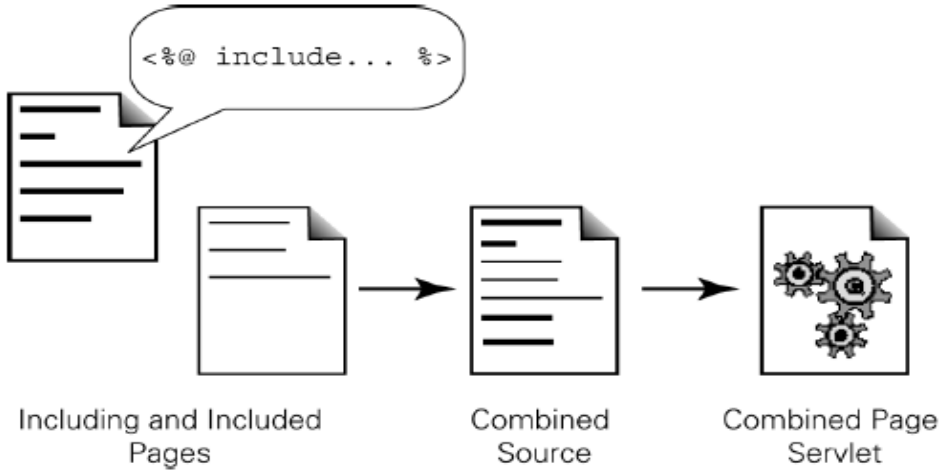
- Sono comandi JSP valutati a tempo di compilazione
- Le più importanti sono:
  - **page**: permette di importare package, dichiarare pagine d'errore, definire il modello di esecuzione JSP relativamente alla concorrenza, ecc.
  - **include**: include un altro documento
  - **taglib**: carica una libreria di custom tag
- Non producono nessun output visibile

```
<%@ page info="Esempio di direttive" %>
<%@ page language="java" import="java.net.*" %>
<%@ page import="java.util.List, java.util.ArrayList" %>
<%@ include file="myHeaderFile.html" %>
```

La direttiva include

- Sintassi: `<%@ include file = "localURL"%>`
- Serve ad includere il contenuto del file specificato
- È possibile nidificare un numero qualsiasi di inclusioni
- L'inclusione viene fatta a tempo di compilazione:
  eventuali modifiche al file incluso non determinano una ricompilazione della pagina che lo include

Esempio: `<%@ include file="/shared/copyright.html"%>`



JSP e modello a componenti

Scriptlet ed espressioni consentono uno sviluppo centrato sulla pagina

- Questo modello non consente una forte separazione tra logica applicativa e presentazione dei contenuti
- Applicazioni complesse necessitano di una architettura a più livelli

A tal fine JSP consentono sviluppo basato su un modello a componenti

Il modello a componenti:

- consente di avere una maggiore separazione fra logica dell'applicazione e contenuti
- Permette di costruire architetture molto più articolate

JavaBeans

- JavaBeans sono il modello di componenti di Java
- Un **JavaBean**, o semplicemente **bean**, non è altro che una classe Java dotata di alcune caratteristiche particolari:
  - Classe **public**
  - Ha un costruttore **public** di default (senza argomenti)
  - Espone proprietà, sotto forma di coppie di metodi di accesso (accessors) costruiti secondo le regole che abbiamo appena esposto (get… set…)
  - Espone eventi con metodi di registrazione che seguono regole precise

Esempio

Creiamo un bean che espone due proprietà in sola lettura (ore e minuti) e ci dà l'ora corrente

```
import java.util.*
public class CurrentTimeBean
{
  private int hours;
  private int minutes;
  public CurrentTimeBean()
  {
    Calendar now = Calendar.getInstance();
    this.hours = now.get(Calendar.HOUR_OF_DAY);
    this.minutes = now.get(Calendar.MINUTE);
  }
  public int getHours()
  { return hours; }
  public int getMinutes()
  { return minutes; }
}
```

