JSP Application Development

Generating Dynamic Content:

JSP is all about generating dynamic content: content that differs based on user input, time of day, the state of an external system, or any other runtime conditions. JSP provides you with lots of tools for generating this content.

Creating JSP Page:

JSP page is just a regular HTML page with a few special elements. A JSP page should have the file extension ***.jsp***, which tells the server that the page needs to be processed by the JSP container. Without this clue, the server is unable to distinguish a JSP page from any other type of file and sends it unprocessed to the browser.

When working with JSP pages, you just need a regular text editor such as Notepad on Windows or Emacs on UNIX. There are a number of tools that may make it easier for you, such as syntax-aware editors that color-code JSP and HTML elements. Some Interactive Development Environments (IDE) even includes a small web container that allows you to easily execute and debug the pages during development.

The first example JSP page, named easy.jsp, is shown below:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
<head>
<title>JSP is Easy</title>
</head>
<body bgcolor="white">
<h1>JSP is as easy as ...</h1>
<%-- Calculate the sum of 1 + 2 + 3 dynamically --%>
1 + 2 + 3 = <c:out value="${1 + 2 + 3}" />
</body>
</html>
```

The easy.jsp page displays static HTML plus the sum of 1, 2, and 3, calculated at runtime and dynamically added to the response.

Installing a JSP Page:

A complete web application may consist of several different resources: JSP pages, Servlets, applets, static HTML pages, custom tag libraries, and other Java class files. Until very recently, an application with all these components had to be installed and configured in different ways for different servers, making it hard for web application developers to provide easy-to-use installation instructions and tools. Starting with the Servlet 2.2 specification, there's a standard, portable way to package all web application resources, along with a ***deployment descriptor***.

The ***deployment descriptor*** is a file named *web.xml*, containing information about security requirements, how all the resources fit together, and other facts about the application. The deployment descriptor and all the other web application files are placed in a WAR file, arranged in a well-defined hierarchy. A WAR file has a .war file extension and can be created with the Java jar command or a ZIP utility program, such as WinZip i.e. the same file format is used for both JAR and ZIP files.

There is a directory named as WEB-INF that contains the application deployment descriptor (web.xml), as well as subdirectories for other types of resources, such as Java class files and configuration files. A browser doesn't have access to the files under this directory, so it's a safe place for files you don't want public. The deployment descriptor file, web.xml, is an XML file with configuration information for the application.

In addition, two WEB-INF subdirectories have special meaning: lib and classes. All application class files (such as servlet and custom tag library classes) must be stored in these two directories. The lib directory is for Java archive (JAR) files (compressed archives of Java class files). Class files that aren't packaged in JAR files must be stored in the classes directory, which can be convenient during development. The files must be stored in subdirectories of the classes directory that mirror their package structure and must follow the standard Java conventions.

Running a JSP Page:

Assuming you have installed the book examples in Tomcat's default directory for applications, called webapps. When you install an application in Tomcat's webapps directory, the subdirectory name is assigned automatically as the URI prefix for the application i.e. /ora in this case.

Now start the Tomcat server and load the book examples main page by typing the URL http://localhost:8080/ora/index.html in the browser address field. Note how the /ora part of the URL matches the Tomcat webapps subdirectory name for the example application. This part of the URL is called the application's context path; every web application has a unique context path, assigned one way or another when you install the application.

Tomcat 4 uses the subdirectory name as the context path by default, but other containers may prompt you for a path in an installation tool or use other conventions. When you make a request for a web application resource like an HTML or JSP page, or a servlet, the first part of the URL (after the hostname and port number) must be the context path, so the container knows which application should handle the request.

There's one exception to this rule; one application per container may be installed as the default, or root, application. For Tomcat 4, this application in stored in the webapps/ROOT directory, by default.

JSP Scripting Elements:

JSP scripting elements let you insert code into the servlet that will be generated from the JSP page. There are three forms:

1. ***Expressions*** of the form <%= expression %>, which are evaluated and inserted into the Servlets output.
2. ***Scriptlets*** of the form <% code %>, which are inserted into the Servlets _jspService method (called by service).
3. ***Declarations*** of the form <%! code %>, which are inserted into the body of the servlet class, outside of any existing methods.

Template text:

In many cases, a large percentage of your JSP page just consists of static HTML, known as template text. In almost all respects, this HTML looks just like normal HTML, follows all the same syntax rules, and is simply "passed through" to the client by the servlet created to handle the page. Not only does the HTML look normal, it can be created by whatever tools you already are using for building Web pages.

However, there are two minor exceptions to the “template text is passed straight through” rule. First, if you want to have <% in the output, you need to put <\% in the template text.

Second, if you want a comment to appear in the JSP page but not in the resultant document, use *<%-- JSP Comment --%>* HTML comments of the form *<!-- HTML Comment -->* are passed through to the resultant HTML normally.

JSP Expressions:

The Expression element contains a Java expression that returns a value. This value is then written to the HTML page. The Expression tag can contain any expression that is valid according to the Java Language Specification. This includes variables, method calls than return values or any object that contains a toString() method. A JSP expression is used to insert values directly into the output. It has the following form:

<%= Java Expression %>

The expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at run time (when the page is requested) and thus has full access to information about the request.

For example, the following shows the date/time that the page was requested:

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

To simplify these expressions, you can use a number of predefined variables. The most important ones are:

1. ***request***, the HttpServletRequest
2. ***response***, the HttpServletResponse
3. ***session***, the HttpSession associated with the request
4. ***out***, the PrintWriter used to send output to the client

Example:

Your hostname: <%= request.getRemoteHost() %>

Finally, note that XML authors can use an alternative syntax for JSP expressions:

<jsp:expression>
Java Expression
</jsp:expression>

Remember that XML elements, unlike HTML ones, are case sensitive. So be sure to use lowercase.

JSP Scriptlet:

The scriptlet can contain any number of language statements, variable or method declarations, or expressions that are valid in the page scripting language. Within a scriptlet, you can do any of the following:

1. Declare variables or methods to use later in the JSP page.
2. Write expressions valid in the page scripting language.
3. Use any of the implicit objects or any object declared with a <jsp:useBean> element.
4. Write any other statement valid in the scripting language used in the JSP page.

Any text, HTML tags, or JSP elements you write must be outside the scriptlet. For example,

