

JSP Basics Explained

Protocol, Simple, TagLib,
JSTL, DB and Deployment
With Oracle and OC4J

When you want to get started quickly with the technology and need a cook-book recipe to get it up and running, refer to this document. I wrote this because I desired a document that detailed pieces of the puzzle that needed to be in place for each of the technologies to work. Even though each step is detailed, this document covers the basics and should not be thought of as a comprehensive guide. This document assumes that you know a little about Java already, yet it does attempt to explain in some greater detail the steps for deploying. The document also talks about deploying to OC4J standalone server, but the concepts apply to all the Servlet Containers I've worked with (Weblogic, Jboss/Tomcat, JRun, Novell, Sun). It is best to read this document from start to finish (its short – don't worry), and then go back and work through the given examples.

Please send me feedback, if you like it. Enjoy!

-Joel Thompson

Document downloadable at:

<http://www.rhinosystemsinc.com/>

(Under latest news on front page – see Nov 2nd speaking announcement)

Author: Joel Thompson

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 1 of 1

Table of Contents

1	Introduction.....	3
2	HTTP	5
2.1	Protocol & Stateless.....	5
2.2	GET and POST.....	5
2.3	Example FTP session with Yahoo.com	6
3	Servlet Basics.....	8
3.1	Basic HttpServlet.....	8
3.2	Deploying Your Servlet	10
4	JSP Basics	12
4.1	Introduction	12
4.2	Declarations.....	13
4.3	Scriptlets.....	14
4.4	Value of Expression.....	15
4.5	Includes & Import.....	16
4.6	Lifecycle of JSP.....	18
4.6.1	Developer Creates File	18
4.6.2	Package into War File	19
4.6.3	Deploy War File.....	21
4.6.4	Invoking your JSP.....	21
4.6.5	Viewing the precompiled Output (Java file)	21
5	Sample Plain JSP.....	24
5.1	Introduction	24
5.2	Code.....	24
6	Sample JSP with TagLib	26
6.1	Introduction	26
6.2	Code.....	27
7	Sample JSP with JSTL	30
7.1	Introduction	30
7.2	Code.....	30
8	Sample JSP with Oracle RDBMS I/O	31
8.1	Introduction	31
8.2	Code.....	32
9	Extra Info	36
9.1	References:.....	36
9.2	Books:	36

1 Introduction

JSP stands for JavaServer Pages, aka JSP Pages. The reason for the “extra” Page in the name is simply convenience for the “developers” to indicate the difference between a JSP Page or HTML Page, or ASP Page.

When the user enters the URL to your website with a *browser* (like Internet Explorer, Netscape, or Mozilla), the browser simply requests a Web Page to be delivered to it. Typically the *browser only understands the HTML markup language* – in other words, if the user navigates to your website with the *browser*, requesting a webpage, then you as the web administrator, must insure that the requested webpage is there in HTML format to be returned to the browser.

As a simple example, let's say the user types in “<http://www.yourwebsite.com/index.html>” into the textfield of the *browser*. Your *browser*, then requests that exact webpage from your webserver. Your website must have the “index.html” *Web Page*, sitting in some *directory* (as configured by your web administrator, typically referenced by the variable “DocumentRoot” under Apache’s httpd.conf configuration file); and your webserver must be *up and running* to handle the request. Once the request for that webpage is received your webserver will read the “index.html” file and *send a copy* of the entire file back to the browser as a response to the request. The *browser then reads and parses* all the HTML in the file and renders the markup as a webpage that is presentable (ie, bold, indents, bullet-items, images, ..etc. all layed out in a cohesive manner) to you (the public).

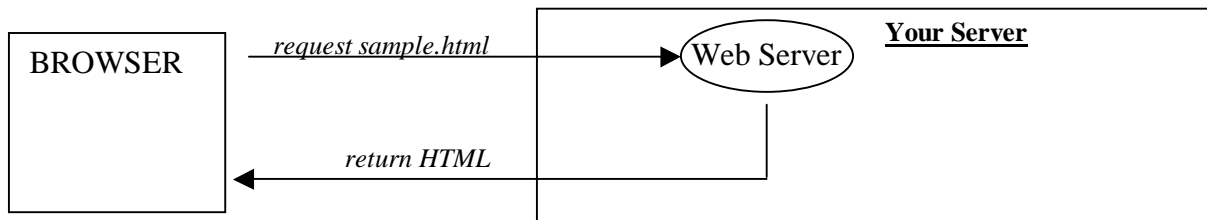


Figure 1: The calling sequence for a WebPage request.

A JSP Page is nothing more than a plain-old HTML page to the browser, but in fact behind the scenes on the server side, it's *HTML is changed “dynamically”*. A request for a JSP Page is typically passed through by the HTTP server (ie, Apache) to another program (something like Oracle’s OC4J or Tomcat) that can handle the JSP formatting and markup – This “other” program is typically called a “Container”, “Servlet Container” or “Servlet Engine”. The *Servlet Engine compiles and executes* the JSP, running the “java code” that you’ve written into the JSP page and as the final step *returns HTML* as the result – Exactly what the *Browser* needs!

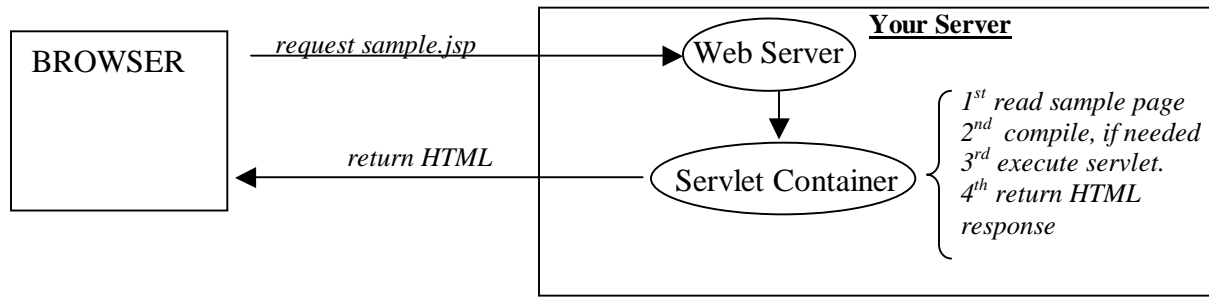


Figure 2: The calling sequence for a JSP Page request.

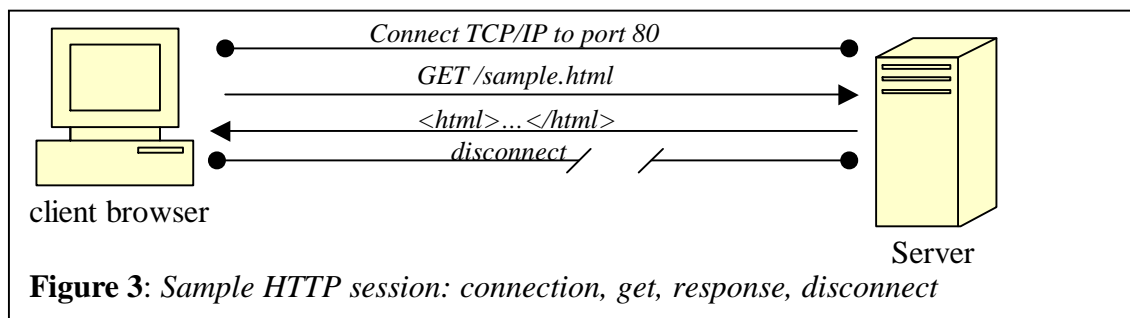
So what exactly is a JSP Page? – The answer is simply this: It is an **HTML page with embedded Java code**. You (the developer) put the JSP file into a directory that your *Servlet Container* knows about, and then upon first request for the JSP Page, the *Servlet Container* compiles the JSP Page into a servlet. From now on the *Servlet Container* **executes** the servlet, and responds to the browser with an HTML stream of data.

