# 3. Java servlets and JSP

## 3.1. Introduction to Java servlets

Java servlets are the Java technology proposal for the development of web applications. A servlet is a program that runs on a web server and builds a web page that is returned to the user. This page is built dynamically and may contain information from databases, be a response to data entered by the user, etc.

Java servlets offer a series of advantages over CGIs, the traditional method of web application development. These are more portable, more powerful, much more efficient, more user-friendly, more scalable, etc.

### 3.1.1. Efficiency

With the traditional CGI model, each request that reaches the server triggers the execution of a new process. If the lifetime of the CGI (the time it takes to be executed) is short, the instantiation time (the time taken to launch a process) can exceed that of execution. With the servlets model, the Virtual Java Machine, the environment from which they are run, starts up when the server starts and remains in operation throughout execution of the same. To deal with each request, instead of launching a new process, a *thread*, a light-weight Java process, is started which is much faster (it is actually instantaneous). Moreover, if we have $x$ simultaneous requests from a CGI, we will have `x` simultaneous processes in memory, thus consuming $x$ times the space of a CGI (which, if interpreted, is usually the case, consumes $x$ times the interpreter). With servlets, there is a certain number of *threads*, but there is only one copy of the Virtual Machine and its classes.

The servlets standard offers additional alternatives to CGIs for optimisation: `caches` of previous calculations, `pools` of database connections, etc.

### 3.1.2. Ease of use

The servlets standard provides a wonderful web application development infrastructure, with methods for the automatic analysis and decoding of HTML form data, access to HTTP request headers, handling of `cookies`, monitoring, control and management of sessions, among many other features.

### 3.1.3. Power

Java servlets can be used for many things that are difficult or impossible to do with traditional CGIs. Servlets can share data with each other, which means that they can share data, database connections etc. They can also maintain information request after request, facilitating tasks such as the monitoring of user sessions, etc.

### 3.1.4. Portability

Servlets are written in Java and use a well documented, standard API. As a result, servlets can be run on all platforms with Java servlet support without the need for recompilation, modification etc., regardless of the platform (Apache, iPlanet, IIS, etc.) and operating system, architecture *hardware*, etc.

## 3.2. Introduction to Java Server Pages or JSP

Java Server Pages (JSP) are a technology that allows us to mix static HTML with HTML generated dynamically using Java code embedded on pages. When we program web applications with CGIs, the bulk of the page generated by the CGIs is static and does not vary from execution to execution. The variable part of the page is truly dynamic and very small. Both CGIs and servlets require us to generate the page fully from our program code, which makes it more difficult for maintenance, graphic design, code comprehension, etc. With JSP, however, we can easily create pages.

