JSPs[1]

As we know, servlets, replacing the traditional CGI technology, can do computation and generate dynamic contents during run-time. However those many `out.println()`, used to produce HTML code, make coding a servlet tedious and awkward. *JavaServer Pages* technology was introduced in this situation to bring simplicity while retaining the capability of customizing web pages dynamically. A simple example have been introduced before to show this approach.

1 Using JSP

Basically, JSPs are HTML files with embedded Java servlet code. Each block of servlet code is surrounded by a leading `<%` tag and a closing `%>` tag. That's HTML code presents the static part while servlet code is in charge of customization. The following gives an example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html><head><title>Hello from a JSP</title></head>
<body>
  <p>Time on our server is <%= new java.util.Date() %></p>
  <h1>Hello from a JSP</h1>
  <p>The rest of the actual HTML page would be here...</p>
</body>
</html>
```

JSPs however are not limited to the above trivial computation. Similar to servlets, they can do much more complicated work based on data submitted by the client. To make the manipulation convenient, a *scriptlet* can use a number of predefined variables. The six most frequently used variables are:

`HttpServletRequest request`

The servlet request.

`HttpServletResponse response`

The servlet response.

[1] This note contains materials from *Java Servlet Programming*, 2nd Edition, O'Reilly.

`javax.servlet.jsp.JspWriter out`

The output writer, used like a `PrintWriter` but it has different buffering characteristics.

The package `javax.servlet.jsp` is a part of Java servlet API, containing classes and interfaces relating to JSP support.

`HttpSession session`

The user's session.

`ServletContext application`

The web application.

`javax.servlet.jsp.PageContext pageContext`

An object primarily used to abstract the implementation of the server but sometimes used directly to share variables between JSP pages and supporting beans and tags.

The following gives an example showing how some of the above variables are used.

```
<HTML>
  <HEAD><TITLE>Hello</TITLE></HEAD>
  <BODY>
    <H1>
    <%
      if (request.getParameter("name") == null) {
         out.println("Hello World");
      }
      else {
        out.println("Hello, " + request.getParameter("name"));
      }
    %>
    </H1>
  </BODY>
</HTML>
```

If you have successfully installed Tomcat on your system, simply save the above page in `hello.jsp` and place it under `$CATALINA_HOME/webapps/ROOT`. Then visit

```
http://host/hello.jsp?name=Jinzhong
```

you will see the page as shown in Figure 1.

Systematically speaking, JSPs may include servlet code in three formats: *Expressions*, *Declarations*, and *Scriptlets*.

A JSP expression begins with `<%=` and ends with `%>`. Any Java expression between the two tags is evaluated, the result is converted to a String, and the text is included directly in the

Figure 1: Saying "Hello" with JSP

page. This technique eliminates the clutter of an `out.println()` call. For example, `<%= foo %>` includes the value of the `foo` variable.

A declaration begins with `<%!` and ends with `%>`. In between the tags, you can include any servlet code that should be placed **outside** the servlet's `service` method. You may declare `static` or instance variables or define new methods. The following example demonstrates with a JSP page that uses a declaration to define the `getName()` method and an expression to print it. The comment at the top of the file shows that JSP comments are surrounded by `<%-- --%>` tags.

```
<%-- hello2.jsp --%>
<HTML>
<HEAD><TITLE>Hello</TITLE></HEAD>
<BODY>
<H1>
Hello, <%= getName(request) %>
</H1>
</BODY>
</HTML>

<%!
private static final String DEFAULT_NAME = "World";