Azioni

- Le azioni sono comandi JSP tipicamente per l'interazione con altre pagine JSP, servlet, o componenti JavaBean
- Sono previsti 6 tipi di azioni definite dai seguenti tag:
  - `useBean`: istanzia JavaBean e gli associa un identificativo
  - `getProperty`: ritorna property indicata come oggetto
  - `setProperty`: imposta valore della property indicata per nome
  - `include`: include nella JSP contenuto generato dinamicamente da un'altra pagina locale
  - `forward`: cede controllo ad un'altra JSP o servlet
  - `plugin`: genera contenuto per scaricare plug-in Java se necessario
- Sono espresse usando sintassi XML

```
<html>
  <body>
    <jsp:useBean id="myBean" class="my.HelloBean"/>
    <jsp:setProperty name="myBean" property="nameProp" param="value"/>
    Hello, <jsp:getProperty name="myBean" property="nameProp"/>!
  </body>
</html>
```

Esempio di uso di bean

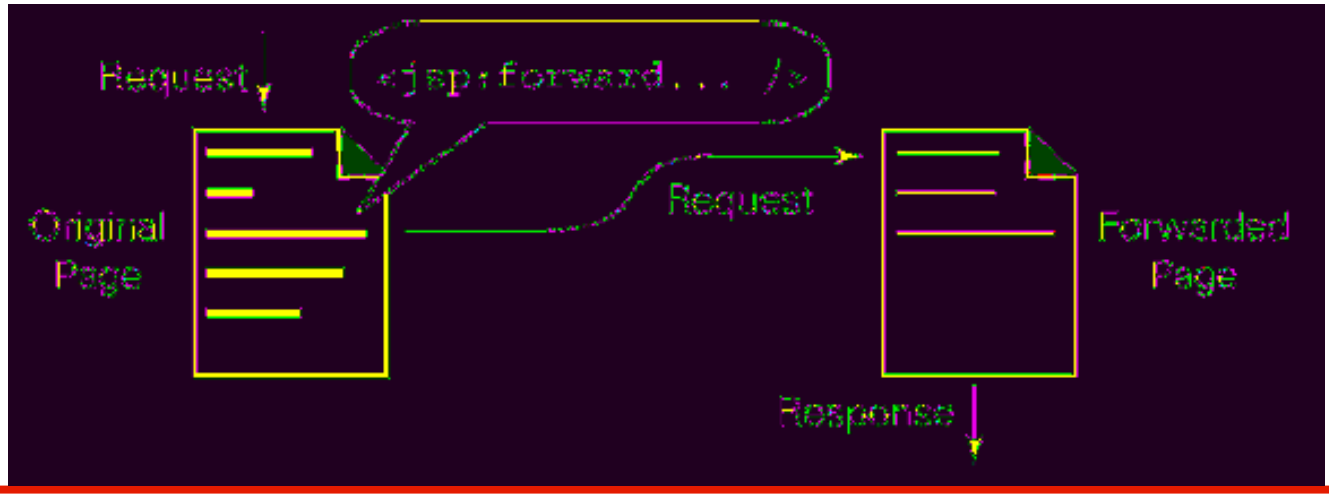
```
<jsp:useBean id="user" class="RegisteredUser" scope="session"/>

<jsp:useBean id="news" class="NewsReports" scope="request">
 <jsp:setProperty name="news" property="category" value="fin."/>
 <jsp:setProprety name="news" property="maxItems" value="5"/>
</jsp:useBean>

<html>
  <body>
    <p>Bentornato
    <jsp:getProperty name="user" property="fullName"/>,
    la tua ultima visita è stata il
    <jsp:getProperty name="user" property="lastVisitDate"/>.
    </p>
    <p>
    Ci sono <jsp:getProperty name="news" property="newItems"/>
    nuove notizie da leggere.</p>
  </body>
</html>
```

Azioni: forward

- Sintassi: `<jsp:forward page="localURL" />`
- Consente trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale
  - L'attributo page definisce l'URL della pagina a cui trasferire il controllo
  - La request viene completamente trasferita in modo trasparente per il client



Azioni: include

Sintassi: `<jsp:include page="localURL" flush="true" />`

Consente di includere il contenuto generato dinamicamente da un'altra pagina locale all'interno dell'output della pagina corrente