**Example**

The parts of the page that do not vary from execution to execution are headers, menus, decorations, etc.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
    <HEAD>
      <TITLE>Store. Welcome.</TITLE>
    </HEAD>
 <BODY>
    <H1>Welcome to our store</H1>
    <SMALL>Welcome,
    < % out.println(Tools.readNameOfCookie(request)); %>
    </SMALL>
 </BODY>
</HTML>
```

As this example shows, a JSP page is nothing more than a HTML page where the special tags `< %` and `%>` allow us to include Java code.

This gives us a series of obvious advantages: firstly, we have practically the same advantages as we do when using Java servlets; in fact, JSP servers "translate" these to servlets before executing them. Secondly, JSPs offer

considerable simplicity and ease of development. It is much easier to write the example page than to write a servlet or CGI that prints each of the lines in the above page.

However, this simplicity is also one of the disadvantages of JSP. With complex applications containing numerous calculations, database accesses, etc., JSP syntax embedded inside HTML becomes tedious. Thus, JSPs and servlets do not usually compete, but rather they complement one another since the standards include capabilities for communication between them.

### 3.3. The servlets/JSP server

To use both servlets and JSP on our web server, we generally need to complement it with a servlets/JSP server (usually called a servlets container). There are many free software and proprietary containers. Sun, the inventors of Java, keep an updated list of servlet containers at:

`http://java.sun.com/products/servlet/industry.html`

- Apache Tomcat. Tomcat is the official implementation of reference for servlet and JSP specifications after versions 2.2 and 1.1, respectively. Tomcat is a very robust, highly efficient product and one of the most powerful servlet containers available. Its only weakness is that it is complicated to configure because there are many options to choose from. For more details, visit the official Tomcat website: `http://jakarta.apache.org/`.

- JavaServer Web Development Kit (JSWDK). JSWDK was the official reference implementation for specifications Servlet 2.1 and JSP 1.0. It was used as a small server to test servlets and JSP pages in development. However, it has now been abandoned in favour of Tomcat. Its website is: `http://java.sun.com/products/servlet/download.html`.

- Enhydra. Enhydra is an applications server whose many functionalities include a very powerful servlet/JSP container. Enhydra (`http://www.enhydra.org`) is a very powerful tool for developing web services and applications, including tools for the control of databases, templates, etc.

- Jetty. This is a very lightweight web server/servlet container written entirely in Java that supports the Servlet 2.3 and JSP 1.2 specifications. It is the ideal server for development because it is small and takes up little memory. Its web page is: `http://jetty.mortbay.org/jetty/index.html`.

### 3.4. A simple servlet

The following example shows the basic structure of a simple servlet that handles HTTP GET requests (servlets can also handle POST requests).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BasicServlet extends HttpServlet
{
 public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // We can use request to access the data of the
    // HTTP request.
    // We can use response to modify the HTTP response
    // that the servlet will generate.

    PrintWriter out = response.getWriter();
    // We can use out to return data to the user
    out.println("¡Hello!\n");
 }
}
```

To write a servlet, we must write a Java class that extends (by inheritance) the `HttpServlet` class (or the most generic `servlet class`) and overwrites the `service` method or one of the more specific request methods (`doGet`, `doPost` etc).

Service methods (`service`, `doPost`, `doGet`, etc.) have two arguments: a `HttpServletRequest` and a `HttpServletResponse`.

The `HttpServletRequest` gives us the methods for reading incoming information such as the data from a HTML form (FORM), HTTP request headers or the *cookies* of the request, etc. In contrast, `HttpServletResponse` has methods for specifying the HTTP response codes (200, 404, etc.), response headers (`Content-Type`, `Set-Cookie` etc). Most importantly, they allow us to obtain a `PrintWriter` (a Java class representing an output "file") used to generate the output data that will be returned to the client. For simple servlets, the bulk of the code is used to work with this `PrintWriter` in `println` statements that generate the desired page.

### 3.5. Compiling and executing servlets

The servlet compilation process is very similar regardless of the web server or servlet container used. If using Sun's Java development programming, the official JDK, we need to make sure that our `CLASSPATH`, the list of libraries and directories where the classes we use in our programs are searched, contains the Java servlets API libraries. The name of this library varies from version to version of the Java API but it is usually: `servlet-version.jar`. Once the servlets library is in our `CLASSPATH`, the servlet compilation process is as follows:

```
javac BasicServlet.java
```

We must locate the resulting `class` file in the directory that our servlet container requires to execute the servlet. To then test it, we need to direct the browser to the URL of our servlet, formed, on the one hand, by the directory where our servlet container displays the servlets (for example, `/servlets`) and, on the other, by the name of the servlet.

For example, in JWS, Sun's test server, `servlets` are located in a servlets subdirectory of the JWS installation directory and the URL is formed thus:

`http://server/servlet/BasicServlet`

In Tomcat, servlets are located in a directory indicating the web application under development, in the `WEB-INF` subdirectory, inside the subdirectory `classes`. Then, if the web application were called `test` for example, the resulting URL would be:

`http://server/test/servlets/BasicServlet`

### 3.6. Generating content from servlets

As we have seen, the API gives us a class called `PrintWriter` to which we can send all of our results. However, this is not enough to make our servlet return HTML to the client.

