JSP, JSTL, and Tomcat Primer

Paul DuBois
paul@kitebird.com

Document revision: 3.0
Last update: 2014-07-29

> **Note:** This is a *MySQL Cookbook* auxiliary document. Much of its content originally was an appendix in the book, but as of the third edition is available separately. There are several references here to the `recipes` software distribution and to the `mcb` web application that it includes. To obtain this distribution, which accompanies *MySQL Cookbook*, visit the companion web site:
>
> `http://www.kitebird.com/mysql-cookbook/`

Among the topics discussed in *MySQL Cookbook* is JavaServer Pages (JSP) programming, including JSP pages that use tags from the JSP Standard Tag Library (JSTL). The discussion uses the Tomcat web server as the deployment vehicle for JSP pages. However, for readers unfamiliar with JSP programming or Tomcat, the background required to use them is fairly extensive. This document provides that background:

- Overview of servlet, JSP, and tag library technologies
- Elements of JSP pages
- Overview of JSTL tags
- Using the Tomcat server
- Tomcat directory structure
- The layout of web applications

For additional information, Oracle's Java site provides access to documentation (including the specifications) for JDBC, servlets, JavaServer Pages (JSP), and the JSP Standard Tag Library (JSTL):

`http://www.oracle.com/technetwork/java/index.html`

Servlet and JavaServer Pages Overview

Java servlet technology enables efficient Java program execution in a web environment. The Java Servlet Specification defines the conventions of this environment, which may be summarized briefly as follows:

- Servlets run inside a servlet container, which itself either runs inside a web server or communicates with one. Servlet containers are also known as *servlet engines*.
- The servlet container receives requests from the web server and executes the appropriate servlet to process the request. The container then receives the response from the servlet and gives it to the web server, which in turn returns it to the client. A servlet container thus provides the connection between servlets and the web server under which they run. The container acts as the servlet runtime environment, with responsibilities that include determining the mapping between client requests and the servlets that handle them, as well as loading, executing, and unloading servlets as necessary.
- Servlets communicate with their container according to established conventions. Each servlet is

expected to implement methods with well-known names to be called in response to various kinds of requests. For example, `get` and `post` requests are routed to methods named `doGet()` and `doPost()`.

- Servlets that can be run by a container are arranged into logical groupings called *contexts*. (Contexts might correspond, for example, to subdirectories of the document tree that is managed by the container.) Contexts also can include resources other than servlets, such as HTML pages, images, or configuration files.

- A context provides the basis for a ‘‘web application,’’ which the Java Servlet Specification defines as follows: ‘‘A Web application is a collection of servlets, HTML pages, classes, and other resources that make up a complete application on a Web server.’’ In other words, an application is a group of related servlets that work together, without interference from other unrelated servlets. Servlets within a given application context can share information with each other. Servlets in different contexts cannot. For example, a gateway or login servlet might establish a user’s credentials, which then are placed into the context environment to be shared with other servlets within the same context as proof that the user has logged in properly. Should those servlets find the proper credentials not present in the environment when they execute, they can redirect to the gateway servlet automatically to require the user to log in. Servlets in other contexts cannot gain access to these credentials. Contexts thus provide security by preventing one application from invading another. They also insulate applications from the effects of another application crashing; the container can keep the noncrashed applications running while it restarts the one that failed.

- An application context can include private information not available to clients. By convention, contexts use their *WEB-INF* directory for private context-specific information. See the section ‘‘Web Application Structure.’’

- Information sharing between servlets can take place at several scope levels, which enables them to work together within the scope of a single request or across multiple requests.

The following listing shows what a simple servlet looks like. It’s a Java program that implements a `SimpleServlet` class. The class has a `doGet()` method to be invoked by the servlet container when it receives a `get` request for the servlet. It also has a `doPost()` method to handle the possibility that a `post` request may be received instead; it’s simply a wrapper that invokes `doGet()`. `SimpleServlet` produces a short HTML page that includes some static text that is the same each time the servlet runs, and two dynamic elements (the current date and client IP address) that vary over time and among clients:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet
{
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
    throws IOException, ServletException
  {
    PrintWriter out = response.getWriter ();

    response.setContentType ("text/html");
    out.println ("<html>");
    out.println ("<head><title>Simple Servlet</title></head>");
    out.println ("<body>");
    out.println ("<p>Hello.</p>");
    out.println ("<p>Current date: " + new Date () + "</p>");
    out.println ("<p>Your IP address: " + request.getRemoteAddr () + "</p>");
    out.println ("</body>");
    out.println ("</html>");
  }
```

```
  public void doPost (HttpServletRequest request,
                      HttpServletResponse response)
    throws IOException, ServletException
  {
    doGet (request, response);
  }
}
```

As you will no doubt observe, this ‘‘simple’’ servlet really isn’t so simple! It requires a fair amount of machinery to import the requisite classes and to establish the `doGet()` and `doPost()` methods that provide the standard interface to the servlet container. Compare the servlet to the following PHP script, which produces the same result with much less fuss:

```
<html>
<head><title>Simple PHP Page</title></head>
<body>
<p>Hello.</p>
<p>Current date: <?php print (date ("D M d H:i:s T Y")); ?></p>
<p>Your IP address: <?php print ($_SERVER["REMOTE_ADDR"]); ?></p>
</body>
</html>
```

The contrast between the Java servlet and the PHP script illustrates a problem with writing servlets—the amount of repetitious overhead:

- A minimal set of several classes must be imported into each servlet.
- The framework for setting up the servlet class and the `doGet()` or `doPost()` methods is highly stereotypical, often varying among servlets only in the servlet class name.
- Each fragment of HTML is produced with an output statement.

The first two points can be addressed by using a prototype file that you copy when beginning a new servlet. The third point (wrapping each line of HTML within a print statement) is not so easily addressed and is perhaps the single most tedious aspect of servlet writing. It also leads to another issue: A servlet’s code may be easy enough to read as Java, but it’s less easy to discern the structure of the HTML that the code generates. By writing Java that writes HTML, you’re writing in two languages at once, which is optimal for neither language.

JSP Pages—An Alternative to Servlets

The approach used in JavaServer Pages (JSP) relieves the burden of creating web pages by means of many print statements. JSP uses a notational approach that is similar to PHP: Write the HTML literally and embed code to be executed within special markers. The following JSP page is equivalent to the `SimpleServlet` servlet, but looks much more like the corresponding PHP script:

```
<html>
<head><title>Simple JSP Page</title></head>
<body>
<p>Hello.</p>
<p>Current date: <%= new java.util.Date () %></p>
<p>Your IP address: <%= request.getRemoteAddr () %></p>
</body>
</html>
```

The JSP page is more concise than the servlet in several ways:

- The standard set of classes required to run a servlet need not be imported. This is done automatically.
- The HTML is written more naturally, without print statements.
- No class definition is required, nor any `doGet()` or `doPost()` methods.

- The `response` and `out` objects need not be declared; they're set up for you and ready to use as implicit objects. For example, the JSP page just shown doesn't refer to `out` explicitly at all because its output-producing constructs write to `out` implicitly.
- The default content type is `text/html`; there's no need to specify it explicitly.
- The script includes literal Java by placing it within special markers. The page just shown uses `<%=` and `%>`, which mean ''evaluate the expression and display its result.'' There are other markers as well, each of which has a specific purpose. (For a brief summary, see ''Elements of JSP Pages'' elsewhere in this document.)

When a servlet container receives a request for a JSP page, it treats the page as a template containing literal text plus executable code embedded within special markers. The container produces an output page from the template. It leaves literal text from the template unmodified, replaces the executable code with any output that it generates, and returns the combined result to the client as the response to the request. That's the conceptual view of JSP processing, at least. When a container processes a JSP request, this is what really happens:

1. The container translates the JSP page into a servlet—that is, into an equivalent Java program. It converts instances of template text to print statements that output the text literally and places other instances of code into the program so that they execute with the intended effect. It places all this within a wrapper that provides a unique class name and includes `import` statements to pull in the standard set of classes necessary for the servlet to run properly in a web environment.
2. The container compiles the servlet to produce an executable class file.
3. The container executes the class file to generate an output page, which is returned to the client as the response to the request.
4. The container caches the executable class so that when the next request for the JSP page arrives, the container can execute the class directly and skip the translation and compilation phases. If the container notices that a JSP page has been modified the next time it is requested, it discards the cached class and recompiles the modified page into a new executable class.

Notationally, JSP pages provide a more natural way to write web pages than do servlets. Operationally, the JSP engine provides the benefits of automatic compilation after the page is installed in the document tree or modified thereafter. When you write a servlet yourself, any changes you make require that you recompile the servlet, unload the old one, and load the new one. That can lead to an emphasis on messing with the servlet itself rather than focusing on the servlet's purpose. JSP reverses the emphasis to enable you to think more about what the page does than about the mechanics of getting it compiled and loaded properly.

Servlets and JSP pages are not mutually exclusive. Application contexts in a servlet container can include both, and because JSP pages are converted into servlets anyway, they can intercommunicate.

JSP is similar enough to certain other technologies that it can provide a migration path away from them. For example, the JSP approach is much like that used in Microsoft's Active Server Pages (ASP), but JSP is vendor and platform neutral, whereas ASP is proprietary. JSP thus may provide an attractive alternative technology for developers looking to move away from the vendor lock-in associated with ASP.

Custom Actions and Tag Libraries

A servlet looks a lot like a Java program because that's what it is. The JSP approach encourages a cleaner separation of HTML (presentation) and code because you need not generate HTML using Java print statements. On the other hand, JSP doesn't *require* separation of HTML and code, so it's still possible to end up with lots of embedded Java in a page if you're not careful.

One way to avoid including literal Java in JSP pages is to use another JSP feature: custom actions. These take the form of special tags that look a lot like HTML tags (because they are written as XML elements). Custom actions enable definition of tags that perform tasks on behalf of the page in which they occur. For example, a `<sql:query>` tag might communicate with a database server to execute a query. Custom actions typically come in groups, which are known as tag libraries and designed as follows:

- The actions performed by the tags are implemented by a set of classes. These are just regular Java classes, written according to a set of interface conventions that enable the servlet container to communicate with them in a standard way. (The conventions define how tag attributes and body content are passed to tag handler classes, for example.) Typically, the set of classes is packaged into a JAR file.
- The library includes a Tag Library Descriptor (TLD) file that specifies how each tag maps onto the corresponding class. This enables the JSP processor to determine which class to invoke for each custom tag that appears in a JSP page. The TLD file also indicates each tag's interface, such as whether it has any required attributes. The JSP processor uses this information at page translation time to determine whether a JSP page uses the tags in the library correctly. For example, if a tag requires a particular attribute and the tag appears in a page without it, the processor detects that problem and issues an appropriate error message.

Tag libraries make it easier to write entire pages using tag notation rather than switching between tags and Java code. The notation is JSP-like, not Java-like, but the effect of placing a custom tag in a JSP page is like making a method call because a tag reference in a JSP page maps onto a method invocation in the servlet that the page translates into.

The rest of this document uses one popular tag library, the JSP Standard Tag Library (JSTL), which consists of several tag sets grouped by function.

To see the difference between the embedded-Java and tag-library approaches, compare two JSP scripts that set up a connection to a MySQL server and display a list of tables in the `cookbook` database. The first script uses Java embedded within the page:

```
<%@ page import="java.sql.*" %>

<html>
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>

<%
  Connection conn = null;
  String url = "jdbc:mysql://localhost/cookbook";
  String user = "cbuser";
  String password = "cbpass";

  Class.forName ("com.mysql.jdbc.Driver").newInstance ();
  conn = DriverManager.getConnection (url, user, password);

  Statement s = conn.createStatement ();
  s.executeQuery ("SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES"
                  + " WHERE TABLE_SCHEMA = 'cookbook'"
                  + " ORDER BY TABLE_NAME");
  ResultSet rs = s.getResultSet ();
  while (rs.next ())
    out.println (rs.getString (1) + "<br />");
  rs.close ();
  s.close ();
  conn.close ();
%>

</body>
```

```
</html>
```

The same thing can be done with the JSTL core and database tags. Converting the preceding JSP page to use these tags yields this result, which entirely avoids use of literal Java:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

<html>
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>

<sql:setDataSource
  var="conn"
  driver="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost/cookbook"
  user="cbuser"
  password="cbpass"
/>

<sql:query dataSource="${conn}" var="rs">
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'cookbook'
  ORDER BY TABLE_NAME
</sql:query>
<c:forEach items="${rs.rowsByIndex}" var="row">
  <c:out value="${row[0]}"/><br />
</c:forEach>

</body>
</html>
```

The `taglib` directives identify which tag libraries the page uses and associate actions from the corresponding tag sets with prefixes of `c` and `sql`. (In effect, a prefix sets up a namespace for a tag set.) The `<sql:dataSource>` tag sets up the parameters for connecting to the MySQL server, `<sql:query>` executes a query, `<c:forEach>` loops through the result, and `<c:out>` adds each table name in the result to the output page. (I'm glossing over details. For more information about these tags, see the section "Overview of JSTL Tags.")

