3 Developing your first tags

In this chapter

- JSP custom tags defined
- Setting up a development environment
- Hello World (the tag way)
- Compiling, deploying, and testing

Thus far we have seen how servlets and JSPs can be used to build a web application. These technologies go some distance toward making web development easier, but do not yet facilitate the separation of Java from HTML in a reusable way. Custom tags make this possible by bundling Java code into concise, HTML-like fragments recognizable by presentation developers. Custom tags are therefore an attractive choice for Java-based web applications and in this chapter, we'll introduce custom tags and walk through examples of their development and use. We'll also look at how to set up a development environment and deploy, test, and troubleshoot tags.

This chapter takes a mountain-top view of custom JSP tags in order to provide a clear, high-level look at the subject's landscape. Later chapters will dive deeper and home in on each of the topics touched upon here. So don't be concerned if the finer details are left for later explanation. The goal now is to jumpstart your tag development and ensure that you're sufficiently comfortable with the basics so that you may start building tags on your own.

3.1 What are JSP custom tags?

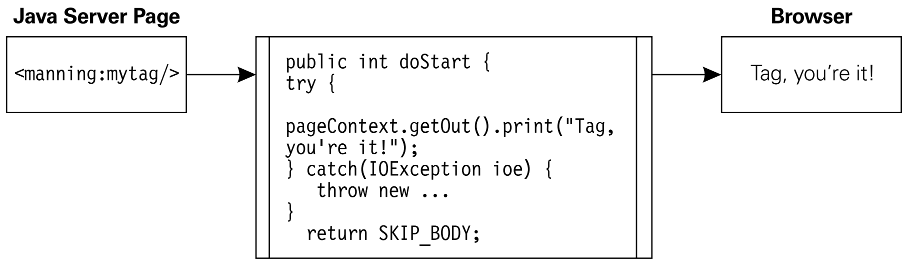
At its most fundamental level, a *tag* is a group of characters read by a program for the purpose of instructing the program to perform an action. In the case of HTML tags, the program reading the tags is a Web browser, and the actions range from painting words or objects on the screen to creating forms for data collection. Custom JSP tags are also interpreted by a program; but, unlike HTML, JSP tags are interpreted on the server side—not client side. The program that interprets custom JSP tags is the runtime engine in your application server (TomCat, JRun, WebLogic, etc.). When the JSP engine encounters a custom tag, it executes Java code that has been specified to go with that tag. Common tasks performed by tag codes include retrieving database values, formatting text, and returning HTML to a browser. Since a tag references some Java code to run when it's encountered, one way to think of a tag is simply as a shorthand notation for a block of code.

Notice in figure 3.1 that when the JSP runtime encounters the tag, it causes a block of Java code to execute and return a message to the client's browser.

3.1.1 Anatomy of a tag

Tags are often structured with a body and/or attributes which are the places where a page author (the user of the tag) can include more information about how the tag should do its job. The following snippet shows the general structure of a tag.

```
<tagname attributename="attributevalue"
        otherattributename="otherattributevalue">
Tag's body... can contain about anything.
</tagname>
```



```
public int doStart {
try {

pageContext.getOut().print("Tag,
you're it!");
} catch(IOException ioe) {
   throw new ...
}
  return SKIP_BODY;
```

Figure 3.1 A tag in action

This syntax should look familiar, since we see it so often in HTML tags, such as:

```
<font face="Tahoma" size=3">
Tag, you're it!
</font>
```

Tags can also appear without a body, meaning that the start tag does not have a matching end tag. These "bodyless" tags look like this:

```
<bodylesstagname attributename="attributevalue"
                 otherattributename="otherattributevalue"/>
```

You've probably seen examples of bodyless tags in HTML, such as:

```
<input type="input" name="body">
```

Bodyless tags usually represent a certain function, as in the printing of the value of a database field onto the page. Tags often have bodies in order to perform an operation on the content in the body, such as formatting, translating, or processing it in some way.

JSP custom tags are merely Java classes that implement one of two special interfaces. Since tags are standard Java classes, they can interact with, delegate to, or integrate with any other Java code in order to make that functionality available through a tag. For instance, we might have a library of utility classes we've written for composing and sending email, or for accessing a particular database that we'd like to make available to HTML developers. We need build only a few tags that collect the necessary information through attributes and pass this information to our utility classes.

