**JAVA SERVER PAGES**

**JSP:**

**What is JavaServer Pages?**

JavaServer Pages (JSP) is a technology for developing web pages that support dynamic content which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages etc.

**Why Use JSP?**

JavaServer Pages often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But JSP offer several advantages in comparison with the CGI.

* Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having a separate CGI files.
* JSP are always compiled before it's processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
* JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP etc.
* JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

**Advantages of JSP:**

Following is the list of other advantages of using JSP over other technologies:

* **vs. Active Server Pages (ASP):** The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.
* **vs. Pure Servlets:** It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.
* **vs. Server-Side Includes (SSI):** SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.
* **vs. JavaScript:** JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.
* **vs. Static HTML:** Regular HTML, of course, cannot contain dynamic information.

**What is Next?**

I would take you step by step to set up your environment to start with JSP. I'm assuming you have good hands on with Java Programming to proceed with learning JSP.

If you are not aware of Java Programming Language then I would recommend to go through [Java Tutorial](http://www.tutorialspoint.com/java/index.htm) to understand Java Programming.

A development environment is where you would develop your JSP programs, test them and finally run them.

**Setting up Java Development Kit:**

This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up PATH environment variable appropriately.

You can download SDK from Oracle's Java site: [Java SE Downloads](http://www.oracle.com/technetwork/java/javase/downloads/index.html).

Once you download your Java implementation, follow the given instructions to install and configure the setup. Finally set PATH and JAVA\_HOME environment variables to refer to the directory that contains java and javac, typically java\_install\_dir/bin and java\_install\_dir respectively.

If you are running Windows and installed the SDK in C:\jdk1.5.0\_20, you would put the following line in your C:\autoexec.bat file.

set PATH=C:\jdk1.5.0\_20\bin;%PATH%

set JAVA\_HOME=C:\jdk1.5.0\_20

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.5.0\_20 and you use the C shell, you would put the following into your .cshrc file.

setenv PATH /usr/local/jdk1.5.0\_20/bin:$PATH

setenv JAVA\_HOME /usr/local/jdk1.5.0\_20

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java.

**Setting up Web Server: Tomcat**

A number of Web Servers that support JavaServer Pages and Servlets development are available in the market. Some web servers are freely downloadable and Tomcat is one of them.

Apache Tomcat is an open source software implementation of the JavaServer Pages and Servlet technologies and can act as a standalone server for testing JSP and Servlets and can be integrated with the Apache Web Server. Here are the steps to setup Tomcat on your machine:

* Download latest version of Tomcat from <http://tomcat.apache.org/>.
* Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\apache-tomcat-5.5.29 on windows, or /usr/local/apache-tomcat-5.5.29 on Linux/Unix and create CATALINA\_HOME environment variable pointing to these locations.

Tomcat can be started by executing the following commands on windows machine:

%CATALINA\_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat

Tomcat can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

$CATALINA\_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh

After a successful startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**. If everything is fine then it should display following result:

Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site: http://tomcat.apache.org

Tomcat can be stopped by executing the following commands on windows machine:

%CATALINA\_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine:

$CATALINA\_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh

**Setting up CLASSPATH:**

Since servlets are not part of the Java Platform, Standard Edition, you must identify the servlet classes to the compiler.

If you are running Windows, you need to put the following lines in your C:\autoexec.bat file.

set CATALINA=C:\apache-tomcat-5.5.29

set CLASSPATH=%CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the CLASSPATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if you are using the C shell, you would put the following lines into your .cshrc file.

setenv CATALINA=/usr/local/apache-tomcat-5.5.29

setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH

**NOTE:** Assuming that your development directory is C:\JSPDev (Windows) or /usr/JSPDev (Unix) then you would need to add these directories as well in CLASSPATH in similar way as you have added above.

The web server needs a JSP engine ie. container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

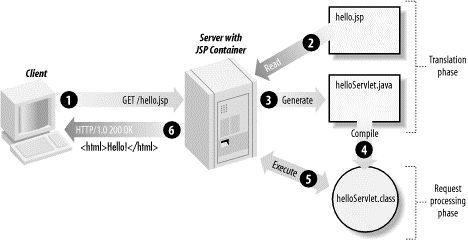
Following diagram shows the position of JSP container and JSP files in a Web Application.

**JSP Processing:**

The following steps explain how the web server creates the web page using JSP:

* As with a normal page, your browser sends an HTTP request to the web server.
* The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of .html.
* The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page.
* The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
* A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
* The web server forwards the HTTP response to your browser in terms of static HTML content.
* Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

All the above mentioned steps can be shown below in the following diagram:



Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with other scripting languages (such as PHP) and therefore faster.

So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet

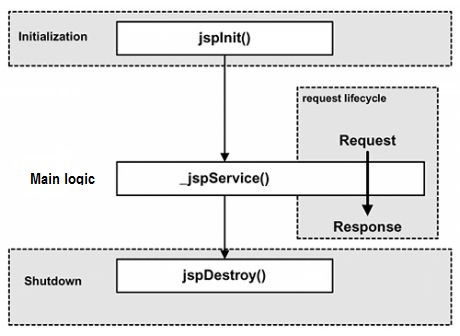
The key to understanding the low-level functionality of JSP is to understand the simple life cycle they follow.

A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

**The following are the paths followed by a JSP:**

1. JSP Compilation
2. JSP Initialization
3. JSP Execution
4. JSP Cleanup

The four major phases of JSP life cycle are very similar to Servlet Life Cycle and they are as follows:



**1.JSP Compilation:**

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps:

* Parsing the JSP.
* Turning the JSP into a servlet.
* Compiling the servlet.

**2.JSP Initialization:**

When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method:

public void jspInit(){

// Initialization code...

}

Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

**3.JSP Execution:**

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **\_jspService()** method in the JSP.

The \_jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

void \_jspService(HttpServletRequest request,

HttpServletResponse response)

{

// Service handling code...

}

The \_jspService() method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

**4.JSP Cleanup:**

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

**The jspDestroy() method has the following form:**

public void jspDestroy()

{

// Your cleanup code goes here.

}

This tutorial will give basic idea on simple syntax (ie. elements) involved with JSP development:

**The Scriptlet:**

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

Following is the syntax of Scriptlet:

<% code fragment %>

You can write XML equivalent of the above syntax as follows:

<jsp:scriptlet>

code fragment

</jsp:scriptlet>

Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple and first example for JSP:

<html>

<head><title>Hello World</title></head>

<body>

Hello World!<br/>

<%

out.println("Your IP address is " + request.getRemoteAddr());

%>

</body>

</html>

**NOTE:** Assuming that Apache Tomcat is installed in C:\apache-tomcat-7.0.2 and your environment is setup as per environment setup tutorial.

Let us keep above code in JSP file hello.jsp and put this file in **C:\apache-tomcat-7.0.2\webapps\ROOT**directory and try to browse it by giving URL http://localhost:8080/hello.jsp. This would generate following result:

**JSP Declarations:**

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

Following is the syntax of JSP Declarations:

<%! declaration; [ declaration; ]+ ... %>

You can write XML equivalent of the above syntax as follows:

<jsp:declaration>

code fragment

</jsp:declaration>

Following is the simple example for JSP Declarations:

<%! int i = 0; %>

<%! int a, b, c; %>

<%! Circle a = new Circle(2.0); %>

**JSP Expression:**

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression.

Following is the syntax of JSP Expression:

<%= expression %>

You can write XML equivalent of the above syntax as follows:

<jsp:expression>

expression

</jsp:expression>

Following is the simple example for JSP Expression:

<html>

<head><title>A Comment Test</title></head>

<body>

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

</body>

</html>

This would generate following result:

|  |
| --- |
| Today's date: 11-Sep-2010 21:24:25 |

**JSP Comments:**

JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out" part of your JSP page.

Following is the syntax of JSP comments:

<%-- This is JSP comment --%>

Following is the simple example for JSP Comments:

<html>

<head><title>A Comment Test</title></head>

<body>

<h2>A Test of Comments</h2>

<%-- This comment will not be visible in the page source --%>

</body>

</html>

This would generate following result:

|  |
| --- |
| A Test of Comments |

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| <%-- comment --%> | A JSP comment. Ignored by the JSP engine. |
| <!-- comment --> | An HTML comment. Ignored by the browser. |
| <\% | Represents static <% literal. |
| %\> | Represents static %> literal. |
| \' | A single quote in an attribute that uses single quotes. |
| \" | A double quote in an attribute that uses double quotes. |

**JSP Directives:**

A JSP directive affects the overall structure of the servlet class. It usually has the following form:

<%@ directive attribute="value" %>

There are three types of directive tag:

|  |  |
| --- | --- |
| **Directive** | **Description** |
| <%@ page ... %> | Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| <%@ include ... %> | Includes a file during the translation phase. |
| <%@ taglib ... %> | Declares a tag library, containing custom actions, used in the page |

We would explain JSP directive in separate chapter [JSP - Directives](http://www.tutorialspoint.com/jsp/jsp_directives.htm)

**JSP Actions:**

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard:

<jsp:action\_name attribute="value" />

Action elements are basically predefined functions and there are following JSP actions available:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| jsp:include | Includes a file at the time the page is requested |
| jsp:include | Includes a file at the time the page is requested |
| jsp:useBean | Finds or instantiates a JavaBean |
| jsp:setProperty | Sets the property of a JavaBean |
| jsp:getProperty | Inserts the property of a JavaBean into the output |
| jsp:forward | Forwards the requester to a new page |
| jsp:plugin | Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin |
| jsp:element | Defines XML elements dynamically. |
| jsp:attribute | Defines dynamically defined XML element's attribute. |
| jsp:body | Defines dynamically defined XML element's body. |
| jsp:text | Use to write template text in JSP pages and documents. |

We would explain JSP actions in separate chapter [JSP - Actions](http://www.tutorialspoint.com/jsp/jsp_actions.htm)

**JSP Implicit Objects:**

JSP supports nine automatically defined variables, which are also called implicit objects. These variables are:

|  |  |
| --- | --- |
| **Objects** | **Description** |
| Request | This is the **HttpServletRequest** object associated with the request. |
| response | This is the **HttpServletResponse** object associated with the response to the client. |
| Out | This is the **PrintWriter** object used to send output to the client. |
| Session | This is the **HttpSession** object associated with the request. |
| application | This is the **ServletContext** object associated with application context. |
| Config | This is the **ServletConfig** object associated with the page. |
| pageContext | This encapsulates use of server-specific features like higher performance **JspWriters**. |
| Page | This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. |
| Exception | The **Exception** object allows the exception data to be accessed by designated JSP. |

We would explain JSP Implicit Objects in separate chapter [JSP - Implicit Objects](http://www.tutorialspoint.com/jsp/jsp_implicit_objects.htm).