2 HTTP

The HTTP protocol call was created in 1990 at CERN to handle sharing documents on a network of servers. The servers needed a standard way of delivering files to the client. **HTTP, being a protocol**, specifies how the calls and results should be implemented. In this sense the HTTP protocol is nothing more than documentation describing how to do it. Implementations of the HTTP protocol include Browsers (the client) and WebServers (the server).

2.1 Protocol & Stateless

The HTTP protocol is a simple request-response protocol with some commands used to specify what you are requesting. It doesn't specify anything about maintaining state of the client application (browser) or server, which means it is a "Stateless" protocol. The sequence of events is as follows:



- 1) CLIENT: The client will "connect" to the server, via TCP/IP socket connection.
- 2) CLIENT: The client issues the command "GET /index.html"
- 3) SERVER: Since the server is waiting on a socket read, it parses the incoming characters looking for the command "GET" and then the argument "/index.html".
- 4) SERVER: The server searches the configured directory where the file "index.html" is supposed to reside, and reads the entire file into memory, and then writes the response back to the CLIENT.
- 5) CLIENT: The client is waiting on a read, and when it receives the character input from the server. The client will read characters into its memory until EOF.
- 6) CLIENT: The client then parses the HTML stream of data and renders the markup in its program for the user to view.

2.2 GET and POST

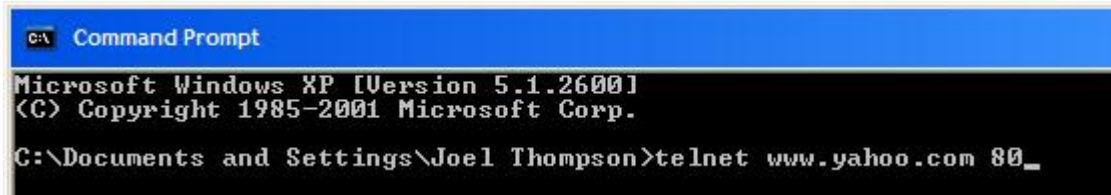
GET and POST are the two most commonly used HTTP commands. The difference between GET and POST, is all the parameters are passed on the command line. With POST the parameters are passed as more input into the request itself – think of it as name=value pairs, that follow in a stream of data after the POST command. This relates directly to GET and POST as used in and HTML FORM "method" attribute.

There are **TWO issues**: #1 with GET the Name/Value pairs are visible from the URL, and POST they are hidden, and #2 the GET is limited on the server size of the environment variables, and GET is virtually unlimited in size, since it is read as a stream of data following the command.

2.3 Example FTP session with Yahoo.com

The best way to illustrate the concept of Client Server and Statelessness, is to show an example of a client (in this case the telnet program), and the server (Yahoo's webserver) interacting with each other to request a page, and Yahoo's response.

Open up a dos window and type "**telnet www.yahoo.com 80**" – You'll see the screen go blank - You are now "connected" to yahoo's webserver at port 80. It is waiting for you to type a command.

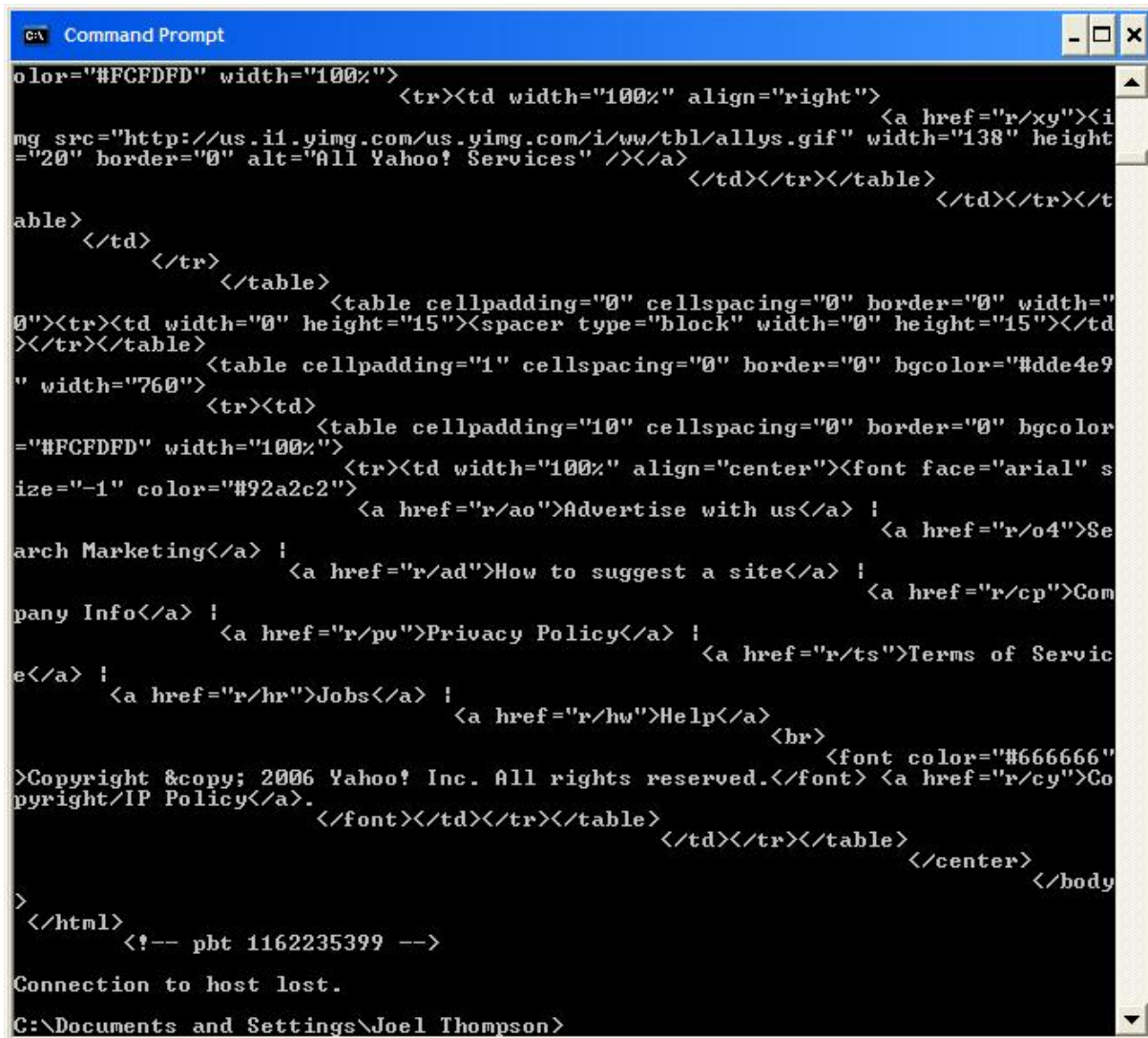
A screenshot of a Windows XP Command Prompt window. The title bar is blue and says "C:\ Command Prompt". The main area is black with white text. It shows the following text: "Microsoft Windows XP [Version 5.1.2600] Copyright 1985-2001 Microsoft Corp." followed by the command prompt "C:\Documents and Settings\Joel Thompson>telnet www.yahoo.com 80_".

```
C:\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Joel Thompson>telnet www.yahoo.com 80_
```

Figure 4: open a dos window and type "**telnet www.yahoo.com 80**"

In the **blank** screen type "**GET /**" (without the double quotes) – you won't see any echo'ing on the screen (meaning you won't see what you type). **Hit return.** You can see the HTML that is returned to your "client" session.