3.1.2 *Using a tag in JSP*

JSP code that uses email and database tags such as those just mentioned might look something like this:

```
<html>
I am sending you an email with your account information
<jspx:sendmail server="mail.corp.com"
               from="john.doe@corp.com"
               to="foo@bar.com"
               subject="mail from a tag">
Look at how easy it is to send an email from a tag... here is
your status.

<jspx:dbaccess>
    <jspx:wdbcon id="con1"/>

    <jspx:wjitdbquery>
        select reserves from account where id='<%= userid %>'
    </jspx:wjitdbquery>

You have <jspx:wdbshow field="reserves "/>$ in your saving account.
</jspx:dbaccess>

</jspx:sendmail>
</html>
```

Among the JSP and HTML fragments are special tags prefixed with `jspx`. Even to the untrained eye, these tags appear to query a database, present the information in the content of an email, and send the message. Notice how the attributes help gather information such as the email sender and subject and the field in the database to display. Also, note how the `<jspx:wjitdbquery>` tag contains a Structured Query Language (SQL) statement within its body that it uses for the database query. This is a good example of what a JSP using custom tags might look like. Consider how much messier this JSP would look if we had to include all the Java code necessary for creating classes, setting properties, catching exceptions, and so forth.

3.1.3 The tag library descriptor

An important step in creating tags is specifying how they will be used by the JSP runtime that executes them. To properly work with a tag, the runtime must know several things about it, such as what (if any) attributes it has, and whether or not it has a body. This information is used by the runtime to verify that the tag is properly employed by a JSP author and to correctly execute the tag during a request. This crucial information is made available to the runtime engine via a standard XML file called a tag library descriptor (TLD), a key component of the JSP Specification and standard across all products that implement it. How to create a TLD is discussed in section 3.2.4, and covered in greater detail in chapter 5 and appendix B.

3.2 Why tags?

JSP already makes it possible to embed *scriptlets* (bits of Java code) and JavaBeans in line with HTML content, so why do we need JSP tags? We need them because tags were never intended to offer more functionality than scriptlets, just better packaging. JSP tags were created to improve the separation of program logic and presentation logic; specifically, to abstract Java syntax from HTML.

Scriptlets are not a suitable solution for all web development because most content developers (art designers, HTML developers, and the like) don't know Java and, perhaps, don't care to. Though much Java code can be encapsulated in beans, their usage in a JSP still requires the presentation developer to have a basic knowledge of Java syntax and datatypes in order to be productive. JSP tags form a new "scriptlet-free" and even a completely "Java-free" component model that is adapted perfectly to the JSP environment with its different developer types. If custom tags are properly constructed, they can be of enormous use to HTML developers, even those who have no working knowledge of Java—they won't even have to know they're using it. Tags can reduce or eliminate the number of scriptlets in a JSP application in four ways:

- A tag is nothing more than a Java component that takes its arguments from attribute and body. Since tags can have attributes and body, any necessary parameters to the tag can be passed within the tag's body or as one of its attributes. No Java code is needed to initialize or set properties on the component.
- JSP requires a considerable quantity of scriptlets for tasks such as iteration, setting of initial values, and performing conditional HTML. All of these tasks can be cleanly abstracted in a few simple tags.
- In many cases, a JavaBean component is configured and activated using scriptlets. One can develop a set of JSP tags to perform this configuration and activation without any Java.
- Tags can implement many utility operations, such as sending email and connecting to a database, and in this way reduce the number of utility scriptlets needed inside JSP.

The benefits of custom tags also include the creation of a neat abstraction layer between logic and presentation. This abstraction creates an interface that allows Java developers to fix bugs, add features, and change implementation without requiring any changes to the JSPs that include those tags. In short, JSP tags help bring you one step closer to the Holy Grail of web development—true abstraction of presentation and control. For more on the benefits of custom tags, see chapter 15.

3.2.1 Comparisons of scriptlets and custom tags

The differences between scriptlets and custom tags are fairly concrete:

1. Custom tags have simpler syntax. Scriptlets are written in Java and require the author to be familiar with Java syntax, whereas tags are HTML-like in syntax and require no Java knowledge.
2. Custom tags are easier to debug and are less error prone than scriptlets, since omitting a curly bracket, a semicolon, or some other minute character in a scriptlet can produce errors that are not easy to understand. Custom tag syntax is extraordinarily simple and, with most JSP runtime products, even the occasional typo in custom tag usage will produce meaningful error messages.
3. Custom tags are easy to integrate in development environments. Since tags are a common component of many web technologies, HTML editors have support for adding tags into the development environment. This allows JSP authors to continue using their favorite integrated development environment (IDE) to build tag-based JSPs. Support for JSP scriptlets syntax in development environments exists, but is only useful to JSP authors well versed in Java.
4. Custom tags can eliminate the need for Java in your JSPs. By containing most of your logic within objects in your scriptlets, you can vastly reduce the amount of Java code in a JSP; however, custom tags still carry the advantage of imposing absolutely no Java syntax, something scriptlets cannot achieve.

