Introduction

The purpose of this book is to provide the beginning JSP (Java Server Pages) an introduction to the topic and some more indepth understand over and above that gained from reading the specification.

With the release of the JSP 1.0 public review, JSP has moved from requiring a solid grounding in HTML, and Java Servlets to also requiring knowledge of XML. Although we do assume a certain amount of knowledge about XML and HTML, we do cover the Servlet interface as it of particular importance to JSP - as all Java Server Pages are compiled into Servlets before running.

There are many topics covered in this book, we range from a JSP history introduction, comparing the technology with ASP, a basic introduction to JSP pages, use of beans, and we further look into using databases as well as the n-tier technology of EJB and CORBA.

This book effort is an outgrowth of the JSP FAQ which is currently located at http://www.esperanto.org.nz/jsp/jspfaq.html.

A note about nomenclature - rather than have a section which repeats much of what is in the JSP 1.0 specification, when we introduce JSP notation of any kind, we will have it appear in a light grey box:

<jsp:useBean> - JSP Using Bean
This is some text about the <jsp:useBean> tag.
XML Equivalent: <jsp:useBean> etc

Furthermore, there are a few symbols indicating certain things:

This symbol indicates that this is an action you should do. It appears when an example starts, or when you need to do something to ensure that something happens properly further on

When you see this symbol, it is a key point. Take special note.

Chapter 1 - JSP in Perspective

Where did it come from?

[I only know the history since 0.92, anyone else?]
The basic idea of a JSP is that of mixing Java code into an HTML (or XML) file, which is then converted into a Java Servlet to allow the creation of dynamic content (i.e. non static HTML pages). With Java Servlets, the focus was on Java code, which could then output HTML as required. This for many people was tedious and frustrating as minor changes to the HTML would lead to the whole compile, test, unload running servlet, reload new servlet situation. It also meant that you could not feasibly use an HTML Editor of any form to actually develop your servlets.

The concepts for JSP were introduced in the initial development of Jeeves, the Java Web Server. The idea was to reverse the process, instead of focusing on Java code with embedded HTML, the focus was changed to HTML with embedded Java code. The HTML page could then be developed by graphics designers as necessary and then the Java developer could take over and make the page actually work as required. The tedious process of unloading and reloading servlets was taken over by the JSP aware server which would detect changes in the JSP and cause a recompliation of the JSP into a servlet and unload/reload itself.

This idea wasn't new - the Microsoft camp had come across the idea and enshrouded it into Active Server Pages (ASP) which Microsoft first introduced in IIS 3.0. ASP performed terribly in 3.0, but with the release of IIS 4.0, ASP became the central focus for the development of dynamic content on the server side in the Microsoft world, and most "Microsoft shops" focus on ASP development as it ties quite well into the Microsoft COM model.

Other dynamic content models

Other models for deploying dynamic content abound for the Internet - the strongest contender in times past being CGI. But there are others of note, CGI/Perl, PHP, Cold Fusion, JHTML, and of course, ASP.

Active Server Pages

JSP is the same basic concept as ASP - there are similarities:

- ASP uses ActiveX objects to provide external functionality and access to business objects. JSP uses JavaBeans.
- ASP focuses on MTS as its Transaction Server for making its solution scalable (but see http://www.esperanto.org.nz/esperanto.asp?inc=techtalk/MTSvsDCE.html). The focus of JSP is on EJB.
- ASP and JSP are essentially interpreted languages. ASP gets converted into pcode and JSP gets converted into Java Byte Code.
- ASP allows the creation of functions for seperating code, JSP uses Inner Classes or Declarations.

and there are differences:

- ASP uses Visual Basic Scripting or Java Script. JSP focussed on Java (but allows others)
- ASP is a “product” (which Chillisoft has copied for other platforms), JSP is an almost specification.

For many, it comes down to a question of personal preference. ASP has had the lead and many, many sites run ASP pages as it is a built in feature of Microsoft’s Internet Information Server.