You should see a bunch of HTML code in your window, and then the final message that your connection is closed. The message “Connection to host lost” message informs the



```
C:\ Command Prompt
olor="#FCFDFD" width="100%")
      <tr><td width="100%" align="right">
        <a href="r/xy"><i
img src="http://us.i1.yimg.com/us.yimg.com/i/ww/tbl/allys.gif" width="138" height
="20" border="0" alt="All Yahoo! Services" /></a>
      </td></tr></table>
    </td></tr></t
able>
  </td>
</tr>
</table>
<table cellpadding="0" cellspacing="0" border="0" width="
0"><tr><td width="0" height="15"><spacer type="block" width="0" height="15"></td
></tr></table>
<table cellpadding="1" cellspacing="0" border="0" bgcolor="#dde4e9
" width="760">
  <tr><td>
    <table cellpadding="10" cellspacing="0" border="0" bgcolor
="#FCFDFD" width="100%">
      <tr><td width="100%" align="center"><font face="arial" s
ize="-1" color="#92a2c2">
        <a href="r/ao">Advertise with us</a> |
        <a href="r/o4">Se
arch Marketing</a> |
        <a href="r/ad">How to suggest a site</a> |
        <a href="r/cp">Com
pany Info</a> |
        <a href="r/pv">Privacy Policy</a> |
        <a href="r/ts">Terms of Servic
e</a> |
        <a href="r/hr">Jobs</a> |
        <a href="r/hw">Help</a>
      <br>
      <font color="#666666">
    >Copyright &copy; 2006 Yahoo! Inc. All rights reserved.</font> <a href="r/cy">Co
pyright/IP Policy</a>.
      </font></td></tr></table>
    </td></tr></table>
  </center>
</body
>
</html>
<!-- pbt 1162235399 -->
Connection to host lost.
C:\Documents and Settings\Joel Thompson>
```

Figure 5: Window showing results of typing “GET /” into a telnet session with Yahoo.

user that we are not “connected” any longer to the server. This is done automatically and can’t be adjusted at runtime by client or server. The conclusion of this **lost connection** is that you don’t have State saved when you interact with the server. The client did the following: connect, issued command, received response and was disconnected, and had no way of issuing more command during that session. **Hypothetically** speaking, if it were stateful, the session might have gone like this: connect, command, response, client-decision, 2nd command response..etc., disconnect.

3 Servlet Basics

Servlets can be thought of as Java code with embedded HTML – The inverse of how we think of JSPs being **HTML with embedded Java**. The developer compiles his Servlet, and then **places the class file into a directory** that the Servlet Container knows about. You can call your Servlet from a Browser by specifying its name. You'll need to configure the Servlet Name in the **web.xml**. The servlet container will **load your servlet class**, and call the **“service”** method.

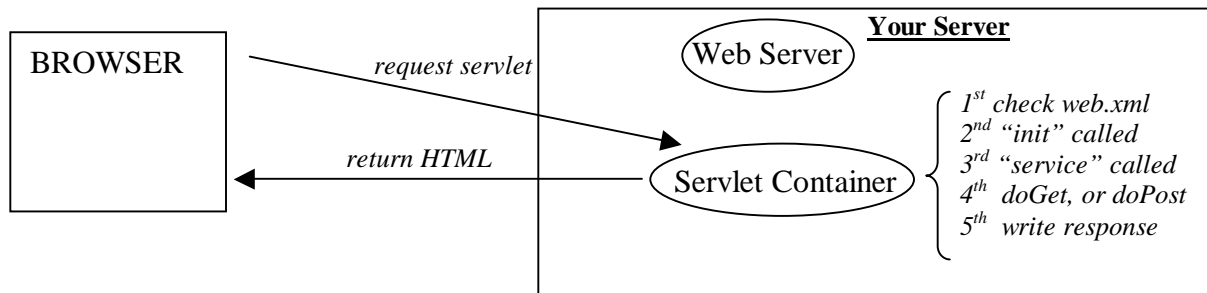


Figure 6: Basic call sequence of a HttpServlet

When you create your Servlet you should **extend HttpServlet**. The HttpServlet will forward the call from “service” to “doGet” and/or “doPost” depending on whether the request is a GET or a POST HTTP request.

You work with **two objects** that are passed to you for “doGet” and “doPost” method calls:

- 1) HttpServletRequest **request** – used to get information and parameters
 - i. find the hostname and ip address of the client
 - ii. request parameters (the name=value pairs, form or url)
 - iii. getting and setting attributes
 - iv. cookies
 - v. getSession()
- 2) HttpServletResponse **response** – used to get output Writer.

An important thing to note is that the Servlet Engine loads only a **single instance** of the Servlet. **Requests** run in their **own thread** but share the Servlet’s instance, and as such the instance variables too. Therefore, you have to take into account that the instance variables are not thread safe. Consider what would happen if you assigned a Database Connection object to an instance variable.

Note: This is important to pay attention to, since in a little while, we’ll see that the JSP is derived from the SERVLET, and we’ll be declaring “instance” variables in the JSP.

3.1 Basic HttpServlet

Here is a basic servlet that shows how you can override the “init”, doGet and doPost methods. Notice the **methods are identical**, and as such is considered poor programming style. If you don’t care whether a GET or POST is sending you the info, then you can

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

create a common routine that both doGet() and doPost() will call. This example also shows how to get the Parameters from the URL, by calling request.getParameterNames() and request.getParameter().

```
package com.rhinosystemsinc;

import java.io.IOException;
import java.io.PrintWriter;

import java.util.Enumeration;

import javax.servlet.*;
import javax.servlet.http.*;

public class srv1 extends HttpServlet {

    private static final String CONTENT_TYPE =
        "text/html; charset=windows-1252";

    //called when the servlet is first accessed to
    //to initialize the servlet
    public void init(ServletConfig config)
        throws ServletException {
        super.init(config);
        System.out.println("init called");
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType(CONTENT_TYPE);

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>srv1</title></head>");
        out.println("<body>");
        out.println("<p>The servlet has received a GET. " +
            "This is the reply.</p>");
        out.println("<p>The servlet Parameters are:<table>");

        Enumeration enum2 = request.getParameterNames();
        String name = null;
        String value = null;
        while (enum2 != null && enum2.hasMoreElements()) {
            name = (String)enum2.nextElement();
            value = request.getParameter(name);
            out.println("<tr><td>" + name + "</td><td>" + value +
                "</td></tr>");
        }
        out.println("</table>");
        out.println("</body></html>");
        out.close();
    }
}
```

```

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

    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>srv1</title></head>");
    out.println("<body>");
    out.println("<p>The servlet has received a POST."+
               " This is the reply.</p>");
    out.println("<p>The servlet Parameters are:<table>");
    Enumeration enum2 = request.getParameterNames();
    String name = null;
    String value = null;
    while (enum2 != null && enum2.hasMoreElements()) {
        name = (String)enum2.nextElement();
        value = request.getParameter(name);
        out.println("<tr>" +
                   "<td>" + name + "</td><td>" + value +
                   "</td>" +
                   "</tr>");
    }
    out.println("</table>");
    out.println("</body></html>");
    out.close();
}
}

```

3.2 Deploying Your Servlet

Here we explain how to deploy to Oracle's default Servlet Container, however, the steps should be similar to all containers.

You can deploy your servlet to Oracle's default Servlet Container by doing the following:

- 1) **Compile your servlet** – make sure to have <jdev10.1.3>\j2ee\home\lib\servlet.jar in your classpath.
- 2) **Copy the class file to:** <jdev10.1.3>\j2ee\home\default-web-app\WEB-INF\classes
- 3) Edit the <jdev10.1.3>\j2ee\home\default-web-app\WEB-INF\&b>web.xml and add the following:

```

<servlet>
  <servlet-name>srv1</servlet-name>
  <servlet-class>com.rhinosystemsinc.srv1</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>srv1</servlet-name>
  <url-pattern>/srv1</url-pattern>
</servlet-mapping>

```

- 4) Startup the OC4J container (restart, if you copied again)
- 5) Test the servlet at:

```
http://localhost:8888/srv1?height=6.1&firstname=joel
```

Another way of deploying your application is by building a **WAR file** (you would put the class in the WEB-INF/classes directory). Please see section on “Packaging into War” ([4.6.2](#)) later in this document.