**Control-Flow Statements:**

JSP provides full power of Java to be embedded in your web application. You can use all the APIs and building blocks of Java in your JSP programming including decision making statements, loops etc.

**Decision-Making Statements:**

The **if...else** block starts out like an ordinary Scriptlet, but the Scriptlet is closed at each line with HTML text included between Scriptlet tags.

<%! int day = 3; %>

<html>

<head><title>IF...ELSE Example</title></head>

<body>

<% if (day == 1 | day == 7) { %>

<p> Today is weekend</p>

<% } else { %>

<p> Today is not weekend</p>

<% } %>

</body>

</html>

This would produce following result:

|  |
| --- |
| Today is not weekend |

Now look at the following **switch...case** block which has been written a bit differentlty using **out.println()**and inside Scriptletas:

<%! int day = 3; %>

<html>

<head><title>SWITCH...CASE Example</title></head>

<body>

<%

switch(day) {

case 0:

out.println("It\'s Sunday.");

break;

case 1:

out.println("It\'s Monday.");

break;

case 2:

out.println("It\'s Tuesday.");

break;

case 3:

out.println("It\'s Wednesday.");

break;

case 4:

out.println("It\'s Thursday.");

break;

case 5:

out.println("It\'s Friday.");

break;

default:

out.println("It's Saturday.");

}

%>

</body>

</html>

This would produce following result:

|  |
| --- |
| It's Wednesday. |

**Loop Statements:**

You can also use three basic types of looping blocks in Java: **for, while,and do…while** blocks in your JSP programming.

Let us look at the following **for** loop example:

<%! int fontSize; %>

<html>

<head><title>FOR LOOP Example</title></head>

<body>

<%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>

<font color="green" size="<%= fontSize %>">

JSP Tutorial

</font><br />

<%}%>

</body>

</html>

This would produce following result:

|  |
| --- |
| JSPTutorial  JSPTutorial  JSP Tutorial |

Above example can be written using **while** loop as follows:

<%! int fontSize; %>

<html>

<head><title>WHILE LOOP Example</title></head>

<body>

<%while ( fontSize <= 3){ %>

<font color="green" size="<%= fontSize %>">

JSP Tutorial

</font><br />

<%fontSize++;%>

<%}%>

</body>

</html>

This would also produce following result:

|  |
| --- |
| JSP Tutorial  JSP Tutorial  JSP Tutorial |

**JSP Operators:**

JSP supports all the logical and arithmetic operators supported by Java. Following table give a list of all the operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**JSP Literals:**

The JSP expression language defines the following literals:

* **Boolean:** true and false
* **Integer:** as in Java
* **Floating point:** as in Java
* **String:** with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
* **Null:** null

JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of JSP processing.

A JSP directive affects the overall structure of the servlet class. It usually has the following form:

<%@ directive attribute="value" %>

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.

The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.

There are three types of directive tag:

|  |  |
| --- | --- |
| **Directive** | **Description** |
| <%@ page ... %> | Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| <%@ include ... %> | Includes a file during the translation phase. |
| <%@ taglib ... %> | Declares a tag library, containing custom actions, used in the page |

The page Directive:

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive:

<%@ page attribute="value" %>

You can write XML equivalent of the above syntax as follows:

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

**Attributes:**

Following is the list of attributes associated with page directive:

|  |  |
| --- | --- |
| **Attribute** | **Purpose** |
| Buffer | Specifies a buffering model for the output stream. |
| autoFlush | Controls the behavior of the servlet output buffer. |
| contentType | Defines the character encoding scheme. |
| errorPage | Defines the URL of another JSP that reports on Java unchecked runtime exceptions. |
| isErrorPage | Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute. |
| Extends | Specifies a superclass that the generated servlet must extend |
| Import | Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes. |
| Info | Defines a string that can be accessed with the servlet's getServletInfo() method. |
| isThreadSafe | Defines the threading model for the generated servlet. |
| Language | Defines the programming language used in the JSP page. |
| Session | Specifies whether or not the JSP page participates in HTTP sessions |
| isELIgnored | Specifies whether or not EL expression within the JSP page will be ignored. |
| isScriptingEnabled | Determines if scripting elements are allowed for use. |

Check more detail related to all the above attributes at [Page Directive](http://www.tutorialspoint.com/jsp/page_directive.htm).

**The include Directive:**

The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.

The general usage form of this directive is as follows:

<%@ include file="relative url" >

The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

You can write XML equivalent of the above syntax as follows:

<jsp:directive.include file="relative url" />

Check more detail related to include directive at [Include Directive](http://www.tutorialspoint.com/jsp/include_directive.htm).

**The taglib Directive:**

The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.

The taglib directive follows the following syntax:

<%@ taglib uri="uri" prefix="prefixOfTag" >

Where the **uri** attribute value resolves to a location the container understands and the **prefix** attribute informs a container what bits of markup are custom actions.

You can write XML equivalent of the above syntax as follows:

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

Check more detail related to taglib directive at [Taglib Directive](http://www.tutorialspoint.com/jsp/taglib_directive.htm).

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard:

<jsp:action\_name attribute="value" />

Action elements are basically predefined functions and there are following JSP actions available:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| jsp:include | Includes a file at the time the page is requested |
| jsp:useBean | Finds or instantiates a JavaBean |
| jsp:setProperty | Sets the property of a JavaBean |
| jsp:getProperty | Inserts the property of a JavaBean into the output |
| jsp:forward | Forwards the requester to a new page |
| jsp:plugin | Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin |
| jsp:element | Defines XML elements dynamically. |
| jsp:attribute | Defines dynamically defined XML element's attribute. |
| jsp:body | Defines dynamically defined XML element's body. |
| jsp:text | Use to write template text in JSP pages and documents. |

**Common Attributes:**

There are two attributes that are common to all Action elements: the **id** attribute and the **scope** attribute.

* **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext
* **Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b)request, (c)session, and (d) application.

**The <jsp:include> Action:**

This action lets you insert files into the page being generated. The syntax looks like this:

<jsp:include page="relative URL" flush="true" />

Unlike the **include** directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

Following is the list of attributes associated with include action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Page | The relative URL of the page to be included. |
| Flush | The boolean attribute. |

Example:

Let us define following two files (a)date.jps and (b) main.jsp as follows:

Following is the content of date.jsp file:

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

Here is the content of main.jsp file:

<html>

<head>

<title>The include Action Example</title>

</head>

<body>

<center>

<h2>The include action Example</h2>

<jsp:include page="date.jsp" flush="true" />

</center>

</body>

</html>

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like this:

|  |
| --- |
| The include action Example  Today's date: 12-Sep-2010 14:54:22 |

**The <jsp:useBean> Action:**

The **useBean** action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows:

<jsp:useBean id="name" class="package.class" />

Once a bean class is loaded, you can use **jsp:setProperty** and **jsp:getProperty** actions to modify and retrieve bean properties.

Following is the list of attributes associated with useBean action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Class | Designates the full package name of the bean. |
| Type | Specifies the type of the variable that will refer to the object. |
| beanName | Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class. |

Let us discuss about **jsp:setProperty** and **jsp:getProperty** actions before giving a valid example related to these actions.

**The <jsp:setProperty> Action:**

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action:

You can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

<jsp:useBean id="myName" ... />

...

<jsp:setProperty name="myName" property="someProperty" .../>

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.

A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

<jsp:useBean id="myName" ... >

...

<jsp:setProperty name="myName" property="someProperty" .../>

</jsp:useBean>

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

Following is the list of attributes associated with setProperty action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Name | Designates the bean whose property will be set. The Bean must have been previously defined. |
| property | Indicates the property you want to set. A value of "\*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods. |
| Value | The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action is ignored. |
| Param | The param attribute is the name of the request parameter whose value the property is to receive. You can't use both value and param, but it is permissible to use neither. |

**The <jsp:getProperty> Action:**

The **getProperty** action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output.

The getProperty action has only two attributes, both of which are required ans simple syntax is as follows:

<jsp:useBean id="myName" ... />

...

<jsp:getProperty name="myName" property="someProperty" .../>

Following is the list of required attributes associated with setProperty action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Name | The name of the Bean that has a property to be retrieved. The Bean must have been previously defined. |
| property | The property attribute is the name of the Bean property to be retrieved. |

Example:

/\* File: TestBean.java \*/

package action;

public class TestBean {

private String message = "No message specified";

public String getMessage() {

return(message);

} public void setMessage(String message) {

this.message = message;

}

}

Compile above code to generated TestBean.class file and make sure that you copied TestBean.class in C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action folder and CLASSPATH variable should also be set to this folder:

Now use the following code in main.jsp file which loads the bean and sets/gets a simple String parameter:

<html>

<head>

<title>Using JavaBeans in JSP</title>

</head>

<body>

<center>

<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"

property="message"

value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>

</body>

</html>

Now try to access main.jsp, it would display following result:

|  |
| --- |
| Using JavaBeans in JSP  Gotmessage.... Hello JSP... |

**The <jsp:forward> Action:**

The **forward** action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.

The simple syntax of this action is as follows:

<jsp:forward page="Relative URL" />

Following is the list of required attributes associated with forward action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Page | Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet. |

Example:

Let us reuse following two files (a) date.jps and (b) main.jsp as follows:

Following is the content of date.jsp file:

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

Here is the content of main.jsp file:

<html>

<head>

<title>The include Action Example</title>

</head>

<body>

<center>

<h2>The include action Example</h2>

<jsp:forward page="date.jsp" />

</center>

</body>

</html>

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like as below. Here it discarded content from main page and displayed content from forwarded page only.

|  |
| --- |
| Today's date: 12-Sep-2010 14:54:22 |

**The <jsp:plugin> Action:**

The **plugin** action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed.

If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean.

The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean.

Following is the typical syntax of using plugin action:

<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"

width="60" height="80">

<jsp:param name="fontcolor" value="red" />

<jsp:param name="background" value="black" />

<jsp:fallback>

Unable to initialize Java Plugin

</jsp:fallback>

</jsp:plugin>

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

The <jsp:element> Action

The <jsp:attribute> Action

The <jsp:body> Action

The <jsp:element>, lt;jsp:attribute> and <jsp:body> actions are used to define XML elements dynamically. The word dynamically is important, because it means that the XML elements can be generated at request time rather than statically at compile time.

