Unit-VI

Java Server Pages (JSP)

JavaServer Pages (**JSP**) is a Java technology that allows software developers to create dynamically-generated web sites, with HTML, XML, or other document types, in response to a Web client request. The technology allows Java code and certain pre-defined actions to be embedded into static content. Architecturally, JSP may be viewed as a high-level abstraction of servlets that is implemented as an extension of the Servlet 2.1 API. Both servlets and JSPs were originally developed at Sun Microsystems

Java Server Pages (JSP) lets you separate the dynamic part of your pages from the static HTML. You simply write the regular HTML in the normal manner, using whatever Web-page-building tools you normally use. You then enclose the code for the dynamic parts in special tags.

Usage of JSP

- JSP is widely used for developing dynamic web sites.
- JSP is used for creating database driven web applications because it provides superior server side scripting support.

Reasons for the popularity

- **Simplifies the process of development:**

  It allows programmers to insert the Java code directly into the JSP file, making the development process easier. Although JSP files are HTML files, they use special tags containing the Java source code, which provides a dynamic ability.

- **Portability:**

  The Java feature of '***write once, run anywhere***' is applicable to JSP. JSP is platform independent, making it portable across any platform and therefore, mulit-platform. It is possible for the programmer to take a JSP file and move it to another platform, JSP-Servlet engine or web server.

- **Because of Efficiency:**

  As soon as the request is received, the JSP pages gets loaded into the web servers' memory. The following transactions are then carried out within a minimal time period, handling JSP pages with efficiency.

- **Reusability:**

  JSP allow component reuse by using JavaBeans and EJBs.

- **Robust:**

  JSP offers a robust platform for web development

- **Independency of Layers:**

  There is a clear separation between presentation and implementation layers. The HTML on the web browser of the client is displayed as a presentation layer. The JSP on the server is displayed in the implementation layer. Programmers who want to work with HTML can work independently without the work of the Java developers', working in the implementation layer, being affected. Java Server Pages, or JSP, is the way to separate the look of the web page from the corresponding content. Integration of JSP with Other source like JDBC, Servlet and so on:

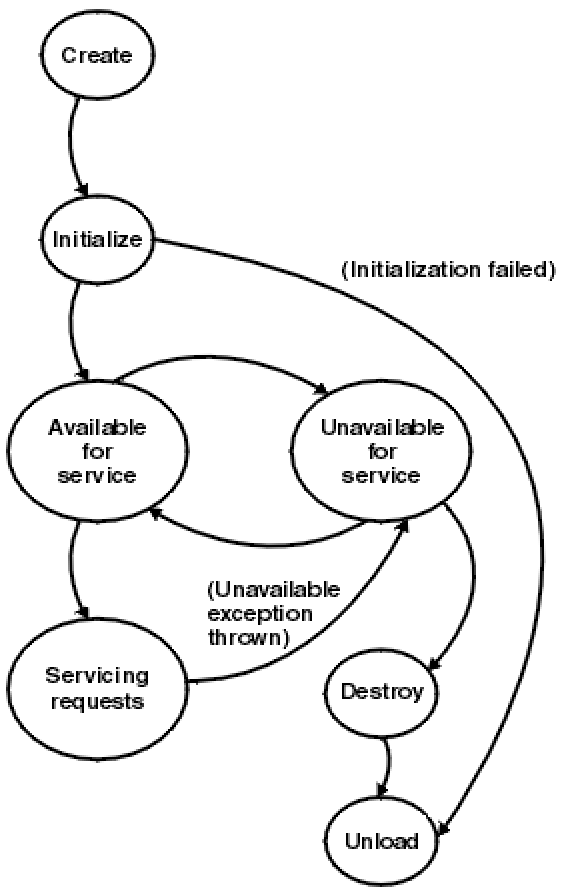
  The combination of JDBC and JSP works very well. JSP is used for generating dynamic web pages. It is essential that data in these systems be maintained efficiently and securely. The programmer must have a good database and database connectivity. This is achieved by JDBC through its excellent database connectivity in heterogeneous database system. This database system is used for the integration of heterogeneous database management systems presenting a single query interface system.

- **Simplification of Process:**

  The JSP language has a simple development and maintenance process. A JSP file that has the extension .jsp is converted into a servlet .java which is dynamically compiled, loaded and executed. Only when there is a change in a JSP file, the Conversion, compilation, and loading process is then performed.