The first step for building a servlet that returns HTML to the client is to tell the servlet container that the return of our servlet is HTML. Remember that HTTP includes the transfer of multiple data types by sending the MIME type marker tag: `Content-Type`. To do this, we have a method for indicating the type returned, `setContentType`. So, before any interaction with the response, we need to mark the content type.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class Helloweb extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
 {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println(
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n" +
      "<HTML>\n" +
      "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
      "<BODY>\n" +
      "<H1>Hello web</H1>\n" +
      "</BODY></HTML>");
 }
}
```

As we can see, generating the result in HTML is a very tedious task, especially if we consider that part of this HTML does not change from servlet to servlet or execution to execution. The solution to this type of problem is to use JSP instead of servlets. However, if you really must use servlets, there are a number of time-saving tricks. The main solution is to declare methods that really return these common HTML parts: the `DOCTYPE` line, t e header and even a common header and footer for the company's whole website.

To do this, we need to build a class containing a series of utilities that we can use in our web application project.

```
public class Utilities
{
 public static final String DOCTYPE =
   "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD"+
   " HTML 4.0 Transitional//EN\">";

 public static String titleHeader(Title string) {
   return(DOCTYPE + "\n" +
     "<HTML>\n" +
     "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
 }
 // Here, we will add some utilities
}
```

### 3.7. Handling form data

Obtaining data sent by a user from a form is one of the most complex and monotonous tasks of CGI programming. Since we have two methods for passing values, `GET` and `POST`, which behave differently, we need to develop two methods to read these values. We must also analyse, *parse* and decode the strings containing coded values and variables.

One of the advantages of using servlets is that the servlets API solves all of these problems. This task is automatic and the values are made available to the servlet through the `getParameter` method of the class called `HttpServletRequest`. This parameter passing system is independent of the method used by the form to pass parameters to the servlet (GET or POST). There are also other methods to help us collect the parameters sent by the form. Firstly, we have a version of `getParameter` called `getParameters` that we need to use if the parameter we are looking for can have more than one value. We also have `getParameterNames`, which returns the name of the parameters passed.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class BasicServlet extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
      throws ServletException, IOException
 {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String tit= "Reading 2 Parameters";
    out.println(Utilities.titleHeader(tit) +
       "<BODY>\n" +
       "<H1 ALIGN=CENTER>" + tit + "</H1>\n" +
       "<UL>\n" +
       " <LI>param1: "
       + request.getParameter("param1") + "\n" +
       " <LI>param2: "
       + request.getParameter("param2") + "\n" +
       "</UL>\n" +
       "</BODY></HTML>");
 }

 public void doPost( HttpServletRequest request,
                     HttpServletResponse response)
       throws ServletException, IOException
```

```
 {
   doGet(request, response);
 }
}
```

This example of a servlet reads two parameters called `param1`, `param2` and displays their values in a HTML list. We can see how `getParameter` is used and how, by making `doPost` call `doGet`, the application is made to respond to the two methods. If required, we have methods for reading the standard input, as in CGI programming.

We will now look at a more complex example to illustrate the full potential of the servlets API. This example receives data from a form, searches for the names of the parameters and prints them, indicating those with the value of zero and those with multiple values.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class Parameters extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                 HttpServletResponse response)
       throws ServletException, IOException
 {
       response.setContentType("text/html");

       PrintWriter out = response.getWriter();
       String tit= "Reading Parameters";

       out.println(Utilities.titleHeader(tit) +
              "<body BGCOLOR=\"#FDF5E6\">\n" +
              "<H1 ALIGN=CENTER>" + tit + "</H1>\n" +
              "<TABLE BORDER=1 ALIGN=CENTER>\n" +
              "<TR BGCOLOR=\"#FFAD00\">\n" +
              "<TH>Parameter Name<TH>Parameter(s) Value");

       // Reading the names of the parameters
       Enumeration params = request.getParameterNames();

       // Going through the names array
       while(params.hasMoreElements())
       {
         // Reading the name
         String param = (String)params.nextElement();
```

```
        // Printing the name
        out.println("<TR><TD>" + paramName + "\n<TD>");

        // Reading the values array of the parameter
        String[] values = request.getParameterValues(param);

        if (values.length == 1)
        {
          // Only one empty value
          String value = values[0];