```
<HTML>
<BODY>
<%
   // This scriptlet declares and initializes "date"
   java.util.Date date = new java.util.Date();
%>
Hello!  The time is now
<%
   out.println( date );
   out.println( "<BR>Your machine's address is " );
   out.println( request.getRemoteHost());
%>
</BODY>
</HTML>
```

Scriptlets are executed at request time, when the JSP container processes the request. If the scriptlet produces output, the output is stored in the out object. If you want to use the characters "%>" inside a scriptlet, enter "%\>" instead. Finally, note that the XML equivalent of <% Code %> is

```
<jsp:scriptlet>
Code
</jsp:scriptlet>
```

JSP Declaration:

A declaration can consist of either methods or variables; static constants are a good example of what to put in a declaration. The JSP you write turns into a class definition. All the Scriptlets you write are placed inside a single method of this class. You can also add variable and method declarations to this class. You can then use these variables and methods from your Scriptlets and expressions.

You can use declarations to declare one or more variables and methods at the class level of the compiled servlet. The fact that they are declared at class level rather than in the body of the page is significant. The class members (variables and methods) can then be used by Java code in the rest of the page.

Syntax:

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

When you write a declaration in a JSP page, remember these rules:

1. You must end the declaration with a semicolon (the same rule as for a Scriptlet, but the opposite of an Expression).
   ```
   <% int i = 0; %>
   ```
2. You can already use variables or methods that are declared in packages imported by the page directive, without declaring them in a declaration element.
3. You can declare any number of variables or methods within one declaration element, as long as you end each declaration with a semicolon. The declaration must be valid in the Java programming language.
   ```
   <% int i = 0; long l = 5L; %>
   ```

For example,

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

A declaration has translation unit scope, so it is valid in the JSP page and any of its static includes files. A static include file becomes part of the source of the JSP page and is any file included with an include directive or a static resource included with a <jsp:include> element. The scope of a declaration does not include dynamic resources included with <jsp:include>.

As with Scriptlets, if you want to use the characters "%>", enter "%\>" instead. Finally, note that the XML equivalent of <%! Code %> is