Java Server Pages (JSP) lifecycle

JSP files are compiled into servlets. After a JSP is compiled, its lifecycle is similar to the servlet lifecycle:



JSP Environment Setup

Steps for Setting JSP Environment

This section deals with the steps for setting JSP environment in Microsoft Windows, setting the PATH and CLASSPATH, steps for downloading and installing the Tomcat web server.

The Java Server Page or JSP is very affordable as most of the software needed for it is easily available for free or at low cost.

- The Java Developer Kit, which is available for free
- The Tomcat web server if used is available for free.
- HTML editor would be needed to purchase.

How to set up the JSP Environment in Microsoft Windows:

There are slight differences between operating systems for the JSP Environment's set up in Windows but below are the basic steps.

Step- 1:

The first and foremost step before setting up the JSP environment is to verify the presence of Java Developer Kit or JDK on the programmer's system. If it is not present, the Java Developer Kit or JDK must be downloaded and installed.

This link provides the JDK download:

http://java.sun.com/j2ee/download.html

or

http://java.sun.com/javase/downloads/index.jsp

This link provides the latest Windows Platform version of Java SDK.

Step- 2:

After the download, the next step is to run the executable setup, or the exe, and follow the prompted instructions on screen.

Step- 3:

Setting the PATH and CLASSPATH:

The third step is to set the `PATH` and `CLASSPATH`. In Windows 95 or Windows 98, edit the `AUTOEXEC.BAT` file. In this file, the user has to add the directory `<JDK installation directory>\bin` to the `PATH` setting in the *autoexec.bat* file. The above sets the new `PATH`. Set the `CLASSPATH` in a similar manner by adding the path `<JDK installation directory>\lib\j2ee.jar` to the `CLASSPATH` setting in the *autoexec.bat* file. The above steps define new `PATH` and `CLASSPATH` settings.

If the system is Windows 2000 or Windows XP, edit the environment variables to set the `PATH` and `CLASSPATH` settings. The environment variables are accessed by navigating as follows:

```
Control Panel -> System -> Environment Variables
```

Repeat the same procedure to set new `PATH` and `CLASSPATH` settings.

Step- 4:

The machine must be rebooted to enable the new settings.

Step- 5:

The JSP environments can be download here

http://java.sun.com/products/jsp/index.jsp.

Step- 6:

The next major step is to download and install the Tomcat web server. This is needed if the programmer does not possess a JSP-capable web server installed on their machine. Tomcat, developed by Apache, is a free open source JSP and Servlet engine.

Steps for downloading and installing the Tomcat web server:

Step- 1:

http://jakarta.apache.org/ downloads the latest version of Tomcat.

Step- 2:

Unzip the downloaded Tomcat zip files, and any files present in the subdirectories that have been placed in a single directory, if any.

Step- 3:

Open the file autoexec.bat file to the entry:

```
SET TOMCAT_HOME=<directory>
```

Where the directory mentioned above would be the directory into which the contents of the .zip file were extracted.

If the directory created by the user was ***test***, all the contents of the .zip file were extracted then the user make an entry in *autoexec.bat* file as follows:

```
SET TOMCAT_HOME=c:\test
```

The above process is only for Windows 95 or Windows 98. If using Windows 2000 or Windows XP, the environment variable `TOMCAT_HOME` can be set as the directory in:

```
c:\test
```

Step- 4:

The next step is to change the server. Navigate to ***test\bin*** as seen in the examples above and type startup.

Step- 5:

The system reboots and opens the web browser. The user types in the address box:

http://localhost:8080/

Step- 6:

The JSP files written by the user are placed in "*webapps*" directory, inside the created directory ***test***, using the above example. Using the JSP file, *example.jsp*, this copies into *webapps/ROOT* directory. Open the web browser and type in the address box

http://localhost:8080/example.jsp

displays the executed JSP file.

Elements of Java Server Pages

The Java code in the page includes:

- **Directives** – these provide global information to the page, for example, import statements, the page for error handling or whether the page is part of a session. In the above example we set the script language to Java.
  - ***Page*** – information for that page
  - ***Include*** – files to be included verbatim
  - ***Taglib*** – the URI for a library of tags that you'll use in the page (unimplemented at the time of writing)
- **Scripting Elements** – JSP scripting elements are used to create and access objects, define methods, and manage the flow of control
  - **Declaratives** - these are for page-wide variable and method declarations.
  - **Scriptlets** - the Java code embedded in the page.
  - **Expressions** - formats the expression as a string for inclusion in the output of the page.
- **JSP Action tags** – Action tag is used to transfer the control between pages and is also used to enable the use of server side JavaBeans.
  - **Include -** include is a type of directive tag. include tag has the same concept as that of the include directive.
  - **Forward -** forward action tag is used to transfer control to a static or dynamic resource.
  - **useBean -** It allows a JSP to create an instance or receive an instance of a Java Bean

Directives

The *directive* tag gives special information about the page to JSP Engine. This changes the way JSP Engine processes the page. Using directive tag, user can import packages, define error handling pages or session information of JSP page.

