Using Java Server Pages – Servlets made simple

Kyle Brown
IBM WebSphere Services
Gary Craig
Superlative Software Solutions, Inc.

In the previous article we saw at a high level how Java Servlets and Java Server Pages (JSP's) can participate in a logical three-tier, MVC architecture. We introduced the Servlet API and showed briefly how servlets can provide a server-side Java interface to the web. Now, astute readers may have already noticed what some perceive as the Achilles heel of the basic servlet approach – namely that it doesn't give a proper separation of content from presentation. When used alone, servlets both directly encode the HTML formatting and layout tags (presentation) of information as well as the dynamic content that they obtain by interacting with Enterprise Java Beans or other back-end Java code. This leads to servlets that require constant maintenance as the presentation of a web site changes, simply because servlet programmers must continually update the HTML tags within their servlets to keep up with new site standards and layout.

Luckily, Sun and IBM have proposed a better solution to this problem in the notion of Java Server Pages (JSP). A JSP is a file that contains extended HTML tags that allow embedding dynamic content (e.g. Java code and special server-side HTML tags) within standard HTML. In this way, we can develop the presentation of information using any standard HTML editor (like NetObjects Fusion or Microsoft FrontPage). This not only allows the flexibility to more cleanly separate the back-end generation of dynamic content from the presentation in HTML, but also permits the two roles (writing HTML and writing Java) be split among different team members, each with complementary skill sets. JSP's are supported by the IBM WebSphere Application Server, and by Sun's Java Web Server. As the standard matures, other application server vendors will probably also announce JSP support.

To understand how all of this works, let's examine the JSP specification and see what it adds to standard HTML, and take a look at the JSP architecture and see how JSPs are used to generate server-side Java.

JSP basics

Let's begin by looking at a simple JSP and discussing how the JSP architecture will make it work at runtime. Take a look at the following file (sample.jsp):

```
<HTML><HEAD><TITLE>Today's Date</TITLE></HEAD>
<BODY> Today's date is <%= new Date() %>. </BODY></HTML>
```

At first glance this appears to be just standard HTML. However, the key to understanding the difference lies in the tag that follows "Today's date is". That tag, <%= new Date() %> is called an *expression.* The syntax is simple:

<%= *Any legal Java expression* %>

When a client browser requests the file, sample.jsp, from a WebServer or Application Server that is able to serve a JSP, the result will be that the expression will be evaluated (e.g. a new Date object representing today's date is created), and the java.lang.String representation of that object embedded in the HTML that is returned to the client.

Similar to an expression is a *scriptlet*. A scriptlet has nearly the same syntax as an expression, but the semantics are slightly different. While an expression says "evaluate this Java code and embed the output in the HTML" a scriptlet just says "evaluate this Java code". The syntax looks like the following:

```
        <% Any legal Java code %>
```

A simple example of using the two tags together shows how calculated values can be used:

```
<HTML><HEAD><TITLE>Welcome</TITLE></HEAD>
<%
        Date today  = new Date();
        Date tomorrow =new Date(today.getTime() + (long) (24*60*60*1000));
%>
<BODY>This page is here on <%= today %> and will be gone by  <%= tomorrow %>.
</BODY></HTML>
```

These two types of tags, scriptlets and expressions, form the basis of the JSP model. However, we haven't yet seen how these tags are executed, or how they relate to servlets. To understand that, we need to take a look at the pagecompile process and understand how web servers and application servers that support JSPs utilize them.

Turning tags to code

The key to JSPs lies in a special servlet called the pagecompile servlet. In short, the pagecompile servlet does the following things:

(1) When a webserver receives a request for a page ending in .jsp, it turns that request over to the pagecompile servlet.
(2) The pagecompile servlet then compares the page to a list of pages it has cached. If the page is a new request, then the pagecompile servlet generates a new servlet class. This servlet class will interleave sending output from a buffer of all non-JSP specific HTML tags from a buffer with the execution of the Java code contained in the JSP tags.
(3) The pagecompile servlet then compiles this page into a Java class, loads the servlet and turns over execution to the service() method of the new servlet. It finally adds the new servlet to its list of cached pages. On all subsequent requests, processing will be immediately turned over to the service() method of the cached servlet.

Understanding how this process works also allows you to understand the other tags in the JSP specification. For instance, there is the directive tag, whose syntax looks like the following:

```
<%@ (directive = value)+ %>
```

The directive argument is one of Language, Method, Extends, Implements, or Import. A directive tag has the scope of the entire servlet class generated by the pagecompile process. For instance, if a JSP file includes the directive

```
<%@ implements = "java.io.Serializable" %>
```

Then the generated servlet will implement the java.io.Serializable interface. Likewise if it includes the tag

```
<%@ import = "java.io.*" %>
```

Then the generated servlet will import the public classes in the java.io package. The method directive is special in that it directs the pagecompile servlet to generate either a doPost() or doGet() method if the argument is set to “POST” or “GET” respectively instead of a service() method. Finally, the language directive is currently only allowed to have the value “Java”, indicating that Java is the only language supported by the generated code. Since this is the default, this tag is not used in JSPs.