```
<jsp:declaration>
Code
</jsp:declaration>
```

Implicit JSP Objects:

Scripting elements can use predefined variables that the container assigns as references to implicit objects as shown in the table to access request and application data. These objects are instances of classes defined by the servlet and JSP specifications.

Variable	Java type
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
exception	java.lang.Throwable
out	javax.servlet.jsp.JspWriter
page	java.lang.Object
pageContext	javax.servlet.jsp.PageContext
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
session	javax.servlet.http.HttpSession

These objects provide access to the same information (and more) as the implicit variables you can use in EL (Expression Language) expressions, but it's not a one-to-one match:

pageContext:

The pageContext variable contains a reference to an instance of the class named javax.servlet.jsp.PageContext. It provides methods for accessing references to all the other objects and attributes for holding data that is shared between components in the same page. It's the same object that you can access with the ${pageContext} EL expression.

Attribute values for this object represent the page scope; they are the same objects as are available to the EL world as a Map represented by the ${pageScope} expression.

request:

The request variable contains a reference to an instance of a class that implements an interface named javax.servlet.http.HttpServletRequest. It provides methods for accessing all the information that's available about the current request, such as request parameters, attributes, headers, and cookies. It's the same object that you can access with the ${pageContext.request} EL expression. Attribute values for this object represent the request scope; they are the same objects as are available to the EL world as a Map represented by the ${requestScope} expression.

response:

The response variable contains a reference to an object representing the current response message. It is the instance of a class that implements the javax.servlet.http.HttpServletResponse interface, with methods for setting headers and the status code, and adding cookies. It also provides methods related to session tracking. These methods are the response methods you're most likely to use. The same object can be accessed with the ${pageContext.response} EL expression.

session:

The session variable allows you to access the client's session data, managed by the server. It's assigned a reference to an instance of a class that implements the javax.servlet.http.HttpSession interface, which provides access to session data as well as information about the session, such as when it was created and when a request for the session was last received. It's the same object that you can access with the ${pageContext.session} EL expression. Attribute values for this object represent the session scope; they are the same objects as are available to the EL world as a Map represented by the ${sessionScope} expression.

application:

The application variable contains a reference to the instance of a class that implements the javax.servlet.ServletContext interface that represents the application. This object holds references to other objects that more than one user may require access to, such as a database connection pool shared by all application users. It also contains log( ) methods you can use to write messages to the container's log file. It's the same object that you can access with the ${pageContext.servletContext} EL expression. Attribute values for this object represent the application scope; they are the same objects as are available to the EL world as a Map represented by the ${applicationScope} expression.

out:

The out object is an instance of javax.servlet.jsp.JspWriter. You can use the print( ) and println( ) methods provided by this object to add text to the response message body. In most cases, however, you will just use template text and JSP action elements instead of explicitly printing to the out object.

exception:

The exception object is available only in error pages and contains information about a runtime error. It's the same object that you can access with the ${pageContext.exception} EL expression.

Conditional Processing:

In most web applications, you produce different output based on runtime conditions, such as the state of a bean or the value of a request header such as UserAgent (containing information about the type of client that is accessing the page).

If the differences are not too great, you can use JSP scripting elements to control which parts of the JSP page are sent to the browser, generating alternative outputs from the same JSP page. However, if the outputs are completely different, it is recommended to use a separate JSP page for each alternative and passing control from one page to another.

Displaying Values:

Besides using Scriptlets for conditional output, one more way to employ scripting elements is by using a JSP expression element to insert values into the response. A JSP expression element can be used instead of the <jsp:getProperty> action in some places, but it is also useful to insert the value of any Java expression that can be treated as a String.

An expression starts with <%= and ends with %>. Note that the only syntax difference compared to a scriptlet is the equals sign (=) in the start identifier. An example is:

<%= userInfo.getUserName( ) %>

The result of the expression is written to the response body. One thing is important to note: as opposed to statements in a scriptlet, the code in an expression must not end with a semicolon. This is because the JSP container combines the expression code with code for writing the result to the response body. If the expression ends with a semicolon, the combined code will not be syntactically correct.

Using an Expression to set an attribute:

In JSP the attributes are set to literal string values. But in many cases, the value of an attribute is not known when you write the JSP page; instead, the value must be calculated when the JSP page is requested. For situations like this, you can use a JSP expression as an attribute value. This is called a request-time attribute value. Here is an example of how this can be used to set an attribute of a fictitious log entry bean:

```
<jsp:useBean id="logEntry" class="com.foo.LogEntryBean" />
<jsp:setProperty name="logEntry" property="entryTime"
value="<%= new java.util.Date( ) %>" />
```

This bean has a property named entryTime that holds a timestamp for a log entry, while other properties hold the information to be logged. To set the timestamp to the time when the JSP page is requested, a <jsp:setProperty> action with a request-time attribute value is used.

The attribute value is represented by the same type of JSP expression as in the previous snippet, here an expression that creates a new java.util.Date object (representing the current date and time). The requesttime attribute is evaluated when the page is requested, and the corresponding attribute is set to the result of the expression.

Not all attributes support request-time values. One reason is that some attribute values must be known when the page is converted into a servlet. For instance, the class attribute value in the <jsp:useBean> action must be known in the translation phase so that the JSP container can generate valid Java code for the servlet. Request-time attributes also require a bit more processing than static string values, so it's up to the custom action developer to decide if request-time attribute values are supported or not.

Declaring Variables and Methods:

In JSP we have three scripting elements like Scriptlets, expressions and declaration elements. A declaration element is a one, which is used to declare Java variables and methods in a JSP page. This is not recommended definition because java variables can be declared either within a method or outside the body of all methods, like this:

```
public class SomeClass {
// Instance variable
private String anInstanceVariable;
// Method
public void doSomething( ) {
String aLocalVariable;
}
}
```

A variable declared outside the body of all methods is called an instance variable. Its value can be accessed from any method in the class, and it keeps its value even when the method that sets it returns.

A variable declared within the body of a method is called a local variable. A local variable can be accessed only from the method where it's declared. When the method returns, the local variable disappears.

A JSP page is turned into a servlet class when it's first requested, and the JSP container creates one instance of this class. If the same page is requested by more than one user at a time, the same instance is used for each request. Each user is assigned what is called a thread in the server, and each thread executes the main method in the JSP object. When more than one thread executes the same code, you have to make sure the code is thread-safe. This means that the code must behave the same when many threads are executing as when just one thread executes the code.

Multithreading and thread-safe code strategies are best left to programmers. However, you should know that using a JSP declaration element to declare variables exposes your page to multithreading problems. That's because a variable declared using a JSP declaration element ends up as an instance variable in the generated servlet, not as a local variable in a method. Since an instance variable keeps its value when the method executed by a thread returns, it is visible to all threads executing code in the same instance.

If one thread changes the value of the instance variable, the value is changed for all threads. To put this in JSP terms, if the instance variable is changed because one user accesses the page, all users accessing the same page will use the new value.

When you declare a variable within a scriptlet element instead of in a JSP declaration block, the variable ends up as a local variable in the generated Servlets request processing method. Each thread has its own copy of a local variable, so local variables can't cause any problems if more than one thread executes the same code. If the value of a local variable is changed, it will not affect the other threads.

Let’s look at a simple example. We use two int variables: one declared as an instance variable using a JSP expression, and the other declared as a local variable. We increment them both by one and display the new values.

Example:

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

```
<%!
int globalCounter = 0;
%>
<html>
<head>
<title>A page with a counter</title>
</head>
<body bgcolor="white">
This page has been visited: <%= ++globalCounter %> times.
<p>
<%
int localCounter = 0;
%>
This counter never increases its value: <%= ++localCounter %>
</body>
</html>
```

The JSP declaration element is right at the beginning of the page, starting with <%! and ending with %>. The exclamation point (!) in the start identifier; that's what makes it a declaration as opposed to a scriptlet.

The declaration element declares an instance variable named globalCounter, shared by all requests for the page. In the body section of the page, JSP expression increments the variable's value. Next comes a scriptlet, enclosed by <% and %>, that declares a local variable named localCounter. The last scriptlet increments the value of the local variable.

When you run this example, the globalCounter value increases every time you load the page, but localCounter stays the same. Again, this is because globalCounter is an instance variable (its value is available to all requests and remains between requests) while localCounter is a local variable (its value is available only to the current request and is dropped when the request ends).

A JSP declaration element can also be used to declare a method that can then be used in Scriptlets in the same page. The only harm this could cause is that your JSP pages end up containing too much code, making it hard to maintain the application. A far better approach is to use JavaBeans and custom actions.

Example: Method Declaration and use

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<%!
String randomColor( ) {
java.util.Random random = new java.util.Random( );
int red = (int) (random.nextFloat( ) * 255);
int green = (int) (random.nextFloat( ) * 255);
int blue = (int) (random.nextFloat( ) * 255);
return "#" +
Integer.toString(red, 16) +
Integer.toString(green, 16) +
```

```
Integer.toString(blue, 16);
}
%>
<h1>Random Color</h1>
<table bgcolor="<%= randomColor( ) %>" >
<tr><td width="100" height="100"> </td></tr>
</table>
</body>
</html>
```