private String getName(HttpServletRequest req) {
  String name = req.getParameter("name");
  if (name == null)
    return DEFAULT_NAME;
  else
    return name;
}
%>
```

This JSP behaves the same as the above `hello.jsp`, which is otherwise an excellent example of scriptlets, meaning blocks of code that may be inserted into `service` method to produce HTML code.

2 Behind the Scenes

How does JSP work? Behind the scenes, the server automatically converts a JSP into a servlet, and then creates, compiles, loads, and runs it to generate the page's content. The whole process is illustrated in Figure 2. To give you a sense of the resulted servlet, the following is the one

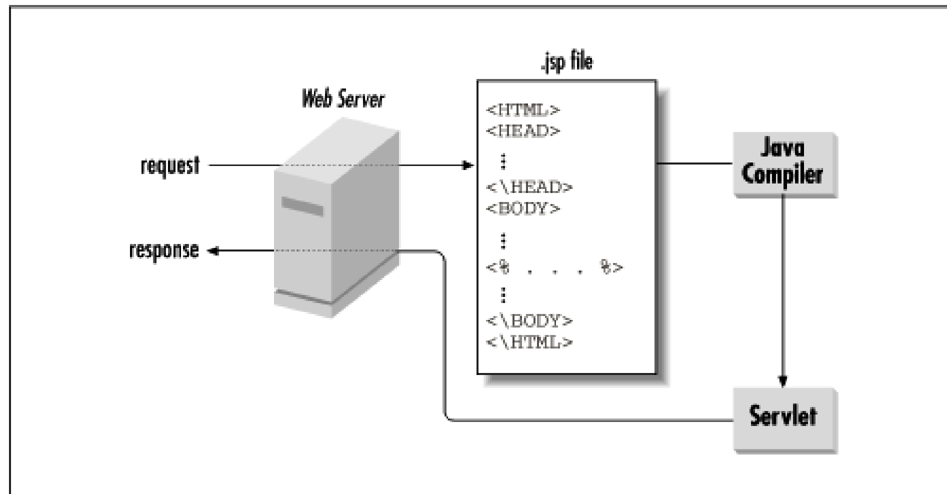

Figure 2: Converting JSP to servlet

generated automatically for `hello.jsp`:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import org.apache.jasper.runtime.*;
import org.apache.jasper.*;

public class _0002fhello_00031_0002ejsphello_jsp_0 extends HttpJspBase {
    private static boolean _jspx_inited = false;

    public _0002fhello_00031_0002ejsphello_jsp_0() { }

    public final void _jspx_init() throws JasperException { }

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

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
```

```
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {
            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                        "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\hello.jsp";from=(0,0);to=(4,0)]
            out.write("<HTML>\r\n<HEAD><TITLE>Hello</TITLE></HEAD>\r\n<BODY>");
            out.write("\r\n<H1>\r\n");
            // end

            // begin [file="C:\\hello.jsp";from=(4,2);to=(11,0)]
            if (request.getParameter("name") == null) {
               out.println("Hello World");
            }
            else {
              out.println("Hello, " + request.getParameter("name"));
            }
            // end

            // HTML // begin [file="C:\\hello.jsp";from=(11,2);to=(15,0)]
            out.write("\r\n</H1>\r\n</BODY></HTML>\r\n\r\n");
            // end
        } catch (Exception ex) {
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

The conversion causes the so-called *first-person penalty* since it always happens at the first

time when the JSP is accessed and may cause a significant delay of response. Subsequent requests should be as fast as ever because the server can reuse the servlet in memory.

You can avoid the first-person penalty by pre-compiling your JSPs. This can be done manually by making a request to each JSP page passing a query string of `jsp_precompile="true"`. The request causes the server to compile the JSPs, but the special query string tells the server not to bother passing the request through to the JSP for handling.

3 Directives

Besides servlet code, JSPs can use *directives* to impose some control upon the request manipulation, which is similar to server directive in the traditional CGI specification.

Directives can be used to tell the generated servlet to set a content type, import a package , control its output buffering and session management, extend a different superclass, and give special handling to errors.

The directive syntax requires a directive name along with an attribute name/value pair, all surrounded by `<%@ %>` tags. The quotes around the attribute value are mandatory:

```
<%@ directiveName attribName="attribValue" %>
```

The `page` directive allows the JSP to control the generated servlet through the setting of special attributes including:

`contentType`

Specifies the content type of the generated page. For example:

```
<%@ page contentType="text/plain" %>
```

The default content type is `text/html; charset=8859_1`.

`import`

Specifies a list of classes and packages the generated servlet should import. Multiple classes can be given in a comma-separated list. For example:

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

The implicit include list is:

```
java.lang.*,javax.servlet.*,javax.servlet.http.*,javax.servlet.jsp.*
```

More directives can be found at http://develop.levity.com/ref/jsp-directives.xtp.

4 JSP and JavaBeans

One of the most interesting and powerful ways to use JSPs is in cooperation with JavaBeans components.

4.1 What are JavaBeans?

JavaBeans is a component technology, comparable to Microsoft's ActiveX. JavaBeans components are reusable Java classes whose methods and variables follow specific naming conventions to give them added abilities. Those conventions make it possible for JavaBeans to be manipulated in a visual builder tool and composed together into applications based on event-driven model and other design patterns. For example, the following gives a Java bean called `MagicBean`, in which a variable `magic` is defined. To comply with JavaBeans specification, `MagicBean` must include two methods for the variable to be read and written, `setMagic`/`getMagic`.