For small projects in which all your JSPs will be authored by developers knowledgeable in Java, scriptlets are a fine solution. For larger projects, where content developers unfamiliar with Java will be handling most of the presentation, JSP custom tags provide a real advantage and are a logical choice.

3.3 Setting up a development environment

Before we can build our first tag, we need to configure our development environment. This development environment should at least make it possible to:

- Compile the tags with the servlet, JSP, and JSP custom tags API[1]
- Test the developed tags
- Browse the JSP custom tags API documentation.

[1] We will take a look at the JSP custom tag API in chapter 4.

There are several Java IDEs in today's market, some of which provide fine support for servlet and JSP development; however, we are not going to work with any particular IDE because it is highly unlikely that you would have the same one that we select. Also, IDEs are notorious for lagging behind the leading edge of the Servlet and JSP API. Instead we explain how to fetch all the ingredients for a minimal development environment and how to set them up so that you may start developing tags immediately. This development environment will be concentrated around Tomcat,[2] the reference implementation of the servlet API, and the JDK1.2.2 or above (as available to most operating systems).

3.3.1 Installing the JDK

The first step in setting up the development environment is to install JDK1.2.2 (or higher) on your development system. More than two years since its first appearance, JDK1.2 can be found in a matured state on most operating systems, and this book uses many of its new classes and interfaces, such as `java.util.Iterator`. Although JDK1.2 is recommended for tag development, a JDK1.1.x version should suffice. Installing the JDK is an operating system-dependent task and will not be covered here, so we'll assume that you have a JDK installed and that you point into the installation directory with an environment variable named `JAVA_HOME`.

3.3.2 Installing Tomcat

Tomcat is the reference implementation of the Servlet and JSP API. It is easy to use and install, has a very small footprint (both on the hard drive and in memory), and is Open Source—all of which makes it a perfect learning tool. Installing Tomcat with the basic functionality of a stand-alone servlet and JSP container is really a cinch:

1. Extract the Tomcat binary distribution archive[3] (available as either .zip or tar.gz archives).
2. Define an environment variable named `TOMCAT_HOME` to point to Tomcat's installation root directory.
3. Make sure that the environment variable `JAVA_HOME` is defined and points to the directory wherein you installed your JDK.

[2] Tomcat's home on the web is at http://www/jakarta.apache.org

[3] You can download the binary distribution directly from Tomcat's web site. The installation directives supplied in this book apply to Tomcat versions 3.1 and 3.2.

3.3.3 Testing your Tomcat installation

To test-drive Tomcat, change the directory to TOMCAT_HOME and execute the startup script in Tomcat's bin directory. Tomcat should start running in the background and you can test it by issuing an HTTP request (i.e., http://your.machine.name:8080/). Once Tomcat is running, the installation of the development environment is complete and you may start immediately to develop tags; but first, let's look at the Tomcat distribution.

servlet.jar

The .jar file is where you find the interfaces and classes constituting the Servlet and JSP API. This file is named servlet.jar and is located in Tomcat's Lib directory. When compiling a servlet or JSP custom tag, you should make sure that this file is in your compilation `CLASSPATH` definition.

webapps directory

Where to place your web applications for Tomcat is the next consideration. Tomcat can generally be configured to take applications from any place you choose, but why bother configuring individual applications when you can simply drop your application into a single directory for deployment? The one directory approach will prove much simpler for your first applications. Under TOMCAT_HOME there is a subdirectory named webapps; and whenever Tomcat starts to run, it inspects this subdirectory, searches for web-application archive files (.war), and automatically deploys them. Moreover, if Tomcat finds subdirectories under webapps, it will assume that these directories contain web applications. Deployment to this directory is thus a simple task.

Javadoc documentation

One last thing to consider with Tomcat is the location of the Javadoc documents for the Servlet and JSP API. These documents are located in an application bundled with the Tomcat samples. In the webapps directory, there's a directory named ROOT, the home of Tomcat default root application. The root application has a subdirectory path named `docs/api` where you can find the Javadoc documents for the Servlet and JSP API (start with the file index.html).[4]

With the environment configured and a basic understanding of the deployment picture, it's time to build our first custom tag.

[4] You can also browse these documents by starting Tomcat and referring to http://your.machine.name:8080/docs/api/index.html.

3.4 Hello World example

Our goal in this section is to create a simple tag that may not be particularly reusable, but it will introduce most of the concepts needed for building useful tags. This simplicity is necessary now, as the myriad details involved with constructing even a Hello World tag can be daunting at first. Later sections in this chapter will present tags that have more real-world relevance.

Our `Hello World` tag is merely going to print `"Hello JSP tag World"` out to an HTML page. Listing 3.1 presents the source code for the `Hello World` implementation.