General notation of *directive* tag is as follows:

There are three types of *directive* tag.

- page
- Include
- Tag Lib

Syntax and usage of *directive* tag:

page directive:

General syntax for the ***page*** directive is

```
<%@ page optional attribute ... %>;
```

There are many optional attributes available for page directive. Each of these attributes is used to give special processing information to the JSP Engine changing the way the JSP Engine processes the page. Some of the optional attributes available for page directive are:

- language
- extends
- import
- session
- buffer
- autoFlush
- isThreadSafe
- info
- errorPage
- IsErrorPage
- contentType

Syntax and usage of some of the optional attributes available for *page* directive discussed below.

language:

This attribute is used to denote the language used by a file. Language denotes the scripting language used in scriptlets, declarations, and expressions in the JSP page and any included files.

Syntax of language attribute available for page directive is

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

In the above statement ***page*** and ***language*** are keywords and one places whatever language the file uses inside " ".

For example if one wants to mention the language as java which is generally mentioned for all it is done as shown below:

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

extends:

This is used to signify the fully qualified name of the Super class of the Java class used by the JSP engine for the translated Servlet.
Syntax of extends attribute available for page directive is

```
<%@ page extends = "package.class"%>
```

In the above statement ***page*** and ***extends*** are keywords.

import:

The import attribute is used to import all the classes in a java package into the current JSP page. With this facility, the JSP page can use other java classes.
Syntax of import attribute available for page directive is

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

In the above statement `page` and `import` are keywords.

If there are many Java packages that the JSP page wants to import, the programmer can use import more than once in a JSP page or separate the Java packages with commas, as shown below:

```
<%@ page import="{package.class | package.*}, ..." %>
```

session:

The session attribute, when set to true, sets the page to make use of sessions.

NOTE: by default, the session attribute value is set to true therefore, all JSP pages have session data available. If the user sets the session attribute to false, it should be performed in this section. When the session attribution is set to false, the user cannot use the session object, or a `<jsp:useBean>` element with `scope=session` in the JSP page which, if used, would give error.
Syntax of session attribute available for page directive is

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

In the above statement `page` and `session` are keywords. And either true or false value can be setted and by default the value is true.

buffer:

If a programmer likes to control the use of buffered output for a JSP page then the buffer attribute can be made use of for achieving this.

Syntax of buffer attribute available for page directive is

```
<%@ page buffer = "none|8kb|sizekb" %>
```

In the above statement `page` and `buffer` are keywords. The size of buffer size is mentioned in kilobytes. The out object to handle output sent from the compiled JSP page to the client web browser uses this. The default value is 8kb. If a user specifies a buffer size then the output is buffered with at least the size mentioned by the user.

For example one can specify as:

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

autoFlush:

`autoFlush` attribute is used to specify whether or not to automatically flush out the output buffer when it is full. Syntax of `autoFlush` attribute available for page directive is written as:

```
<%@ page autoFlush = "true|false" %>
```

In the above example, `page` and `autoFlush` are keywords. True or false value can be set to `autoFlush` attribute, by default, its value is true . This means, the buffer will be flushed automatically when it is full. When the `autoflush` attribute is set to false, then an exception is thrown when the output buffer gets full and results in overflow. **NOTE:** The user should not to set the `autoflush` to false when the buffer attribute is set to none.

isThreadSafe:

`isThreadSafe` attribute is used to set whether the generated Servlet handles multiple requests or single requests, depending on the set value of true or false. `isThreadSafe` attribute is used to set and ensure whether thread safety is implemented in the JSP page.

Syntax of `isThreadSafe` attribute available for page directive is:

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

In the above statement, `page` and `isThreadSafe` are keywords. A true or false value can be set and, by default, the value is set to true. It implies that the JSP container can handle or send multiple concurrent client requests to the JSP page by starting a new thread. If the value of this attribute is set to false, then the JSP container sends client requests only one at a time to the JSP page.

info:

Programmers make use of `info` attribute to place the information or documentation for a page. Details such as: author, version, copyright and date are placed in this attribute. This is actually text string that is written as input text in the compiled JSP page.