Following is a simple example to define XML elements dynamically:

<%@page language="java" contentType="text/html"%>

<html xmlns="http://www.w3c.org/1999/xhtml"

xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>

<body>

<jsp:element name="xmlElement">

<jsp:attribute name="xmlElementAttr">

Value for the attribute

</jsp:attribute>

<jsp:body>

Body for XML element

</jsp:body>

</jsp:element>

</body>

</html>

This would produce following HTML code at run time:

<html xmlns="http://www.w3c.org/1999/xhtml"

xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>

<body>

<xmlElement xmlElementAttr="Value for the attribute">

Body for XML element

</xmlElement>

</body>

</html>

**The <jsp:text> Action:**

The <jsp:text> action can be used to write template text in JSP pages and documents. Following is the simple syntax for this action:

<jsp:text>Template data</jsp:text>

The body fo the template cannot contain other elements; it can only contain text and EL expressions ( Note: EL expressions are explained in subsequent chapter). Note that in XML files, you cannot use expressions such as ${whatever > 0}, because the greater than signs are illegal. Instead, use the gt form, such as ${whatever gt 0} or an alternative is to embed the value in a CDATA section.

<jsp:text><![CDATA[<br>]]></jsp:text>

If you need to include a DOCTYPE declaration, for instance for XHTML, you must also use the <jsp:text> element as follows:

<jsp:text><![CDATA[<!DOCTYPE html

PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"DTD/xhtml1-strict.dtd">]]>

</jsp:text>

<head><title>jsp:text action</title></head>

<body>

<books><book><jsp:text>

Welcome to JSP Programming

</jsp:text></book></books>

</body>

</html>

Try above example with and without <jsp:text> action.

JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

JSP supports nine Implicit Objects which are listed below:

|  |  |
| --- | --- |
| **Object** | **Description** |
| Request | This is the **HttpServletRequest** object associated with the request. |
| response | This is the **HttpServletResponse** object associated with the response to the client. |
| Out | This is the **PrintWriter** object used to send output to the client. |
| Session | This is the **HttpSession** object associated with the request. |
| application | This is the **ServletContext** object associated with application context. |
| config | This is the **ServletConfig** object associated with the page. |
| pageContext | This encapsulates use of server-specific features like higher performance **JspWriters**. |
| page | This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. |
| Exception | The **Exception** object allows the exception data to be accessed by designated JSP. |

**The request Object:**

The request object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

We would see complete set of methods associated with request object in coming chapter: [JSP - Client Request](http://www.tutorialspoint.com/jsp/jsp_client_request.htm).

**The response Object:**

The response object is an instance of a javax.servlet.http.HttpServletResponse object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

We would see complete set of methods associated with response object in coming chapter: [JSP - Server Response](http://www.tutorialspoint.com/jsp/jsp_server_response.htm).

**The out Object:**

The out implicit object is an instance of a javax.servlet.jsp.JspWriter object and is used to send content in a response.

The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions.

Following are the important methods which we would use to write boolean char, int, double, object, String etc.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **out.print(dataType dt)** | Print a data type value |
| **out.println(dataType dt)** | Print a data type value then terminate the line with new line character. |
| **out.flush()** | Flush the stream. |

**The session Object:**

The session object is an instance of javax.servlet.http.HttpSession and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests. We would see complete usage of session object in coming chapter: [JSP - Session Tracking](http://www.tutorialspoint.com/jsp/jsp_session_tracking.htm).

The application Object:

The application object is direct wrapper around the ServletContext object for the generated Servlet and in reality an instance of a javax.servlet.ServletContext object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

You can check a simple use of Application Object in chapter: [JSP - Hits Counter](http://www.tutorialspoint.com/jsp/jsp_hits_counter.htm)

**The config Object:**

The config object is an instantiation of javax.servlet.ServletConfig and is a direct wrapper around the ServletConfig object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following config method is the only one you might ever use, and its usage is trivial:

config.getServletName();

This returns the servlet name, which is the string contained in the <servlet-name> element defined in the WEB-INF\web.xml file

**The pageContext Object:**

The pageContext object is an instance of a javax.servlet.jsp.PageContext object. The pageContext object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object.

The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including PAGE\_SCOPE, REQUEST\_SCOPE, SESSION\_SCOPE, and APPLICATION\_SCOPE, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the javax.servlet.jsp. JspContext class.

One of the important methods is **removeAttribute**, which accepts either one or two arguments. For example, pageContext.removeAttribute ("attrName") removes the attribute from all scopes, while the following code only removes it from the page scope:

pageContext.removeAttribute("attrName", PAGE\_SCOPE);

You can check a very good usage of pageContext in coming chapter: [JSP - File Uploading](http://www.tutorialspoint.com/jsp/jsp_file_uploading.htm).

**The page Object:**

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the **this** object.

**The exception Object:**

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

We would see complete usage of this object in coming chapter: [JSP - Exception Handling](http://www.tutorialspoint.com/jsp/jsp_exception_handling.htm).

When a browser requests for a web page, it sends lot of information to the web server which can not be read directly because this information travel as a part of header of HTTP request. You can check [HTTP Protocol](http://www.tutorialspoint.com/http/index.htm) for more information on this.

Following is the important header information which comes from browser side and you would use very frequently in web programming:

|  |  |
| --- | --- |
| **Header** | **Description** |
| Accept | This header specifies the MIME types that the browser or other clients can handle. Values of **image/png** or **image/jpeg** are the two most common possibilities. |
| Accept-Charset | This header specifies the character sets the browser can use to display the information. For example ISO-8859-1. |
| Accept-Encoding | This header specifies the types of encodings that the browser knows how to handle. Values of **gzip** or **compress** are the two most common possibilities. |
| Accept-Language | This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc. |
| Authorization | This header is used by clients to identify themselves when accessing password-protected Web pages. |
| Connection | This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or other browser to retrieve multiple files with a single request. A value of**Keep-Alive** means that persistent connections should be used |
| Content-Length | This header is applicable only to POST requests and gives the size of the POST data in bytes. |
| Cookie | This header returns cookies to servers that previously sent them to the browser. |
| Host | This header specifies the host and port as given in the original URL. |
| If-Modified-Since | This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means **Not Modified** header if no newer result is available. |
| If-Unmodified-Since | This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date. |
| Referer | This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2. |
| User-Agent | This header identifies the browser or other client making the request and can be used to return different content to different types of browsers. |

## The HttpServletRequest Object:

The request object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

There are following important methods which can be used to read HTTP header in your JSP program. These method are available with *HttpServletRequest* object which represents client request to webserver.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **Cookie[] getCookies()** Returns an array containing all of the Cookie objects the client sent with this request. |
| 2 | **Enumeration getAttributeNames()** Returns an Enumeration containing the names of the attributes available to this request. |
| 3 | **Enumeration getHeaderNames()** Returns an enumeration of all the header names this request contains. |
| 4 | **Enumeration getParameterNames()** Returns an Enumeration of String objects containing the names of the parameters contained in this request. |
| 5 | **HttpSession getSession()** Returns the current session associated with this request, or if the request does not have a session, creates one. |
| 6 | **HttpSession getSession(boolean create)** Returns the current HttpSession associated with this request or, if if there is no current session and create is true, returns a new session. |
| 7 | **Locale getLocale()** Returns the preferred Locale that the client will accept content in, based on the Accept-Language header |
| 8 | **Object getAttribute(String name)** Returns the value of the named attribute as an Object, or null if no attribute of the given name exists. |
| 9 | **ServletInputStream getInputStream()** Retrieves the body of the request as binary data using a ServletInputStream. |
| 10 | **String getAuthType()** Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected |
| 11 | **String getCharacterEncoding()** Returns the name of the character encoding used in the body of this request. |
| 12 | **String getContentType()** Returns the MIME type of the body of the request, or null if the type is not known. |
| 13 | **String getContextPath()** Returns the portion of the request URI that indicates the context of the request. |
| 14 | **String getHeader(String name)** Returns the value of the specified request header as a String. |
| 15 | **String getMethod()** Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT. |
| 16 | **String getParameter(String name)** Returns the value of a request parameter as a String, or null if the parameter does not exist. |
| 17 | **String getPathInfo()** Returns any extra path information associated with the URL the client sent when it made this request. |
| 18 | **String getProtocol()** Returns the name and version of the protocol the request. |
| 19 | **String getQueryString()** Returns the query string that is contained in the request URL after the path. |
| 20 | **String getRemoteAddr()** Returns the Internet Protocol (IP) address of the client that sent the request. |
| 21 | **String getRemoteHost()** Returns the fully qualified name of the client that sent the request. |
| 22 | **String getRemoteUser()** Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. |
| 23 | **String getRequestURI()** Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request. |
| 24 | **String getRequestedSessionId()** Returns the session ID specified by the client. |
| 25 | **String getServletPath()** Returns the part of this request's URL that calls the JSP. |
| 26 | **String[] getParameterValues(String name)** Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist. |
| 27 | **boolean isSecure()** Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS. |
| 28 | **int getContentLength()** Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known. |
| 29 | **int getIntHeader(String name)** Returns the value of the specified request header as an int. |
| 30 | **int getServerPort()** Returns the port number on which this request was received. |

## HTTP Header Request Example:

Following is the example which uses **getHeaderNames()** method of HttpServletRequest to read the HTTP header infromation. This method returns an Enumeration that contains the header information associated with the current HTTP request.

Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using*hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>HTTP Header Request Example</title>

</head>

<body>

<center>

<h2>HTTP Header Request Example</h2>

<table width="100%" border="1" align="center">

<tr bgcolor="#949494">

<th>Header Name</th><th>Header Value(s)</th>

</tr>

<%

Enumeration headerNames = request.getHeaderNames();

while(headerNames.hasMoreElements()) {

String paramName = (String)headerNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getHeader(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

%>

</table>

</center>

</body>

</html>

Now put the above code in main.jsp and try to access it. This would produce result something as follows:

# HTTP Header Request Example

|  |  |
| --- | --- |
| **Header Name** | **Header Value(s)** |
| Accept | \*/\* |
| accept-language | en-us |
| user-agent | Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; MS-RTC LM 8) |
| accept-encoding | gzip, deflate |
| Host | localhost:8080 |
| connection | Keep-Alive |
| cache-control | no-cache |

To become more comfortable with other methods you can try few more above listed methods in the same fashion.