The method named randomColor( ), declared between <%! and %>, returns a randomly generated String in a format that can be used as an HTML color value. This method is then called from an expression element to set the background color for a table. Every time you reload this page, you see a single table cell with a randomly selected color.

Error Handling and Debugging:

When you develop any application that's more than a trivial example, errors are inevitable. A JSP-based application is no exception. There are many types of errors you will deal with. Simple syntax errors in the JSP pages are almost a given during the development phase. And even after you have fixed all the syntax errors, you may still have to figure out why the application doesn't work as you intended due to design mistakes.

The application must also be designed to deal with problems that can occur when it's deployed for production use. Users can enter invalid values and try to use the application in ways you never imagined. External systems, such as databases, can fail or become unavailable due to network problems.

Since a web application is the face of a company, making sure it behaves well, even when the users misbehave and the world around it falls apart, is extremely important for a positive customer perception. Proper design and testing is the only way to accomplish this goal. Unfortunately, many developers seem to forget the hard-learned lessons from traditional application development when designing web applications.

Dealing with Syntax Errors:

The first type of error you will encounter is the one you, or your co-workers, create by simple typos: in other words, syntax error. The JSP container needs every JSP element to be written exactly as it's defined in the specification in order to turn the JSP page into a valid servlet class. When it finds something that's not right, it will tell you. But how easy it is to understand what it tells you depends on the type of error, the JSP container implementation, and sometimes, on how fluent you are in computer gibberish.

Example: Improperly Terminated Directive

```
<%@ page language="java" contentType="text/html" >
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
The current time at the server is:
<ul>
<li>Date: <jsp:getProperty name="clock" property="date" />
<li>Month: <jsp:getProperty name="clock" property="month" />
<li>Year: <jsp:getProperty name="clock" property="year" />
```

<li>Hours: <jsp:getProperty name="clock" property="hours" />

<li>Minutes: <jsp:getProperty name="clock" property="minutes" />

***
***

</body>

</html>

The syntax error here is that the page directive on the first line is not closed properly with %>; the percent sign is missing.

Tomcat reports the error by sending an error message to the browser. This is the default behavior for Tomcat, but it's not mandated by the JSP specification. The specification requires only that a response with the HTTP status code for a severe error (500) is returned, but how a JSP container reports the details is vendor-specific.

The actual error message is called an exception stack trace. When something goes really wrong in a Java method, it typically throws an exception. An exception is a special Java object, and throwing an exception is the method's way of saying it doesn't know how to handle a problem.

Sometimes another part of the program can take care of the problem in a graceful manner, but in many cases the best that can be done is to tell the user about it and move on. That's what the Tomcat container does when it finds a problem with a JSP page during the translation phase: it sends the exception stack trace to the browser.

The stack trace contains a message about what went wrong and where the problem occurred. The message is intended to be informative enough for a user to understand, but the actual trace information is of value only to a programmer.

Debugging a JSP-Based Application:

A debugger steps through the program line by line or runs until it reaches a break point that you have defined, and let you inspect the values of all variables in the program. With careful analysis of the program flow in runtime, you can discover why it works the way it does, and not the way you want it to.

There are debuggers for JSP as well, such as IBM's Visual Age for Java. This product lets you debug a JSP page exactly the same way as you would a program written in a more traditional programming language.