Many people further examine JSP for a number of reasons

- the Java language is more powerful than Visual Basic Scripting/JavaScript for server side work,
- Java is the chosen development language for server side development (and perhaps client side development)
- JSP allows the “scripting” language and the language for the development of the business logic (beans) to be the same but still interact with business logic embedded in other languages via CORBA (if necessary).
- CORBA and/or EJB play a large part in an organisation’s technology architecture
- An organisation focuses on Unix, AS400 or mainframe architectures and there is greater support on those platforms for the JSP model. ASP doesn’t have the support of ActiveX on non-Windows platforms.
- Greater choice in the vendors - there were 11 vendors who offered JSP implementations at the time of writing.

JSP vs CGI

CGI has one major disadvantage in relation to other environments available (including ASP and JSP) - performance and maintenance of state.

Both ASP and JSP (and many of the other products available for performing similar functionality) hook directly into the web server (through DLLs or being dynamically linked in some other manner), and thus there is no need to go through the time consuming, resource intensive process of writing the request to a file or pipe, running a program and reading back the output. CGI web applications also tend to suffer from a lack of cohesiveness in their structure more than the other technologies as there is no focus with CGI technologies for tools that manage a group of pages as an "application".

The other problem of CGI is that of state management. With the JSP/Servlet host running all of the time, the host can store state easily and make sure that it is provided to the JSP page in an easy, effective manner. With CGI applications, if libraries are not already written or otherwise available, they have to be written to manage all of the state management mechanisms.

The future of JSP

The future of JSP is not entirely clearly mapped out. As of the time of writing, we are working with the JSP 1.0 Public Draft. This has introduced a number of things (particularly XML syntax and buffered output to help with redirection) and removed other items (particularly extension tags for HTML which has a number of developers wanting to stick with the 0.92 specification). There is an indication in the Public Draft that XML will become required in the 1.1 specification, and that JSP will be tied more heavily into the Enterprise Java architecture (J2EE - Java 2 Enterprise Edition) that Sun is working on. At present, there is no integration with any of the Enterprise features

Chapter 2 - QuickStart to JSP

Getting the reference implementation going

The reference implementation of JSP 1.0 is a good place to start. The reference implementation implements the basic web server capability that was in 0.92, which just provides you with a very simple web server running on a port of your choosing (defaulting to 8080).

Getting the reference implementation

The reference implementation exists at http://java.sun.com/products/jsp. It has a Windows and a UNIX version - I assume a Windows audience here. Once downloaded, unzip the file to wherever you want to run it from - I installed it on my E: drive, so it created a directory called E\JSP1.0 - I then renamed this to JSP to avoid having to type in the 1.0 when I cd'd into the directory.

To run it, you need to run the STARTSERVER.BAT batch file - but this assumes that you have the 1.2 JDK installed. If you don't have it, then go and get it. Make sure that the java.exe program is on your path.

There is a problem with the STARTSERVER.BAT file. It won't actually be able to compile your JSPs until you tell it where the location of your tools.jar file is (it is in the lib directory of your JDK 1.2 installation). You need to alter the STARTSERVER.BAT file to ensure that it adds it to the class path before running.

I use JBuilder 3, so my tools.jar file was located in C:\JBUILDER3\JAVA\LIB\TOOLS.JAR - which makes my STARTSERVER.BAT file look as follows:

```
@echo off
rem $Id: startup.bat,v 1.10 1999/04/22 23:48:27 akv
Exp $
rem Startup batch file for servlet runner.
rem This batch file written and tested under
Windows NT
rem Improvements to this file are welcome
if "%CLASSPATH%" == "" goto noclasspath
rem else
set _CLASSPATH=%CLASSPATH%
set
CLASSPATH=server.jar;servlet.jar;classes;examples
\WEB-
```

```
INF\jsp\beans;c:\jbuilder3\java\lib\tools.jar;%CL
ASSPATH%
goto next
:noclasspath
set _CLASSPATH=
set
CLASSPATH=server.jar;servlet.jar;examples\WEB-
INF\jsp\beans;c:\jbuilder3\java\lib\tools.jar
goto next
:next
rem echo Using classpath: %CLASSPATH%
java -Djsp.keepgenerated=false
com.sun.web.shell.Startup %1 %2 %3 %4 %5 %6 %7 %8
%9
rem clean up classpath after
set CLASSPATH=%_CLASSPATH%
set _CLASSPATH=
```