When a Web server responds to a HTTP request to the browser, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

HTTP/1.1 200 OK

Content-Type: text/html

Header2: ...

...

HeaderN: ...

(Blank Line)

<!doctype ...>

<html>

<head>...</head>

<body>

...

</body>

</html>

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example).

Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side and you would use them very frequently in web programming:

|  |  |
| --- | --- |
| **Header** | **Description** |
| Allow | This header specifies the request methods (GET, POST, etc.) that the server supports. |
| Cache-Control | This header specifies the circumstances in which the response document can safely be cached. It can have values **public, private** or**no-cache** etc. Public means document is cacheable, Private means document is for a single user and can only be stored in private (nonshared) caches and no-cache means document should never be cached. |
| Connection | This header instructs the browser whether to use persistent in HTTP connections or not. A value of **close** instructs the browser not to use persistent HTTP connections and **keep-alive** means using persistent connections. |
| Content-Disposition | This header lets you request that the browser ask the user to save the response to disk in a file of the given name. |
| Content-Encoding | This header specifies the way in which the page was encoded during transmission. |
| Content-Language | This header signifies the language in which the document is written. For example en, en-us, ru, etc. |
| Content-Length | This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection. |
| Content-Type | This header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. |
| Expires | This header specifies the time at which the content should be considered out-of-date and thus no longer be cached. |
| Last-Modified | This header indicates when the document was last changed. The client can then cache the document and supply a date by an **If-Modified-Since** request header in later requests. |
| Location | This header should be included with all responses that have a status code in the 300s. This notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document. |
| Refresh | This header specifies how soon the browser should ask for an updated page. You can specify time in number of seconds after which a page would be refreshed. |
| Retry-After | This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request. |
| Set-Cookie | This header specifies a cookie associated with the page. |

**The HttpServletResponse Object:**

The response object is an instance of a javax.servlet.http.HttpServletRequest object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

There are following methods which can be used to set HTTP response header in your servlet program. These method are available with *HttpServletResponse* object which represents server response.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **String encodeRedirectURL(String url)** Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged. |
| 2 | **String encodeURL(String url)** Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. |
| 3 | **boolean containsHeader(String name)** Returns a boolean indicating whether the named response header has already been set. |
| 4 | **boolean isCommitted()** Returns a boolean indicating if the response has been committed. |
| 5 | **void addCookie(Cookie cookie)** Adds the specified cookie to the response. |
| 6 | **void addDateHeader(String name, long date)** Adds a response header with the given name and date-value. |
| 7 | **void addHeader(String name, String value)** Adds a response header with the given name and value. |
| 8 | **void addIntHeader(String name, int value)** Adds a response header with the given name and integer value. |
| 9 | **void flushBuffer()** Forces any content in the buffer to be written to the client. |
| 10 | **void reset()** Clears any data that exists in the buffer as well as the status code and headers. |
| 11 | **void resetBuffer()** Clears the content of the underlying buffer in the response without clearing headers or status code. |
| 12 | **void sendError(int sc)** Sends an error response to the client using the specified status code and clearing the buffer. |
| 13 | **void sendError(int sc, String msg)** Sends an error response to the client using the specified status. |
| 14 | **void sendRedirect(String location)** Sends a temporary redirect response to the client using the specified redirect location URL. |
| 15 | **void setBufferSize(int size)** Sets the preferred buffer size for the body of the response. |
| 16 | **void setCharacterEncoding(String charset)** Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8. |
| 17 | **void setContentLength(int len)** Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header. |
| 18 | **void setContentType(String type)** Sets the content type of the response being sent to the client, if the response has not been committed yet. |
| 19 | **void setDateHeader(String name, long date)** Sets a response header with the given name and date-value. |
| 20 | **void setHeader(String name, String value)** Sets a response header with the given name and value. |
| 21 | **void setIntHeader(String name, int value)** Sets a response header with the given name and integer value. |
| 22 | **void setLocale(Locale loc)** Sets the locale of the response, if the response has not been committed yet. |
| 23 | **void setStatus(int sc)** Sets the status code for this response. |

**HTTP Header Response Example:**

Following example would use **setIntHeader()** method to set **Refresh** header to simulate a digital clock:

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>Auto Refresh Header Example</title>

</head>

<body>

<center>

<h2>Auto Refresh Header Example</h2>

<%

// Set refresh, autoload time as 5 seconds

response.setIntHeader("Refresh", 5);

// Get current time

Calendar calendar = new GregorianCalendar();

String am\_pm;

int hour = calendar.get(Calendar.HOUR);

int minute = calendar.get(Calendar.MINUTE);

int second = calendar.get(Calendar.SECOND);

if(calendar.get(Calendar.AM\_PM) == 0)

am\_pm = "AM";

else

am\_pm = "PM";

String CT = hour+":"+ minute +":"+ second +" "+ am\_pm;

out.println("Current Time is: " + CT + "\n");

%>

</center>

</body>

</html>

Now put the above code in main.jsp and try to access it. This would display current system time after every 5 seconds as follows. Just run the JSP and wait to see the result:

|  |
| --- |
| Auto Refresh Header Example  Current Time is: 9:44:50 PM |

To become more comfortable with other methods you can try few more above listed methods in the same fashion.

The format of the HTTP request and HTTP response messages are similar and will have following structure:

* An initial status line + CRLF ( Carriage Return + Line Feed ie. New Line )
* Zero or more header lines + CRLF
* A blank line ie. a CRLF
* An optional message body like file, query data or query output.

For example, a server response header looks as follows:

HTTP/1.1 200 OK

Content-Type: text/html

Header2: ...

...

HeaderN: ...

(Blank Line)

<!doctype ...>

<html>

<head>...</head>

<body>

...

</body>

</html>

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example).

Following is a list of HTTP status codes and associated messages that might be returned from the Web Server:

|  |  |  |
| --- | --- | --- |
| **Code:** | **Message:** | **Description:** |
| 100 | Continue | Only a part of the request has been received by the server, but as long as it has not been rejected, the client should continue with the request |
| 101 | Switching Protocols | The server switches protocol. |
| 200 | OK | The request is OK |
| 201 | Created | The request is complete, and a new resource is created |
| 202 | Accepted | The request is accepted for processing, but the processing is not complete. |
| 203 | Non-authoritative Information |  |
| 204 | No Content |  |
| 205 | Reset Content |  |
| 206 | Partial Content |  |
| 300 | Multiple Choices | A link list. The user can select a link and go to that location. Maximum five addresses |
| 301 | Moved Permanently | The requested page has moved to a new url |
| 302 | Found | The requested page has moved temporarily to a new url |
| 303 | See Other | The requested page can be found under a different url |
| 304 | Not Modified |  |
| 305 | Use Proxy |  |
| 306 | *Unused* | This code was used in a previous version. It is no longer used, but the code is reserved. |
| 307 | Temporary Redirect | The requested page has moved temporarily to a new url. |
| 400 | Bad Request | The server did not understand the request |
| 401 | Unauthorized | The requested page needs a username and a password |
| 402 | Payment Required | *You can not use this code yet* |
| 403 | Forbidden | Access is forbidden to the requested page |
| 404 | Not Found | The server can not find the requested page. |
| 405 | Method Not Allowed | The method specified in the request is not allowed. |
| 406 | Not Acceptable | The server can only generate a response that is not accepted by the client. |
| 407 | Proxy Authentication Required | You must authenticate with a proxy server before this request can be served. |
| 408 | Request Timeout | The request took longer than the server was prepared to wait. |
| 409 | Conflict | The request could not be completed because of a conflict. |
| 410 | Gone | The requested page is no longer available. |
| 411 | Length Required | The "Content-Length" is not defined. The server will not accept the request without it. |
| 412 | Precondition Failed | The precondition given in the request evaluated to false by the server. |
| 413 | Request Entity Too Large | The server will not accept the request, because the request entity is too large. |
| 414 | Request-url Too Long | The server will not accept the request, because the url is too long. Occurs when you convert a "post" request to a "get" request with a long query information. |
| 415 | Unsupported Media Type | The server will not accept the request, because the media type is not supported. |
| 417 | Expectation Failed |  |
| 500 | Internal Server Error | The request was not completed. The server met an unexpected condition |
| 501 | Not Implemented | The request was not completed. The server did not support the functionality required. |
| 502 | Bad Gateway | The request was not completed. The server received an invalid response from the upstream server |
| 503 | Service Unavailable | The request was not completed. The server is temporarily overloading or down. |
| 504 | Gateway Timeout | The gateway has timed out. |
| 505 | HTTP Version Not Supported | The server does not support the "http protocol" version. |

## Methods to Set HTTP Status Code:

There are following methods which can be used to set HTTP Status Code in your servlet program. These method are available with *HttpServletResponse* object.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **public void setStatus ( int statusCode )** This method sets an arbitrary status code. The setStatus method takes an int (the status code) as an argument. If your response includes a special status code and a document, be sure to call setStatus before actually returning any of the content with the *PrintWriter*. |
| 2 | **public void sendRedirect(String url)** This method generates a 302 response along with a *Location* header giving the URL of the new document. |
| 3 | **public void sendError(int code, String message)** This method sends a status code (usually 404) along with a short message that is automatically formatted inside an HTML document and sent to the client. |

## HTTP Status Code Example:

Following is the example which would send 407 error code to the client browser and browser would show you "Need authentication!!!" message.

<html>

<head>

<title>Setting HTTP Status Code</title>

</head>

<body>

<%

// Set error code and reason.

response.sendError(407, "Need authentication!!!" );

%>

</body>

</html>

Now calling the above JSP would display following result:

|  |
| --- |
| HTTP Status 407 - Need authentication!!! **type** Status report  **message** Need authentication!!!  **description** The client must first authenticate itself with the proxy (Need authentication!!!). APACHE TOMCAT/5.5.29 |

To become more comfortable with HTTP status codes, try to set different status codes and their description.

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your backend program. The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

## GET method:

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows:

http://www.test.com/hello?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server.

The GET method has size limitation: only 1024 characters can be in a request string.

This information is passed using QUERY\_STRING header and will be accessible through QUERY\_STRING environment variable which can be handled using getQueryString() and getParameter() methods of request object.

## POST method:

A generally more reliable method of passing information to a backend program is the POST method.

This method packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes to the backend program in the form of the standard input which you can parse and use for your processing.

JSP handles this type of requests using getParameter() method to read simple parameters and getInputStream() method to read binary data stream coming from the client.