          // Empty value.
          if (value.length() == 0)
        out.print("Empty");
     else
     out.print(value);
    }
    else
    {
      // Multiple values
       out.println("<UL>");
      for(int i=0; i<values.length; i++)
  {
      out.println("<LI>" + values[i]);
  }
      out.println("</UL>");
  }
  }
      out.println("</TABLE>\n</BODY>\n</HTML>");
 }

 public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
   throws ServletException, IOException
 {
   doGet(request, response);
 }
}
```

We first look for the names of all parameters using the method `getParameterNames`. This returns an `enumeration`. We then use the standard method to run `enumeration` (using `hasMoreElements` to determine when to stop and `nextElement` to obtain each input). Since `nextElement` returns an `object` object, we convert the result to `String` and we use them with `getParameterValues` to obtain a `String`. If this array only has one entry and contains only one empty `String`, the parameter has

no values and the servlet will generate an "empty" entry in italics. If the array contains more than one entry, the parameter has multiple values, which are displayed in an unsorted list. Otherwise, the only value is displayed.

This is an HTML form that will be used to test the servlet, as it sends a group of parameters to it. Since the form contains a `PASSWORD` type field, we will use the POST method to send the values.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <HEAD>
 <TITLE>Form with POST</TITLE>
</head>

<BODY BGCOLOR="#FDF5E6">
 <H1 ALIGN="CENTER">Form with POST</H1>

 <FORM ACTION="/examples/servlets/Parameters" METHOD="POST">
    Code: <INPUT TYPE="TEXT" NAME="code"><BR>
    Quantity: <INPUT TYPE="TEXT" NAME="quantity"><BR>
    Price: <INPUT TYPE="TEXT" NAME="price" VALUE="\$"><BR>
    <HR>
    Name:
    <INPUT TYPE="TEXT" NAME="Name"><BR>
    Surname:
    <INPUT TYPE="TEXT" NAME="Surname"><br>

    Address:
    <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><br>

    Credit card:<BR>
       <INPUT TYPE="RADIO" NAME="cred"
              VALUE="Visa">Visa<BR>
       <INPUT TYPE="RADIO" NAME="cred"
              VALUE="MasterCard">MasterCard<BR>
       <INPUT TYPE="RADIO" NAME="cred"
              VALUE="Amex">American Express<BR>
       <INPUT TYPE="RADIO" NAME="cred"
              VALUE="Maestro">Maestro<BR>
Card number:
<INPUT TYPE="PASSWORD" NAME="cardno"><br>

 Re-enter card number:
 <INPUT TYPE="PASSWORD" NAME="cardno"><BR><BR>
 <CENTER>
         <INPUT TYPE="SUBMIT" VALUE="Place order">
 </CENTER>
</FORM>
```

```
</BODY>
</HTML>
```

### 3.8. The HTTP request: HttpRequest

When an HTTP client (the browser) sends a request, it can send a specific number of optional headers, except for Content-Length, which is required in POST requests. These headers provide additional information to the web server, which can use them to adapt its response to suit the browser request.

Some of the most common and useful headers are:

- Accept. The MIME types preferred by the browser.
- Accept-Charset. The character set accepted by the browser.
- Accept-Encoding. The types of data encoding accepted by the browser. For example, it can indicate that the browser accepts compressed pages, etc.
- Accept-Language. The language preferred by the browser.
- Authorization. Authorisation information, usually in response to a server request.
- Cookie. XML `cookies` stored in the browser that correspond to the server.
- Host. Server and port of the original request.
- If-Modified-Since. Send only if it has been modified since the specified date.
- Referrer. The URL of the page containing the link followed by the user to obtain the current page.
- User-Agent. Type and brand of browser, useful for adapting the response to specific browsers.

To read the headers, we simply need to call the method `getHeader` of `HttpServletRequest`. This will return a `String`, if the indicated header was sent in the request, and `null` if it was not.

Some header fields are used so often that they have their own methods. The `getCookies` method is used to access *cookies* sent with the HTTP request, analysing and storing them in a `cookie`. The `getAuthType` and `getRemoteUser` methods allow access to each of the components of the

`Authorization` field in the header. The `getDateHeader` and `getIntHeader` methods read the specific header and convert it to the values `Date` and `int`, respectively.

Instead of searching for a specific header, we can use `getHeaderNames` to obtain an `enumeration` of all the header names of a specific request. If this is the case, we can run through this list of headers, etc.