4 JSP Basics

4.1 Introduction

Keep in mind that a JSP is really **precompiled into Servlet** file. So all that we learned from our Servlet section still applies. Some differences are **calling** your JSP from the url is done directly to the name of the JSP file, and we need to **deploy** it differently to the Servlet Container (on this note, you be happy to know that you don't need to restart your container when deploying to the default webapp directory on OC4J).

In the simplest form to create & run a JSP page, all you have to do is open up any plain-text editor and create a **normal HTML file**, and name it <whatever>.jsp.

To deploy your JSP file (meaning put it in a directory that the Servlet Container knows about), you can simply **copy to the default web application directory**. With Oracle' OC4J standalone container, that place is:

```
<jdev10.1.3>\j2ee\home\default-web-app
```

That's it! Since the plain-old HTML file doesn't contain any JSP features, then it won't do anything but display the HTML that is in the file.

As an example, create hello.jsp with the following HTML:

```
<html>
<body>
Hi there Mom!
</body>
</html>
```

And put the file in the "default-web-app" (as indicated above). Then start the OC4J container and run it from the browser URL:

```
http://localhost:8888/hello.jsp
```

You should get:



Figure 7: *Output of simple hello.jsp file*

Notice that you **don't have to compile** anything as you did with a Servlet. Since you specify a ".jsp" extension, the Servlet Container knows that it needs to compile the file, load the class file, and then run it. This process is done dynamically, and you don't need to restart the Servlet Container, as you did when you redeploy your Servlet's class file.

What happens **behind the scenes** is the Servlet Container parses the .JSP file and creates a Servlet dynamically (<some name>.java). Next the Servlet Container compiles the .JAVA class into a .CLASS file and loads a shared instance of this class.

In this section we covered what it takes from a bare minimum to put in a JSP file. Next, we'll see what features of JSP we can put into our file...

4.2 Declarations

You can declare instance variables (remember: This is compiled into a Servlet, and Servlet's instance variables and methods are **not thread safe**). This declaration section is also useful for creating methods that will be used in the body of the .JSP page.

Note: *You can declare local variables within Scriptlets, as you'll see in the next section.*

You declare a section with

```
<%!  
    ...<some java declaration code>  
%>
```

Example hello2.jsp:

```
<html>  
<%!  
    String sharedString="Hi Mom!";  
    int thisPageCounter=0;
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 13 of 13

```

    public String whatDinner()
    {
        return "Chicken soup";
    }
%>
<body>
Hi there Mom!
</body>
</html>

```

Right now these declarations aren't used in our JSP page. We'll see how to do that next!

4.3 Scriptlets

With Scriptlets you can access declared variables (either local or instance), the “out”, “request” and “response” object (automatically provided for you), and do any other business I/O or logic you'd like. In the Scriptlet you put your java code. You can have as many Scriptlets as you'd like in one page. Each of the Scriptlets will be precompiled into the jspService method for the Servlet.

You use Scriptlets with the following syntax:

```

<%
... your Java Code
%>

```

Notice no exclamation after the first %.

Example hello3.jsp:

```

<html>
<%!
    String sharedString="Hi Mom!";
    int thisPageCounter=0;
    public String whatDinner()
    {
        return "Chicken soup";
    }
%>
<body>
<%
    thisPageCounter++;
    out.println(sharedString + "<br/>");
    out.println("What's for dinner?<br/>");
    out.println(whatDinner() + "<br/>");
    out.println("<strong> You asked " + thisPageCounter +
        "</strong> times<br/>");
%>
</body>
</html>

```

You should see:

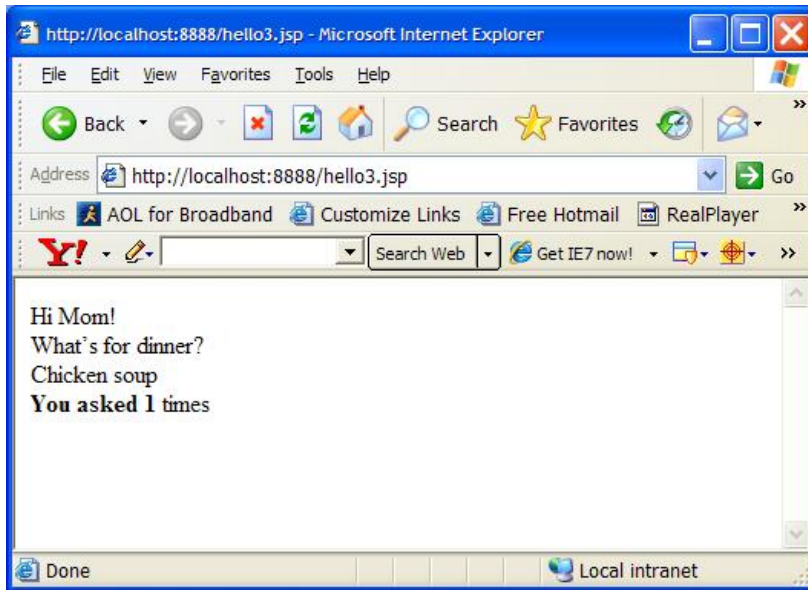


Figure 8: Output of simple *hello3.jsp* file with expressions and counter.

Hit refresh a bunch of times, and what the counter increase. Try it from a different browser, do see that it is a shared instance variable!

4.4 Value of Expression

Notice that the previous example, *hello3.jsp*, used “`out.println (...)`” to create the HTML tags and merge the values. However, there is another way. You can get the value of an expression. Where the expression can simply be the variable name, method or more complex boolean operations...etc. The only requirement for using an expression is that the **expression needs to be convertible to a String**.

You use the following syntax in your JSP page (Notice the ***equals*** after the *percent*)

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

As an example, we'll rewrite *hello3.jsp*.

Example hello4.jsp:

```
<html>
<%!
    String sharedString="Hi Mom!";
    int thisPageCounter=0;
    public String whatDinner()
    {
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 15 of 15

```

        return "Chicken soup";
    }
%>
<body>
<%=sharedString%> <br/>
What's for dinner?<br/>
<%=whatDinner()%> <br/>
<strong> You asked <%= (thisPageCounter++) %>
                </strong> times<br/>

</body>
</html>

```

This example is a little **cleaner**, and works better with **HTML editors!!!**

4.5 Includes & Import

Includes:

You can include other JSP files into your JSP file in two ways – this is similar in concept to how C preprocessor #include works. It basically inlines your included file.

```
<% @ include file="header.jsp" %>
```

This is often used for building up **common sections** of webpages, like a header, navbar and footer, with each being in their own JSP file. As an example, we might build a new webpage for our site called products.jsp as follows:

Example products.jsp file:

```

<html>
<body>
<table border="0" cellspacing="0" cellpadding="0">
<tr><td colspan="2"><%@ include file="header.jsp" %></td></tr>
<tr><td ><%@ include file="navbar.jsp" %></td>
        <td><%@ include file="product_list.jsp"%></td>
</tr>
<tr><td colspan="2"><%@ include file="header.jsp" %></td></tr>
</body>
</html>

```

Imports:

In any java file beyond the basics, you need to import other packages into your program. This is telling the compiler that you want to use **class files that are packaged in JAR files or reside in a directory**, with location determined by the **CLASSPATH** environment variable.

JAR & CLASS FILE LOCATIONS:

In your OC4J Server, you have some options with where these CLASS & JAR files go.

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 16 of 16

See section “[Package into War File](#)” for a sample directory structure with classes and jar files placed according.