## Reading Form Data using JSP:

JSP handles form data parsing automatically using the following methods depending on the situation:

* **getParameter():** You call request.getParameter() method to get the value of a form parameter.
* **getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
* **getParameterNames():** Call this method if you want a complete list of all parameters in the current request.
* **getInputStream():** Call this method to read binary data stream coming from the client.

## GET Method Example Using URL:

Here is a simple URL which will pass two values to HelloForm program using GET method.

**http://localhost:8080/main.jsp?first\_name=ZARA&last\_name=ALI**

Below is **main.jsp** JSP program to handle input given by web browser. We are going to use**getParameter()** method which makes it very easy to access passed information:

<html>

<head>

<title>Using GET Method to Read Form Data</title>

</head>

<body>

<center>

<h1>Using GET Method to Read Form Data</h1>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first\_name")%>

</p></li>

<li><p><b>Last Name:</b>

<%= request.getParameter("last\_name")%>

</p></li>

</ul>

</body>

</html>

Now type *http://localhost:8080/main.jsp?first\_name=ZARA&last\_name=ALI* in your browser's Location:box. This would generate following result:

|  |
| --- |
| Using GET Method to Read Form Data  * **First Name**: ZARA * **Last Name**: ALI |

## GET Method Example Using Form:

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same JSP main.jsp to handle this input.

<html>

<body>

<form action="main.jsp" method="GET">

First Name: <input type="text" name="first\_name">

<br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

</body>

</html>

Keep this HTML in a file Hello.htm and put it in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access *http://localhost:8080/Hello.htm*, here is the actual output of the above form.

Top of Form

FirstName:    
 Last Name: 

Try to enter First Name and Last Name and then click submit button to see the result on your local machine where tomcat is running. Based on the input provided, it will generate similar result as mentioned in the above example.

## POST Method Example Using Form:

Let us do little modification in the above JSP to handle GET as well as POST methods. Below is**main.jsp** JSP program to handle input given by web browser using GET or POST methods.

Infact there is no change in above JSP because only way of passing parameters is changed and no binary data is being passed to the JSP program. File handling related concepts would be explained in separate chapter where we need to read binary data stream.

<html>

<head>

<title>Using GET and POST Method to Read Form Data</title>

</head>

<body>

<center>

<h1>Using GET Method to Read Form Data</h1>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first\_name")%>

</p></li>

<li><p><b>Last Name:</b>

<%= request.getParameter("last\_name")%>

</p></li>

</ul>

</body>

</html>

Following is the content of Hello.htm file:

<html>

<body>

<form action="main.jsp" method="POST">

First Name: <input type="text" name="first\_name">

<br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

</body>

</html>

Now let us keep main.jsp and hello.htm in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access *http://localhost:8080/Hello.htm*, below is the actual output of the above form.

Top of Form

FirstName:    
Last Name: 

Bottom of Form

Try to enter First and Last Name and then click submit button to see the result on your local machine where tomcat is running.

Based on the input provided, it would generate similar result as mentioned in the above examples.

## Passing Checkbox Data to JSP Program:

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code, CheckBox.htm, for a form with two checkboxes

<html>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="maths" checked="checked" /> Maths

<input type="checkbox" name="physics" /> Physics

<input type="checkbox" name="chemistry" checked="checked" />

Chemistry

<input type="submit" value="Select Subject" />

</form>

</body>

</html>

The result of this code is the following form

Top of Form

 Maths  Physics  Chemistry

Bottom of Form

Below is main.jsp JSP program to handle input given by web browser for checkbox button.

<html>

<head>

<title>Reading Checkbox Data</title>

</head>

<body>

<center>

<h1>Reading Checkbox Data</h1>

<ul>

<li><p><b>Maths Flag:</b>

<%= request.getParameter("maths")%>

</p></li>

<li><p><b>Physics Flag:</b>

<%= request.getParameter("physics")%>

</p></li>

<li><p><b>Chemistry Flag:</b>

<%= request.getParameter("chemistry")%>

</p></li>

</ul>

</body>

</html>

For the above example, it would display following result:

|  |
| --- |
| Reading Checkbox Data  * **Maths Flag :**: on * **Physics Flag:**: null * **Chemistry Flag:**: on |

## Reading All Form Parameters:

Following is the generic example which uses **getParameterNames()** method of HttpServletRequest to read all the available form parameters. This method returns an Enumeration that contains the parameter names in an unspecified order.

Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using*hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>HTTP Header Request Example</title>

</head>

<body>

<center>

<h2>HTTP Header Request Example</h2>

<table width="100%" border="1" align="center">

<tr bgcolor="#949494">

<th>Param Name</th><th>Param Value(s)</th>

</tr>

<%

Enumeration paramNames = request.getParameterNames();

while(paramNames.hasMoreElements()) {

String paramName = (String)paramNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getHeader(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

%>

</table>

</center>

</body>

</html>

Following is the content of Hello.htm:

<html>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="maths" checked="checked" /> Maths

<input type="checkbox" name="physics" /> Physics

<input type="checkbox" name="chemistry" checked="checked" /> Chem

<input type="submit" value="Select Subject" />

</form>

</body>

</html>

Now try calling JSP using above Hello.htm, this would generate a result something like as below based on the provided input:

# Reading All Form Parameters

|  |  |
| --- | --- |
| **Param Name** | **Param Value(s)** |
| Maths | on |
| chemistry | on |

You can try above JSP to read any other form's data which is having other objects like text box, radio button or drop down box etc.

Servlet and JSP Filters are Java classes that can be used in Servlet and JSP Programming for the following purposes:

* To intercept requests from a client before they access a resource at back end.
* To manipulate responses from server before they are sent back to the client.

There are various types of filters suggested by the specifications:

* Authentication Filters.
* Data compression Filters
* Encryption Filters
* Filters that trigger resource access events.
* Image Conversion Filters .
* Logging and Auditing Filters.
* MIME-TYPE Chain Filters.
* Tokenizing Filters .

Filters are deployed in the deployment descriptor file **web.xml**and then map to either servlet or JSP names or URL patterns in your application's deployment descriptor. The deployment descriptor file web.xml can be found in *<Tomcat-installation-directory>\conf* directory.

When the JSP container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

**Servlet Filter Methods:**

A filter is simply a Java class that implements the javax.servlet.Filter interface. The javax.servlet.Filter interface defines three methods:

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **public void doFilter (ServletRequest, ServletResponse, FilterChain)** This method is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. |
| 2 | **public void init(FilterConfig filterConfig)** This method is called by the web container to indicate to a filter that it is being placed into service. |
| 3 | **public void destroy()** This method is called by the web container to indicate to a filter that it is being taken out of service. |

**JSP Filter Example:**

Following is the JSP Filter Example that would print the clients IP address and current date time each time it would access any JSP file. This example would give you basic understanding of JSP Filter, but you can write more sophisticated filter applications using the same concept:

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Implements Filter class

public class LogFilter implements Filter {

public void init(FilterConfig config)

throws ServletException{

// Get init parameter

String testParam = config.getInitParameter("test-param");

//Print the init parameter

System.out.println("Test Param: " + testParam);

}

public void doFilter(ServletRequest request,

ServletResponse response,

FilterChain chain)

throws java.io.IOException, ServletException {

// Get the IP address of client machine.

String ipAddress = request.getRemoteAddr();

// Log the IP address and current timestamp.

System.out.println("IP "+ ipAddress + ", Time "

+ new Date().toString());

// Pass request back down the filter chain

chain.doFilter(request,response);

}

public void destroy( ){

/\* Called before the Filter instance is removed

from service by the web container\*/

}

}

Compile **LogFilter.java** in usual way and put your LogFilter.class class file in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes.

**JSP Filter Mapping in Web.xml:**

Filters are defined and then mapped to a URL or JSP file name, in much the same way as Servlet is defined and then mapped to a URL pattern in web.xml file. Create the following entry for filter tag in the deployment descriptor file **web.xml**

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>LogFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>LogFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

The above filter would apply to all the servlets and JSP because we specified **/\*** in our configuration. You can specify a particular servlet or JSP path if you want to apply filter on few servlets or JSP only.

Now try to call any servlet or JSP in usual way and you would see generated log in you web server log. You can use Log4J logger to log above log in a separate file.

**Using Multiple Filters:**

Your web application may define several different filters with a specific purpose. Consider, you define two filters *AuthenFilter* and *LogFilter*. Rest of the process would remain as explained above except you need to create a different mapping as mentioned below:

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>LogFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter>

<filter-name>AuthenFilter</filter-name>

<filter-class>AuthenFilter</filter-class>

<init-param>

<param-name>test-param</param-name>

<param-value>Initialization Paramter</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>LogFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<filter-mapping>

<filter-name>AuthenFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

**Filters Application Order:**

The order of filter-mapping elements in web.xml determines the order in which the web container applies the filter to the servlet or JSP. To reverse the order of the filter, you just need to reverse the filter-mapping elements in the web.xml file.

For example, above example would apply LogFilter first and then it would apply AuthenFilter to any servlet or JSP but the following example would reverse the order:

<filter-mapping>

<filter-name>AuthenFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<filter-mapping>

<filter-name>LogFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying returning users:

* Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
* Browser stores this information on local machine for future use.
* When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them using JSP programs.

**The Anatomy of a Cookie:**

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this:

HTTP/1.1 200 OK

Date: Fri, 04 Feb 2000 21:03:38 GMT

Server: Apache/1.3.9 (UNIX) PHP/4.0b3

Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;

path=/; domain=tutorialspoint.com

Connection: close

Content-Type: text/html

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this:

GET / HTTP/1.0

Connection: Keep-Alive

User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)

Host: zink.demon.co.uk:1126

Accept: image/gif, \*/\*

Accept-Encoding: gzip

Accept-Language: en

Accept-Charset: iso-8859-1,\*,utf-8

Cookie: name=xyz

A JSP script will then have access to the cookies through the request method *request.getCookies()*which returns an array of *Cookie* objects.