The other thing that you have to be careful about when running this batch file under Windows 95 is to increase the space available for environment variables. There are instructions in the README.TXT that cater for this problem.

Running the server

You should be able to start the server from Explorer without having to drop to a DOS prompt. If you are having trouble getting it going then it is best to drop to the DOS prompt and edit the STARTSERVER.BAT file and comment out the line that says @ECHO OFF - this will let you see what is going on. By default, the server will start on port 8080, looking something like:

```
JSDK WebServer Version 2.1
Loaded configuration from file:E:\jsp/default.cfg
endpoint created: :8080
```

when it is running.

You can now go to your browser and type http://localhost:8080 to bring up the index.html page (that exists in jsp/webpages/index.html).

Configuring the server

There are a number of parameters that you can change in the configuration file. The file is called DEFAULT.CFG and should look like this:

```
# $Id: default.cfg,v 1.9 1999/04/23 22:31:51 gonzo
Exp $
# Default configuration file for Project Tomcat
server.port=8080
```

```
server.hostname=
server.inet=
server.docbase=webpages
server.workdir=work
# All sections of the server's namespace are
defined in terms of
# a 'webapp'. There are two attributes that define
a webapp, 1)
# a mapping of where the webapp is placed into the
namespace of
# the server and 2) the base of the webapp. The base
can be composed
# of a simple string as shown below which indicates
a file
# relative to the current working directory (the
directory in which
# the server was started) or can be a full URL.
server.webapp.examples.mapping=/examples
server.webapp.examples.docbase=examples
# server.app.wombat.mapping=/wombat
# server.app.wombat.base=http://wombat.eng/apps/
wombat.web
```

There are a few key configurable areas here:

- **server.port=8080** - this specifies what port that the server will run on. With port on 8080, it means that your server will require a http://localhost:8080 to get to it. If you don't have any other web servers running, you can change this down to 80 which is the default for web servers, allowing you to use http://localhost to connect. If you find at any stage that the server doesn't want to run on port 8080, or you wish to run two web servers (each on different ports) you can change this value or run the server with the -p PORT# flag.

  ```
  STARTSERVER -P 80
  ```

  for example will start the server at port 80, overriding the information in the default.cfg file.
- **server.docbase=webpages** - this line specifies which directory (off the directory that you are running the server from) that the web pages are located. In this case, all of the web pages that come with the server for the examples and so forth are in the /jsp/webpages directory.
- **server.workdir=work** - this specifies the directory that the JSP engine will use to compile the JSP page from a JSP to a servlet. It also specifies the directory for storage of the classes which are compiled from the servlet code. Be careful this directory path *does not contain values that would not be valid Java identifiers*, as JSP servers often use their path as part of the package name. Do not, for example, make a directory

called 1.

- The next two items, **server.webapp.examples.mapping=/examples** and **server.webapp.examples.docbase=examples** are designed to create an alias path for the web server. It means that when the web server sees a request for /examples, go to the examples directory instead. For example, there is a file called /examples/jsp/cal/cal1.jsp - this request won't go to /jsp/webpages/examples/jsp/cal/cal1.jsp, it will go to /jsp/examples/jsp/cal/cal1.jsp - as the /examples is mapped to the examples directory. You can and should do this for your own work as well.

