Technologies
− J2EE
− J2SE
− J2ME
− Web Services
− Wireless
− XML
− Other

Downloads
− Early Access

Documentation
− APIs
− Tutorials
− Code Samples
− See All

Industry News

Developer Services
− Bug Database
− Forums
− Support
− See All

Java BluePrints

Printable Page

Articles Index

JavaServer Pages™ : A Developer's Perspective
By Scott McPherson
April 2000

JavaServer™ Pages(JSP) technology provides an easy way to create dynamic web pages and simplify the task of building web applications that work with a wide variety of web servers, application servers, browsers and development tools. This article provides an overview of JSP from a developer's perspective, and includes examples of JSP in action.

Introduction

JavaServer Pages technology allows web developers and designers to easily develop and maintain dynamic web pages that leverage existing business systems. As part of the Java™ technology family, JSP enables rapid development of web−based applications that are platform−independent. JSP separates user interfaces from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.

So what exactly *is* a JavaServer Page? In its basic form, a JSP page is simply an HTML web page that contains additional bits of code that execute application logic to generate dynamic content. This application logic may involve JavaBeans™, JDBC™ objects, Enterprise Java Beans™ (EJB), and Remote Method Invocation (RMI) objects, all of which can be easily accessed from a JSP page. For example, a JSP page may contain HTML code that displays static text and graphics, as well as a method call to a JDBC object that accesses a database; when the page is displayed in a user's browser, it will contain both the static HTML content and dynamic information retrieved from the database.

The separation of user interface and program logic in a JSP page allows for a very convenient delegation of tasks between web content authors and developers. It also allows developers to create flexible code that can easily be updated and reused. Because JSP pages are automatically compiled as needed, web authors can make changes to presentation code without recompiling application logic. This makes JSP a more flexible method of generating dynamic web content than Java servlets, whose functionality JavaServer Pages extend.

JSP and Servlets

If you've worked with Java servlets you know that servlets allow you to create dynamically−generated web pages that include data from server−side Java objects. But you also know that the servlet approach to generating web pages is to embed HTML tags and presentation code within a Java class. This means that changes to presentation code requires modification and recompilation of the servlet source file. Because web authors who design HTML pages may not be the same folks as the developers who write servlet code, updating servlet−based web

applications can be an involved process.

Enter JavaServer Pages, which are an extension of the Servlet API. In fact, JSP pages are compiled into servlets before they are used, so they have all of the benefits of servlets, including access to Java APIs. Because JSP pages are generally presentation code with application logic embedded in them, they can be thought of as "inside−out" servlets.

While JSP pages mainly provide a higher−level method of creating servlets, they bring other benefits as well. Even if you're already content writing servlets for web applications, there are plenty advantages to using JSP:

- JSP pages easily combine static templates, including HTML or XML fragments, with code that generates dynamic content.
- JSP pages are compiled dynamically into servlets when requested, so page authors can easily make updates to presentation code. JSP pages can also be precompiled if desired.
- JSP tags for invoking JavaBeans components manage these components completely, shielding the page author from the complexity of application logic.

- Developers can offer customized JSP tag libraries that page authors access using an XML−like syntax.
- Web authors can change and edit the fixed template portions of pages without affecting the application logic. Similarly, developers can make logic changes at the component level without editing the individual pages that use the logic.

In general, JSP allows developers to easily distribute application functionality to a wide range of page authors. These authors do not have to know the Java programming language or know anything about writing servlet code, so they can concentrate on writing their HTML code while you concentrate on creating your objects and application logic.

Creating JSP Pages

At first glance, a JSP page looks similar to an HTML (or XML) page−−both contain text encapsulated by tags, which are defined between <angle brackets>. While HTML tags are processed by a user's web browser to display the page, JSP tags are used by the web server to generate dynamic content. These JSP tags can define individual operations, such as making a method call to a JavaBean, or can include blocks of standard Java code (known as *scriptlets*) that are executed when the page is accessed.

To see how this all happens, here's a sample JSP page that includes both static HTML content and dynamic data generated from a JavaBean. When the user accesses this page, it prints the current day of the month and the year, and adds a greeting based on the time of day (either "Good Morning" or "Good Afternoon"). For simplicity, JSP tags are shown in **bold**:

```
<HTML>
<%@ page language="java" imports="java.util.*" %>

<H1>Welcome</H1>

<P>Today is </P>
<jsp:useBean id="clock" class="jspCalendar" />
<UL>
<LI>Day: <%= clock.getDayOfMonth() %>
<LI>Year: <%= clock.getYear() %>
</UL>
<%-- Check for AM or PM --%>
<%! int time =
Calendar.getInstance().get(Calendar.AM_PM); %>
<%
if (time == Calendar.AM) {
%>
Good Morning
<%
}
else {
%>
Good Afternoon
<%
}
%>
<%@ include file="copyright.html" %>
</HTML>
```