If you plan to connect to the database server the same way from multiple JSP pages in your application context, move the `<sql:dataSource>` tag to an include file to achieve a further simplification. For example, name the file *jstl-mcb-setup.inc* and place it in the application's *WEB-INF* directory. Then any page within the application context can set up the connection to the MySQL server by accessing the file with an `include` directive. Modifying the preceding page to use the include file results in a page that looks like this:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ include file="/WEB-INF/jstl-mcb-setup.inc" %>

<html>
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>

<sql:query dataSource="${conn}" var="rs">
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'cookbook'
  ORDER BY TABLE_NAME
</sql:query>
```

```
<c:forEach items="${rs.rowsByIndex}" var="row">
  <c:out value="${row[0]}"/><br />
</c:forEach>

</body>
</html>
```

You're still *using* Java when you use a tag library because tag actions map onto Java class invocations. But the notation follows XML conventions, so it's less like writing program code and more like writing HTML. If you are part of a team that produces web content using a ''separation of powers'' workflow, custom actions enable elements of the page that are produced dynamically to be packaged in a way that is easier for designers and other non-programmers to deal with. They need not develop or work directly with the classes that implement tag actions; that's left to the programmers who write the classes that correspond to the tags.

Elements of JSP Pages

The earlier section ''Servlet and JavaServer Pages Overview'' describes general characteristics of JSP pages. This section discusses the permitted constructs in more detail.

JSP pages are templates that contain static parts and dynamic parts:

- Literal text in a JSP page not enclosed within special markers is static; it's sent to the client unmodified. JSP pages commonly produce HTML pages, so their static parts are written in HTML. But you can also write JSP pages that produce other types of output, such as plain text, XML, or WML.
- The nonstatic (dynamic) parts of JSP pages consist of code to be evaluated, distinguished from static text by special markers. Some markers indicate page-processing directives or scriptlets. A directive gives the JSP engine information about how to process the page, whereas a scriptlet is a mini-program that is evaluated and replaced by the output it produces. Other markers take the form of tags written as XML elements; they are associated with classes that act as tag handlers to perform the desired actions.

The following sections discuss the types of dynamic elements that JSP pages can contain.

Scripting Elements

Several scripting markers enable you to embed Java code or comments in a JSP page.

<% ... %>

The <% and %> markers indicate a scriptlet—that is, embedded Java code. The following scriptlet invokes `print()` to write a value to the output page:

```
<% out.print (1+2); %>
```

<%= ... %>

These markers indicate an expression to evaluate. The result is added to the output page, which makes it easy to display values with no explicit print statement. For example, these two constructs both display the value 3, but the first is easier to write:

```
<%= 1+2 %>
<% out.print (1+2); %>
```

<%! ... %>

The <%! and %> markers declare class variables and methods.

<%-- ... --%>

These markers indicate a comment. JSP comments disappear entirely and do not appear in the output returned to the client. If you want a comment to appear in the final output page, use an HTML comment instead:

```
<%-- this comment will not be part of the final output page --%>
<!-- this comment will be part of the final output page -->
```

When a JSP page is translated into a servlet, all scripting elements effectively become part of the same servlet. This means that a variable declared in one element can be used by other elements later in the page. It also means that if you declare a given variable in two elements, the resulting servlet is illegal and an error occurs.

The <% ... %> and <%! ... %> markers both enable you to declare variables, but differ in their effect. A variable declared within <% ... %> is an object (or instance) variable, initialized each time the page is requested. A variable declared within <%! ... %> is a class variable, initialized only at the beginning the life of the page. Consider the following JSP page, *counter.jsp*, which declares `counter1` as an object variable and `counter2` as a class variable:

```
<%-- counter.jsp: demonstrate object and class variable counters --%>

<% int counter1 = 0; %>     <%-- object variable --%>
<%! int counter2 = 0; %>    <%-- class variable --%>
<% counter1 = counter1 + 1; %>
<% counter2 = counter2 + 1; %>
<p>Counter 1 is <%= counter1 %></p>
<p>Counter 2 is <%= counter2 %></p>
```

If you install the page where your servlet container can access it and request it from a browser several times, the value of `counter1` is 1 for every request. The value of `counter2` increments across successive requests (even if different clients request the page), until the servlet container restarts.

In addition to variables that you declare yourself, JSP pages have access to a number of objects that are declared for you implicitly; see ‘‘Implicit JSP Objects.’’

JSP Directives

The <%@ and %> markers indicate a JSP directive that provides the JSP processor with information about the kind of output the page produces, the classes or tag libraries it requires, and so forth.

<%@ `page` ... %>

`page` directives provide several kinds of information, indicated by one or more *`attribute`*`="`*`value`*`"` pairs following the `page` keyword. The following directive specifies that the page scripting language is Java and that it produces an output page with a content type of `text/html`:

```
<%@ page language="java" contentType="text/html" %>
```

If the page produces HTML, this particular directive can be omitted because `java` and `text/html` are the default values for their respective attributes. If a JSP page produces non-HTML output, override the default content type. For example, in a page that produces plain text, use this directive:

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

An `import` attribute causes Java classes to be imported. A regular Java program does this using an `import` statement. In a JSP page, use a `page` directive instead:

```
<%@ page import="java.util.Date" %>
<p>The date is <%= new Date () %>.</p>
```

If you refer to a particular class only once, it may be more convenient to omit the `page` directive and refer to the class by its full name when you use it:

```
<p>The date is <%= new java.util.Date () %>.</p>
```

<%@ include ... %>

The `include` directive inserts the contents of a file into the page translation process. Include files enable easy sharing of content (either static or dynamic) among a set of JSP pages. For example, you can use them to provide standard headers or footers for a set of JSP pages, or to execute code for common operations such as setting up a connection to a database server.

The directive is replaced by the contents of the included file, which is then translated itself. The following directive causes inclusion of a file named *my-setup-stuff.inc* from the application's *WEB-INF* directory:

```
<%@ include file="/WEB-INF/my-setup-stuff.inc" %>
```

A leading `/` indicates a filename relative to the application directory (a context-relative path). With no leading `/`, the file is relative to the location of the page containing the `include` directive.

<%@ taglib ... %>

A `taglib` directive indicates that the page uses custom actions from a given tag library. For example, a page that uses the core and database-access tags from JSTL includes the following `taglib` directives:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

The `prefix` attribute value is chosen by you and indicates how the rest of the page refers to tags from the library. The directives just shown specify that references to core and database tags in the page will have the forms <c:*xxx*> and <sql:*xxx*>. For example, you can use the `out` tag from the core library as follows to display a value:

```
<c:out value="Hello, world."/>
```

Or you might issue a query with the database `query` tag like this:

```
<sql:query dataSource="${conn}" var="result">
  SELECT id, name FROM profile ORDER BY id
</sql:query>
```

The `uri` (Uniform Resource Identifier) attribute uniquely identifies the tag library. This value must match that defined in the library's TLD file (see ''Custom Actions and Tag Libraries''). The TLD file defines the interface to the actions so that the JSP processor can verify that the page uses the library's tags correctly. `uri` values commonly take the form of a string that includes the host from which the tag library originates. That makes the `uri` value look like a URL, but it's just an identifier; the JSP engine doesn't actually fetch the descriptor file from that host.

To use custom tags from a tag library, the library must be installed where the application can find it; see ''Using a Tag Library.''

Action Elements

Action element tags can refer to standard (predefined) JSP actions, or to custom actions in a tag library. Tag names include a prefix and a specific action:

- Tag names with a `jsp` prefix indicate predefined action elements. For example, `<jsp:forward>` forwards the current request to another page. This action is available to any page run under a standard

JSP processor.

- Custom actions are implemented by tag libraries. The prefix of the tag name must match the `prefix` attribute of a `taglib` directive that appears earlier in the page, to enable the JSP processor to determine which library defines the tag.

Actions appear as XML elements within a JSP page, and their syntax follows normal XML rules. An element with a body is written with separate opening and closing tags:

```
<c:if test="${x == 0}">
  x is zero
</c:if>
```

If the tag has no body, the opening and closing tags can be combined:

```
<jsp:forward page="some_other_page.jsp"/>
```

Implicit JSP Objects

When a servlet runs, the servlet container passes it two arguments representing the request and the response, but the servlet must declare other objects itself. For example, the `response` argument can be used to create an output-writing object like this:

```
PrintWriter out = response.getWriter ();
```

A convenience that JSP provides in comparison to servlet writing is a set of implicit objects—that is, standard objects provided as part of the JSP execution environment. You can refer to any of these objects without explicitly declaring them. Thus, in a JSP page, the `out` object is already set up and available for use. Some useful implicit objects are:

`pageContext`

The object that provides the environment for the page.

`request`

The object that contains information about the request received from the client, such as the parameters submitted in a form.

`response`

The response being constructed for transmission to the client. For example, you can use it to specify response headers.

`out`

The output object. To add text to the response page, write to this object using methods such as `print()` or `println()`.

`session`

Tomcat provides access to a session object that carries information from request to request. This enables you to write applications that interact with the user across what seems to the user as a cohesive series of events. Web-based session management in JSP is described more fully in *MySQL Cookbook*.

`application`

This object provides access to information that is shared on an application-wide basis.

Scope Levels in JSP Pages

JSP pages can store and access information at several scope levels, which enables pages to control how widely available the information is:

Page scope

> Information available only to the current JSP page.

Request scope

> Information available to any of the JSP pages or servlets that are servicing the current client request. It's possible for one page to invoke another during request processing; placing information in request scope enables such pages to communicate with each other.

Session scope

> Information available to any page servicing a request that is part of a given session. Session scope can span multiple requests from a given client.

Application scope

> Information available to any page that is part of the application context. Application scope can span multiple requests, sessions, or clients.

One application context knows nothing about other contexts, but pages served from within the same context can share information with each other by registering attributes (objects) in one of the scopes that are higher than page scope.

To move information into or out of a given scope, use the `setAttribute()` or `getAttribute()` methods of the implicit object corresponding to that scope (`pageContext`, `request`, `session`, or `application`). For example, to place a string value `tomcat.example.com` into request scope as an attribute named `myhost`, use the `request` object:

```
<% request.setAttribute ("myhost", "tomcat.example.com"); %>
```

`setAttribute()` stores the value as an `Object`. To retrieve the value later, fetch it by name using `getAttribute()` and coerce it back to string form:

```
<%
  Object obj = request.getAttribute ("myhost");
  String host = obj.toString ();
%>
```

When used with the `pageContext` object, `setAttribute()` and `getAttribute()` default to page context. Alternatively, they can be invoked with an additional parameter of `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, or `APPLICATION_SCOPE` to specify a scope level explicitly. The following statements have the same effect as those just shown:

```
<%
  pageContext.setAttribute ("myhost", "tomcat.example.com",
                            pageContext.REQUEST_SCOPE);
  obj = pageContext.getAttribute ("myhost", pageContext.REQUEST_SCOPE);
  host = obj.toString ();
%>
```

Using a Tag Library

A tag library consists of one or more JAR files. To make the library available to the JSP pages of a given application, copy its JAR file or files to the application's *WEB-INF/lib* directory and restart Tomcat.

A JSP page that uses the tag library must include an appropriate `taglib` directive before it refers to any of the actions the library provides:

```
<%@ taglib prefix="name" uri="taglib-identifier" %>
```

For example:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

The `prefix` attribute tells Tomcat how to recognize references to tags from the library in the rest of the JSP page. If you use a `prefix` value of `c`, you can refer to tags later in the page like this:

```
<c:sometag attr1="attribute value 1" attr2="attribute value 2">
tag body
</c:sometag>
```

The `prefix` value is a name of your own choosing, but you must use it consistently throughout the page, and you cannot use the same value for two different tag libraries.

The `uri` attribute tells the JSP processor how the tag library identifies itself. The processor looks in tag library JAR files to discover which one of them matches this value. Documentation for a given tag library should tell you the `uri` value to use when you include references to tags from the library. (Alternatively, unpack the library's JAR file or files and examine its TLD files to find the `<uri>` element that defines the value.)

Overview of JSTL Tags

This section discusses the syntax for some of the JSTL tags used most frequently by JSP pages from the `mcb` application included in the `recipes` distribution that accompanies *MySQL Cookbook*. The descriptions are very brief, and many of these tags have additional attributes that enable use in ways other than those shown here. For more information, consult the JSTL specification.

A JSP page that uses JSTL must include a `taglib` directive for each tag set that the page uses. Examples here use the core and database tags, identified by the following `taglib` directives:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

The `prefix` attribute value is chosen by you and indicates how the rest of the page refers to tags from the library. The `uri` (Uniform Resource Identifier) attribute uniquely identifies the tag library, as described in the section ''Using a Tag Library.''

JSTL tags are written in XML format, using a special syntax for tag attributes to include expressions. Within tag attributes, text is interpreted literally unless enclosed within `${...}`, in which case it is interpreted as an expression to be evaluated. The following sections summarize some of the commonly used core and database tags.

The JSTL Core Tag Set

The following tags are part of the JSTL core tag set:

`<c:out>`

This tag evaluates its `value` attribute and is replaced by the result. It's commonly used to provide content for the output page. The following tag produces the value 3:

```
<c:out value="${1+2}"/>
```

<c:set>

This tag assigns a value to a variable. To assign a string to a variable named `title`, then include that value in the `<title>` element of the output page, do this:

```
<c:set var="title" value="JSTL Example Page"/>

<html>
<head><title><c:out value="${title}"/></title></head>
...
```

This example illustrates a principle that is generally true for JSTL tags: to specify a variable into which a value is to be stored, name it without using `${...}` notation. To refer to that variable's value later, use it within `${...}` so that it is interpreted as an expression to be evaluated.

<c:if>

This conditional tag evaluates the expression given in its `test` attribute. If the expression result is true, the tag body is evaluated and becomes the tag's output; if the result is false, the body is ignored:

```
<c:if test="${1 != 0}">
1 is not equal to 0
</c:if>
```

The comparison operators are `==`, `!=`, `<`, `>`, `<=`, and `>=`. The alternative operators `eq`, `ne`, `lt`, `gt`, `le`, and `ge` make it easier to avoid using special HTML characters in expressions. Arithmetic operators are `+`, `-`, `*`, `/` (or `div`), and `%` (or `mod`). Logical operators are `&&` (or `and`), `||` (or `or`), and `!` (or `not`). The special `empty` operator is true if a value is empty or `null`:

```
<c:set var="x" value=""/>
<c:if test="${empty x}">
x is empty
</c:if>
<c:set var="y" value="hello"/>
<c:if test="${!empty y}">
y is not empty
</c:if>
```

The `<c:if>` tag provides no ''else'' clause. To perform if/then/else testing, use the `<c:choose>` tag.

<c:choose>

This conditional tag enables multiple conditions to be tested. Include a `<c:when>` tag for each condition to test explicitly, and a `<c:otherwise>` tag if there is a ''default'' case:

```
<c:choose>
  <c:when test="${count == 0}">
    Please choose an item
  </c:when>
  <c:when test="${count gt 1}">
    Please choose only one item
  </c:when>
  <c:otherwise>
    Thank you for choosing exactly one item
  </c:otherwise>
</c:choose>
```

`<c:forEach>`

This iterator tag enables you to loop over a set of values. The following example uses a `<c:forEach>` tag to loop through a set of rows in the result set from a query (represented here by the `rs` variable):

```
<c:forEach items="${rs.rows}" var="row">
  id = <c:out value="${row.id}"/>,
  name = <c:out value="${row.name}"/>
  <br />
</c:forEach>
```

Each iteration of the loop assigns the current row to the variable `row`. Assuming that the query result includes columns named `id` and `name`, their values are accessible as `row.id` and `row.name`.

The JSTL Database Tag Set

The JSTL database tags enable you to execute SQL statements and access their results:

`<sql:setDataSource>`

This tag sets up connection parameters to be used when JSTL contacts the database server. For example, to specify parameters for using the Connector/J JDBC driver to access the `cookbook` database, the tag looks like this:

```
<sql:setDataSource
  var="conn"
  driver="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost/cookbook"
  user="cbuser"
  password="cbpass"
/>
```

The `driver`, `url`, `user`, and `password` attributes specify the connection parameters, and the `var` attribute names the variable to associate with the connection. The example uses the variable `conn`; tags occurring later in the page that require a data source refer to the connection using the expression `${conn}`.

To avoid writing connection parameters in multiple JSP pages that use MySQL, put the `<sql:setDataSource>` tag in a file and include the file from each page that needs a database connection. For the `mcb` application from the `recipes` distribution, the include file is *WEB-INF/jstl-mcb-setup.inc*. JSP pages access the file as follows to set up the database connection:

```
<%@ include file="/WEB-INF/jstl-mcb-setup.inc" %>
```

To change the connection parameters used by the `mcb` pages, edit *jstl-mcb-setup.inc*.

`<sql:update>`

To execute a statement such as `UPDATE`, `DELETE`, or `INSERT` that doesn't return rows, use a `<sql:update>` tag. A `dataSource` tag attribute indicates the data source. The rows-affected count resulting from the statement is returned in the variable named by its `var` attribute, and tag body specifies the statement itself:

```
<sql:update dataSource="${conn}" var="count">
  DELETE FROM profile WHERE id > 100
</sql:update>
Number of rows deleted: <c:out value="${count}"/>
```

`<sql:query>`

To process statements that return a result set, use `<sql:query>`. As with `<sql:update>`, the `dataSource` attribute indicates the data source, and the tag body provides the text of the statement. The `<sql:query>` tag also takes a `var` attribute that names a variable to associate with the result set so that you can access the rows of the result:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT id, name FROM profile ORDER BY id
</sql:query>
```

The example uses `rs` as the name of the result set variable. Strategies for accessing result set contents are outlined shortly.

`<sql:param>`

You can write data values literally into a statement string, but JSTL also supports the use of placeholders, which is helpful for values that contain characters that are special in SQL statements. Use a `?` character for each placeholder in the statement string, and provide values to be bound to the placeholders using `<sql:param>` tags in the body of the statement-issuing tag. Specify a data value either in the body of an `<sql:param>` tag or in its `value` attribute:

```
<sql:update dataSource="${conn}" var="count">
  DELETE FROM profile WHERE id > ?
  <sql:param>100</sql:param>
</sql:update>

<sql:query dataSource="${conn}" var="rs">
  SELECT id, name FROM profile WHERE cats = ? AND color = ?
  <sql:param value="1"/>
  <sql:param value="green"/>
</sql:query>
```

The contents of a result set returned by `<sql:query>` are accessible several ways. Assuming that you have associated a variable named `rs` with the result set, you can access row *`i`* of the result either as `rs.rows[`*`i`*`]` or as `rs.rowsByIndex[`*`i`*`]`, where row numbers begin with 0. The first form produces a row with columns accessible by name. The second form produces a row with columns accessible by column number (beginning with 0). For example, if a result set has columns named `id` and `name`, access the values for the third row using column names like this:

```
<c:out value="${rs.rows[2].id}"/>
<c:out value="${rs.rows[2].name}"/>
```

To use column numbers instead, do this:

```
<c:out value="${rs.rowsByIndex[2][0]}"/>
<c:out value="${rs.rowsByIndex[2][1]}"/>
```

To loop through the rows in a result set, use the `<c:forEach>` iterator tag. For column values accessible by name, iterate using `rs.rows`:

```
<c:forEach items="${rs.rows}" var="row">
  id = <c:out value="${row.id}"/>,
  name = <c:out value="${row.name}"/>
  <br />
</c:forEach>
```

For column values accessible by number, iterate using `rs.rowsByIndex`:

```
<c:forEach items="${rs.rowsByIndex}" var="row">
  id = <c:out value="${row[0]}"/>,
  name = <c:out value="${row[1]}"/>
  <br />
</c:forEach>
```

To obtain the number of result set rows, use `rs.rowCount`:

```
Number of rows selected: <c:out value="${rs.rowCount}"/>
```

To obtain the result set column names, use `rs.columnNames`:

```
<c:forEach items="${rs.columnNames}" var="name">
  <c:out value="${name}"/>
  <br />
</c:forEach>
```

Using the Tomcat Server

The preceding parts of this document provide an introduction to servlets, JSP pages, and JSTL tags, but say nothing about how you actually set up a web server to use them. This section describes how to install Tomcat, a JSP-aware web server. Tomcat, like Apache, is a development effort of the Apache Software Foundation. Tomcat distributions and documentation are available here:

```
http://tomcat.apache.org/
```

As described in the section ‘‘Servlet and JavaServer Pages Overview,’’ servlets execute inside a container, which is an engine that communicates with or plugs into a web server to handle requests for pages that are produced by executing servlets. Some servlet containers operate in standalone fashion, such that they function both as container and web server. That is how Tomcat works. By installing it, you get a fully functioning server with servlet-processing capabilities. In fact, Tomcat is a reference implementation for both the servlet and JSP specifications, so it also acts as a JSP engine, providing JSP-to-servlet translation services. The servlet container part is named Catalina. The JSP processor is named Jasper.