Listing 3.1 Source code for the HelloWorldTag handler class

```
package book.simpletasks;

import java.io.IOException;

import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;

public class HelloWorldTag            ❶
        extends TagSupport {

    public int doStartTag()                 ❷
            throws JspException
    {
        try {
            pageContext.getOut().print("Hello JSP tag World");    ❸
        } catch(IOException ioe) {    ❹
            throw new JspTagException("Error:
   IOException while writing to the user");
        }
        return SKIP_BODY;
    }
}
```

❶ **TagSupport is an abstract class which is part of the JSP tag APIs** Listing 3.1 presents a Java class that implements a tag handler, but it also contains methods and objects that are new to you unless you already have a very solid background in servlets and JSPs. We mentioned earlier that tags are Java classes that implement one of two special interfaces. These interfaces define all the methods the JSP runtime uses to get at the tag's functionality. As with many Java interfaces, some utility-only classes that provide basic implementations of these interfaces are available, making development easier. In the case of our `HelloWorldTag`, we extend one such utility class called `TagSupport`. `TagSupport` and the interface it implements, `Tag`, are both part of the custom JSP tag API. Don't worry too much over the specifics of this interface. For now it's important to know only that we need to implement `Tag` to create a tag, and we've done so by extending `TagSupport`.

❷ **JSP runtime calls `doStartTag()` to execute the tag** Here we note that there is no explicit constructor for this tag, nor is there a `main()` method for invoking the class. This is because a tag handler is not a stand-alone class, but is instantiated by the JSP runtime that invokes its methods. The JSP custom tags API defines a set of methods for custom tags (which are included in the two special interfaces previously mentioned) that the JSP runtime calls throughout a tag's life cycle. One of these methods, `doStartTag()`, can be seen in our example and is called by the JSP runtime when it starts executing a tag (more about the `Tag` methods in chapter 4). The `doStartTag()` method is a repository for code that we wish to have executed whenever the JSP runtime encounters our tag within the page.[5]

❸ **Tag echoes the `hello` message to the user** In our implementation of `doStartTag()`, we perform three operations. We print the `hello` message using an `out` object that we got from the `PageContext` (in chapter 2).

❹ **Aborts the execution upon errors** We watch out for `IOException`s that may be thrown by the response `Writer`, catch them, and abort the tag's execution by throwing a `JspTagException`. Finally, as required by the method, we return an integer value which tells the JSP runtime how to proceed after encountering our tag. A value of `SKIP_BODY` tells the runtime engine to simply ignore the tag's body, if there is one, and go on evaluating the rest of the page. There are, of course, other valid return values for `doStartTag()`, which we'll explore in future chapters.

As listing 3.1 shows, the tag is only a few lines long and, indeed, all it does is write out to the page, but a few details that will reappear in other tags are already evident.

Now that we have the Java source of our tag, it is time to compile it.

3.4.1 Compiling the tag

Compiling Java source into its class (without an IDE) requires careful setting of the compilation `CLASSPATH` (a list of all directories and .jar files that hold the classes referenced in our source code). Basically, the `CLASSPATH` for a tag handler must include the Servlet and JSP APIs; you should also include any additional classes or libraries that you are using within the tag handler (such as JavaMail and JNDI). In

[5] Though this would seem to imply that the runtime evaluates a JSP each time a page is requested, we know from JSP development that the page is only interpreted and compiled into a servlet once. Tags are no exception; this is just a convenient way to think about how the tag will behave at runtime.

the case of `HelloWorldTag`, we are not using any additional libraries, and can settle with the following Javac command line (assuming that `JAVA_HOME` and `TOMCAT_HOME` are both defined and we are compiling the source file into a directory named classes):

For `UNIX`:

```
$JAVA_HOME/bin/javac -d ../classes -classpath $TOMCAT_HOME/lib/servlet.jar
  book/simpletasks/HelloWorldTag.java
```

For Windows:

```
%JAVA_HOME%\bin\javac -d ..\classes -classpath %TOMCAT_HOME%\lib\servlet.jar
  book\simpletasks\HelloWorldTag.java
```

Both command lines use the `TOMCAT_HOME` environment variable to add the Servlet and JSP API into the `CLASSPATH`, and this is actually the only JSP-`Tags`-specific portion in the compilation command. When the compilation ends, we have our compiled tag handler in the classes directory and we are ready to continue to the next step—creating the tag library descriptor (TLD).

3.4.2 Creating a tag library descriptor (TLD)

The JSP runtime requires your assistance if it is to understand how to use your custom tag. For example, it has to know what you want to name your tag and any tag attributes. To do this you need to create a file called a tag library descriptor for your tag. An in-depth explanation of the exact use of a TLD will be covered in chapter 5, and its syntax is explained in appendix B, so we needn't go into great detail on these now. Instead, if we look at our example for the `HelloWorldTag`, the ways to use a TLD will emerge.