- ▶ Option 1: You can place them in the **WEB-INF/lib** when you bundle your web application into a WAR file or copy to the Default Web Application.
- ▶ Option 2: You can put the class files (without JAR) (with their fully qualified package directory) in the **WEB-INF/classes** directory of your WAR file or Default Web Application.
- ▶ Option 3: You can put it in the EAR file during enterprise application bundling (and specify the location in “Class-Path:” in the **META-INF/MANIFEST.MF** file). The files can be at the top-level of the EAR file, or can be subdirectoryed (name of your choosing), as long as the names in the MANIFEST.MF’s Class-Path: variable. Sample in your MANIFEST.MF would be:

```
Class-Path: mylib/sqllite_lib.jar mylib/rhino_util.jar
mylib/classes12.jar
```

Put the above entry in a manifest.txt file and then when you create your ear, you specify the txt file to be used as the MANIFEST.MF, instead of the generated one.

```
jar -cmvf manifest.txt ../bigApp.ear *
```

- ▶ Option 4: Put the JAR files into the **shared library** directory (available to the entire container), known as <jdev10.1.3>\home\applib.

IMPORTING IN YOUR JSP:

In your JSP file you can now use the classes (as located in the JAR files), by “importing” them into your JSP file.

The following is an example of a page declarative used to import classes:

```
<%@ page import="java.io.*,java.text.*,com.rhinosystems.beans.*" %>
```

Above we specified a list of classes to import, separated by commas. You can also specify these on separate lines for readability.

```
<%@ page import="java.io.*,
                 java.text.*,
                 com.rhinosystems.beans.*"
%>
```

4.6 Lifecycle of JSP

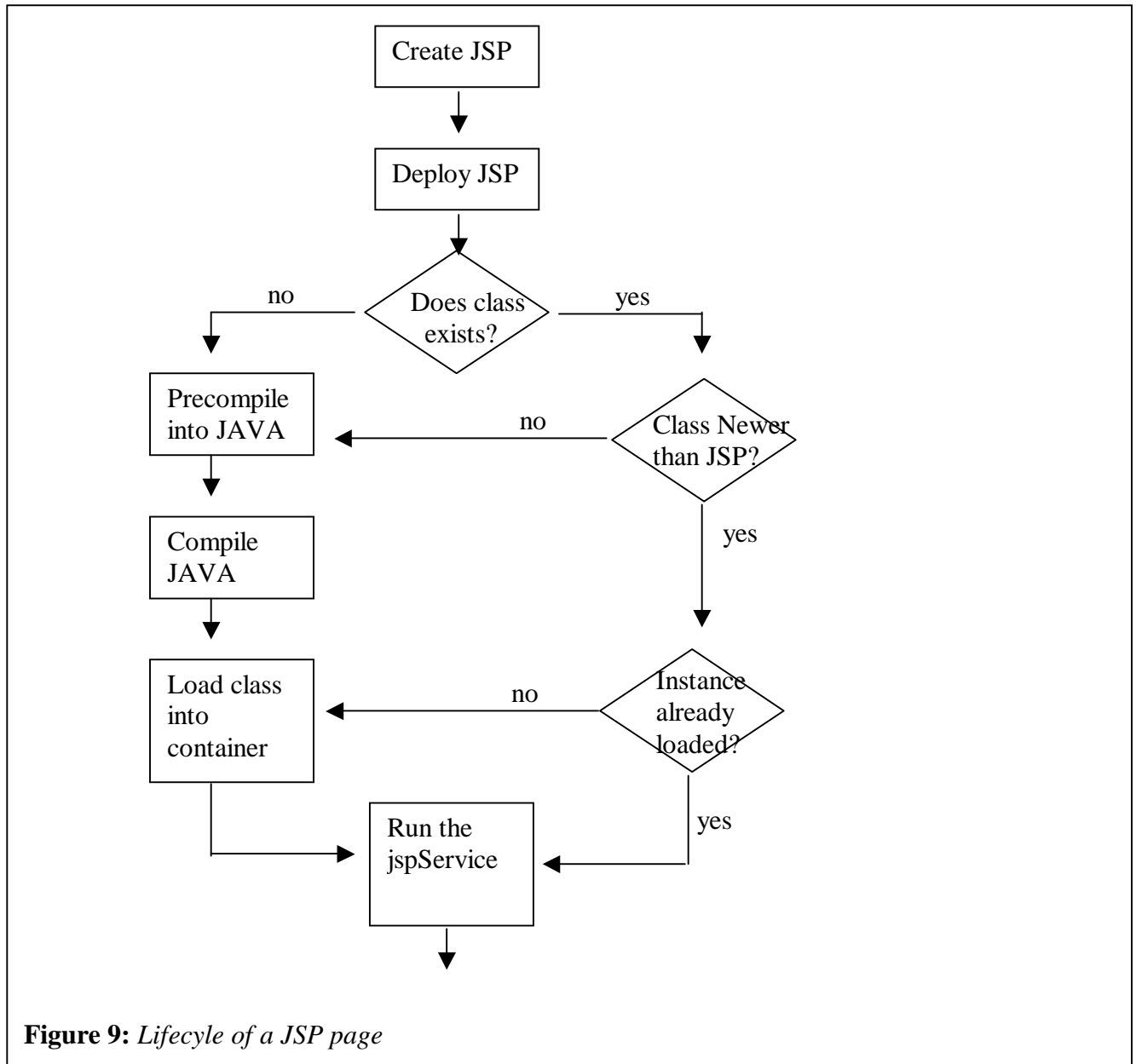


Figure 9: Lifecycle of a JSP page

4.6.1 Developer Creates File

The file needs to be created with a text editor and named with a **.jsp extension**. The extension is important because it indicates to the webserver and servlet container that it is a JSP type file and needs to have its last-modification timestamp checked against what is already loaded in the container and recompiled if timestamp is greater than what is loaded.

4.6.2 Package into War File

(Optionally you don't need to create & deploy a WAR file and can simply copy the JSP file into the default webapplication as shown in previous examples – see “JSP Basics | [Introduction](#)” section for more details on this)

Once the file is created it can be packaged (or bundled) into a WAR file – WAR stands for Web Archive. Basically a WAR file can be named whatever you'd like, but has a **.war extension** to it. You create the WAR file by running the following JAR command (JAR can be found in the <JDK>/bin distribution, and typically you want it in your systems PATH environment variable). Think of JAR'ing up a WAR file as being equal to ZIP'ing up a directory.

The **directory structure** and contents is critical for the WAR file, since the container will look inside the WAR file for specific files and directory locations to extract the need information to load it into the container. Here is a minimal directory structure w/ minimal contents starting with the “webapp” directory:

```
webapp/
|-- WEB-INF/
|   |-- classes/
|   |   |-- com/
|   |   |   |-- rhinosystemsinc/
|   |   |   |   |-- beans/
|   |   |   |   |   |-- Employee.class
|   |-- lib/
|   |   |-- mytools.jar
|   |-- web.xml
|-- hello.jsp
|-- images/
|   |-- home.jpg
```

The **WEB-INF/classes** and **WEB-INF/lib** directory are optional, but if you wanted to include classes and/or jars you need to put them in these directories, since that is where the container will look for them by inspecting the WAR file. In my example the “**images**” directory is a completely arbitrary name, yet it will be the name of the directory that your home.jsp is referencing the images it wants to use in it's webpage. In this case the “hello.jsp” file would reference it by relative path:

```

```

Or it can specify the entire path as in:

```

```

We can use “webapp”, since that is what we are going to name the WAR file in the next step. But first there is the “**web.xml**” file that we need to address.

The web.xml file

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

The web.xml file contains information about your WAR file at a minimum it needs the following information:

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app>
</web-app>
```

An example with a Servlet specified would be:

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee">
  <description>Empty web.xml file for Web Application</description>
  <servlet>
    <servlet-name>srv1</servlet-name>
    <servlet-class>com.rhinosystemsinc.srv1</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>srv1</servlet-name>
    <url-pattern>/srv1</url-pattern>
  </servlet-mapping>
</web-app>
```

The **attributes (ie, xmlns and xsi:schemaLocation)** in the <web-app> tag, indicate what version and format of web.xml file we are dealing with. It is best to check examples that come with your container for what goes here. Sometimes your application will not work properly if you don't have this specified correctly. The <servlet> & <servlet-mapping> tags are mapping the name of the "servlet" that we showed in the earlier example of a Servlet.