1. We will add in two lines:

```
server.webapp.esperanto.mapping=/esperanto
server.webapp.esperanto.docbase=esperanto
```

2. Unfortunately, that isn't all you need to do. We have created a "webapp" called esperanto, but we need to configure the Servlet engine to let it know what to do with the JSP extension tag. The examples directory does this with four files that exist under the Web-Inf directory. So we need to copy this directory (except for the JSP and SERVLETS directories) to the /jsp/esperanto directory. You should then end up with a sub directory called Web-Inf with four files in it: mappings.properties, mime.properties, servlets.properties and webapp.properties.

Finally, there is one more configuration option that configures the JSP engine rather than the web server and that is a command line flag:

```
-Djsp.keepgenerated=false
```

This option tells JSP to not keep the servlet source that was generated. It is worthwhile to make this true as we want to actually see what is being generated by the servlet. So change the STARTSERVER.BAT file to:

```
java -Djsp.keepgenerated=true
com.sun.web.shell.Startup %1 %2 %3 %4 %5 %6 %7 %8
%9
```

Examining the examples

There are two key aspects of HTML that make JSP (and any dynamic page generation mechanism) work:

- Inputs given when an HTML page is requested. This is given in the name=value[&name=value[...]] notation. For example:

  ```
  http://localhost:8080/
  req.jsp?searchName=bob&wantWorksFor=true
  ```

  This will cause the JSP page to be loaded with two parameters - searchName being equal to “bob” and wantWorksFor being “true”. This is a feature primarily of HTML Forms which we won’t cover here (see Appendix A) as it is covered in much better detail in many books and other resources. We assume in this text that you have a basic grasp of HTML Forms.
- Cookies - cookies are the ability for a web server to store some information on a clients machine, which gets presented back to the web server again. We cover the creation of your own cookies later on (??? where?), but take it as given that the Java Servlet engine uses Cookies to keep track of who is who so that we can provide “state” on the server side.

Example 1 - Whats the time?

This example is going to tell us what the current time is. It is a very simple example that does not make use of any input from the client and introduces one JSP tag.

1. Under the /jsp/esperanto directory create a directory called example1
2. In the example1 directory create a new file called jsp1.jsp - the easiest way to do this is right click on the directory, choose New, then Text Document. It gives you the option to rename it, so change it to jsp1.jsp and accept the fact that you are changing the extension.

3. Enter the following code (we explain the tags in a little while):

```
<H1>Time JSP</H1>
<%
java.util.Date dt = new java.util.Date(
System.currentTimeMillis() );
%>
<%=dt.getHours()%>:<%=dt.getMinutes()%>:<%=dt.ge
tSeconds()%>
```

/jsp/esperanto/example1/jsp1.jsp

4. Save it and load up your browser. Assuming your server is running on port 8080, type http://localhost:8080/esperanto/example1/jsp1.jsp
5. You should get back a page that looks similar to:

Example1 - Examining the Servlet generated

From the code shown below, we can see a number of things:

- The package name includes the full directory path of the location of the JSP. Remember the warning above about using directories that are not valid Java identifiers.
- The Servlet only imports what it needs to import - by specifying the full path to the Date class we got around importing java.util.Date (or java.util.*). We will show you how you can import packages later in the book.
- The JSP page compiler generated a servlet descended from HttpJspBase. This is a special class which takes over the processRequest/doGet/doPost mechanism used in Servlets and calls a single method - _jspService to process the request.
- There are a number of variables created (session, application, config, etc) that we won't discuss. One we will is the ***out*** object. The out object is created for the express purpose of sending data back down the wire to the client browser. It is a special form of a PrintWriter (similar in use to System.out) which actually talks to a buffered output buffer - rather than directly to the output stream. We will talk about the signifi-