The above example demonstrates the simplicity of JSP, and also offers a glimpse of the various components of a JSP page. These components include the following:

- **JSP actions** (or **JSP tags**) perform a variety of functions and extend the capabilities of JSP. JSP actions use XML−like syntax, and are used to (among other things) manage JavaBean components. In the sample page, a `jsp:useBean` action initializes a JavaBean that is used in subsequent portions of the page:

  ```
  <jsp:useBean id=="clock" class=="jspCalendar" />
  ```

  If the sample page had needed to get or set properties of this bean, other JSP actions would have been given, using the following syntax:

```
<jsp:getProperty name="bean" property="property" />
<jsp:setProperty name="bean" property="property"
value="value" />
```

- **Directives** are instructions that are processed by the JSP engine when the page is compiled to a servlet. Directives are used to set page−level instructions, insert data from external files, and specify custom tag

libraries. Directives are defined between <%@ and %>. In the above example, directives define the language of the page (Java), import the Java classes needed by the embedded code, and specify that the contents of an HTML file should be inserted at the bottom of the page:

```
<%@ page language=="java" imports=="java.util.*" %>
<%@ include file=="copyright.html" %>
```

- **Declarations** are similar to variable declarations in Java, and define variables for subsequent use in expressions or scriptlets. Declarations are defined between <%! and %>. In the above sample page an `int` is declared and given a value corresponding to the time of day (AM or PM):

```
<%! int time =
Calendar.getInstance().get(Calendar.AM_PM); %>
```

- **Expressions** are variables or constants that are inserted into the data returned by the web server, and are defined with the <%= and %>. In the sample page, expressions make calls on a JavaBean component and insert the resulting data into the page:

```
<%= clock.getDayOfMonth() %>
<%= clock.getYear() %>
```

- **Scriptlets** are blocks of Java code embedded within a JSP page. Scriptlet code is inserted verbatim into the servlet generated from the page, and is defined between <% and %>. A scriptlet in the above sample determines the time of day and greets the user accordingly:

```
<%
if (time == Calendar.AM) {
%>
Good Morning
<%
}
else {
%>
Good Afternoon
<%
}
%>
```

- **Comments** are similar to HTML comments, and are stripped from the page by the JSP engine when it is executed. This means that JSP comments are not returned to the user's browser. Unlike HTML comments, which are given between <!-- and --> tags, JSP comments are given between <%-- and --%>. For example:

```
<%-- Check for AM or PM --%>
```

Using Custom Tags

Although you can embed Java code within a JSP page to execute server−side processing, JSP also supports an alternative method of inserting dynamic content with *custom tags*, a mechanism that allows you to invent your own HTML−like tags for your JSP pages. In other words, your JSP pages can generate dynamic content using simple tag syntax instead of Java code. Custom tags are incredibly useful because they provide further separation of responsibilities between developers (who create the custom tags) and page authors (who use them).

Creating a custom tag is more complicated than using simple scriptlets in JSP pages, since custom tags require several steps to connect your JSP code to the business logic of your Java components. However, custom tags are also easier to distribute and reuse, and support for custom tags is being implemented in JSP authoring tools.

The following example shows a JSP page that generates dynamic content (in this case, today's lunch special) using a custom tag. Notice that in this case we don't need to import Java classes, declare variables, or write any Java code:

```
<HTML>
<%@ taglib uri="/tlds/menuDB.tld" prefix="menu" %>

<H1>Today's Menu</H1>

<P>Lunch</P>
<%@ include file="lunch_menu.html" %>

<P>Our Special of the Day</P>
<menu: insertCatchOfDay meal="lunch" >

</HTML>
```

The syntax for this page is obviously simpler than the scriptlet example shown the previous section, since it doesn't involve initializing objects and executing their methods. But the JSP page code is only part of the story; for each custom tag, the following three components are present:

- **JSP pages** that include the custom tag, such as the above code fragment that uses the `insertCatchOfDay` custom tag. Before using a custom tag, a page must specify the `taglib` directive to provide the location of the tag library descriptor (see below) that defines the tag. When executing a custom tag, a page typically defines one or more tag attributes (such as `meal` in this example) to determine dynamic content.
- A **tag library descriptor**, an XML file that defines the custom tag and connects it to its tag handler class. A tag library descriptor includes the various attributes of the tag, the name (and location) of the tag handler class associated with it, and any other information that JSP engine needs to process the custom tag.
- A **tag handler**, a Java class that executes the operations associated with the custom tag. For example, for the above `insertCatchOfDay` tag,

the tag handler is the Java class that executes the database query to retrieve the appropriate menu item.

We've already seen a JSP page that uses a custom tag, so let's look at the other two components.