- Trasferisce temporaneamente controllo ad un'altra pagina
- L'attributo page definisce l'URL della pagina da includere
- L'attributo flush stabilisce se sul buffer della pagina corrente debba essere eseguito flush prima di effettuare l'inclusione
- Gli oggetti session e request per pagina da includere sono gli stessi della pagina chiamante, ma viene istanziato un nuovo contesto di pagina

Expression Language Introduzione

- Per snellire ulteriormente la struttura delle pagine jsp, dalla versione 2.0 di JSP è stato introdotto (in realtà potenziato e reso più efficace) un formalismo per gestire la stampa di *espressioni*
- Se il progetto è organizzato bene, nelle pagine Jsp abbiamo quasi esclusivamente scriplet del tipo `<% out.print(…)%>` perchè la Jsp dovrebbe avere l'unica responsabilità di produrre la risposta

Expression Language

- Formalismo per snellire sintassi e semantica
- Sintassi: usa il marcatore di espressioni

  `${`*`espressione`***`}`**

  All'interno delle parentesi graffe ci può essere una **espressione** Java
- Semantica: il risultato dell'espressione è passato come argomento a `out.print()`
  Se il risultato dell'espressione è `null`, viene passata la stringa vuota

Expression Language

- Una caratteristica importante dell'EL è quella di consentire un accesso molto agevole agli oggetti presenti nella richiesta o nella sessione e alle proprietà di questi oggetti
- Abbiamo visto che si accede agli attributi nella richiesta (o nella sessione) specificandone il nome
  - Al posto di

    ```
    <% request.getAttribute("codeErr")!=null:
        out.print(request.getAttribute("codeErr"))?out.print(""); %>
    ```

  - scriviamo
    - ${codeErr}
- NB: è furbo! se la stringa non esiste non stampa nulla

Expression Language

- Di più: in una espressione è possibile indicare direttamente il nome di un attributo (della richiesta o della sessione) e specificare l'accesso ad uno dei suoi campi (purchè questi abbiano un metodo getter pubblico)

Expression Language

- Esempio:

```
<html>
…
   <h1>Nome: ${product.name}</h1>
   …
```

- Semantica:
  - –Viene cercato (nell'ordine) nella pagina, nella richiesta, nella sessione, nel contesto dell'applicazione un attributo con chiave "product"
  - –Se l'attributo esiste, viene invocato il metodo getName()
  - –Il valore restituito viene passato a out.print()