The TLD is nothing more than a simple extended markup language (XML[6]) file, a text file including a cluster of tags with some predefined syntax. Since the TLD is just a text file, you can create it with your preferred editor (Emacs, VI, notepad, etc.) as long as you keep to some rudimentary guidelines as explained in appendix B. The TLD created for the `HelloWorld` tag is presented in listing 3.2.

Listing 3.2 Tag library descriptor for the HelloWorldTag

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
```

[6] XML is briefly described in appendix A.

```
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>simp</shortname>
    <uri> http://www.manning.com/jsptagsbook/simple-taglib </uri>
    <info>
        A simple sample tag library
    </info>

    <tag>
        <name>hello</name>
        <tagclass>book.simpletasks.HelloWorldTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>
            Say hello.
        </info>
    </tag>
</taglib>
```

Listing 3.2 defines a tag whose name is "hello," and whose implementing class is `HelloWorldTag`, which we just developed. This means that whenever the JSP runtime sees the tag `<hello/>` it should actually execute the methods contained in our `HelloWorldTag`.

The portion of listing 3.2 unique to this tag is in bold face and, as it demonstrates, creating a tag library involves many "overhead lines" that specify such information as the desired version of JSP and the like. Normally you can just grab (and update) these overhead lines from a pre-existing library descriptor and add your own tags below them.

Let's assume that we saved the TLD in a file named simpletags.tld. We now have our tag handler class and the TLD to help the JSP runtime use it. These two files are all we need to deploy our `HelloWorldTag` and begin using it in a JSP.

3.4.3 *Testing HelloWorldTag*

Testing `HelloWorldTag` involves deploying it to a JSP container and writing a JSP file to use the tag. To do this:

1 Create a web application for your tags (in our case, `HelloWorldTag`).
2 Deploy your tags in the application.
3 Write a JSP file that will use `HelloWorldTag`.
4 Execute the JSP file created in step 3 and look at the results.

Creating a web application

What must be done to create a new web application in Tomcat? This can be accomplished either by deploying a web application archive or creating an application directory that follows the WAR structure. We are going to create an application directory, as follows:

1. Make a directory named testapp in Tomcat's webapps directory.
2. Under the testapp directory make another directory named WEB-INF, and inside this create directories named lib and classes.

Create a file named web.xml in the WEB-INF directory and add the content of listing 3.3 into it; web.xml is going to be your web application deployment descriptor; and listing 3.3 contains an "empty" deployment descriptor content.

Listing 3.3 An empty web application deployment descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
</web-app>
```

Deploying a tag

You now have an application structure under the testapp directory into which you may deploy your tags. Tag deployment takes the following steps:

1. Copy your tag implementation classes or jar files into the application directory; .jar files should go into the newly created lib directory, .class files should go into the classes directory. In the present case, we will copy the compiled class into the classes directory (while preserving the package directory structure).
2. Copy the TLD into a location in the application's directory structure (WEB-INF is a good location). In our example we will copy our TLD from listing 3.2 (simpletags.tld) into the WEB-INF directory.
3. Add a tag library reference into the web application deployment descriptor. In our case, edit web.xml and add the content of listing 3.4 into the `<web-app>` section (these last two steps set up a reference to the TLD as will be explained in chapter 5).

Listing 3.4 A TLD reference entry for the tags described in simpletags.tld

```
<taglib>
    <taglib-uri>
        http://www.manning.com/jsptagsbook/simple-taglib
    </taglib-uri>
    <taglib-location>
        /WEB-INF/simpletags.tld
    </taglib-location>
</taglib>
```

The tag was deployed into the web application; all we need to do now is to create a JSP that uses the tag and verify whether it works.

Creating a JSP file to test HelloWorldTag

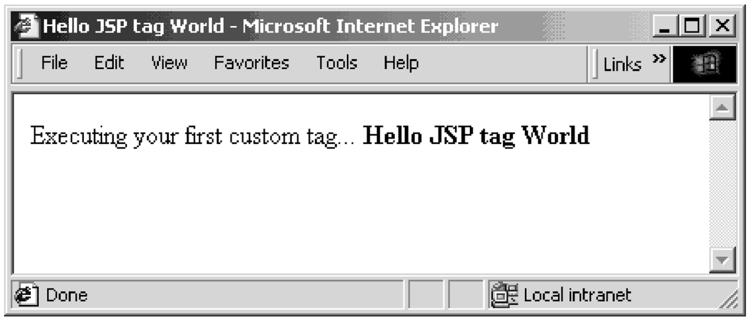
Developing a JSP file to test `HelloWorldTag` is a relatively simple task. All we need to do is craft a JSP file similar to the one presented in listing 3.5.