Syntax of info attribute available for page directive is:

```
<%@ page info = "text" %>
```

In the above statement, `page` and `info` are keywords. The details of documentation or information are placed inside `"  "`. This is actually text string written as input text in the compiled JSP page.

For example:

```
<%@ page info = "AAIDU-JSP-Info example,2009" %>
```

The above text, `exforsys.com example, 2006`, is a text string written as text in the compiled JSP page.

errorPage:

If the programmer wants to place errors in a different page then the URL to the error page can be mentioned in this attribute as `errorPage`.

Syntax of `errorPage` attribute available for page directive is as below:

```
<%@ page errorPage = "relativeURL" %>
```

In the above statement, `page` and `errorPage` are keywords.

For example, the user specifies the attribute `errorpage` is:

```
<%@ page errorPage = "/handleerr/testerr.jsp" %>
```

isErrorPage:

`isErrorPage` attribute is used to specify whether or not a JSP page displays an error page by setting the value as true or false. By default, the value is set to false, meaning that the user cannot make use of the exception object in the JSP page. If the value is set to true, then it means that the user can make use of the exception object in the JSP page.

Syntax of `isErrorPage` attribute available for page directive is:

```
<%@ page isErrorPage="true|false" %>
```

In the above statement, `page` and `isErrorPage` are keywords. By default, the value is false.

contentType:

`contentType` attribute is used to set the mime type and character set of the JSP. The user can make use of any MIME type or character set valid for the JSP container.

Syntax of `contentType` attribute available for page directive is:

```
<%@ page contentType="mimeType [; charset=characterSet]" %>
```

In the above statement, `page` and `contentType` are keywords. The default, MIMEtype is text/html, and the default character set is ISO-8859-1. For example, the user specifies the attribute `contentType` is:

```
<%@ page contentType="text/html;charset=ISO-8859-1" %>
```

Include directive:

Include directive is a type of directive tag. If a programmer wants to include contents of a file inside another, then the Include directive is used. The file included can be either static (HTML file) or dynamic (i.e., another tag file). The include directive is used to include the contents of other file in the current tag file.

Syntax of Include directive is:

```
<%@ include file = "xxx.html/xx.tagf"%>
```

In the above statement, include and file are keywords. This includes either .html (static file) or .tagf file (dynamic file) between " " in the include directive.

For instance:

```
<%@ include file = "include/test.html"%>
```

Above example shows how to include a static resource called exforsys.html in an include directive. The above example includes the html from exforsys.html found in the include directory into the current JPS page.

For example:

The user includes a dynamic file in the include directive:

```
<%@ include file="test.tagf" %>
```

The above example shows how to include a dynamic resource called test.tagf in the include directive.

Tag Lib directive:

A user can customize actions from their tag file using the Tag Lib directive. The Tag Lib directive is a collection of custom tags.

The syntax of Tag Lib directive is:

```
<%@ taglib uri = "tag library URI" prefix = "tag Prefix" %>
```

In the above, taglib and uri are both keywords. The URI, which can be absolute or relative, uniquely identifies the tag library descriptor and is enclosed with " " in tag library URI. The tag prefix included inside " " is a string that will become the prefix to distinguish a custom action.

If the custom tag has a content body, then the format for using a custom tag in taglib directive is as follows:

```
<prefix:tagName>body</prefix:tagName>
```

If the custom tag does not have content body, then the format for using a custom tag in taglib directive is as follows:

```
<prefix:tagName/>
```

Scripting Elements

JSP Scripting elements are used to include scripting code (normally Java code) within the JSP. They allow to declare variables and methods, include arbitrary scripting code, and evaluate an expression.
The three types of scripting elements are:

- Declarations
- Scriptlets
- Expressions

Declarations

Declaration tag is used to define functions, methods and variables that will be used in Java Server Pages. A declaration is a block of Java code in a JSP that is used to define class-wide variables and methods in the generated class file. Declarations are initialized when the JSP page is initialized and have “class” scope. Anything defined in a declaration is available throughout the JSP, to other declarations, expression or code.

Notation of the *Declaration* tag is shown below:

```
<%! %>
```

At the start of *Declaration* tag one must place `<%!` Inside *Declaration* tag one can declare variables or methods. *Declaration* tag ends with the notation `%>.` Also care must be taken to place a semicolon that is `;` at the end of each code placed inside *Declaration* tag.