cance of that when we talk about redirection & forwarding.


```
package E_0003a.jsp.esperanto.example1;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
import com.sun.jsp.runtime.*;
import java.beans.*;
import com.sun.jsp.JspException;

public class jsp1_jsp_1 extends HttpJspBase {

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

        HttpSession session = null;
        ServletContext application;
        ServletConfig config;
        JspWriter out = null;
        String  _value = null;
        try {
            application =
getServletConfig().getServletContext();
            config = getServletConfig();
            session = request.getSession(true);
            out = new
JspWriter(response.getWriter(), 8192);
           response.setContentType("text/html");
            out.write("<H1>Time JSP</H1>\r\n");
         java.util.Date dt = new java.util.Date(
System.currentTimeMillis() );
            out.write("\r\n\r\n");
            out.println(dt.getHours());
            out.write(":");
         out.println(dt.getMinutes());
            out.write("");
            out.println(dt.getSeconds());
            out.write("\r\n");
        } catch (Throwable t) {
            throw new JspException("Unknown
exception: "+t);
        } finally {
        }
        out.flush();
    }
}
```

/jsp/work/????esperanto/jsp1_jsp.java - generated file

- Perhaps one last key thing to note is that the servlet will throw a new JspException if something goes wrong. We will cover

this more in Error Pages for Exception Handling

Now it is time to explain a little of what we did. Notice that there are no tags inside the JSP file to indicate that it is a JSP file, that is all done with the Servlet engine recognising that the .jsp suffix is to be processed by the JSP engine (it is specified in the servlets.properties file in Web-Inf - the line is jsp.code=com.sun.jsp.runtime.JspServlet).

We did use a couple of tags though, so lets explain what those are:

<% %> - JSP Scriplets

JSP allows you to embed arbitrary blocks of Java code inside your JSP files. These blocks of Java code get added directly into the servlet code that is to be generated. This means that you can get compile time errors as well as runtime errors in your JSP page.

The scriptlet tags start with <% after which any valid Java code can appear. The code can be multi line, and it ends with %>

These tags must start and end in the same file - you cannot issue an include (see later) which may open a scriptlet and then close back in your main code.

XML Equivalent: <jsp:scriptlet> Java Code </jsp:scriptlet>

<%= %> - JSP Expressions

JSP Expressions are a quick mechanism to avoid having to write the full code for out.println. The <%= is a replacement for the out.println statement and it is directly converted into that.

XML Equivalent: <jsp:expr> Java Code </jsp:expr>

Given that we now know the syntax for XML, we can give the equiva-

```
<?xml version="1.0" ?>
<H1>Time JSP</H1>
<jsp:scriptlet>
java.util.Date dt = new java.util.Date(
System.currentTimeMillis() );
</jsp:scriptlet>
<jsp:expr>dt.getHours()</jsp:expr>:
<jsp:expr>dt.getMinutes()</jsp:expr>:
<jsp:expr>dt.getSeconds()</jsp:expr>
```

/jsp/esperanto/example1/jsp2.jsp

lent XML form of the same JSP file. As you can see, it is significantly more wordy.

Why XML?

XML has been introduced into the JSP 1.0 specification (it wasn't in earlier versions) because Sun want to provide a standard mechanism for tag extensibility and for page developers to be able to use the ever widening range of XML tools. It is envisioned that JSP will eventually be able to be done with a smart, JSP aware software tool. Unfortunately, the JSP 1.0 public draft does not come with a DTD (Document Type Definition) which makes it difficult to code for by hand.

JSP 1.0 does specifically cater for non-XML tags primarily because JSP is still at an early stage in its acceptance by the developer community. Most pages require hand coding, and the use of XML tags with their cumberson CDATA mechanisms for inserting program code mean that it is important for the specification to enable those not using XML editor.

That said, the current JSP tags tend to confuse such editors (for example, they change <% into <%) so you have to use non-graphical editors for JSP, like vi, emacs, HomeSite or Notepad.

Example 2 - Adding two numbers

The following example will enable you to add two numbers together. It uses HTML forms

An introduction to useBean

Typical use of useBean - forcing a login