JAR UP THE WAR FILE

First CD to the "webapp" directory and then create the final war file by issuing the following JAR command:

```
jar -cf webapp.war *
```

- c tells jar to create a new file
- f indicates that a filename for the jar follows
- * is all the files in the current directory

You might be inclined to use a "." instead "*" since "." would get the entire directory and contents, however you **should avoid this** since your classes will not be found. "." will create a jar with entries like "./com/rhinosystemsinc/beans/Employee.class" instead of "com/rhinosystemsinc/beans/Employee.class".

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

4.6.3 Deploy War File

You need to copy the WAR file into your servers directory. In OC4J that means copying the file to: <jdevstudio1013>\j2ee\home\applications directory and then issuing the commands to **deploy and bind** in OC4J. See the [OC4J manual](#) for more details on this.

With some Servlet containers like Tomcat, you can simply copy the WAR file into the correct directory and it will **automatically** deploy. For Tomcat that directory is <tomcat>/webapps directory.

(Remember, optionally we didn't need to deploy with a WAR file at all, we could have simply copied the JSP to the default webapplication directory and modified the WEB-INF/web.xml file already resident, as shown in previous examples – see "[JSP Basics](#)" section for more details on this)

4.6.4 Invoking your JSP

After deploying your WAR file, you can invoke your JSP from a web browser. In our example the URL would be:

```
http://localhost:8888/webapp/hello.jsp
```

Reminder: The very first time you access the JSP, the container checks to see if the hello.jsp class file exists in the container already; and if not it precompiles the hello.jsp into a java servlet file, and then compiles that file into a class, loads it, and then finally run's it. If the class file does exist, then the container checks JSP file's modification timestamp to see if newer than the class, and if so, repeats the precompile/compile/load step.

The container only precompiles/compiles & loads the JSP if it doesn't exists in the container or the JSP is newer; else it RUN the JSP by calling the "jspService" (a method created for your servlet during precompile time).

4.6.5 Viewing the precompiled Output (Java file)

The precompilation step that was mentioned in "Invoking your JSP" creates an **intermediate JAVA file** that you can view and debug later. Here is the JAVA that was created in our hello4.jsp

Recall that the hello4.jsp file looks like:

```
<html>
<%!
    String sharedString="Hi Mom!";
    int thisPageCounter=0;
    public String whatDinner()
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 21 of 21

```

    {
        return "Chicken soup";
    }
%>
<body>
<%=sharedString%> <br/>
What's for dinner?<br/>
<%=whatDinner()%> <br/>
<strong> You asked <%= (thisPageCounter++) %>
        </strong> times<br/>

</body>
</html>

```

And the results of the container precompiling the JSP into a JAVA file. Notice the **jspService** section in the file as well as the **supplied variables(out,page,session,...etc.)** for your JSP program.

Note: *All JSP containers will do precompile into a JAVA file, but the specifics about the “generated” name, and other details of this file are Container specific.*

```

import oracle.jsp.runtime.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import oracle.jsp.el.*;
import javax.servlet.jsp.el.*;

public class _hello4 extends com.orionserver.http.OrionHttpJspPage {

    // ** Begin Declarations

    String sharedString="Hi Mom!";
    int thisPageCounter=0;
    public String whatDinner()
    {
        return "Chicken soup";
    }

    // ** End Declarations

    public void _jspService(HttpServletRequest request,
    HttpServletResponse response) throws java.io.IOException,
    ServletException {

        response.setContentType( "text/html");
        /* set up the intrinsic variables using the pageContext goober:
        ** session = HttpSession
        ** application = ServletContext
        ** out = JspWriter
        ** page = this

```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 22 of 22

```

** config = ServletConfig
** all session/app beans declared in globals.jsa
*/
PageContext pageContext =
JspFactory.getDefaultFactory().getPageContext( this, request, response,
null, true, JspWriter.DEFAULT_BUFFER, true);
// Note: this is not emitted if the session directive == false
HttpSession session = pageContext.getSession();
int __jsp_tag_starteval;
ServletContext application = pageContext.getServletContext();
JspWriter out = pageContext.getOut();
_hello4 page = this;
ServletConfig config = pageContext.getServletConfig();
javax.servlet.jsp.el.VariableResolver __ojsp_varRes =
(VariableResolver)new OracleVariableResolverImpl(pageContext);

com.evermind.server.http.JspCommonExtraWriter __ojsp_s_out =
(com.evermind.server.http.JspCommonExtraWriter) out;
try {

    __ojsp_s_out.write(__oracle_jsp_text[0]);
    __ojsp_s_out.write(__oracle_jsp_text[1]);
    out.print(sharedString);
    __ojsp_s_out.write(__oracle_jsp_text[2]);
    out.print(whatDinner());
    __ojsp_s_out.write(__oracle_jsp_text[3]);
    out.print((thisPageCounter++));
    __ojsp_s_out.write(__oracle_jsp_text[4]);
    out.print((thisPageCounter>3));
    __ojsp_s_out.write(__oracle_jsp_text[5]);

}
catch( Throwable e) {
    if (!(e instanceof javax.servlet.jsp.SkipPageException)){
        try {
            if (out != null) out.clear();
        }
        catch( Exception clearException) {
        }
        pageContext.handlePageException( e);
    }
}
finally {
    OracleJspRuntime.extraHandlePCFinally(pageContext,false);
    JspFactory.getDefaultFactory().releasePageContext(pageContext);
}

}
private static final byte __oracle_jsp_text[][]=new byte[6][];
static {
    try {
        __oracle_jsp_text[0] =
"<html>\r\n".getBytes("ISO8859_1");
        __oracle_jsp_text[1] =

```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 23 of 23

```

    "\r\n<body>\r\n".getBytes("ISO8859_1");
    __oracle_jsp_text[2] =
    " <br/>\r\nWhat's for dinner?<br/>\r\n".getBytes("ISO8859_1");
    __oracle_jsp_text[3] =
    " <br/>\r\n<strong> You asked ".getBytes("ISO8859_1");
    __oracle_jsp_text[4] =
    " \r\n                </strong>
times<br/>\r\n\r\n".getBytes("ISO8859_1");
    __oracle_jsp_text[5] =
    "\r\n</body>\r\n</html>\r\n".getBytes("ISO8859_1");
    }
    catch (Throwable th) {
        System.err.println(th);
    }
}
}

```

5 Sample Plain JSP

5.1 Introduction

This example shows a list of employees in an HTML table. The list is derived from a static array coded directly into the JSP page. Notice the **Array of Employees** and how we are **looping** through the results – we'll be changing both of these out in later examples of TagLibraries and JSTL.

5.2 Code

JSP File employees1.jsp

```

<%@ page import="java.io.*,
                java.text.*,
                com.rhinosystemsinc.beans.Employee"
    %>
<html>
<body>
<%
    Employee emps[]=new Employee[]{new Employee("Joel",10),
                                    new Employee("Bob",11),
                                    new Employee("Mary",12)};
%>
<table border="1">
<%
    for(int jj=0;jj<emps.length;jj++)
    {
%>
<tr>
    <td><%=emps[jj].getName()%></td>
    <td><%=emps[jj].getEmpid()%></td>
</tr>
<%
    }
%>
</table>

```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

```
</body>
</html>
```

Reminder: *The class file from this Java file needs to be in the WEB-INF/classes directory and deployed to Default Web Application directly or deployed in WAR file with above JSP and web.xml (as explained earlier).*

Java File com\rhinosystemsinc\beans\Employee.java