Example: Accessing Scoped Variables

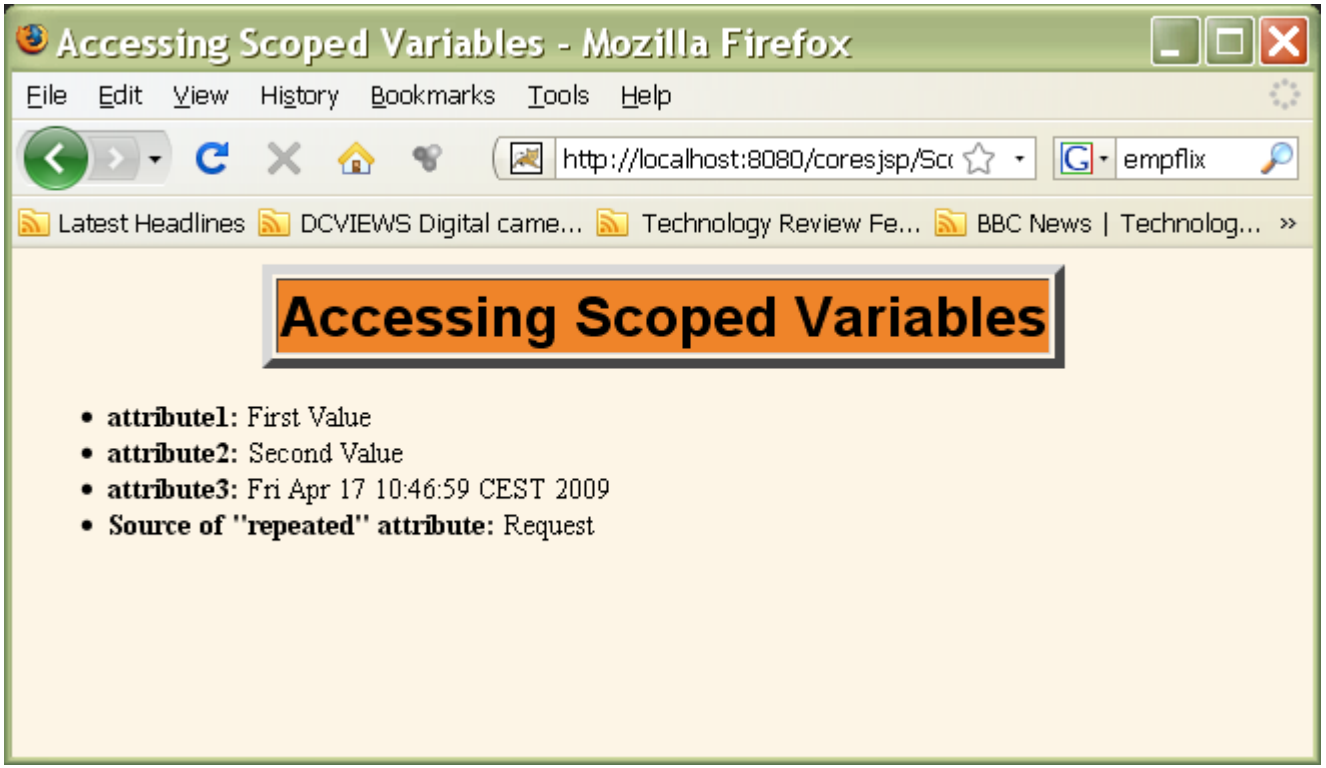
```
public class ScopedVars extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    request.setAttribute("attribute1", "First Value");
    HttpSession session = request.getSession();
    session.setAttribute("attribute2", "Second Value");
    ServletContext application = getServletContext();
    application.setAttribute("attribute3",
                             new java.util.Date());
    request.setAttribute("repeated", "Request");
    session.setAttribute("repeated", "Session");
    application.setAttribute("repeated",
   "ServletContext");
    RequestDispatcher dispatcher =
      request.getRequestDispatcher
                      ("scoped-vars.jsp");
    dispatcher.forward(request, response);
  }
}
```

Example: Accessing Scoped Variables (cont.)

The called JSP

```
<!DOCTYPE …>
…
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
  Accessing Scoped Variables
</TABLE>
<P>
<UL>
  <LI><B>attribute1:</B> ${attribute1}
  <LI><B>attribute2:</B> ${attribute2}
  <LI><B>attribute3:</B> ${attribute3}
  <LI><B>Source of "repeated" attribute:</B>
      ${repeated}
</UL>
</BODY></HTML>
```

Example: Accessing Scoped Variables (Result)



call

Expression Language: collezioni

- Se si accede ad un attributo che rappresenta una collezione di oggetti (es. Lista, array, map) è possibile accedere ai singoli elementi attraverso l'operatore []
- `expr-a[expr-b]`, valuta `expr-a` in `value-a` e `expr-b` in `value-b`. Quindi:
  - Se **`value-a`** o **`value-b`** è **`null`**, ritorna **`null`**.
  - Se **`value-a`** è un oggetto **`Map`**, ritorna **`value-a.get(value-b)`**.
  - Se !**`value-a.containsKey(value-b)`**, ritorna **`null`**.
  - Se **`value-a`** è un oggetto **`List`** o un array, forza **`value-b`** a **`int`** e ritorna **`value-a.get(value-b)`** o **`Array.get(value-a, value-b)`**. Se la forzatura non si applica, ritorna un errore. Se la chiamata a **`get()`** solleva una **`IndexOutOfBoundsException`**, ritorna `null`. Se la chiamata a **`get()`** solleva un'altra eccezione, ritorna un errore.