Listing 3.5 A JSP file to drive HelloWorldTag

```
<%@ taglib                                                      ❶
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="jspx" %>
<html>
<title><jspx:hello/></title>    ❷
<body>
Executing your first custom tag... <b><jspx:hello/></b>    ❷
</body>
</html>
```

❶ **Declares that the JSP file uses the library referenced by the URI and that the library's tags are referenced by jspx** Listing 3.5 is elementary, yet it illustrates a few important points about tags. The first is the `taglib` directive at the beginning of the JSP file. The `taglib` directive is further discussed in chapter 5, but for now we need to note that it indicates to the JSP runtime where the tag library lives and the prefix by which we'll refer to tags in this library. With this directive in place, the JSP runtime will recognize any usage of our tag throughout the JSP, as long as we precede our tag name with the prefix "jspx."

❷ **Uses the `hello` tag through the JSP file** We also see how the custom tag can be used through the JSP file. We use the `HelloWorldTag` twice, and we could, of course, have used it as much as we wanted. All that's needed is to add it to the JSP content. Note that our tag is bodyless, necessitating the use of the trailing backslash.


Hello JSP tag World - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Done
Local intranet

Executing your first custom tag... **Hello JSP tag World**

Figure 3.2 Output generated using the `hello` tag driver JSP

Figure 3.2 shows the results achieved by executing the JSP file in listing 3.5. Observe that wherever we had the `<hello>` tag, we now have the content generated by it.

Executing HelloWorldTag

Once we've created a web application, deployed the tag, and created and deployed a JSP to use it, all that's left is to view the page in a browser.

3.4.4 Did it work?

If your tag didn't work properly there is always some recourse. The error messages you see will vary, depending on which JSP runtime engine you've chosen. If, however, the messages you're seeing aren't helpful, here are a couple of suggestions:

- Make sure there are no spelling errors in the URL that you specified for the browser when asking for the JSP file (it should look like http://www.host.name/appname/jspfile.jsp).
- Make sure there are no spelling errors in your TLD file and that you've specified the fully qualified class name for your tag—package names and all.
- Verify that your TLD file is in a location where the JSP engine will be seeking it, such as the WEB-INF directory in your web application.
- Make sure the `taglib` directive has been properly placed at the top of the JSP. Without this, the engine doesn't know where to find the code for your tags and will just ignore them. When that happens, you'll actually see the tag in the HTML source.

3.4.5 A tag with attributes

Our `HelloWorldTag` is predictable; in fact, it always does exactly the same thing. In the dynamic world of web development, that is seldom the case, so let's look at a tag that behaves realistically, based on some user-specified attributes.

A web page might, for instance, need to display the value stored in a cookie such as a user name. Rather than forcing the page author to learn Java to access that value, we'll build a simple tag that does this for him. The tag should be flexible enough to be used in retrieving the value of any accessible cookie, so we'll create a tag attribute called `cookieName` to allow this. The first step in supporting this new attribute is to modify our tag handler class to receive and make use of this new attribute(listing 3.6):

Listing 3.6 Source code for the CookieValueTag handler class

```
package book.simpletasks;

import java.io.IOException;

import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.http.*;

public class CookieValueTag extends TagSupport {

   String cookieName;    ❶
    public int doStartTag()
               throws JspException
    {
      try {
        Cookie[] cookies =
        ((HttpServletRequest)pageContext.getRequest()).getCookies();
        if ( cookies != null ) {
          for ( int i=0; i < cookies.length; i++ ) {
            if ( cookies[i].getName().equalsIgnoreCase( cookieName ) ) {
              pageContext.getOut().print( cookies[i].getValue() );    ❷
              break;
            }
          }
        }
      } catch(IOException ioe) {
    throw new JspTagException("Error: IOException while writing to the user");
      }
      return SKIP_BODY;    ❸
    }

    public void setCookiename( String value ) {    ❹
    cookieName = value;
    }
    }
```

❶ **The field that will get set by the attribute.**

❷ Prints the value of the cookie to the response.

❸ Returns `SKIP_BODY` to tell the JSP runtime to skip the body if one exists.

❹ Invokes the set method when the JSP runtime encounters this attribute.