Lastly, as well as accessing the header fields of the request, we can obtain information about the request itself. The `getMethod` returns the method used for the request (usually `GET` or `POST`, but HTTP has other, less common methods, such as `HEAD`, `PUT` and `DELETE`). The `getRequestURI` method returns the `URI` (the part of the `URL` that appears after the name of the server and port but before the form data). The `getRequestProtocol` method returns the protocol used, generally `"HTTP/1.0"` or `"HTTP/1.1"`.

### 3.9. Additional request information

Besides the headers of the HTTP request, we can obtain a series of values that will provide us with further information about the request. Some of these values are available for CGI programming as environment variables. They are all available as `HttpRequest`.

getAuthType (). If an `Authorization` header s supplied, this is the specified schema (`basic` or `digest`). CGI variable: `AUTH_TYPE`.

getContentLength (). Only for POST requests, the number of bytes sent.

getContentType (). The MIME type of the attached data, if specified. CGI variable: `CONTENT_TYPE`.

getPathInfo (). Information on the *path* attached to the URL. CGI variable: `PATH_INFO`.

getQueryString (). For GET requests; these are the data sent as a single string with encoded values. They are not generally used in servlets, since direct access to the decoded parameters is available. CGI variable: `QUERY_STRING`.

getRemoteAddr (). The IP address of the client. CGI variable: `REMOTE_ ADDR`.

getRemoteUser (). If an Authorization header is supplied, the user part. CGI variable: `REMOTE_USER`.

getMethod (). The request type is normally GET or POST, but it can also be HEAD, PUT, DELETE, OPTIONS or TRACE. CGI variable: `REQUEST_METHOD`.

## 3.10. Status and response codes

When a browser's web request is processed, the response usually contains a numerical code that tells the browser whether the request has been fulfilled and, where applicable, the reasons why this is not the case. It also includes some headers to give the browser further information about the response. Servlets can be used to indicate the HTTP return code and the value of some of these headers. This means that we can redirect the user to another page, indicate the type of response content, request a password from the user, etc.

### 3.10.1. Status codes

To return a specific status code, our servlets can use the `setStatus`. which tells the web server and servlet container the status that they should return to the client. In the `HttpServletResponse` class, the servlets API provides a table of constants to facilitate the use of response codes. These constants have names that are easy to remember and use.

> For example, the constant for code 404 (qualified in standard HTTP as *not found*), is SC_NOT_FOUND.

If the code we return is not the default one (200, SC OK), we will need to call `setStatus` before using `PrintWriter` to return the client content. We can also use `setStatus` to return error codes for two more specialised methods: `sendError` to return errors (code 404), which allows us to add a HTML message to the numerical code, and `sendRedirect` (code 302), which is used to specify the address to which the client is redirected.

### 3.10.2. Return headers

Besides including a numerical code when responding to the http request, the server can add a series of values in response headers. These headers tell the browser about the expiry of the information sent (`Expires`), that it must refresh the information after a specific time (`Refresh`), etc. We can modify the value of these headers or add new ones from our servlets. To do so, we can use the `setHeader` method of the class called `HttpServletResponse` class, hich allows us to assign random values to the headers we return to the client. As with return codes, we must select the headers before sending a value to the client. There are two auxiliary methods for `setHeader` for times when we want to send headers containing dates or integers. These methods, `setDateHeader` and `setIntHeader`, do not rule out the need for converting dates and integers to `String`, the parameter accepted by `setHeader`.

There are also specialised methods for some of the more common headers:

`SetContentType`. Provides a value for the `Content-Type` header and must be used in most servlets.

`SetContentLength`. Allows us to assign a value to the `Content-Length`.

`AddCookie`. Assigns a *cookie* to the response.

`SendRedirect`. As well as assigning status code 302, as we saw, it assigns the address to which the user is redirected in the header `Location`.

### 3.11. Session monitoring

HTTP is a stateless protocol, which means that each request is totally independent of the previous one. This means that we cannot link two consecutive requests, which is disastrous if we want to use the web for something more than simply viewing documents. If we are developing an e-commerce application such as an on-line store, we need control over the products that our client has selected to ensure that we have the correct shopping list when the client reaches the order page. How can we obtain the list of objects selected for purchase when this screen is reached?