Equivalence of Dot and Array Notations

- Equivalent forms
  - `${name.property}`
  - `${name["property"]}`
- Reasons for using array notation
  - To access arrays, lists, and other collections
    - See upcoming slides
  - To calculate the property name at request time.
    - `{name1[name2]}` (no quotes around name2)
  - To use names that are illegal as Java variable names
    - `{foo["bar-baz"]}`
    - `{foo["bar.baz"]}`

Accessing Collections

- `${attributeName[entryName]}`
- Works for
  - Array. Equivalent to
    - `theArray[index]`
  - List. Equivalent to
    - `theList.get(index)`
  - Map. Equivalent to
    - `theMap.get(keyName)`
- Equivalent forms (for HashMap)
  - `${stateCapitals["maryland"]}`
  - `${stateCapitals.maryland}`
  - But the following is illegal since 2 is not a legal var name: `${listVar.2}`

Referencing Implicit Objects (I)

- `pageContext`: The PageContext object
  - E.g. `${pageContext.session.id}`
- Using the `pageContext object` you can obtain:
  - `Request: pageContext.request`
  - `Response: pageContext.response`
  - `Session: pageContext.session`
  - `Out: pageContext.out`
  - `ServletContext: pageContext.out`
- `param` and `paramValues`: Request params
  - E.g. `${param.custID}`

The value(s) of custID parameter

Referencing Implicit Objects (II)

- `header` and `headerValues`: Request headers
  - E.g. `${header.Accept}` or `${header["Accept"]}`
  - `${header["Accept-Encoding"]}`
- `cookie`: Cookie object (not cookie value)
  - E.g. `${cookie.userCookie.value}` or `${cookie["userCookie"].value}`
- `pageScope`, `requestScope`, `sessionScope`, `applicationScope`
  - Instead of searching scopes
  - `${requestScope.name}` look only in the `HttpServletRequest` object.

Java Standard Tag Library

- Insieme di tag speciali che consentono di scrivere le istruzioni di controllo Java (istruzioni condizionali e istruzioni di ciclo) mediante tag
- Vantaggio: nelle pagine ci sono solo tag
  - utile agli strumenti di produzione di pagine HTML

JSTL: tag di iterazione

- Cicli for

```
<c:forEach var="name" begin="x" end="y" step="z">
Bla, bla, bla ${name}
</c:forEach>
```

- Cicli for-each (iterare una struttura di dati)
  - Si applica con array, collezioni, mappe

```
<c:forEach var="name" items="array-or-
collection">
Bla, bla, bla ${name}
</c:forEach>
```

Tag di iterazione: esempio

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Esempio tag for</title>
  </head>
  <body>
  <ul>
    <c:forEach var="i" begin="1" end="10">
      <li>${i}</li>
    </c:forEach>
  </ul>
  </body>
</html>
```

Tag di iterazione su collezioni: esempio

- Ipotizzando che products sia una collezione di oggetti Product presente nella richiesta (o nella sessione)

- JSP senza JSTL

```
<ul>
<%
for(int i=0; i<products.size(); i++) {
    String nome = (products.get(i)).getName();
%>
    <li><% out.print(name); %></li>
<% } %>
</ul>
```

- JSP con l'uso di JSTL

```
<ul>
    <c:forEach var="product" items="${products}">
        <li>${product.name}</li>
    </c:forEach>
</ul>
```

JSTL: tag condizionali

- Una scelta: if

```
<c:if test="${product.price > 1000}">
<p>some content</p>
</c:if>
```

- Scelta multipla: choose

```
<c:choose>
    <c:when test="test1">Content1</c:when>
    <c:when test="test2">Content2</c:when>
    ...
    <c:when test="testN">ContentN</c:when>
    <c:otherwise>Default Content</c:otherwise>
</c:choose>
```

JSTL: URL encoding

- L'encoding degli URL può essere fatto con lo standard custom tag url.
- Sintassi: <c:url value="*url*" />
- Esempi:

```
<form action="<c:url value="product" />" method="get">
<a href="<c:url value="newProduct.jsp" />">Insert new</a>
```