All we needed to do was add a set method called `setCookieName()` and assign a variable within it. The value of that variable is examined within our tag handler's `doStartTag()` to decide which cookie value to return. Now we need to inform the JSP runtime of this new tag and its attribute. Recall that the TLD is where we specify this kind of information, so we need to modify our previous TLD to support `CookieValueTag`. The tag declaration in our TLD file (listing 3.7) now looks like the following:

Listing 3.7 The new TLD file with our CookieValueTag

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>simp</shortname>
    <uri> http://www.manning.com/jsptagsbook/simple-taglib </uri>
    <info>
        A simple sample tag library
    </info>

    <tag>
        <name>hello</name>
        <tagclass>book.simpletasks.HelloWorldTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>
            Say hello.
        </info>
    </tag>
    <tag>
      <name>cookievalue</name>
      <tagclass>book.simpletasks.CookieValueTag</tagclass>
      <bodycontent>empty</bodycontent>
      <info>
          Get a cookie's value.
      </info>
      <attribute>                         ❶
        <name>cookiename</name>
        <required>true</required>         ❷
      </attribute>
    </tag>
</taglib>
```

❶ **This tag will have an attribute called `cookiename`.**

❷ **Specifies that this attribute is always required for this tag.**

The tag definition itself should look familiar, since it is very similar to our `HelloWorldTag`. The important difference is, of course, the attribute we've included. Note that the name of an attribute, in our case `cookiename`, is used by the JSP runtime to find `setCookieName()` to use in the tag handler; therefore, these need to match exactly for the tag to function.

To use this attribute within a JSP, syntax such as in listing 3.8 works well:

Listing 3.8 A JSP file to drive HelloWorldTag

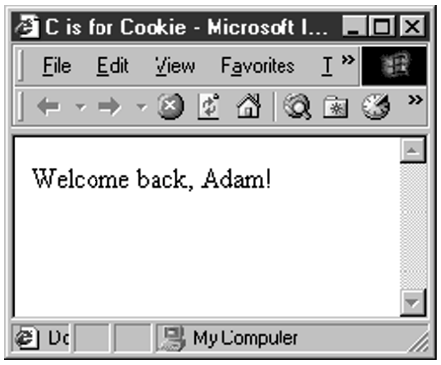
```
<%@ taglib                                                        ❶
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="jspx" %>
<html>
<title>C is for Cookie</title>
<body>
Welcome back, <jspx:cookievalue cookiename="username">      ❷
</body>
</html>
```

❶ **Declares that the JSP file uses the library referenced by the URI and that the library's tags are referenced by jspx.**

❷ **Uses the `cookeivalue` tag to retrieve a cookie called "username".**

Assuming we've used this tag in a case where a cookie named "username" will be accessible, we'll see a message like that shown in figure 3.3.

Adding attributes to your tags makes them much more flexible and useful to the web pages where they are used. We explore the use of tag attributes in further detail in chapters 4 and 6.



Welcome back, Adam!

Figure 3.3 CookieValueTag in action.

3.4.6 Packaging tags for shipment

Once the tags have been tested to your satisfaction, it's time to package them in a standard deployable manner. Packaging tags means putting the implementation classes along with the library descriptor in a .jar file following a convention that further instructs you to:

- Put your tag class files inside the .jar archive while maintaining their package structure.
- Put your TLD in the .jar file in a directory called META-INF.

For example, packaging our lone `HelloWorldTag` will require the following .jar file structure:

```
/book/simpletasks/HelloWorldTag.class
/META-INF/simpletags.tld
```

This .jar packaging need not be complicated; all that's required is to create the desired directory structure on your file system and use the `jar` command (bundled with the JDK) to archive this structure into the .jar file. The command to place our class and TLD in a jar called hello.jar looks like this:

```
jar cf hello.jar META-INF book
```

Now you can distribute your tag.

3.5 A tag with a body

Remember that tags can have a body or be bodyless. Our `HelloWorldTag` was an example of a tag without a body, so let's see an example of a tag with one. We create them whenever we want to take a block of content (typically HTML) and modify it or include it in the server's response. Think back to the HTML `<font>` tag. The body of the `<font>` is where you put text to which you wish to apply a particular font. Tags with bodies are great for translating content (from, say, HTML to WML), applying formatting, or indicating that a grouping of content should be treated in a special way, as is the case with the HTML `<form>` tag.

Here is an extremely simplified example that illustrates how a tag with a body works. Suppose we need to create a tag that will change a block of text from capital letters to lower case. We'll be creative and call this tag `LowerCaseTag`. Our new tag will have a lot in common with `HelloWorldTag`, but there are a few differences. The first is that `LowerCaseTag` doesn't extend from `TagSupport`, rather from `BodyTag-Support`. The formula is elementary: if your custom tag doesn't have a body or will include just its body verbatim, it should either implement the `Tag` interface or extend its utility class, `TagSupport`. If, however, your tag will modify or control its body, it needs to implement `BodyTag` or extend its utility class called `BodyTagSupport`. We'll cover several additional examples of both types in the next chapters.

3.5.1 LowerCaseTag handler

Here is the code for our `LowerCaseTag` handler class:

Listing 3.9 Source code for the LowerCaseTag handler class