But a real debugger is often overkill for JSP pages. If your pages are so complex that you feel you need a debugger, you may want to move code from the pages into JavaBeans or custom actions instead. These components can then be debugged with a standard Java debugger, which can be found in most Java Interactive Development Environments (IDEs).

Example: Testing Header Values in the Wrong Order

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

<html>

<body bgcolor="white">

<% if (request.getHeader("User-Agent").indexOf("Mozilla") != -1) { %>

You're using Netscape.

<%

} else

if (request.getHeader("User-Agent").indexOf("MSIE") != 1) {

%>

You're using Internet Explorer.

```
<% } else { %>
You're using a browser I don't know about.
<% } %>
</body>
</html>
```

If you run this example in a Netscape browser, it responds with "You're using Netscape," as expected. The problem is that if you run it with Internet Explorer, you get the same response. Clearly there's something wrong with the way the User-Agent header value is tested.

Sharing Data between JSP pages, Requests and Users:

Any real application consists of more than a single page, and multiple pages often need access to the same information and server-side resources. When multiple pages are used to process the same request, for instance one page that retrieves the data the user asked for and another that displays it, there must be a way to pass data from one page to another.

In an application in which the user is asked to provide information in multiple steps, such as an online shopping application, there must be a way to collect the information received with each request and get access to the complete set when the user is ready.

Other information and resources need to be shared among multiple pages, requests, and all users. Examples are information about currently logged-in users, database connection pool objects, and cache objects to avoid frequent database lookups.

Passing Control and Data between Pages:

One of the most fundamental features of JSP technology is that it allows for separation of request processing, business logic, and presentation, using what's known as the Model-View-Controller (MVC) model.

Using different JSP pages as Controller and View means that more than one page is used to process a request. To make this happen, you need to be able to do two things:

Pass control from one page to another.
Pass data from one page to another.

Passing Control from One Page to Another:

JSP supports this through the <jsp:forward> action:

```
<jsp:forward page="userinfovalid.jsp" />
```

This action stops processing one page and starts processing the page specified by the page attribute instead, called the target page. The control never returns to the original page. The target page has access to all information about the request, including all request parameters. You can also add additional request parameters when you pass control to another page by using one or more nested <jsp:param> action elements:

```
<jsp:forward page="userinfovalid.jsp" >
<jsp:param name="msg" value="Invalid email address" />
</jsp:forward>
```

Parameters specified with <jsp:param> elements are added to the parameters received with the original request. The target page, therefore, has access to both the original parameters and the new ones, and can access both types in the same way. If a parameter is added to the request using the name of a parameter that already exists, the new value is added to the list of values for the existing parameter.

The page attribute is interpreted relative to the location of the current page if it doesn't start with /. This is called a page-relative path. If the source and target page are located in the same directory, just use the name of the target page as the page attribute value, as in the previous example. You can also refer to a file in a different directory using notation like ../foo/bar.jsp or /foo/bar.jsp. When the page reference starts with /, it's interpreted relative to the top directory for the application's web page files. This is called a context-relative path.

Let's look at some concrete examples to make this clear. If the application's top directory is C:\Tomcat\webapps\myapp, page references in a JSP page located in C:\Tomcat\webapps\myapp\registration\userinfo are interpreted like this:

page="bar.jsp"

C:\Tomcat\webapps\myapp\registration\userinfo\ bar.jsp

page="../foo/bar.jsp"

C:\Tomcat\webapps\myapp\registration\foo\ bar.jsp

page="/foo/bar.jsp"

C:\Tomcat\webapps\myapp\foo\ bar.jsp

Passing data from one page to another:

JSP provides different scopes for sharing data objects between pages, requests, and users. The scope defines for how long the object is available and whether it's available only to one user or to all application users. The following scopes are defined:

- Page
- Request
- Session
- Application

Objects placed in the default scope, the *page scope*, are available only to actions and Scriptlets within one page. The *request scope* is for objects that need to be available to all pages processing the same request. The *session scope* is for objects shared by multiple requests by the same user, and the *application scope* is for objects shared by all users of the application.

The <jsp:useBean> action has a scope attribute that you use to specify in what scope the bean should be placed. Here is an example:

<jsp:useBean id="userInfo" scope="request" class="com.ora.jsp.beans.userinfo.UserInfoBean" />

Sharing Session and Application Data:

HTTP is a stateless, request-response protocol. This means that the browser sends a request for a web resource, and the web server processes the request and returns a response. The server then forgets this transaction ever happened. So when the same browser sends a new request, the web server has no idea that this request is related to the previous one. This is fine if you're dealing with static files, but it's a problem in an interactive web application.