**Servlet Cookies Methods:**

Following is the list of useful methods associated with Cookie object which you can use while manipulating cookies in JSP:

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **public void setDomain(String pattern)** This method sets the domain to which cookie applies, for example tutorialspoint.com. |
| 2 | **public String getDomain()** This method gets the domain to which cookie applies, for example tutorialspoint.com. |
| 3 | **public void setMaxAge(int expiry)** This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session. |
| 4 | **public int getMaxAge()** This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown. |
| 5 | **public String getName()** This method returns the name of the cookie. The name cannot be changed after creation. |
| 6 | **public void setValue(String newValue)** This method sets the value associated with the cookie. |
| 7 | **public String getValue()** This method gets the value associated with the cookie. |
| 8 | **public void setPath(String uri)** This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories. |
| 9 | **public String getPath()** This method gets the path to which this cookie applies. |
| 10 | **public void setSecure(boolean flag)** This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections. |
| 11 | **public void setComment(String purpose)** This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user. |
| 12 | **public String getComment()** This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment. |

**Setting Cookies with JSP:**

**Setting cookies with JSP involves three steps:**

**(1) Creating a Cookie object:** You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

Cookie cookie = new Cookie("key","value");

Keep in mind, neither the name nor the value should contain white space or any of the following characters:

[ ] ( ) = , " / ? @ : ;

**(2) Setting the maximum age:** You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

cookie.setMaxAge(60\*60\*24);

**(3) Sending the Cookie into the HTTP response headers:** You use **response.addCookie** to add cookies in the HTTP response header as follows:

response.addCookie(cookie);

Example:

Let us modify our [Form Example](http://www.tutorialspoint.com/jsp/jsp-form-processing.htm) to set the cookies for first and last name.

<%

// Create cookies for first and last names.

Cookie firstName = new Cookie("first\_name",

request.getParameter("first\_name"));

Cookie lastName = new Cookie("last\_name",

request.getParameter("last\_name"));

// Set expiry date after 24 Hrs for both the cookies.

firstName.setMaxAge(60\*60\*24);

lastName.setMaxAge(60\*60\*24);

// Add both the cookies in the response header.

response.addCookie( firstName );

response.addCookie( lastName );

%>

<html>

<head>

<title>Setting Cookies</title>

</head>

<body>

<center>

<h1>Setting Cookies</h1>

</center>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first\_name")%>

</p></li>

<li><p><b>Last Name:</b>

<%= request.getParameter("last\_name")%>

</p></li>

</ul>

</body>

</html>

Let us put above code in main.jsp file and use it in the following HTML page:

<html>

<body>

<form action="main.jsp" method="GET">

First Name: <input type="text" name="first\_name">

<br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

</body>

</html>

Keep above HTML content in a file hello.jsp and put hello.jsp and main.jsp in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access *http://localhost:8080/hello.jsp*, here is the actual output of the above form.

Top of Form

FirstName:    
Last Name: 

Bottom of Form

Try to enter First Name and Last Name and then click submit button. This would display first name and last name on your screen and same time it would set two cookies firstName and lastName which would be passed back to the server when next time you would press Submit button.

Next section would explain you how you would access these cookies back in your web application.

**Reading Cookies with JSP:**

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the**getCookies( )** method of *HttpServletRequest*. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.

Example:

Let us read cookies which we have set in previous example:

<html>

<head>

<title>Reading Cookies</title>

</head>

<body>

<center>

<h1>Reading Cookies</h1>

</center>

<%

Cookie cookie = null;

Cookie[] cookies = null;

// Get an array of Cookies associated with this domain

cookies = request.getCookies();

if( cookies != null ){

out.println("<h2> Found Cookies Name and Value</h2>");

for (int i = 0; i < cookies.length; i++){

cookie = cookies[i];

out.print("Name : " + cookie.getName( ) + ", ");

out.print("Value: " + cookie.getValue( )+" <br/>");

}

}else{

out.println("<h2>No cookies founds</h2>");

}

%>

</body>

</html>

Now let us put above code in main.jsp file and try to access it. If you would have set first\_name cookie as "John" and last\_name cookie as "Player" then running *http://localhost:8080/main.jsp* would display the following result:

|  |
| --- |
| Found Cookies Name and Value  Name : first\_name, Value: John  Name : last\_name, Value: Player |

**Delete Cookies with JSP:**

To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:

* Read an already existing cookie and store it in Cookie object.
* Set cookie age as zero using **setMaxAge()** method to delete an existing cookie.
* Add this cookie back into response header.

Example:

Following example would delete and existing cookie named "first\_name" and when you would run main.jsp JSP next time it would return null value for first\_name.

<html>

<head>

<title>Reading Cookies</title>

</head>

<body>

<center>

<h1>Reading Cookies</h1>

</center>

<%

Cookie cookie = null;

Cookie[] cookies = null;

// Get an array of Cookies associated with this domain

cookies = request.getCookies();

if( cookies != null ){

out.println("<h2> Found Cookies Name and Value</h2>");

for (int i = 0; i < cookies.length; i++){

cookie = cookies[i];

if((cookie.getName( )).compareTo("first\_name") == 0 ){

cookie.setMaxAge(0);

response.addCookie(cookie);

out.print("Deleted cookie: " +

cookie.getName( ) + "<br/>");

}

out.print("Name : " + cookie.getName( ) + ", ");

out.print("Value: " + cookie.getValue( )+" <br/>");

}

}else{

out.println(

"<h2>No cookies founds</h2>");

}

%>

</body>

</html>

Now let us put above code in main.jsp file and try to access it. It would display the following result:

|  |
| --- |
| Cookies Name and Value  Deletedcookie:first\_name Name:first\_name,Value:John Name : last\_name, Value: Player |

Now try to run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows:

|  |
| --- |
| Found Cookies Name and Value  Name : last\_name, Value: Player |

You can delete your cookies in Internet Explorer manually. Start at the Tools menu and select Internet Options. To delete all cookies, press Delete Cookies.

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Still there are following three ways to maintain session between web client and web server:

## Cookies:

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

## Hidden Form Fields:

A web server can send a hidden HTML form field along with a unique session ID as follows:

<input type="hidden" name="sessionid" value="12345">

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session\_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

## URL Rewriting:

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.

For example, with http://tutorialspoint.com/file.htm;sessionid=12345, the session identifier is attached as sessionid=12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

## The session Object:

Apart from the above mentioned three ways, JSP makes use of servlet provided HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

<%@ page session="false" %>

The JSP engine exposes the HttpSession object to the JSP author through the implicit **session** object. Since **session** object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().

Here is a summary of important methods available through session object:

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name. |
| 2 | **public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session. |
| 3 | **public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. |
| 4 | **public String getId()** This method returns a string containing the unique identifier assigned to this session. |
| 5 | **public long getLastAccessedTime()** This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT. |
| 6 | **public int getMaxInactiveInterval()** This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. |
| 7 | **public void invalidate()** This method invalidates this session and unbinds any objects bound to it. |
| 8 | **public boolean isNew(** This method returns true if the client does not yet know about the session or if the client chooses not to join the session. |
| 9 | **public void removeAttribute(String name)** This method removes the object bound with the specified name from this session. |
| 10 | **public void setAttribute(String name, Object value)** This method binds an object to this session, using the name specified. |
| 11 | **public void setMaxInactiveInterval(int interval)** This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session. |

## Session Tracking Example:

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

<%@ page import="java.io.\*,java.util.\*" %>

<%

// Get session creation time.

Date createTime = new Date(session.getCreationTime());

// Get last access time of this web page.

Date lastAccessTime = new Date(session.getLastAccessedTime());

String title = "Welcome Back to my website";

Integer visitCount = new Integer(0);

String visitCountKey = new String("visitCount");

String userIDKey = new String("userID");

String userID = new String("ABCD");

// Check if this is new comer on your web page.

if (session.isNew()){

title = "Welcome to my website";

session.setAttribute(userIDKey, userID);

session.setAttribute(visitCountKey, visitCount);

}

visitCount = (Integer)session.getAttribute(visitCountKey);

visitCount = visitCount + 1;

userID = (String)session.getAttribute(userIDKey);

session.setAttribute(visitCountKey, visitCount);

%>

<html>

<head>

<title>Session Tracking</title>

</head>

<body>

<center>

<h1>Session Tracking</h1>

</center>

<table border="1" align="center">

<tr bgcolor="#949494">

<th>Session info</th>

<th>Value</th>

</tr>

<tr>

<td>id</td>

<td><% out.print( session.getId()); %></td>

</tr>

<tr>

<td>Creation Time</td>

<td><% out.print(createTime); %></td>

</tr>

<tr>

<td>Time of Last Access</td>

<td><% out.print(lastAccessTime); %></td>

</tr>

<tr>

<td>User ID</td>

<td><% out.print(userID); %></td>

</tr>

<tr>

<td>Number of visits</td>

<td><% out.print(visitCount); %></td>

</tr>

</table>

</body>

</html>

Now put above code in main.jsp and try to access *http://localhost:8080/main.jsp*. It would display the following result when you would run for the first time:

# Welcome to my website

## Session Infomation

|  |  |
| --- | --- |
| **Session info** | **value** |
| Id | 0AE3EC93FF44E3C525B4351B77ABB2D5 |
| Creation Time | Tue Jun 08 17:26:40 GMT+04:00 2010 |
| Time of Last Access | Tue Jun 08 17:26:40 GMT+04:00 2010 |
| User ID | ABCD |
| Number of visits | 0 |

Now try to run the same JSP for second time, it would display following result.

# Welcome Back to my website

## Session Infomation

|  |  |
| --- | --- |
| **info type** | **value** |
| Id | 0AE3EC93FF44E3C525B4351B77ABB2D5 |
| Creation Time | Tue Jun 08 17:26:40 GMT+04:00 2010 |
| Time of Last Access | Tue Jun 08 17:26:40 GMT+04:00 2010 |
| User ID | ABCD |
| Number of visits | 1 |

## Deleting Session Data:

When you are done with a user's session data, you have several options:

* **Remove a particular attribute:** You can call *public void removeAttribute(String name)* method to delete the value associated with a particular key.
* **Delete the whole session:** You can call *public void invalidate()* method to discard an entire session.
* **Setting Session timeout:** You can call *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.
* **Log the user out:** The servers that support servlets 2.4, you can call **logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.
* **web.xml Configuration:** If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

<session-config>

<session-timeout>15</session-timeout>

</session-config>

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat.

The getMaxInactiveInterval( ) method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, getMaxInactiveInterval( ) returns 900.

A JSP can be used with an HTML form tag to allow users to upload files to the server. An uploaded file could be a text file or binary or image file or any document.