There are three possible solutions to this problem:

**1)** Use *cookies*. *Cookies* are small pieces of information sent by the server to the browser, which the latter resends every time it accesses the website. Despite excellent support from *cookies*, using this technique to monitor a session is still an arduous task:
- Control the *cookie* containing the session identifier.
- Control expiry of the latter.
- Associate the contents of the *cookie* with information from a session.

**2)** Rewrite the URL. We can use the URL to add further information to identify the session. This solution has the advantage that it works with browsers that have no *cookies* support or where it is disabled. However, it is still a tedious method:
- We need to ensure that all URLs reaching the user have the right session information.

- It causes problems for users trying to add addresses to their *bookmarks*, because these contain expired session information.

**3)** Hidden fields in forms. We can use the `HIDDEN` fields of HTML forms to spread information in our interest. Clearly, this suffers from the same problems as the above solutions.

Fortunately, the servlets API has a solution to this problem. Servlets have a high-level API, `HttpSession`, for session management, which is carried out using *cookies* and URL rewriting. This API isolates the author from the servlets of the details of session management.

### 3.11.1. Obtaining the session associated with the request

To obtain the session associated with the HTTP request in course, we can use a `getSession` method of the class called `HttpServletRequest`. If a session exists, this method will return a `HttpSession`. If it does not exist, it will return `null`. We can call `getSession` using an additional parameter that will create the session automatically if it does not exist.

```
HttpSession session = request.getSession(true);
```

### 3.11.2. Accessing the associated information

The `HttpSession` objects representing the information associated with a session allow us to store a series of named values inside. To read these values, we can use `getAttribute`, and to modify them, we have `setAttribute`.

**Note**

In versions prior to 2.2 of the servlets API, the functions for accessing information were: `getValue` and `setValue`.

One schema for accessing this session data might be:

```
HttpSession session = request.getSession(true);

 Languagestring=(String)session.getAttribute("language");
 if (language == null)
 {
   language=new String("Spanish");
   response.setAttribute("language",language);
 }
 // we can now display the data in the language
 // preferred by the user
```

There are methods for accessing the list of attributes saved in the session, such as `getAttributeNames`, which returns an `enumeration`, similar to the `getHeaders` and `getParameterNames` methods of `HttpServletRequest`.

**Note**

In versions prior to 2.2 of the servlets API, the list of value names function was `getValueNames`.

There are also some useful functions for accessing session information:

**getId** returns a unique identifier generated for each session.

**isNew** returns *true* if the client has never seen the session because it has just been created.

**getCreationTime** returns the time in milliseconds since 1970, the year in which the session was created.

**getLastAccessedTime** returns the time in milliseconds since 1970, the year in which the session was sent to the client for the last time.

### 3.12. Java Server Pages: JSP

Java Server Pages, or JSP, are a HTML extension developed by Sun used to embed Java instructions (*scriptlets*) in the HTML code. This simplifies matters when it comes to designing dynamic websites. We can use any of the many HTML editors to create our web or we can leave this to the designers, focusing instead on the development of the Java code that will generate the dynamic parts of the page so that we can subsequently embed this code in the page.

An example of a basic JSP page that will introduce us to some of the main concepts of the standard is as follows:

```
<HTML>
 <BODY>
 <H1>Welcome. Date: < %= date %> </h1>
 <B>
 < % if(name==null)
      out.println("New user");
   else
      out.println("Welcome back");
 %>
 </b>
 </BODY>
</HTML>
```

JSP pages normally have the extension `.jsp` and are located in the same directory as HTML files. As we can see, a `.jsp` page is simply a HTML page in which we embed pieces of Java code, delimited by `< %` and `%>`. Constructions delimited by `< %` and `%>` can be of three types:

- *Script* elements allowing us to enter a code that will form part of servlet resulting from translation of the page.
- Directives, used to tell the servlet container how we want the servlet to be generated.
- Actions allow us to specify components that should be used.