It’s possible to use the container part of Tomcat in conjunction with other web servers. For example, you can set up a cooperative arrangement between Apache and Tomcat under which Apache acts as a frontend that passes servlet and JSP requests through to Tomcat and handles other requests itself. Check the Connectors section of the Tomcat web site for information about setting up Apache and Tomcat to work together this way.

To run a Tomcat server, you also need a Java Development Kit (JDK). This is required because Tomcat compiles Java servlets as part of its operation. If a JDK is not already installed on your system, obtain one and install it. Mac OS X comes with a JDK. For Solaris, Linux, and Windows, JDKs are available here:

```
http://www.oracle.com/technetwork/java/index.html
```

In addition, some knowledge of XML is helpful. Tomcat configuration files are written as XML documents, and many scripting elements within JSP pages follow XML syntax rules.

Installing a Tomcat Distribution

To install Tomcat, get a binary distribution from *tomcat.apache.org*. (I assume that you don’t intend to build it from source, which is more difficult.) The instructions here apply to Tomcat 7.0, but may also work for other versions. Tomcat distributions are available in several packaging formats, distinguished by filename extension:

.tar.gz

A compressed *tar* file, usually used for installing on Unix

.zip

A ZIP archive, applicable to either Unix or Windows

.exe

An executable installer, used on Windows only

Your operating system vendor might make Tomcat available in the OS native packaging format. For example, vendors of Linux might provide Tomcat on the distribution media or online via an APT or RPM repository. To install such a package, consult the OS documentation or contact your vendor for details.

For a distribution packaged as a compressed *tar* or ZIP file, place it in the directory under which you want to install Tomcat, and then run the installation command in that directory to unpack the distribution there. The Windows *.exe* installer prompts you to indicate where to install Tomcat, so it can be run from any directory. The following commands are representative of those needed to install each distribution type. Change the version numbers in the filenames to reflect the actual Tomcat distribution that you're using.

To install Tomcat from a compressed *tar* file, unpack it like this:

```
% tar zxf apache-tomcat-7.0.47.tar.gz
```

If you have trouble unpacking a Tomcat *tar* file distribution, use a GNU-compatible version of *tar*, such as *gnutar*.

If you use a ZIP archive, you can unpack it with the *jar* utility or any other program that understands ZIP format, such as the Windows *WinZip* application. For example, with *jar*, use the following command:

```
% jar xf apache-tomcat-7.0.47.zip
```

The Windows *.exe* distribution is directly executable. Launch it, then indicate where to place Tomcat when the installer prompts for a location. The installer also gives you the option of installing Tomcat as a service that starts automatically at system boot time.

The top-level directory of the resulting unpacked Tomcat distribution is the Tomcat root directory. I assume here that the Tomcat root is */usr/local/apache-tomcat* under Unix and *C:\apache-tomcat* under Windows. (The actual directory name likely has a version number at the end.) The Tomcat root contains various text files that may be useful if you have general or platform-specific problems. It also contains several directories. To explore these now, see the section ''Tomcat Directory Structure.'' Otherwise, proceed to the next section, ''Starting and Stopping Tomcat,'' to find out how to run Tomcat.

Starting and Stopping Tomcat

Tomcat can be controlled manually, and also set to run automatically when your system starts. It's good to become familiar with the Tomcat startup and shutdown commands that apply to your platform because you may find yourself needing to stop and restart Tomcat often—at least while you're setting it up initially. For example, if you modify Tomcat's configuration files or install a new application, restarting Tomcat causes it to notice the changes.

Before running Tomcat, set these environment variables:

- Set `JAVA_HOME` to the pathname of your JDK so that Tomcat can find it.
- Set `CATALINA_HOME` to the pathname of the Tomcat root directory.

To control Tomcat manually under Unix, change location into the *bin* directory under the Tomcat root. The following two shell scripts start and stop Tomcat:

```
% sh startup.sh
% sh shutdown.sh
```

To run Tomcat automatically at system boot time, look for a startup script such as */etc/rc.local* or */etc/rc.d/rc.local* and add a few lines to it (the location of the startup script and pathnames in the lines you add to it are system dependent; adjust accordingly):

```
export JAVA_HOME=/usr/local/java/jdk
export CATALINA_HOME=/usr/local/apache-tomcat
$CATALINA_HOME/bin/startup.sh &
```

When the system invokes the script at startup, those commands run Tomcat as `root`. To run Tomcat under a different user account, modify the last line to invoke Tomcat with *su* instead and specify the username:

```
su -c $CATALINA_HOME/bin/startup.sh user_name &
```

If you use *su* to specify a username, make sure that Tomcat's directory tree is accessible to that user or Tomcat will have file permission problems when it tries to access files in that tree. One way to ensure accessibility is to run the following command as `root` in the Tomcat root directory:

```
# chown -R user_name .
```

Linux users who install Tomcat from vendor-supplied package files may find that the installation creates a script named *tomcat* (or perhaps something like *tomcatN*, where *N* is the Tomcat major version number) in the */etc/rc.d/init.d* directory that can be used manually or for automatic startup. To use the script manually, change location into that directory and use these commands:

```
% sh tomcat start
% sh tomcat stop
```

For automatic startup, activate the script by running the following command as `root`:

```
# chkconfig --add tomcat
```

Linux package installation also might create a user account with a login name such as `tomcat` or `tomcat7` intended for use in running Tomcat.

On Windows, the distribution includes a pair of batch files in the *bin* directory for controlling Tomcat manually:

```
C:\> startup.bat
C:\> shutdown.bat
```

If you elected to install Tomcat as a service on Windows, you should control Tomcat using the services console. You can use this to start or stop Tomcat, or to set Tomcat to run automatically when Windows starts. (The service name is `TomcatN`, where *N* is the Tomcat major version number.)

After you start Tomcat using the preceding instructions, request the default page using your browser. The URL looks something like this:

```
http://localhost:8080/
```

Adjust your hostname and port number appropriately. For example, Tomcat normally runs on port 8080, but if you install from package files under Linux, Tomcat might use a port number such as 8180. If your browser receives the page correctly, you should see the Tomcat logo and links to examples and documentation. It's useful at this point to follow the examples link and try a few JSP pages there, to check whether they compile and execute properly.

If you find that Tomcat can't find your Java compiler, despite setting the `JAVA_HOME` variable for the environment in which Tomcat runs, try setting the `PATH` environment variable to explicitly include the directory containing the compiler. Normally, this is the *bin* directory under your JDK installation. If `PATH` is already set, add the *bin* directory to its current value.

Tomcat Directory Structure

To write JSP pages, it's not strictly necessary to be familiar with the hierarchy of Tomcat's directory layout. But it certainly doesn't hurt, so change location into the Tomcat root directory and have a look around. You'll find a number of standard directories, shown in the following table.