```
package book.simpletasks;

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

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class LowerCaseTag extends BodyTagSupport {      ❶

    public int doAfterBody()           ❷
                throws JspException
    {

        try {
            BodyContent body = getBodyContent();    ❸
            JspWriter writer = body.getEnclosingWriter();     ❹
            String bodyString = body.getString();
            if ( bodyString != null ) {
            writer.print( bodyString.toLowerCase());    ❺
}


        } catch(IOException ioe) {
     throw new JspException("Error: IOException while writing to the user");
        }
        return SKIP_BODY;    ❻
    }
```

❶ **BodyTagSupport is an abstract class which is part of the JSP tag APIs.**

❷ **The method `doAfterBody()` is executed by the JSP runtime, once it has read in the tag's body.**

❸ **Retrieves the body that was just read in by the JSP runtime.**

❹ **Gets JspWriter to output the lowercase content.**

❺ **Writes the body out to the user in lowercase.**

❻ **Returns `SKIP_BODY` is returned to tell the JSP runtime to continue processing the rest of the page.**

With the tag handler class written, the next step is, once again, to create a TLD. This time our tag entry looks like this:

Listing 3.10 Tag entry for LowerCaseTag

```
<tag>
    <name>lowercase</name>
    <tagclass>book.simpletasks.LowerCaseTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
        Put body in lowercase.
    </info>
</tag>
```

The only difference in this listing is that the `<bodycontent>` field is no longer empty but now must be JSP. This is the way to indicate to the runtime that `LowerCaseTag` will have a body, unlike our `HelloWorldTag` that did not. There will be much more about bodycontent and other TLD fields in chapters 5 and 6.

We have returned to the stage where we need to use this new tag in a JSP file. Our JSP looks like this:

Listing 3.11 A JSP file to drive the LowerCaseTag

```
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"      ❶
    prefix="jspx" %>
<html>
<title>LowerCaseTag </title>
<body>
<jspx:lowercase>

I've got friends in low places.</jspx:lowercase>      ❷
</body>
</html>
```

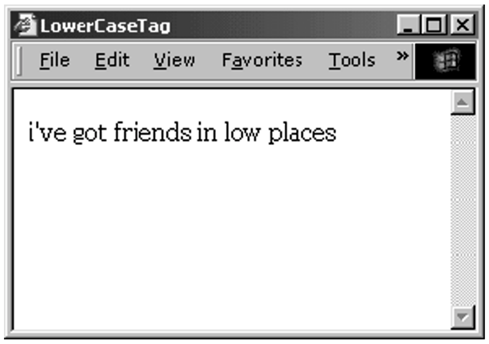
❶ **Declares that the JSP file uses the library referenced by the URI and that the library's tags are referenced by jspx.**

❷ **Uses the lowercase tag to change its body to lowercase.**

Now we add our tag to our deployment directory, pull up the JSP in our browser (figure 3.4), and voila!

This tag doesn't do anything especially useful, however it is always possible to modify it to do something worthwhile with the body. Some examples might include the body as the message of an email, translating the body from one markup language to another, or parsing the body of XML and outputting certain nodes or attributes. In the next chapters, we'll see how the body of a custom tag can include other custom tags to allow cooperation with very powerful results.

3.6 Summary



i've got friends in low places

Figure 3.4 Output generated using the lowercase tag driver JSP

What are custom tags? Why use them? Custom tags are unique JSP components that make it easy to integrate portions of Java logic into a JSP file in an easy-to-use, well-recognized format. Custom tags also answer to well-known API and life cycle definitions (to be discussed in chapter 4) that make it clear how tags behave in any development or runtime environment.

Why use custom tags? Custom tags represent a great way to separate the business logic and presentation, thus enhancing manageability and reducing overall maintenance costs. Another benefit is their ease of use. By using tag syntax, many of the scriptlets and other portions of Java code associated with the classic JSP programming are no longer needed, and the JSP development can be opened to content (commonly, HTML) developers.

We also discussed the mechanics related to tag development, and saw that it is not so difficult to develop simple, but useful, tags.

This chapter provided a solid foundation for you to start developing custom JSP tags. It presented four important tools that you will use in your daily tag development:

- How to configure a simple (and free) development environment with which you can compile and test your tags.
- How to develop, compile, and test simple tags using this development environment.
- How to write a TLD file to describe your tag's runtime behavior and attributes.
- How to package your tag library in a distributable .jar file.

If you have a lot of questions at this point, that's good. We've only lightly touched on many of the nuances of tag development in order to help you get started right away. In the next chapters, we will dive in and explore more fully each of the topics presented here.