When the server/servlet container processes a JSP page, it converts this into a servlet in which all of the HTML that we have entered in the JSP page is printed on output and subsequently used for compiling this servlet and passing the

request to it. This conversion/compilation step is generally only carried out the first time we access the page or if the JSP file has been modified since the last time it was compiled.

### 3.12.1. *Script elements*

*Script* elements allow us to insert Java code inside a servlet produced by the compilation of our JSP page. There are three options when it comes to inserting code:

- Expressions of the type `< %= expression %>` which are evaluated and inserted in the output.
- *Scriptlets* of the type `< % code %>` that are inserted within the servlet's Service method.
- Declarations of the type `< %! code %>` that are inserted in the body of the servlet class, outside any existing method.

#### Expressions

JSP expressions are used to insert a Java value directly in the output. Their syntax is:

```
< %= expression %>
```

The expression is evaluated and produces a result that is converted into a string, which is inserted in the resulting page. The evaluation is carried out in execution time, when the page is requested. Hence, expressions can access HTTP request data. For example,

```
< %= request.getRemoteUser() > logged on on
< %= new java.util.Date() >
```

This code will display the remote user (if authenticated) and the date on which the page was requested.

We can see in our example that we are using a variable, `request`, which represents the HTTP request. This predefined variable belongs to a series of predefined variables that we can use:

- `request`: the `HttpServletRequest`
- `response`: the `HttpServletResponse`
- `session`: the `HttpSession` associated with `request` (if it exists)

- `out`: the `PrintWriter` used to send the output to the client

There is an alternative syntax for entering expressions. This syntax was introduced to make JSP compatible with XML editors, *parsers*, etc. It is based on the concept of *tagActions*. The syntax for an expression is:

```
<jsp:expression> expression</jsp:expression>
```

### *Scriptlets*

XML *scriptlets* are used to insert random Java code in the servlet that will result from compilation of the JSP page. A *scriptlet* looks like this:

```
< % code %>
```

In a *scriptlet* we can access the same predefined variables as in an expression. For example:

```
< %
 String user = request.getRemoteUser();
 out.println("User: " + user);
%>
```

XML *scriptlets* are inserted in the resulting servlet as they are written, while the HTML code entered is converted into `println`. This means that we can create constructions such as:

```
<% if (obtainTemperature() < 20) { %>
       <B>Wrap up! It's cold! </B>
<% } else { %>
       <B>Have a nice day!</B>
< % } %>
```

In this example, we see that the Java code blocks can affect and include the HTML defined on the JSP pages. Once the page has been compiled and the servlet generated, the above code will look something like this:

```
if (obtainTemperature() < 20) {
    out.println("<B>Wrap up! It's cold! </B>");
} else {
    out.println("<B>Have a nice day!</B>");
}
```

The XML equivalent for *scriptlets* is:

```
<jsp:scriptlet> code </jsp:scriptlet>
```

### JSP declarations

Declarations are used to define methods or fields that are subsequently inserted in the servlet outside the `service`. They look similar to this:

```
< %! code %>
```

Declarations do not generate output. As a result, they are usually used to define global variables, etc. For example, the following code adds a counter to our page:

```
< %! private int visits = 1; %>
Visits to the page while server is running:
< %= visits++ %>
```

This counter is restored to one each time the servlet container is restarted or each time we modify the servlet or JSP file (which requires the server to reload it). The equivalent to declarations for XML is:

```
<jsp:declaration> code </jsp:declaration>
```

## 3.12.2. JSP directives

Directives affect the general structure of the servlet class. They look like this:

```
< %@ attribute directive1="value1"
                  attribute2="value2"
                  ...
                  %>
```

There are three main directives:

`page` allowing us to modify compilation of the JSP page to the servlet.

**Example**

For example, we can import classes, modify the servlet class, etc.

`include` which allows us to insert another file in the resulting servlet (this is inserted when translating JSP to servlet).

`taglib` which is used to indicate which tag libraries we wish to use. JSP allows us to define our own tag libraries.