General syntax of Declaration Tag:

```
<%!                  //start of declaration tag
     statement1;
     statement2;     //variables or methods declaration
     ..........;
     ..........;
%>                   //end of declaration tag
```

For example:

```
<%!
     private int example = 0 ;
     private int test = 5 ;
%>
```

To add a declaration, you must use the **<%!** and **%>** sequences to enclose your declarations, as shown below.

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%!
    Date theDate = new Date();
    Date getDate()
    {
        System.out.println( "In getDate() method" );
        return theDate;
    }
%>
Hello!  The time is now <%= getDate() %>
</BODY>
</HTML>
```

The example has been created a little contrived, to show variable and method declarations.

Here we are declaring a Date variable `theDate`, and the method `getDate`. Both of these are available now in our scriptlets and expressions.

But this example no longer works! The date will be the same, no matter how often you reload the page. This is because these are declarations, and will only be evaluated once when the page is loaded! (Just as if you were creating a class and had variable initialization declared in it.)

Note: Now that you know how to do this -- it is in general not a good idea to use variables as shown here. The JSP usually will run as multiple *threads* of one single instance. Different threads would interfere with variable access, because it will be the same variable for all of them. If you do have to use variables in JSP, you should use *synchronized* access, but that hurts the performance. In general, any data you need should go either in the *session* object or the *request* object if passing data between different JSP pages. Variables you declare inside *scriptlets* are fine, e.g. <% int i = 45; %> because these are declared inside the local scope and are not shared.

Scriptlets:

JSP also allows you to write blocks of Java code inside the JSP. You do this by placing your Java code between `<%` and `%>` characters. This block of code is known as a "scriptlet". By itself, a scriptlet doesn't contribute any HTML. A scriptlet contains Java code that is executed every time the JSP is invoked. We can embed any amount of java code in the JSP Scriptlets. JSP Engine places these code in the *_jspService()* method.

General syntax of Scriptlets Tag:

```
<%                  //start of scriptlet tag
     statement1;
     statement2;
     ..........;
     ..........;
%>                  // end of scriptlet tag
```

Example

```
 <%
   for (int j = 0; j<5; j++)
   {
     out.print(j);
   }
 %>
```

In the above example, the Java code embeds inside the tag <% and %>

NOTE: at the end of the statement, a semicolon ; is appended.

Here is a modified version of our JSP from previous section, adding in a scriptlet.

```
<HTML>
<BODY>
<%
    // This is a scriptlet.  Notice that the "date"
    // variable we declare here is available in the
    // embedded expression later on.
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

If you run the above example, you will notice the output from the "`System.out.println`" on the server log. This is a convenient way to do simple debugging (some servers also have techniques of debugging the JSP in the IDE. See your server's documentation to see if it offers such a technique.)

By itself a scriptlet does not generate HTML. If a scriptlet wants to generate HTML, it can use a variable called "`out`". This variable does not need to be declared. It is already predefined for scriptlets, along with some other variables. The following example shows how the scriptlet can generate HTML output.

```
<HTML>
<BODY>
<%
    // This scriptlet declares and initializes "date"
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello!  The time is now
<%
    // This scriptlet generates HTML output
    out.println( String.valueOf( date ));
%>
</BODY>
</HTML>
```

Here, instead of using an expression, we are generating the HTML directly by printing to the "`out`" variable. The "`out`" variable is of type `javax.servlet.jsp.JspWriter`.

Another very useful pre-defined variable is "`request`". It is of type `javax.servlet.http.HttpServletRequest`

Expression

Expression tag is used to display output of any data on the generated page. The data placed in *Expression* tag prints on the output stream and automatically converts data into string. The *Expression* tag can contain any Java expression used for printing output equivalent to `out.println()`.Thus, an expression tag contains a scripting language expression which is evaluated, automatically converts data to a String and the outputs are displayed.

Notation of *Expression* tag is shown below:

```
<%= %>
```

Expression tag must begin with `<%=` Inside *Expression* tag, the user embeds any Java expression. *Expression* tag ends with the notation `%>`.

NOTE: *Expression* should not contain a semicolon between codes, as with *Declaration* tag.

General syntax of Expression Tag:

```
<%!                     //start of declaration tag

     statement          //Java Expression

%>                      //end of declaration tag
```

For example:

```
<%!

      Exfdate: <%= new java.util.Date() %>

%>
```

Above example displays current date and time as `Date()` is placed inside the *Expression* tag `<%= %>`

JSP Action tags

Action tag which is another type of tag, include action tag, forward action tag and useBean action tag used in Java Server Pages.