JSP problems

So far, we’ve seen how JSP tags can be used to add server-side Java functionality directly to an HTML page, making Servlet writing easier in the case where the amount of HTML to be output greatly outweighs the amount of Java code that needs to be executed. However, this leads to a problem. By allowing the direct embedding of arbitrary Java code into an HTML page, we have inadvertently led to violating the rule of separating presentation from content.

Most organizations doing significant web development are likely to have people fulfilling at least two distinct roles in developing applications. Web Page Designers will be mostly concerned with page layout and graphic arts. Their primary tools set may include tools like NetObjects Fusion, Microsoft FrontPage, or HotMetal. This is a separate skill set from a Web Page Developer, who is more skilled at Java and Javascript, and who uses a different tool set for Java development. The two roles meet in the JSP. This can lead to conflict – either both will be working on the same page, leading to possible source code management problems, or one or the other will have to take primary responsibility for the JSP pages – leading them to potentially perform tasks outside of their skill set.

Another problem with this is that as the amount of Java code in a JSP grows, it defeats the purpose of OO programming in general – to facilitate reuse by proper factoring of responsibility. Since the major reuse facility provided for JSPs is copy-cut-and-paste programming, duplicated code will abound, with the result that errors can be propagated from page to page and become difficult to track down and fix.

Again, Sun and IBM have come to the rescue with a way out of this predicament. There is another set of JSP tags that are specifically aimed at making common JSP use scenarios simple. They do this by facilitating communication between a JSP and other servlets through the mechanism of JavaBeans. To understand how this happens let’s examine the <BEAN> tag and see how it works.

Connecting Servlets & JSPs with the <Bean> tag

The JSP <BEAN> tag[1] provides a way for a JSP to access a JavaBean. There are a number of scenarios supported by the <BEAN> tag. One common scenario involves accessing an existing JavaBean. This JavaBean can have a lifetime limited to the current HTTP request or the current Session[2]. To access this Bean, the JSP needs to know:

1. Where to locate the Bean (its scope)
2. What key is to used to access the Bean (its name)
3. The Java type the Bean conforms to

An example Bean tag following this scenario is:

```
<BEAN name=beanKey type=com.abc.beanType scope=session></BEAN>
```

[1] The <BEAN> tag from the JSP 0.91 specification has been renamed <USEBEAN> in the JSP 1.0 specification

[2] The JSP Specification Version 1.0 specifies three distinct lifetimes, Request, Session, and Application. In the 0.91 version of the Specification, only two lifetimes are discussed and they are referred to as the “scope” of the Bean, rather than lifetime.

Here the JSP accesses a Bean of type com.abc.beanType by retrieving it using the key, beanKey, from the current HttpSession object. Alternatively, the Bean could have been retrieved from the HttpServletRequest context by specifying scope=request.

A second common scenario instantiates a Bean and places it into a scope. To create and access a Bean this way, the JSP needs to:

1. Specify the Java type the Bean conforms to
2. Specify that the Bean is to be created if not found in the specified scope (create=yes)
3. Provide the filename from which the Bean is to be instantiated
4. Indicate whether the Bean's properties are to be set from parameters in the current request object
5. Provide a key used to store the Bean in the specified scope

An example Bean tag for this scenario is:

<BEAN create=yes type=com.abc.beanType beanName=beanFilename introspect=no name=beanKey scope=request><PARAM propertyA="1234"></BEAN>

Here the beanFilename is used to instantiate a Bean of type com.abc.beanType. The Bean is subsequently stored in the request object as an attribute under the key beanKey.

This <BEAN> tag might result in the following Java code (this code would typically appear in the service() method of the Servlet which is generated from the JSP file).

```
com.abc.beanType beanKey =
                      (com.abc.beanType)request.getAttribute("beanKey");
//scenario assumes the above fails but the code will be generated anyway
if (beanKey == null) {
  // instantiate Bean from "beanFilename"
  beanKey = (com.abc.beanType)Beans.
         instantiate(this.getClassLoader(), "beanFilename");
  if (beanKey == null) throw new ServletException(…);
  // store Bean in request scope
  ((com.sun.server.http.HttpServiceRequest)request).
                           setAttribute("beanKey", beanKey);
}

// now set property from PARAM
beanKey.setPropertyA("1234");
```

The scope of a <BEAN> defines the *lifetime* of the Bean. When the enclosing object (HttpSession or HttpServletRequest) goes out of scope, the Bean will become available for garbage collection. If the content of the Bean is applicable only for the current request, the Bean should be managed within the request scope. Beans whose content is applicable across requests within a Session should be managed within the session scope.