Directory Name	Directory Contents
bin	Startup and shutdown scripts
conf	Configuration files
lib	Class files, JAR file libraries
logs	Logfiles
temp	Temporary files
webapps	Applications
work	Servlet files compiled from JSP pages

The following discussion describes these directories, grouped by function. Your installation layout may not be exactly as described here: some distribution formats may omit some directories, and certain operational directories might not be created until you start Tomcat for the first time.

Application directories. From the point of view of an application developer, the *webapps* directory is the most important part of Tomcat's directory hierarchy. Each application context has its own directory, located in the *webapps* directory under the Tomcat root.

Tomcat processes client requests by mapping them onto locations under the *webapps* directory. For a request that begins with the name of a directory located under *webapps*, Tomcat looks for the appropriate page within that directory. For example, Tomcat serves the following two requests using the *index.html* and *simple.jsp* pages in the *mcb* directory:

```
http://localhost:8080/mcb/index.html
http://localhost:8080/mcb/simple.jsp
```

For requests that don't begin with the name of a *webapps* subdirectory, Tomcat serves them from a special subdirectory named *ROOT* that provides the default application context. (The *webapps/ROOT* directory is distinct from the Tomcat root directory; the latter is the top-level directory of the Tomcat tree.) For the following request, Tomcat serves the *index.html* page from the *ROOT* directory:

```
http://localhost:8080/index.html
```

Applications typically are packaged as web archive (WAR) files and Tomcat by default looks for WAR files that need to be unpacked when it starts. Thus, to install an application, copy its WAR file to the *webapps* directory, restart Tomcat, and let Tomcat unpack it. The section ''Web Application Structure'' describes the layout of individual application directories.

A web application is a group of related servlets that work together, without interference from other unrelated servlets. For Tomcat, this means is that an application is everything under a subdirectory of the *webapps* directory, and that scripts in one application directory cannot interfere with anything in another application directory.

Control and configuration directories. Two directories contain control and configuration files. The *bin* directory contains control scripts for Tomcat startup and shutdown, and *conf* contains Tomcat's configuration files, which are written as XML documents. Tomcat reads its configuration files only at startup time. If you modify any of them, restart Tomcat so your changes take effect.

The most important configuration file is *server.xml*, which controls Tomcat's overall behavior. Another file, *web.xml*, provides application configuration defaults. Tomcat uses this file in conjunction with any *web.xml* file an application may have of its own. The *tomcat-users.xml* file defines credentials for users that have access to protected server functions, such as the Manager application that enables you to control applications from your browser. (See ''Restarting Applications Without Restarting Tomcat.'') This file can be

superseded by other user information storage mechanisms. For example, you can store Tomcat user records in MySQL instead. For instructions, look in the *tomcat* directory of the `recipes` distribution.

Class directories. Tomcat uses the *lib* directory for class files and JAR file libraries. Files installed here are available both to Tomcat itself and to all applications. (Tomcat actually looks in the *lib* directories under the locations named by the `CATALINA_BASE` and `CATALINA_HOME` environment variables. But if `CATALINA_HOME` is set to the Tomcat root directory, Tomcat looks in *lib* under the root.)

Operational directories. Tomcat writes logfiles and temporary files to the *logs*, *temp*, and *work* directories, respectively.

The files in the *logs* directory can be useful for diagnosing problems. For example, if Tomcat has a problem starting properly, it usually writes the reason to one of the logfiles.

The Java virtual machine writes temporary files to the *temp* directory.

When Tomcat translates a JSP page into a servlet and compiles it into an executable class file, it stores the resulting *.java* and *.class* files under the *work* directory. To better understand the relationship between JSP pages and servlets, you may find it instructive to have a look under the *work* directory to compare your original JSP pages with the corresponding servlets that Tomcat produces.

Restarting Applications Without Restarting Tomcat

If you modify a JSP page, Tomcat recompiles it automatically when the page is next requested. But if the page uses a JAR or class file under the application's *WEB-INF* directory and you update one of those, Tomcat normally won't notice the change until you restart it.

One way to avoid restarts for an application is to provide a `<Context>` element for the application in Tomcat's *server.xml* file that specifies a `reloadable` attribute of `true`. That causes Tomcat to look for changes not only in JSP pages that are requested directly, but also for changes in classes and libraries under the *WEB-INF* directory that the pages use. For example, to write such a `<Context>` element for an application named `mcb`, add a line like this to Tomcat's *server.xml* file:

```
<Context path="/mcb" docBase="mcb" debug="0" reloadable="true"/>
```

The `<Context>` element attributes tell Tomcat four things:

`path`

: The URL that maps to pages from the application context. The value is the part of the URL that follows the hostname and port number.

`docBase`

: The application context directory location, relative to the *webapps* directory in the Tomcat tree.

`debug`

: The context debugging level. A value of zero disables debug output; higher numbers generate more output.

`reloadable`

: Tomcat recompilation behavior when a client requests a JSP page located in the application context. By default, Tomcat recompiles a page only after noticing a modification to the page itself. Setting `reloadable` to `true` tells Tomcat to also check any classes or libraries stored under the application's *WEB-INF* directory that the page uses.

After modifying *server.xml* to add the `<Context>` element, restart Tomcat so the change takes effect.

Having Tomcat check for class and library changes can be useful during application development to avoid repeated restarts. However, as you might expect, automatic class checking adds a lot of processing overhead and incurs a significant performance penalty. It's better used on development systems than on production systems.

Another way to have Tomcat recognize application changes without restarting the entire server is to use the Manager application. This enables you to reload applications on request from a browser, without the overhead caused by enabling the `reloadable` attribute. Invoke the Manager application using the path */manager/html* at the end of the URL you use to access your Tomcat server. The URL also includes the command that you want to execute. For example, the following request shows which contexts are running:

```
http://localhost:8080/manager/html/list
```

The manager displays a page that enables contexts to be stopped, started, reloaded, or undeployed. For more information on using the Manager application and what its permitted commands are, see the Manager App HOW-TO:

```
http://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html
```

That document may also be available locally by following the documentation link on your Tomcat server's home page. Note particularly the section on configuring manager application access that describes how to set up a Tomcat user with the `manager-gui` role; you must provide a name and password to gain access to the Manager application. By default, user records are defined in Tomcat's *tomcat-users.xml* configuration file, but you can store Tomcat user records in MySQL instead. For instructions, look in the *tomcat* directory of the `recipes` distribution.

Web Application Structure

Each web application corresponds to a single servlet context and exists as a collection of resources. Some of these resources are visible to clients, others not. For example, an application's JSP pages may be available to clients, but the configuration, property, or class files used by the JSP pages can be hidden. The location of components within the application hierarchy determines whether clients can see them, so you can make resources public or private depending on where you place them.

The Java Servlet Specification defines the standard for web application layout. This helps application developers by providing conventions that indicate where to put what, along with rules that define which parts of the application the container makes available to clients and which parts it hides.