Tag Handler

A tag handler is a Java class that is somewhat similar to a servlet. Whereas servlets implement the `Servlet` interface and are typically executed by an HTML `GET` or `POST` request, tag handlers implement a `Tag` interface (`javax.servlet.jsp.Tag`) and are executed when a custom tag is processed by the JSP engine. Each tag handler implements the method

```
public int doStartTag()
```

which defines the action taken when the custom tag is processed. If the custom tag includes attributes then the tag handler must define these attributes and get/set methods for each. For example, when defining the tag handler for the `insertCatchOfDay` custom tag shown above, we must define the `meal` attribute and its associated get and set methods:

```
private String meal = null;

public void setMeal(String s) {
meal = s;
}
public String getMeal() {
return meal;
}
```

Tag Library Descriptor

If you spend all of your time working with Java technology and don't know a thing about XML then the tag library descriptor component of JSP programming may seem strange. Have no fear, you don't need to learn a new language or master a whole new way of programming. Tag library descriptors simply use HTML−like tag syntax to define the name of your custom tag and its attributes, much like defining an object.

The following tag library descriptor defines the `insertCatchOfDay` tag used above. Note that this file defines the name of the custom tag, its attributes, and the associated tag handler class:

```
<? xml version="1.0" ?>
<taglib>
<tag>
<name>insertCatchOfDay</name>
<tagclass>com.sun.CatchOfDayHandler</tagclass>
<info>
```

```
Queries menu database for the catch of the day.
</info>

<attribute>
<name>meal</name>
</attribute>
</tag>

</taglib>
```

Along with defining the name of an attribute, a tag library descriptor can also define its data type and specify whether the attribute is required; this allows the JSP engine to do certain error checking before the tag handler is executed. Additional information, such as a name and version number for the library, may be included in a tag library descriptor for use with JSP authoring tools. (See the JSP 1.1 specification for complete information on tag library descriptor syntax.)

More Examples

This section contains additional examples of JSP in action. In the first example, a JSP page uses the HTTP request object (`HttpServletRequest`) to determine the user's browser version and returns appropriate content from one of three HTML pages. This page could be used to serve up more sophisticated content to newer browsers while maintaining support for older browsers:

```
<!-- example1.jsp -->
<%@ page language=="java" info="Example JSP #1" %>
<html>
<body>
<%! String agent; %>
<%
agent = request.getHeader("User-Agent");
if ( agent.startsWith("Mozilla/4.0") {
%>
<%-- Return content for 4.0 browsers --%>
<%@ include file="ver4.html" %>
<%
}
else if ( agent.startsWith("Mozilla/3.0") {
%>
<%-- Return content for 3.0 browsers --%>
<%@ include file="ver3.html" %>
<%
}
else {
%>
<%-- Return content for other/unknown browsers --%>
<%@ include file="other.html" %>
<%
}
%>
```

</body>
</html>

Note that this page accesses the HTTP request object without declaring or initializing the object. Both the `request` and `response` (`HttpServletResponse`) objects are always implicitly available to JSP pages. As with servlets, JSP pages can use the request object to get parameter values from HTML forms, as in the following example:

```
<!--example2.jsp -->
<%@ page language="java" info="Example JSP #2" %>
<html>
<body>
<%@ include file="header.html" %>
<jsp:useBean id="db" class="DbBean" />
<%! String selections[], info; %>
<p>Here are your current selections:</p>
<%
<!-- Get the form parameter values -->
selections = request.getParameterValues("items");
if (selections != null) {
%><ul>
<%
for(int x = 0; x < selections.length; x++) {
%><li>
<%= selections[x] %> : <%= db.getInfo(selections[x])
%>
<%
}
%></ul><%
}
else {
%>
<p>(no items selected)</p>
<%
}
%>
<br>
<%@ include file="footer.html" %>
</body>
</html>
```

In this example, static HTML content that defines page presentation is inserted with `include` directives, while dynamic content is generated from parameter values retrieved from the `request` object. After each parameter value is read, a JavaBean component is queried for information about each selection. Using beans in JSP pages this way makes it easy to retrieve dynamic web content from a database.

Conclusion

If you're looking for a convenient way to create web applications that connect to server−side Java components, JavaServer Pages is the way to go. Besides the inherent portability of Java and JSP's ready access to technologies like EJB, RMI, JDBC, and JavaBeans, the separation of HTML presentation code and application logic makes JSP pages very easy for organizations to work with. In fact, because web authors can create JSP pages with little or no help from you, the Java developer, you may no longer need to worry about creating web pages and writing HTML code.

About the Author

Scott McPherson *is the founder of MochaMail Corporation, a Silicon Valley Java technology start−up. MochaMail uses a variety of Java technologies to provide the next generation in web−based email access. Scott can be reached at scottm@mochamail.com.*

Reader Feedback

Tell us what you think of this article.

Very worth reading Worth reading Not worth reading

If you have other comments or ideas for future articles, please type them here:

[This page was last updated Aug−14−2002]