### The `page directive`

We can define the following attributes using the page directive, which will modify translation of JSP to servlet:

- `import="package.class"` or `import="package.class1, ... ,package.classN". Import` allows us to specify the packets and classes

that need to be imported by Java to compile the resulting servlet. This attribute can appear several times in each JSP. For example:

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

- `contentType="MIME-Type"` or `contentType="MIME-Type; charset=Character-Set"` This directive is used to specify the resulting MIME type of the page. The default value is `text/html`. For example:

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

This is equivalent to using the *scriptlet*:

```
< % response.setContentType("text/plain"); %>
```

- `isThreadSafe="true|false"`. A `true` value (the default value) indicates that the resulting servlet will be a normal servlet, in which multiple requests can be processed simultaneously, assuming that the instance variables shared between *threads* will be synchronised by the author. A `false` value indicates that the servlet must implement a `SingleThreadModel`.

- `session="true|false"`. A `true` value (the default value) indicates that there must be a predefined `session` variable (of the type `Httpsession`) with the session or, if there is no session, one must be created. A `false` value indicates that sessions will not be used and attempts to access them will result in errors when it comes to translating to servlet.

- `extends="package.class"`. This indicates that the servlet generated must extend a different superclass. It must be used with extreme caution, since the servlet container we use may require the use of a specific superclass.

- `errorPage="URL"`. Specifies which JSP will be processed if an exception is launched (an object of the type `Throwable`) and it is not captured on the current page.

- `isErrorPage="true|false"`. Indicates whether the current page is an error processing page.

The equivalent XML syntax is:

```
<jsp:directive.Directive attribute=value />
```

For example, the following two lines are equivalents:

```
< %@ page import="java.util.*" %>
<jsp:directive.page import="java.util.*" />
```

### The `include` directive

The `include` directive is used to include files in the JSP page when translated to servlet. The syntax is as follows:

```
< %@ include file="file to be included" %>
```

The file to be included can be relative to the position of the JSP on the server, for example, `examples/example1.jsp` or absolute, for example, `/general/header.jsp`. and it can contain any JSP construction: html, scriptlets, directives, actions, etc.

The `include` directive can save us a lot of work because it allows us to write elements like the menus of our website on a single page, which means that we only need to include them in each JSP we use.

```
1.
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
   <HEAD>
     <TITLE>Website</TITLE>
     <META NAME="author" CONTENT="carlesm@asic.udl.es">
     <META NAME="keywords" CONTENT="JSP, Servlets">
     <meta NAME="description" CONTENT="One page">
     <LINK REL=STYLESHEET HREF="style.css" TYPE="text/css">
   </HEAD>
   <body>

2.
<HR>
 <CENTER><small>&#169; Web developer, 2003. All
rights reserved</SMALL></center>

</BODY> </html>

3.
< %@ include file="/header.html" %>
 <!-- JSP page -->
   .
   .
< %@ include file="/footer.html" %>
```

In this example, we have three files: `header.html, footer.html` and a JSP page of the website, respectively. As we can see, having a fragment of the page content in separate files considerably simplifies the writing and maintenance of JSP pages.

One thing to bear in mind is that it is included when the JSP page is translated to servlet. If we change anything in the files included, we will need to force re-translation of the entire site. Although this may seem a problem, it is greatly compensated by the benefits gained by the efficiency of only having to include the files once.

If we want them to be included in each request, we have an alternative in the XML version of the directive:

```
<jsp:include file="/header.html">
 <!-- JSP page -->
   .
   .
<jsp:include file="/header.html">
```

In this case, the inclusion is made when the page is served. However, we cannot include any JSPs in the file we are going to include; it can only be in HTML.

### 3.12.3. Predefined variables

In JSPs, we have a group of defined variables to make code development easier.

- `request`. The `HttpServletRequest` object associated with the request. This allows access to the request parameters (through `getParameter`), the type of request and the HTTP headers (*cookies*, `referrer` etc).

- `response`. This is the `HttpServletResponse` object associated with the servlet response. Since the *stream* output object (the `out` variable defined later) has a *buffer*, we can select the status codes and response headers.

- `out`. This is the `PrintWriter` object used to send the output to the client.

- `session`. This is the `HttpSession` object associated with the request. Sessions are created automatically by default. This variable exists even if there is no reference session. The only exception is if we use the `session` attribute of the `page directive`.

- `application`. This is the `ServletContext` object obtained through `getServletConfig().getContext()`.