```
/*
A simple bean that contains a single "magic" string.
*/
public class MagicBean {

  private String magic;

  public MagicBean(String string) {
    magic = string;
  }

  public MagicBean() {
    magic = "Woo Hoo"; // default magic string
  }

  public String getMagic() {
    return(magic);
  }

  public void setMagic(String magic) {
    this.magic = magic;
  }
}
```

You may go to http://java.sun.com/ for more details about JavaBeans technology.

4.2 Embedding a Bean

Beans are embedded in a JSP page using the `<jsp:useBean>` action. It has the following syntax (case sensitive, and the quotes are mandatory):

```
<jsp:useBean id="name" scope="page|request|session|application"
             class="className" type="typeName">
</jsp:useBean>
```

You can set the following attributes of the `<jsp:useBean>` action:

`id`

Specifies the name of the bean. This is the key under which the bean is saved if its scope extends beyond the page. If a bean instance saved under this name already exists in the given scope, that instance is used with this page. Otherwise a new bean is created. For example:

```
id="userPreferences"
```

`scope`

Specifies the scope of the bean's visibility. The value must be `page`, `request`, `session`, or `application`. If `page`, the variable is created essentially as an instance variable; if `request`, the variable is stored as a request attribute; if `session`, the bean is stored in the user's session; and if `application`, the bean is stored in the servlet context. For example:

```
scope="session"
```

The default value is `page`.

`class`

Specifies the class name of the bean. This is used when initially constructing the bean. The class name must be fully qualified. For example:

```
class="com.company.tracking.UserPreferencesImpl"
```

The class attribute is not necessary if the bean already exists in the current scope, but there must still be a type attribute to allow the system to cast the object to the proper type. If there's a problem constructing the given class, the JSP page throws an `InstantiationException`.

`type`

Specifies the type of the bean as it should be held by the system, used for casting when the object is retrieved from the request, session, or context. The type value should be the fully qualified name of a superclass or an interface of the actual class. For example:

```
type="com.company.tracking.UserPreferences"
```

If unspecified, the value defaults to be the same as the `class` attribute. If the type does not match the object's true type, the JSP page throws a `ClassCastException`.

`beanName`

The `class` attribute may be replaced by a `beanName` attribute. The difference between the two is that `beanName` uses `Beans.instantiate()` to create the bean instance, which looks for a serialized version of the bean (a `.ser` file) before creating an instance from scratch. This allows the use of preconfigured beans. See the `Beans.instantiate()` Javadoc documentation for more information. For example:

```
beanName="com.company.tracking.UserPreferencesImpl"
```

The body of the `<jsp:useBean>` element – everything between the opening `<jsp:useBean>` and the closing `</jsp:useBean>` – is interpreted after the bean's creation. If the bean is not created (because an existing instance was found in the given scope) then the body is ignored. To demonstrate:

```
<jsp:useBean id="prefs" class="com.company.tracking.UserPreferencesImpl">
```

If you see this we must have created a new bean! `</jsp:useBean>` If no body is needed, the XML shorthand for an empty tag `/>` can be used:

```
<jsp:useBean id="prefs" class="com.company.tracking.UserPreferencesImpl" />
```

4.3 Controlling Bean Properties

The `<jsp:setProperty>` action provides the ability for request parameters to automatically (via introspection) set properties on the beans embedded within a page. This gives beans automatic access to the parameters of the request without having to call `getParameter()`. This feature can be used in several ways. *First*:

```
<jsp:setProperty name="beanName" property="*" />
```

This says that any request parameter with the same name and type as a property on the given bean should be used to set that property on the bean. For example, if a bean has a `setSize(int size)` method and the request has a parameter `size` with a value of 12, the server will automatically call `bean.setSize(12)` at the beginning of the request handling. If the parameter value can't be converted to the proper type, the parameter is ignored. (Note that an empty string parameter value is treated as if the parameter does not exist and will not assign an empty string value to a `String` property.)

Notice that the action uses the XML shorthand for an empty tag. This is very important. The JSP specification requires all JSP actions to be well-formed XML, even when placed in HTML files. Here's the *second* `<jsp:setProperty>` usage:

```
<jsp:setProperty name="beanName" property="propertyName" />
```

This says that the given property should be set, if there's a request parameter with the same name and type. For example, to set the `size` property but no other property, use this action:

```
<jsp:setProperty name="prefs" property="size" />
```

Here's a *third* `<jsp:setProperty>` usage:

```
<jsp:setProperty name="beanName" property="propertyName" param="paramName" />
```

This says that the given property should be set, if there's a request parameter with the given name and the same type. This lets a parameter with one name set a property with another name. For example, to set the `size` property from the `fontSize` parameter:

```
<jsp:setProperty name="prefs" property="size" param="fontSize" />
```

This is the *final* usage:

```
<jsp:setProperty name="beanName" property="propertyName" value="constant" />
```

This says that the given property should be set to the given value, which will be converted to the appropriate type if necessary. For example, to set the default size to 18 with a parameter override:

```
<jsp:setProperty name="prefs" property="size" value="18" />
<jsp:setProperty name="prefs" property="size" param="fontSize" />
```

For advanced users, the value attribute does not need to be a constant; it can be specified using an expression with what's called a *request-time attribute expression*. For example, to ensure the size never drops below 6:

```
<jsp:setProperty name="prefs" property="size" param="fontSize" />
<jsp:setProperty name="prefs" property="size"
    value="<%= Math.max(6, prefs.getSize()) %>" />
```

Finally, the `<jsp:getProperty>` action provides a mechanism for retrieving property values without using Java code in the page. Its usage looks much like that of `<jsp:setProperty>`:

```
<jsp:getProperty name="beanName" property="propertyName" />
```

This says to include at this location the value of the given property on the given bean. It's longer to type than an expression but eliminates the need to place Java code in the page. Whether to use `<jsp:getProperty>` or an expression is a matter of personal taste. Note that `<jsp:getProperty>` embeds the property value directly, so if it contains characters that HTML treats as special, that will cause problems.

4.4 Saying “Hello” Using a Bean

The following demonstrates the use of a JavaBeans component with a JSP page; it says “Hello” with the help of a `HelloBean`.

```
<%-- hello.jsp --%>

<%@ page import="HelloBean" %>

<jsp:useBean id="hello" class="HelloBean">
  <jsp:setProperty name="hello" property="*" />
</jsp:useBean>

<HTML>
<HEAD><TITLE>Hello</TITLE></HEAD>
<BODY>
<H1>
Hello, <jsp:getProperty name="hello" property="name" />
</H1>
</BODY>
</HTML>
```

As you can see, using a JavaBeans component with JavaServer Pages can reduce the amount of code placed into the page. The `HelloBean` class contains the business logic for determining a user’s name, while the JSP page acts only as a template.

The code for `HelloBean` is shown as follows. Its class file should be placed in the standard directory for support classes (`WEB-INF/classes`).

```
public class HelloBean {
  private String name = "World";

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

5 Includes and Forwards

Through a combination of directives and actions, JSP provides support for includes and forwards. There are two kinds of include, and one kind of forward.

The first include is an `include` directive:

```
<%@ include file="pathToFile" %>
```

This include occurs at *translation time* which means it happens during the creation of the background servlet. All content from the external file is included as if it were typed into the JSP page directly. For C programmers, this works like a `#include`. The file path is rooted at the context root (so `/index.jsp` refers to the `index.jsp` for the current web application) and the path may not extend outside the context root (don't try `../../../otherContext`). The content within the file is typically a page fragment, not a full page, so you may want to use the `.inc` or `.jin` file extension to indicate this. The following JSP page uses several include directives to construct a page from a set of components (the components aren't shown):

```
<%@ include file = "/header.html" %>
<%@ include file = "/nameCalculation.inc" %>
Hello, <%= name %>
<%@ include file = "/footer.html" %>
```

Where does the `name` variable come from? It's created by the `nameCalculation.inc` file. Because the included file's content is included directly, there's no separation of variable scope.

The second kind of include is a `<jsp:include>` action:

```
<jsp:include page="pathToDynamicResource" flush="true" />
```

This include occurs at request time. It's not a raw include like the `include` directive; instead the server executes the specified dynamic resource and includes its output into the content sent to the client. Behind the scenes the `<jsp:include>` action generates code that calls the RequestDispatcher's `include()` method, and so the same usage rules apply for `<jsp:include>` as apply for `include()` – the included page must be dynamic, cannot set the status code, and cannot set any headers, and the path to the page is rooted at the context root. The `flush` attribute is required and in JSP 1.1 must always be set to `true`. This indicates the response buffer should be flushed before the include takes place. Given the capabilities of Servlet API 2.2, a value of `false` cannot be supported; it's expected JSP 1.2 built on Servlet API 2.3 will allow a value of `false`.

As an example, the following `<jsp:include>` action includes the content generated by a call to the `greeting.jsp` page. The `greeting.jsp` page could look a lot like `hello.jsp`, except you'd want to remove the surrounding HTML that makes `hello.jsp` a complete HTML page:

```
<jsp:include page="/greeting.jsp" />
```

The `<jsp:include>` directive can optionally take any number of `<jsp:param>` tags within its body. These tags add request parameters to the include request. For example, the following action passes the greeting a default name:

```
<jsp:include page="/greeting.jsp">
  <jsp:param name="defaultName" value="New User" />
</jsp:include>
```

Finally, a JSP may forward a request using the `<jsp:forward>` action:

```
<jsp:forward page="pathToDynamicResource" />
```

The forward causes control of the request handling to be passed to the specified resource. As with the `<jsp:include>` action it's executed at request time and is built on the RequestDispatcher 's `forward()` method. The `forward()` usage rules apply – at the time of the forward all buffered output is cleared and if output has already been flushed, the system throws an exception. The following action forwards the request to a special page if the user is not logged in:

```
<% if (session.getAttribute("user") == null) { %>
  <jsp:forward page="/login.jsp" />
<% } %>
```

The `<jsp:forward>` action accepts `<jsp:param>` tags as well.