```
package com.rhinosystemsinc.beans;

public class Employee
{
    String name=null;
    int empid=-1;
    public Employee()
    {
        name=" ";
        empid= -1;
    }
    public Employee(String nm, int i)
    {
        name=nm;
        empid=i;
    }
    public String getName() {return name;}
    public void setName(String nm){name=nm;}
    public int getEmpid(){return empid;}
    public void setEmpid(int ei){empid=ei;}
}
```

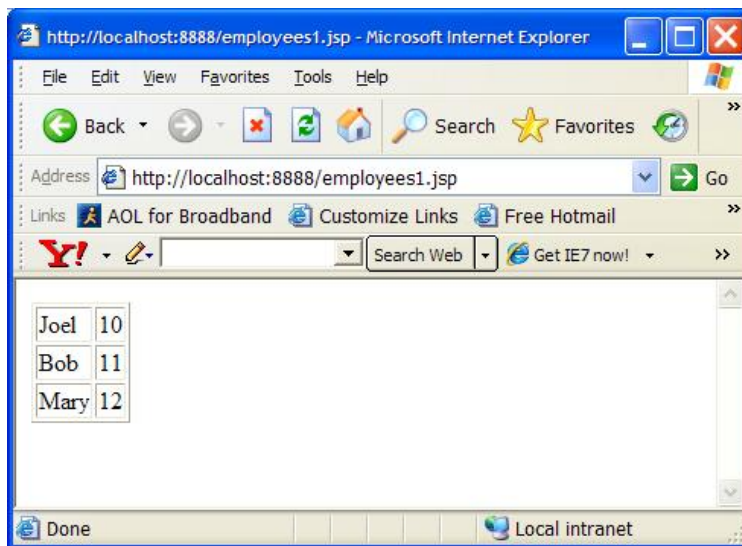


Figure 10: *Output of employees1.jsp simple array of Employee beans.*

6 Sample JSP with TagLib

6.1 Introduction

A tag library is basically a way to **create a new HTML** tag to include into your JSP file without having to write the Scriptlet sections. Most people would agree that a “best practice” would be to hide all your Scriptlets from the JSP - **Scriptlets get messy and are not modularized** and don't work well with **HTML Editors**.

So, let's create a simple tag library to create a list of Employees with a filter by name as an attribute of the tag. We want our tag to look like this when we are finished:

```
<rhino:Employees filterby="Jo"/>
```

To create a tag library you need to do the following things:

- 1) **CREATE TLD FILE:** create a tld file and make available to the JSP file at runtime – usually this means just placing in the WAR file or at the same directory of the JSP file.
- 2) **CREATE A TAGSUPPORT JAVA CLASS:** create a java class that extends TagSupport – compile class and put in your WEB-INF/classes directory.
 - a. **IDENTIFY SUPPORTING JAR FILES:** In order to compile the Java class you created in step #2, you need to find the JAR files from your Servlet Container that supports TagLibs, and put them in your CLASSPATH env variable.

I typically run this command (need cygwin unix SHELL) in the Servlet Containers root directory to find the appropriate JAR file.

```
for f in home/lib/*.jar; do
  jar -tvf $f |grep TagSupport;
  if [ $? -eq 0 ]; then
    echo found in $f;
  fi;
done
```

In OC4J it is found in <jdev10.1.3>\j2ee\home\lib\servlet.jar

Here is my compile command:

```
javac -cp .;C:\software\jdevstudio1013\j2ee\home\lib\servlet.jar
com\rhinosystemsinc\taglibs\EmployeeTag.java
```

You could do this from JDeveloper and the "servlet.jar" file should be in your JDeveloper path already.

- 3) **CREATE THE JSP FILE USING THE TAG:** You now can create the JSP file and use your tag in the JSP file.
- 4) **DEPLOY THE FILES TO THE CONTAINER:** In my case I simply wanted to run the samples, so I copied the files into the default web application dir. However, you can follow the directions to create a WAR file and deploy as such. (see section "Package into War File" section [4.6.2](#))

Here is my copy command (the \ indicates next line should be on this line):

```
copy employees2.jsp \
C:\software\jdevstudio1013\j2ee\home\default-web-app

copy EmployeeTag.tld \
C:\software\jdevstudio1013\j2ee\home\default-web-app

xcopy /S /Y /Q com \
C:\software\jdevstudio1013\j2ee\home\default-web-app\WEB-
INF\classes\com
```

Note: *there is no need to modify the web.xml for this sample.*

6.2 Code

employees2.jsp file:

```
<%@ page import="java.io.*,
                java.text.*,
                java.util.*,
                com.rhinosystemsinc.beans.Employee"
%>

<%@taglib prefix="rhino" uri="EmployeeTag.tld"%>

<html>
<body>

<!-- here is my custom TAG!!! It will
     store the filtered list in an Attribute of the
     request Object -->

    <rhino:Employees filterby="Jo"/>

<table border="1">
<%
    ArrayList al=(ArrayList)request.getAttribute("emps");

    for(int jj=0;jj<al.size();jj++)
    {
%>
<tr>
    <td><%=((Employee)al.get(jj)).getName()%></td>
    <td><%=((Employee)al.get(jj)).getEmpid()%></td>
</tr>
<%
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

```
}
%>
</table>
</body>
</html>
```

EmployeeTag.tld file:

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.0</tlib-version>
  <short-name>Employees</short-name>
  <description>Employee Sample rhino web apps</description>

  <tag>
    <name>Employees</name>
    <tag-class>com.rhinosystemsinc.taglibs.EmployeeTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>filterby</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

com\rhinosystemsinc\taglib\EmployeeTag.java file:

```
package com.rhinosystemsinc.taglibs;

import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.util.*;
import com.rhinosystemsinc.beans.*;

/**
 * Created by IntelliJ IDEA.
 * User: Joel Thompson - joel@rhinosystemsinc.com
 * All rights reserved.
 */
public class EmployeeTag extends TagSupport
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 28 of 28

```

{
    String filterby="";

    Employee emps[]=new Employee[]{new Employee("Joel",10),
                                    new Employee("Joe",13),
                                    new Employee("Joseph",14),
                                    new Employee("Bob",11),
                                    new Employee("Mary",12)
                                    };

    public String getFilterby()
    {
        return filterby;
    }

    public void setFilterby(String f)
    {
        this.filterby = f;
    }

    //this method is called when the tag is found on the
    //JSP page. Our example is
    //<rhino:Employees filterby="Jo">
    //</rhino:Employees>
    //or <rhino:Employees filterby="Jo"/>
    // doStartTag is called by the processor when
    // <rhino:Employees ...> is encountered.
    public int doStartTag() throws JspException
    {
        HttpServletRequest request=
            (HttpServletRequest) pageContext.getRequest();

        ArrayList al=new ArrayList();

        //ServletResponse res=pageContext.getResponse();

        //you can get a handle to the output stream
        //if you'd like to create new HTML (msg,..etc) for the
        //response. IN our case it is not used.

        //JspWriter out=pageContext.getOut();

        //you can get a handle to the user's Session object
        //HttpSession session=request.getSession(true);

        //In our case were are going to put the filtered list
        //in a variable stored in the request object.
        for(int jj=0;jj<emps.length;jj++)
        {
            if(emps[jj].getName().toUpperCase().contains(
                filterby.toUpperCase()))
            {
                al.add(emps[jj]);
            }
        }
    }
}

```

```

        //we'll pull this out later in our JSP file...with a
        //scriptlet
        request.setAttribute("emps",al);

        //tell the processor to skip the BODY of the tag, ie
        //the part between the beginning and end of tag itself.
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException
    {
        return super.doEndTag();
    }

    public int doAfterBody() throws JspException
    {
        return super.doAfterBody();
    }
}

```

7 Sample JSP with JSTL

7.1 Introduction

With JSTL (JSP Standard Tag Libraries), you can take advantage of TagLibraries that already exist. This further **reduces the complexity** of your JSP page. In our previous example with “employees2.jsp” we used our custom tag library to put the list of employees that mathed the “**filterby**” attribute of our tag in the **request’s Attribute list**. We are now going to take advantage of that with a simple JSTL tag library that

7.2 Code

employees3.jsp file:

```

<%@ page import="java.io.*,
                java.text.*,
                java.util.*,
                com.rhinosystemsinc.beans.Employee"
    %>