Action Tag:

Action tag is used to transfer the control between pages and is also used to enable the use of server side JavaBeans. Instead of using Java code, the programmer uses special JSP action tags to either link to a Java Bean set its properties, or get its properties.

General syntax of Action Tag:

```
<jsp:action attributes />
```

In the above statement `jsp` is a keyword.

There are many action tags that are available for Java Server Pages. The most commonly used action tags are three of them and they are namely:

- include
- forward
- useBean

Syntax, attributes and usage of above three action tags are provided in brief.

include action tag

`include` is a type of directive tag. `include tag` has the same concept as that of the include directive. The `include tag` is used to include either static or dynamic content, wherever required.

General syntax of include action Tag:

```
<jsp:include page="{relativeURL | <%= expression %>}"
flush="true" />
```

In the above statement, the words `jsp, include, flush` and `page` are keywords. The `include action tag` includes a static file or it is used to send a request to a dynamic file. If the included file is a static file, the contents are included as such in the called JSP file. In contrast, if the included file is dynamic, then a request sends to the dynamic file. Once the include action is completed, the result is sent back.

If the file is dynamic one can use the syntax as below:

```
<jsp:include page="{relativeURL | <%= expression %>}"
flush="true" />
 <jsp:param name="parameterName" value="{parameterValue | <%=
expression %>}" />
 </jsp:include>
```

The file is dynamic and extra attribute clause passed is `jsp:param` clause, and this is used to pass the name and value of a parameter to the dynamic file. In the above, the words `jsp, include, flush, page, param, name` and `value` are keywords. In the above syntax for dynamic file inclusion, the name attribute specifies the parameter name and its case-sensitive literal string. The value attribute specifies the parameter value, taking either a case-sensitive literal string or an expression that is evaluated at request time.

In the syntax of `include action Tag` relative URL placed within `" "` denotes the location of the file to be included or denotes the pathname to be included. This can also be an expression, which is taken automatically as string, denoting the relative URL.

NOTE: the pathname must not contain the protocol name, port number, or domain name. Both absolute and relative representations can be made for denoting the URL.

The flush attribute can only take on value `"true"` as mentioned in syntax notation.

NOTE: Do not to include the value if false for the flush attribute.

The syntax used for including dynamic file had `<jsp:param>` clause allows the user to pass one or more name/value pairs as parameters to an included file. This is the syntax for dynamic file, meaning that the included file is a dynamic file (i.e. either a JSP file or a servlet, or other dynamic file). Also, the user can pass more than one parameter to the included file by using more than one `<jsp:param>` clause.

For example:

An example to include a static file test.html in the include action tag:

```
     <jsp:include page="test.html" flush="true" />
```

Let us see an example of dynamic file inclusion in the include action tag which is done as below:

```
<jsp:include page="example/test.jsp">
<jsp:include name="username" value="test" />

</jsp:include>
```

forward action tag

The forward action tag is used to transfer control to a static or dynamic resource. The static or dynamic resource to which control has to be transferred is represented as a URL. The user can have the target file as an HTML file, another JSP file, or a servlet. Any of the above is permitted.

NOTE: The target file must be in the same application context as the forwarding JSP file.

General syntax of forward action Tag:

```
<jsp:forward page="{relativeURL | <%= expression %>}" />
```

Another syntax representation for `forward action Tag` is as below:

```
<jsp:forward page ="{relativeURL | <%= expression %>}" />
<jsp:param   name ="parameterName"
                   value="parameterValue" | <%= expression %>}"
/>

</jsp:forward>
```

In the above statement, `jsp`, `forward`, `page`, `param`, `name` and `value` are all keywords. The relative URL of the file to which the request is forwarded is represented as a String, an expression or as absolute or relative representation to the current JSP file denoting the pathname.

NOTE: Do not include protocol name, port number, or domain name in the pathname. The file can be a JSP file or a servlet, or any other dynamic file.

If the user follows the second syntax representation of forward action Tag, then the target file should be dynamic where `<jsp:param>` clause is used. This allows the sending of one or more name/value pairs as parameters to a dynamic file. The user can pass more than one parameter to the target file by using more than one `<jsp:param>` clause. In the above second representation, the name attribute is used to specify the parameter name and its case-sensitive literal string. The value attribute can be specified by the parameter value taking either a case-sensitive literal string or an expression that is evaluated at request time.

For example:

```
<jsp:forward page ="test.html" />
<jsp:forward page ="/example/test" />
<jsp:param   name ="username" value="test" />

</jsp:forward>
```