In Tomcat, web applications are represented by directories under the *webapps* directory. Within a given application directory, you'll find a *WEB-INF* subdirectory, and usually other files such as HTML pages, JSP pages, or image files. The files located in the application's top-level directory are public and may be requested by clients. The *WEB-INF* directory has special significance. Its mere presence signifies to Tomcat that its parent directory actually represents an application. *WEB-INF* is thus the only required component of a web application; it must exist, even if it's empty. If *WEB-INF* is nonempty, it typically contains application-specific configuration files, classes, and possibly other information. Three of its most common primary components are:

```
WEB-INF/web.xml
WEB-INF/classes
WEB-INF/lib
```

The *web.xml* file is the web application deployment descriptor. It gives the container a standard way to discover how to handle the resources that make up the application. The deployment descriptor is used for purposes such as defining the behavior of JSP pages and servlets, setting up access control for protected information, and specifying error pages to be used when problems occur.

The *classes* and *lib* directories under *WEB-INF* hold class files and libraries, and sometimes other information. Individual class files go under *classes*, using a directory structure that corresponds to the class

hierarchy. (For example, a class file *MyClass.class* that implements a class named `com.kitebird.jsp.MyClass` must be stored in the *classes/com/kitebird/jsp* directory.) Class libraries packaged as JAR files go in the *lib* directory instead. Tomcat looks in the *classes* and *lib* directories automatically when processing requests for pages from the application. This enables pages to use application-specific information with a minimum of fuss.

The *WEB-INF* directory is private. Its contents are available to the application's servlets and JSP pages but cannot be accessed directly through a browser. For example, you can store a properties file under *WEB-INF* that contains connection parameters for a database server. Or if you have an application that permits image files to be uploaded with one page and downloaded later from another page, putting the images into a directory under *WEB-INF* makes them private. Because Tomcat will not serve the contents of *WEB-INF* directly, your JSP pages can implement an access control policy that determines who can perform image operations. (A simple policy might require clients to specify a name and password before being permitted to upload images.) The *WEB-INF* directory is also beneficial in giving you a known location for private files that is fixed with respect to the application's root directory, no matter the host machine or where on that machine you deploy the application.

Tomcat interprets the *WEB-INF* directory name in case-sensitive fashion, even on systems with filenames that are not case sensitive, such as Windows. On such systems, take care not to create the *WEB-INF* directory with a name like *Web-Inf*, *web-inf*, and so forth. The operating system itself may not consider the name different from *WEB-INF*, but Tomcat does, with the result that the resources in the directory are unavailable to your JSP pages.

The preceding discussion describes web application layout in terms of a directory hierarchy because that's the easiest way to explain it. However, an application need not necessarily exist that way. A web application typically is packaged as a WAR file, using the standard layout for components prescribed by the servlet specification. But some containers can run an application directly from its WAR file without unpacking it. Furthermore, a container that does unpack WAR files is free to do so into any filesystem structure it wishes.

Tomcat uses the simplest approach, which is to store an application in the filesystem using a directory structure identical to the directory tree from which the file was originally created. To see this correspondence, compare the structure of a WAR file to the directory hierarchy that Tomcat creates by unpacking it. For example, to examine the WAR file for the `mcb` application, use this command:

```
% jar tf mcb.war
```

The list of pathnames displayed by the command corresponds to the layout of the *mcb* directory created by Tomcat when it unpacks the file under the *webapps* directory. To verify this, recursively list the contents of the *mcb* directory using one of these commands:

```
% ls -R mcb                    (Unix)
C:\> dir /s mcb                (Windows)
```

Were you to set up a context manually for an application named `myapp`, the steps would be like those described in the following procedure. (To see what the resulting application hierarchy should be, have a look at the *tomcat/myapp* directory of the `recipes` distribution.)

1. Change location into the *webapps* subdirectory of the Tomcat directory tree.
2. Create a directory in the *webapps* directory with the same name as the application (*myapp*), then change location into that directory.
3. In the *myapp* directory, create a directory named *WEB-INF*. The presence of this directory signals to Tomcat that *myapp* is an application context, so it must exist. Then restart Tomcat so it notices the new application.
4. Create a short test page named *page1.html* in the *myapp* directory that you can request from a browser to verify that Tomcat serves pages for the application. This is a plain HTML file, to avoid complications that arise from use of embedded Java, tag libraries, and so forth:

```
<html>
<head><title>Test Page</title></head>
<body>
<p>This is a test.</p>
</body>
</html>
```

To request the page, use a URL like this, adjusting appropriately for your own server hostname and port number:

```
http://localhost:8080/myapp/page1.html
```

5. To try a simple JSP page, make a copy of *page1.html* named *page2.jsp*. That creates a valid JSP page (even though it contains no executable code), so you should be able to request it and see output identical to that produced by *page1.html*:

```
http://localhost:8080/myapp/page2.jsp
```

6. Copy *page2.jsp* to *page3.jsp* and modify the latter to contain some embedded Java code by adding lines that print the current date and client IP number:

```
<html>
<head><title>Test Page</title></head>
<body>
<p>This is a test.</p>
<p>Current date: <%= new java.util.Date () %></p>
<p>Your IP address: <%= request.getRemoteAddr () %></p>
</body>
</html>
```

The `Date()` method returns the current date, and `getRemoteAddr()` returns the client IP number. After making the changes, request *page3.jsp* from your browser and the output should include the current date and the IP number of the host from which you requested the page.

At this point, you have a simple application context that consists of three pages (one of which contains executable code) and an empty *WEB-INF* directory. For many applications, *WEB-INF* contains a *web.xml* web application deployment descriptor file that tells Tomcat how the application is configured. If you examine *web.xml* files from other applications, you'll find that they can be complex. A minimal deployment descriptor file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         version="2.5" metadata-complete="true">

</web-app>
```

Adding information to the *web.xml* file is a matter of placing new elements between the `<web-app>` and `</web-app>` tags. As a simple illustration, add a `<welcome-file-list>` element to specify a list of files that Tomcat should look for when clients send a request URL that ends with *myapp* and no specific page. Whichever file Tomcat finds first becomes the default page that is sent to the client. For example, to specify that Tomcat should consider *page3.jsp* and *index.html* to be valid default pages, in that order, create a *web.xml* file in the *WEB-INF* directory that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         version="2.5" metadata-complete="true">

  <welcome-file-list>
    <welcome-file>page3.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Restart Tomcat so it reads the new application configuration information, then issue a request that specifies no explicit page:

```
http://localhost:8080/myapp/
```

The *myapp* directory contains a page named *page3.jsp*, which is listed as one of the default pages in the *web.xml* file, so Tomcat should execute *page3.jsp* and send the result to your browser.

Revision History

- 1.0—The version in *MySQL Cookbook*, first edition, Appendix B.
- 2.0—The version in *MySQL Cookbook*, second edition, Appendix C.
- 3.0—Initial standalone version, to accompany *MySQL Cookbook*, third edition.