    <%@taglib prefix="rhino" uri="EmployeeTag.tld"%>

    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

    <html>
    <body>

    <!--
        here is my custom TAG!!! It filters by
        the filter attribute and stores the
        filtered list in an Attribute of the
        request Object, named "emps".
    -->

```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 30 of 30

```

-->
<rhino:Employees filterby="Jo"/>

<table border="1">
<!--
  "emps" is stored as a request Attribute
  The items in emps is iterable and each item
  Supports the java bean symantics with getters
  And setters. JSTL will look in request, page,
  session, and application for the attribute
  with this name "emps".
-->
-->
<c:forEach var="e" items="{emps}">
  <tr> <!-- notice the two different ways to get to the Employee's
    attributes - one is with . notation and the other is
    by array with string name. Either way, doesn't matter.
  -->
    <td><c:out value='${e["name"]}'/> </td>
    <td><c:out value="{e.empid}"/></td>
  </tr>
</c:forEach>
</table>

</table>
</body>
</html>

```

Deployment Notes: In order to deploy this, make sure the Employee class and the EmployeeTag.tld are deployed from previous example. Also note, that some containers don't support the JSTL by default and you may need to provide the JAR files that implement these tags. The two needed jar files are **standard.jar** and **jstl.jar** and must be in the containers lib directory or in your WAR file (under WEB-INF/lib). OC4J does provide these libraries.

8 Sample JSP with Oracle RDBMS I/O

8.1 Introduction

Here we build on the previous example and place the **JDBC code into the Tag Library**. You should replace the connection code that is called from the "doStartTag()" method with **JNDI lookups** into a connection pool. One other consideration would be to **nest tags**, ie have an outer <dbconnect> tag that evaluates it's body elements, which would be the tags rendering the table-HTML with the items retrieved from the database. Although nesting tags gets a bit complicated and beyond the scope of an "introduction" to TagLibraries. If you'd like to pursue the "**nesting**" approach then look into extending your Tag class from **BodyTagSupport** – this way you can get the content of the Body. There is a good tutorial on JSP Tags at:

<http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html>

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Note: You could use JSTL SQL Tag Library to do this example. See the tutorial on Sun's website: <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSTL7.html>

One other thing to consider now is that you have to make sure Oracle's JDBC library (ojdbc14.jar) is in the container. For OC4J it is by default, as you'd expect.

8.2 Code

employees4.jsp file:

```
<%@ page import="java.io.*,
                java.text.*,
                java.util.*,
                com.rhinosystemsinc.beans.Employee"
%>

<%@taglib prefix="rhino" uri="EmployeeDBTag.tld"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<body>

<!-- here is my custom TAG!!! It will
     store the filtered list in an Attribute of the
     request Object -->

<rhino:Employees filterby="Jo"/>

<table border="1">
<c:forEach var="e" items="{emps}">
  <tr>
    <td><c:out value='${e["name"]}' /> </td>
    <td><c:out value='${e.empid}' /></td>
  </tr>
</c:forEach>
</table>

</table>
</body>
</html>
```

Only change from previous to
"EmployeeDBTag"

EmployeeDBTag.tld file:

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd"
        version="2.0">

  <tlib-version>1.0</tlib-version>
```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

Page 32 of 32

```

<short-name>Employees</short-name>
<description>Employee Sample rhino web apps</description>

<tag>
  <name>Employees</name>
  <tag-class>com.rhinosystemsinc.taglibs.EmployeeDBTag</tag-
class>
  <body-content>empty</body-content>
  <attribute>
    <name>filterby</name>
    <required>true</required>
  </attribute>
</tag>
</taglib>

```

Only change from previous to
"EmployeeDBTag"

com\rhinosystemsinc\taglibs\EmployeeDBTag.java file:

```

package com.rhinosystemsinc.taglibs;

import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.util.*;
import com.rhinosystemsinc.beans.*;

//added for DB support
import java.sql.*;
import java.io.IOException;

/**
 * Created by IntelliJ IDEA.
 * User: Joel Thompson
 * Date: Nov 30, 2005
 * Time: 3:16:41 PM
 */
public class EmployeeDBTag extends TagSupport
{
    Connection conn = null;
    String filterby="";

    public String getFilterby()
    {
        return filterby;
    }

    public void setFilterby(String f)

```

By joel@rhinosystemsinc.com 10/2006 - copyright 2006 – all rights reserved.

<http://www.rhinosystemsinc.com>

```

{
    this.filterby = f;
}

public void getConnection()
{
    System.out.println("getConnection called");
    try
    {
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        conn = DriverManager.getConnection(
            "jdbc:oracle:thin:joel/welcome@localhost:1521:SID2");
        conn.setAutoCommit(false);
        System.out.println("got connection");
    }
    catch (SQLException sqle)
    {
        System.out.print("Sorry, the database is " +
            " not available.");
        System.out.print("Exception: " + sqle);
        System.out.println("getConnection.jsp:SQLException: "
            + sqle);
    }
    catch (Exception e)
    {
        System.out.print("Exception occured: " + e);
        System.out.println("getConnection.jsp:Exception: " +
            e);
    }
}

public ArrayList findEmpsByName()
{
    ArrayList al=new ArrayList();
    if(conn!=null)
    {
        PreparedStatement ps=null;
        ResultSet rs=null;
        String name="";
        int id=0;
        try
        {
            System.out.println("prepareStatement");
            ps=conn.prepareStatement("select name,empid " +
                " from employee " +
                " where UPPER(name) like '% ' || UPPER(?) || '% '");
            ps.setString(1,filterby);
            System.out.println("executeQuery");
            rs=ps.executeQuery();
            System.out.println("executed query");

            while(rs!=null&&rs.next())
            {
                name=rs.getString("NAME");
                System.out.println("found emp, by " + name);
                id=rs.getInt("EMPID");
            }
        }
    }
}

```

```

        al.add(new Employee(name,id));
    }
}catch(SQLException se)
{
    System.out.println("Could not excute query ERROR:" +
        se.getMessage());
}finally
{
    if(rs!=null) try{rs.close();}catch(SQLException e){}
    if(ps!=null) try{ps.close();}catch(SQLException e){}
    if(conn!=null) try{conn.close();}catch(SQLException e){}
}
}
return al;
}
//this method is called when the tag is found on the
//JSP page. Our example is
//<rhino:Employees filterby="Jo">
//</rhino:Employees>
//or <rhino:Employees filterby="Jo"/>
// doStartTag is called by the processor when
// <rhino:Employees ...> is encountered.

public int doStartTag() throws JspException
{
    System.out.println("doStartTag called");
    getConnection();
    HttpServletRequest request=
        (HttpServletRequest) pageContext.getRequest();

    ArrayList al=findEmpsByName();

    //we'll pull this out later in our JSP file...with a
    //scriptlet
    request.setAttribute("emps",al);

    //tell the processor to skip the BODY of the tag, ie
    //the part between the beginning and end of tag itself.
    return SKIP_BODY;
}

public int doEndTag() throws JspException
{
    return super.doEndTag();
}

public int doAfterBody() throws JspException
{
    return super.doAfterBody();
}
}

```

Besides the function
getConnection() and
findEmpsByName(),
this is the only change

SQL to create table and populate it:

```
create table Employee
(
    name varchar2(32),
    empid number
);
insert into Employee values('Joel',10);
insert into Employee values('Bob',11);
insert into Employee values('Joseph',12);
insert into Employee values('Mary',13);
insert into Employee values('Joe',14);
commit;
```

9 Extra Info

9.1 References:

<http://java.sun.com>

TagLibrary Tutorial

<http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html>

JSTL SQL Tags tutorial

<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSTL7.html>

OC4J 10.1.3 Manual

http://download-west.oracle.com/otn_hosted_doc/ias/preview/web.1013/b14433.pdf

9.2 Books:

“JSP - The Complete Reference”, by Phil Hanna