**Creating a File Upload Form:**

The following HTM code below creates an uploader form. Following are the important points to be noted down:

* The form **method** attribute should be set to **POST** method and GET method can not be used.
* The form **enctype** attribute should be set to **multipart/form-data**.
* The form **action** attribute should be set to a JSP file which would handle file uploading at backend server. Following example is using **uploadFile.jsp** program file to upload file.
* To upload a single file you should use a single <input .../> tag with attribute type="file". To allow multiple files uploading, include more than one input tags with different values for the name attribute. The browser associates a Browse button with each of them.

<html>

<head>

<title>File Uploading Form</title>

</head>

<body>

<h3>File Upload:</h3>

Select a file to upload: <br />

<form action="UploadServlet" method="post"

enctype="multipart/form-data">

<input type="file" name="file" size="50" />

<br />

<input type="submit" value="Upload File" />

</form>

</body>

</html>

This will display following result which would allow to select a file from local PC and when user would click at "Upload File", form would be submitted along with the selected file:

**File Upload:**

Selectafiletoupload:

NOTE: Above form is just dummy form and would not work, you should try above code at your machine to make it work.

**Writing Backend JSP Script:**

First let us define a location where uploaded files would be stored. You can hard code this in your program or this directory name could also be added using an external configuration such as a **context-param** element in web.xml as follows:

<web-app>

....

<context-param>

<description>Location to store uploaded file</description>

<param-name>file-upload</param-name>

<param-value>

c:\apache-tomcat-5.5.29\webapps\data\

</param-value>

</context-param>

....

</web-app>

Following is the source code for UploadFile.jsp which can handle multiple file uploading at a time. Before proceeding you have make sure the followings:

* Following example depends on FileUpload, so make sure you have the latest version of**commons-fileupload.x.x.jar** file in your classpath. You can download it from<http://commons.apache.org/fileupload/>.
* FileUpload depends on Commons IO, so make sure you have the latest version of **commons-io-x.x.jar** file in your classpath. You can download it from <http://commons.apache.org/io/>.
* While testing following example, you should upload a file which has less size than *maxFileSize*otherwise file would not be uploaded.
* Make sure you have created directories c:\temp and c:\apache-tomcat-5.5.29\webapps\data well in advance.

<%@ page import="java.io.\*,java.util.\*, javax.servlet.\*" %>

<%@ page import="javax.servlet.http.\*" %>

<%@ page import="org.apache.commons.fileupload.\*" %>

<%@ page import="org.apache.commons.fileupload.disk.\*" %>

<%@ page import="org.apache.commons.fileupload.servlet.\*" %>

<%@ page import="org.apache.commons.io.output.\*" %>

<%

File file ;

int maxFileSize = 5000 \* 1024;

int maxMemSize = 5000 \* 1024;

ServletContext context = pageContext.getServletContext();

String filePath = context.getInitParameter("file-upload");

// Verify the content type

String contentType = request.getContentType();

if ((contentType.indexOf("multipart/form-data") >= 0)) {

DiskFileItemFactory factory = new DiskFileItemFactory();

// maximum size that will be stored in memory

factory.setSizeThreshold(maxMemSize);

// Location to save data that is larger than maxMemSize.

factory.setRepository(new File("c:\\temp"));

// Create a new file upload handler

ServletFileUpload upload = new ServletFileUpload(factory);

// maximum file size to be uploaded.

upload.setSizeMax( maxFileSize );

try{

// Parse the request to get file items.

List fileItems = upload.parseRequest(request);

// Process the uploaded file items

Iterator i = fileItems.iterator();

out.println("<html>");

out.println("<head>");

out.println("<title>JSP File upload</title>");

out.println("</head>");

out.println("<body>");

while ( i.hasNext () )

{

FileItem fi = (FileItem)i.next();

if ( !fi.isFormField () )

{

// Get the uploaded file parameters

String fieldName = fi.getFieldName();

String fileName = fi.getName();

boolean isInMemory = fi.isInMemory();

long sizeInBytes = fi.getSize();

// Write the file

if( fileName.lastIndexOf("\\") >= 0 ){

file = new File( filePath +

fileName.substring( fileName.lastIndexOf("\\"))) ;

}else{

file = new File( filePath +

fileName.substring(fileName.lastIndexOf("\\")+1)) ;

}

fi.write( file ) ;

out.println("Uploaded Filename: " + filePath +

fileName + "<br>");

}

}

out.println("</body>");

out.println("</html>");

}catch(Exception ex) {

System.out.println(ex);

}

}else{

out.println("<html>");

out.println("<head>");

out.println("<title>Servlet upload</title>");

out.println("</head>");

out.println("<body>");

out.println("<p>No file uploaded</p>");

out.println("</body>");

out.println("</html>");

}

%>

Now try to upload files using the HTML form which you created above. When you would try http://localhost:8080/UploadFile.htm, it would display following result which would help you uploading any file from your local machine.

**File Upload:**

Selectafiletoupload:

If your JSP script works fine, your file should be uploaded in c:\apache-tomcat-5.5.29\webapps\data\ directory.

One of the most important advantages of using JSP is that you can use all the methods available in core Java. This tutorial would take you through Java provided **Date** class which is available in **java.util**package, this class encapsulates the current date and time.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

Date( )

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970

Date(long millisec)

Once you have a Date object available, you can call any of the following support methods to play with dates:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **boolean after(Date date)** Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false. |
| 2 | **boolean before(Date date)** Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false. |
| 3 | **Object clone( )** Duplicates the invoking Date object. |
| 4 | **int compareTo(Date date)** Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date. |
| 5 | **int compareTo(Object obj)** Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException. |
| 6 | **boolean equals(Object date)** Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false. |
| 7 | **long getTime( )** Returns the number of milliseconds that have elapsed since January 1, 1970. |
| 8 | **int hashCode( )** Returns a hash code for the invoking object. |
| 9 | **void setTime(long time)** Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970 |
| 10 | **String toString( )** Converts the invoking Date object into a string and returns the result. |

## Getting Current Date & Time:

This is very easy to get current date and time in JSP program. You can use a simple Date object with*toString()* method to print current date and time as follows:

<%@ page import="java.io.\*,java.util.\*, javax.servlet.\*" %>

<html>

<head>

<title>Display Current Date & Time</title>

</head>

<body>

<center>

<h1>Display Current Date & Time</h1>

</center>

<%

Date date = new Date();

out.print( "<h2 align=\"center\">" +date.toString()+"</h2>");

%>

</body>

</html>

Now let us keep about code in CurrentDate.jsp and then call this JSP using URL http://localhost:8080/CurrentDate.jsp. This would produce following result:

|  |
| --- |
| Display Current Date & TimeMon Jun 21 21:46:49 GMT+04:00 2010 |

Try to refresh URL http://localhost:8080/CurrentDate.jsp and you would find difference in seconds everytime you would refresh.

## Date Comparison:

As I mentioned above you can use all the available Java methods in your JSP scripts. In case you need to compare two dates, following are the methods:

* You can use getTime( ) to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
* You can use the methods before( ), after( ), and equals( ). Because the 12th of the month comes before the 18th, for example, new Date(99, 2, 12).before(new Date (99, 2, 18)) returns true.
* You can use the compareTo( ) method, which is defined by the Comparable interface and implemented by Date.

## Date Formatting using SimpleDateFormat:

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting.

Let us modify above example as follows:

<%@ page import="java.io.\*,java.util.\*" %>

<%@ page import="javax.servlet.\*,java.text.\*" %>

<html>

<head>

<title>Display Current Date & Time</title>

</head>

<body>

<center>

<h1>Display Current Date & Time</h1>

</center>

<%

Date dNow = new Date( );

SimpleDateFormat ft =

new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

out.print( "<h2 align=\"center\">" + ft.format(dNow) + "</h2>");

%>

</body>

</html>

Compile above servlet once again and then call this servlet using URL http://localhost:8080/CurrentDate. This would produce following result:

|  |
| --- |
| Display Current Date & TimeMon 2010.06.21 at 10:06:44 PM GMT+04:00 |

## Simple DateFormat format codes:

To specify the time format use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| G | Era designator | AD |
| Y | Year in four digits | 2001 |
| M | Month in year | July or 07 |
| D | Day in month | 10 |
| H | Hour in A.M./P.M. (1~12) | 12 |
| H | Hour in day (0~23) | 22 |
| M | Minute in hour | 30 |
| S | Second in minute | 55 |
| S | Millisecond | 234 |
| E | Day in week | Tuesday |
| D | Day in year | 360 |
| F | Day of week in month | 2 (second Wed. in July) |
| W | Week in year | 40 |
| W | Week in month | 1 |
| A | A.M./P.M. marker | PM |
| K | Hour in day (1~24) | 24 |
| K | Hour in A.M./P.M. (0~11) | 10 |
| Z | Time zone | Eastern Standard Time |
| ' | Escape for text | Delimiter |
| " | Single quote | ` |

For a complete list of constant available methods to manipulate date, you can refer to standard Java documentation.

Page redirection is generally used when a document moves to a new location and we need to send the client to this new location or may be because of load balancing, or for simple randomization.

The simplest way of redirecting a request to another page is using method **sendRedirect()** of response object. Following is the signature of this method:

public void response.sendRedirect(String location)

throws IOException

This method sends back the response to the browser along with the status code and new page location. You can also use setStatus() and setHeader() methods together to achieve the same redirection:

....

String site = "http://www.newpage.com" ;

response.setStatus(response.SC\_MOVED\_TEMPORARILY);

response.setHeader("Location", site);

....

## Example:

This example shows how a JSP performs page redirection to an another location:

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>Page Redirection</title>

</head>

<body>

<center>

<h1>Page Redirection</h1>

</center>

<%

// New location to be redirected

String site = new String("http://www.photofuntoos.com");

response.setStatus(response.SC\_MOVED\_TEMPORARILY);

response.setHeader("Location", site);

%>

</body>

</html>

Now let us put above code in PageRedirect.jsp and call this JSP using URL http://localhost:8080/PageRedirect.jsp. This would take you given URL http://www.photofuntoos.com

A hit counter tells you about the number of visits on a particular page of your web site. Usually you attach a hit counter with your index.jsp page assuming people first land on your home page.

To implement a hit counter you can make use of Application Implicit object and associated methods getAttribute() and setAttribute().

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method.

Following is the syntax to set a variable at application level:

application.setAttribute(String Key, Object Value);

You can use above method to set a hit counter variable and to reset the same variable. Following is the method to read the variable set by previous method:

application.getAttribute(String Key);

Every time a use access your page, you can read current value of hit counter and increase it by one and again set it for future use.

Example:

This example shows how you can use JSP to count total number of hits on a particular page. If you want to count total number of hits of your website then you would have to include same code in all the JSP pages.