Servlets and JSP's

Using JavaBeans in this way permits delivery of dynamic content to the JSP. Using JavaBeans allows encapsulation of server-side business logic with reusable components. Generally, the interaction controller (servlet) will select one or more JavaBeans to provide to the view layer. For the JSP, the available dynamic content is represented through the "type" of the JavaBean.

The type represents a contract between the supplier of the JavaBean and the Bean *consumer*, the JSP. JavaBeans are tool friendly and this "contract" of available Properties and Services is easily viewable through Bean introspection. This enhances the ability for different tools (used by varying *Roles* in the organization) to participate in the development of the Web Application. The participation may occur little interaction between the developers other than the communication of expectations (API) as expressed by this Bean contract.

The following scenario illustrates how this would be used: Suppose that our hypothetical web site for downloading articles included a page for reviewing the articles that you've paid for. There are two parts to this page – the static content (the layout of the page, headers, banners, etc.) and the dynamic content (the particular list of articles for a user). In our architecture, the initial URL that represents that page would be that of a servlet that acts as a Controller.

Our controller would first look for certain preconditions – Is the user logged in? Do they have access to this particular function? If these preconditions are not met they would be directed to one or more error pages – this is part of the Mediation aspect of a Controller servlet discussed in the previous article. If the preconditions are fulfilled, then the servlet would contact some back-end server logic to obtain the list of articles (perhaps using EJBs). It would then format this information into a JavaBean and invoke a JSP that can display the list. This is the heart of the second role of the Controller servlet – adapting the underlying data (from the server logic) to the specification of a display bean needed by the JSP.

The mechanics of how this is done are very simple – first the JavaBean is placed in the appropriate context (either in the HttpSession, or in the HttpServletRequest) and then the JSP is called through the HttpServletResponse method callPage(String, HttpServletRequest). The callPage() method simply performs some necessary housekeeping, and then locates the servlet corresponding to the file name passed as the first parameter and calls its service() method. The generated code we've seen earlier would then be called, and the cycle is complete.

The two sides of this contract are the type of JavaBean that the servlet creates, and the corresponding type information in the <BEAN> tag of the displaying JSP. The underlying representation of the bean is immaterial to the JSP, as is the way in which the bean is displayed is immaterial to the servlet. Only the specification of the type is needed to join the two parts. This distinction is crucial for division of labor in assigning roles in an organization, and for allowing each of the two parts to survive changes made to the other.

Where JSPs are going – the JSP 1.0 Specification

This last notion – that the division between servlets (which act as controllers) and JSP's (which act as views) is due to a division in role between page designers (who build HTML pages) and web developers (who write Java code for Controllers and Models) is crucial to understanding some of the new features that are emerging in the 1.0 version of the JSP specification[3]. For instance, there are two new tags called <DISPLAY> and <LOOP> that are used to embed String representations of bean properties and iterate through indexed properties respectively. While a Java programmer would be perfectly comfortable performing these tasks through Java code in an expression or scriptlet, someone whose primary knowledge is of HTML might not be as facile at writing the code to do this. These more HTML-like tags (which grew out of two similar tags named <INSERT> and <REPEAT> that are supported in the current version of IBM WebSphere) are meant to facilitate these simple, yet common tasks by reducing the amount of Java coding knowledge necessary to write a JSP.

[3] This information is based on the 0.92 version of the specification, which was released for public comment in September, plus additional private conversations with Sun developers.

The greater purpose of these new tags is to make it easier for tools vendors to support the tags so that even someone without knowledge of Java would be able to write JSP's that use pre-existing JavaBeans. For instance, you can easily imagine a tool that would allow you to "drag" a JavaBean from a palette onto an HTML layout page and then pick properties for that JavaBean that would be displayed and "drop" them into appropriate places in the resulting HTML. Making these tasks automated will make it simpler to maintain the division of roles, and make the resulting systems more maintainable as a result due to the reasons cited in the previous section.

Summary

We've now seen the second leg of the MVC architecture laid out in the previous article. We've seen more clearly how JSPs are used as Views, and how Beans are used as Models. We've also introduced the notion of using servlets to call JSP's so that they can be used as Controllers. In the final installment of this series we'll examine the last part of the triad, Enterprise Java Beans. We'll also explore how they are used in the WebSphere Application Server to bring the same sort of rigor to business logic coding and distribution that we've seen JSP's and servlets bring to separation of presentation from business logic.