In a travel agency application, for instance, it's important to remember the dates and destination entered to book the flight so the customer doesn't have to enter the same information again when it's time to make hotel and rental car reservations.

The way to solve this problem is to let the server send a piece of information to the browser that the browser then includes in all subsequent requests. This piece of information, called a session ID, is used by the server to recognize a set of requests from the same browser as related: in other words, as part of the same session.

A session starts when the browser makes the first request for a JSP page in a particular application. The session can be ended explicitly by the application, or the JSP container can end it after a period of user inactivity (the default value is typically 30 minutes after the last request).

With the help of session ID, the server knows that all requests from the same browser are related. Information can therefore be saved on the server while processing one request and accessed later when another request is processed. The server uses the session ID to associate the requests with a session object, a temporary in memory storage area where servlets and JSP pages can store information.

The session ID can be transferred between the server and browser in a few different ways. The Servlet 2.2 API, which is the foundation for the JSP 1.1 specification, identifies three methods: using cookies, using encoded URLs, and using the session mechanism built into the Secure Socket Layer (SSL), the encryption technology used by HTTPS.

SSL-based session tracking is currently not supported by any of the major servlet containers, but all of them support the cookie and URL rewriting techniques. JSP hides most of the details about how the session ID is transferred and how the session object is created and accessed, providing you with the session scope to handle session data at a convenient level of abstraction. Information saved in the session scope is available to all pages requested by the same browser during the lifetime of the session.

However, some information is needed by multiple pages independent of who the current user is. JSP supports access to this type of shared information through another scope, the application scope. Information saved in the application scope by one page can later be accessed by another page, even if the two pages were requested by different users. Examples of information typically shared through the application scope are database connection pool objects, information about currently logged-in users, and cache objects to avoid frequent database lookups.

Memory Usage Considerations:

One should be aware that all objects you save in the application and session scopes take up memory in the server process. It's easy to calculate how much memory is used for application objects since you have full control over the number of objects you place there. But the total number of objects in the session scope depends on the number of concurrent sessions, so in addition to the size of each object; you also need to know how many concurrent users you have and how long a session lasts.

Let us consider the example website CartBean, which stores only references to ProductBean instances, not copies of the beans. An object reference in Java is 8 bytes, so with three products in the cart we need 24 bytes. The java.util.Vector object used to hold the references adds some overhead, say 32 bytes. All in all, we need 56 bytes per shopping cart bean with three products.

If this site has a modest number of customers, you may have 10 users shopping per hour. The default timeout for a session is 30 minutes, so let's say that at any given moment, you have 10 active users and another 10 sessions that are not active but have not timed out yet. This gives a total of 20 sessions times 56 bytes per session, a total of 1,120 bytes.

Now let's say the website becomes extremely popular, with 2,000 customers per hour. Using the same method to calculate the number of concurrent sessions, you now have 4,000 sessions at 56 bytes, a total of roughly 220 KB - still nothing to worry about. However, if you store larger objects in each session, for instance the results of a database search, with an average of 10 KB per active session, that corresponds to roughly 40 MB for 4,000 sessions.

A lot more, but still not extreme, at least not for a site intended to handle this amount of traffic. However, it should become apparent that with that many users, you have to be a bit more careful with how you use the session scope. Here are some things you can do to keep the memory requirements under control:

- Place only those objects that really need to be unique for each session in the session scope. In the shopping cart example, for instance, each cart contains references only to the shared product beans, and the catalog bean is shared by all users.

- Set the timeout period for sessions to a lower value than the default. If you know it's rare that your users leave the site for 30 minutes and then return, use a shorter period. You can change the timeout for all sessions in an application through the application's Deployment Descriptor or call session.setMax-InactiveInterval( ) to change it for an individual session.

- Provide a way to end the session explicitly. A good example is a logout function. Another possibility is to invalidate the session when something is completed (such as submitting the order form). You can use the session.invalidate( ) method to invalidate a session and make all objects available for garbage collection.