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>Applcation object in JSP</title>

</head>

<body>

<%

Integer hitsCount =

(Integer)application.getAttribute("hitCounter");

if( hitsCount ==null || hitsCount == 0 ){

/\* First visit \*/

out.println("Welcome to my website!");

hitsCount = 1;

}else{

/\* return visit \*/

out.println("Welcome back to my website!");

hitsCount += 1;

}

application.setAttribute("hitCounter", hitsCount);

%>

<center>

<p>Total number of visits: <%= hitsCount%></p>

</center>

</body>

</html>

Now let us put above code in main.jsp and call this JSP using URL http://localhost:8080/main.jsp. This would display hit counter value which would increase every time when you refresh the page. You can try to access the page using different browsers and you will find that hit counter will keep increasing with every hit and would display result something as follows:

|  |
| --- |
| Welcome back to my website!  Total number of visits: 12 |

**Hit Counter Resets:**

What about if you re-start your application ie. web server, this will reset your application variable and your counter will reset to zero. To avoid this loss, you can implement your counter in professional way which is as follows:

* Define a database table with a single count, let us say hitcount. Assign a zero value to it.
* With every hit, read the table to get the value of hitcount.
* Increase the value of hitcount by one and update the table with new value.
* Display new value of hitcount as total page hit counts.
* If you want to count hits for all the pages, implement above logic for all the pages.

Consider a webpage which is displaying live game score or stock market status or currency exchange ration. For all such type of pages, you would need to refresh your web page regularly using refresh or reload button with your browser.

JSP makes this job easy by providing you a mechanism where you can make a webpage in such a way that it would refresh automatically after a given interval.

The simplest way of refreshing a web page is using method **setIntHeader()** of response object. Following is the signature of this method:

public void setIntHeader(String header, int headerValue)

This method sends back header "Refresh" to the browser along with an integer value which indicates time interval in seconds.

**Auto Page Refresh Example:**

Following example would use **setIntHeader()** method to set **Refresh** header to simulate a digital clock:

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>Auto Refresh Header Example</title>

</head>

<body>

<center>

<h2>Auto Refresh Header Example</h2>

<%

// Set refresh, autoload time as 5 seconds

response.setIntHeader("Refresh", 5);

// Get current time

Calendar calendar = new GregorianCalendar();

String am\_pm;

int hour = calendar.get(Calendar.HOUR);

int minute = calendar.get(Calendar.MINUTE);

int second = calendar.get(Calendar.SECOND);

if(calendar.get(Calendar.AM\_PM) == 0)

am\_pm = "AM";

else

am\_pm = "PM";

String CT = hour+":"+ minute +":"+ second +" "+ am\_pm;

out.println("Crrent Time: " + CT + "\n");

%>

</center>

</body>

</html>

Now put the above code in main.jsp and try to access it. This would display current system time after every 5 seconds as follows. Just run the JSP and wait to see the result:

|  |
| --- |
| Auto Refresh Header Example  Current Time is: 9:44:50 PM |

To become more comfortable with other methods you can try few more above listed methods in the same fashion.

To send an email using a JSP is simple enough but to start with you should have **JavaMail API** and**Java Activation Framework (JAF)** installed on your machine.

* You can download latest version of [JavaMail (Version 1.2)](http://java.sun.com/products/javamail/) from Java's standard website.
* You can download latest version of JavaBeans Activation Framework [JAF (Version 1.0.2)](http://www.oracle.com/technetwork/java/javase/jaf-136260.html) from Java's standard website.

Download and unzip these files, in the newly created top level directories you will find a number of jar files for both the applications. You need to add **mail.jar** and **activation.jar** files in your CLASSPATH.

## Send a Simple Email:

Here is an example to send a simple email from your machine. Here it is assumed that your **localhost**is connected to the internet and capable enough to send an email. Same time make sure all the jar files from Java Email API package and JAF package ara available in CLASSPATH.

<%@ page import="java.io.\*,java.util.\*,javax.mail.\*"%>

<%@ page import="javax.mail.internet.\*,javax.activation.\*"%>

<%@ page import="javax.servlet.http.\*,javax.servlet.\*" %>

<%

String result;

// Recipient's email ID needs to be mentioned.

String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned

String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost

String host = "localhost";

// Get system properties object

Properties properties = System.getProperties();

// Setup mail server

properties.setProperty("mail.smtp.host", host);

// Get the default Session object.

Session mailSession = Session.getDefaultInstance(properties);

try{

// Create a default MimeMessage object.

MimeMessage message = new MimeMessage(mailSession);

// Set From: header field of the header.

message.setFrom(new InternetAddress(from));

// Set To: header field of the header.

message.addRecipient(Message.RecipientType.TO,

new InternetAddress(to));

// Set Subject: header field

message.setSubject("This is the Subject Line!");

// Now set the actual message

message.setText("This is actual message");

// Send message

Transport.send(message);

result = "Sent message successfully....";

}catch (MessagingException mex) {

mex.printStackTrace();

result = "Error: unable to send message....";

}

%>

<html>

<head>

<title>Send Email using JSP</title>

</head>

<body>

<center>

<h1>Send Email using JSP</h1>

</center>

<p align="center">

<%

out.println("Result: " + result + "\n");

%>

</p>

</body>

</html>

Now let us put above code in SendEmail.jsp file and call this JSP using URL http://localhost:8080/SendEmail.jsp which would send an email to given email ID *abcd@gmail.com*and would display following response:

|  |
| --- |
| Send Email using JSP Result: Sent message successfully.... |

If you want to send an email to multiple recipients then following methods would be used to specify multiple email IDs:

void addRecipients(Message.RecipientType type,

Address[] addresses)

throws MessagingException

Here is the description of the parameters:

* **type:** This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example *Message.RecipientType.TO*
* **addresses:** This is the array of email ID. You would need to use InternetAddress() method while specifying email IDs

## Send an HTML Email:

Here is an example to send an HTML email from your machine. Here it is assumed that your **localhost**is connected to the internet and capable enough to send an email. Same time make sure all the jar files from Java Email API package and JAF package ara available in CLASSPATH.

This example is very similar to previous one, except here we are using setContent() method to set content whose second argument is "text/html" to specify that the HTML content is included in the message.

Using this example, you can send as big as HTML content you like.

<%@ page import="java.io.\*,java.util.\*,javax.mail.\*"%>

<%@ page import="javax.mail.internet.\*,javax.activation.\*"%>

<%@ page import="javax.servlet.http.\*,javax.servlet.\*" %>

<%

String result;

// Recipient's email ID needs to be mentioned.

String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned

String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost

String host = "localhost";

// Get system properties object

Properties properties = System.getProperties();

// Setup mail server

properties.setProperty("mail.smtp.host", host);

// Get the default Session object.

Session mailSession = Session.getDefaultInstance(properties);

try{

// Create a default MimeMessage object.

MimeMessage message = new MimeMessage(mailSession);

// Set From: header field of the header.

message.setFrom(new InternetAddress(from));

// Set To: header field of the header.

message.addRecipient(Message.RecipientType.TO,

new InternetAddress(to));

// Set Subject: header field

message.setSubject("This is the Subject Line!");

// Send the actual HTML message, as big as you like

message.setContent("<h1>This is actual message</h1>",

"text/html" );

// Send message

Transport.send(message);

result = "Sent message successfully....";

}catch (MessagingException mex) {

mex.printStackTrace();

result = "Error: unable to send message....";

}

%>

<html>

<head>

<title>Send HTML Email using JSP</title>

</head>

<body>

<center>

<h1>Send Email using JSP</h1>

</center>

<p align="center">

<%

out.println("Result: " + result + "\n");

%>

</p>

</body>

</html>

Now try to use above JSP to send HTML message on a given email ID.

## Send Attachment in Email:

Here is an example to send an email with attachment from your machine:

<%@ page import="java.io.\*,java.util.\*,javax.mail.\*"%>

<%@ page import="javax.mail.internet.\*,javax.activation.\*"%>

<%@ page import="javax.servlet.http.\*,javax.servlet.\*" %>

<%

String result;

// Recipient's email ID needs to be mentioned.

String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned

String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost

String host = "localhost";

// Get system properties object

Properties properties = System.getProperties();

// Setup mail server

properties.setProperty("mail.smtp.host", host);

// Get the default Session object.

Session mailSession = Session.getDefaultInstance(properties);

try{

// Create a default MimeMessage object.

MimeMessage message = new MimeMessage(mailSession);

// Set From: header field of the header.

message.setFrom(new InternetAddress(from));

// Set To: header field of the header.

message.addRecipient(Message.RecipientType.TO,

new InternetAddress(to));

// Set Subject: header field

message.setSubject("This is the Subject Line!");

// Create the message part

BodyPart messageBodyPart = new MimeBodyPart();

// Fill the message

messageBodyPart.setText("This is message body");

// Create a multipar message

Multipart multipart = new MimeMultipart();

// Set text message part

multipart.addBodyPart(messageBodyPart);

// Part two is attachment

messageBodyPart = new MimeBodyPart();

String filename = "file.txt";

DataSource source = new FileDataSource(filename);

messageBodyPart.setDataHandler(new DataHandler(source));

messageBodyPart.setFileName(filename);

multipart.addBodyPart(messageBodyPart);

// Send the complete message parts

message.setContent(multipart );

// Send message

Transport.send(message);

String title = "Send Email";

result = "Sent message successfully....";

}catch (MessagingException mex) {

mex.printStackTrace();

result = "Error: unable to send message....";

}

%>

<html>

<head>

<title>Send Attachement Email using JSP</title>

</head>

<body>

<center>

<h1>Send Attachement Email using JSP</h1>

</center>

<p align="center">

<%

out.println("Result: " + result + "\n");

%>

</p>

</body>

</html>

Now try to run above JSP to send a file as an attachment along with a message on a given email ID.

## User Authentication Part:

If it is required to provide user ID and Password to the email server for authentication purpose then you can set these properties as follows:

props.setProperty("mail.user", "myuser");

props.setProperty("mail.password", "mypwd");

Rest of the email sending mechanism would remain as explained above.

## Using Forms to send email:

You can use HTML form to accept email parameters and then you can use **request** object to get all the information as follows:

String to = request.getParameter("to");

String from = request.getParameter("from");

String subject = request.getParameter("subject");

String messageText = request.getParameter("body");

Once you have all the information, you can use above mentioned programs to send email.