

1

Getting Started with JavaServer Pages

JavaServer Pages (JSP) is a Java-based technology that is run on a server to facilitate the processing of Web-based requests. Many of the Web sites that you visit daily may be using JSP to format and display the data that you see. This chapter reveals what JSP is, how it works, and why it is important.

The evolution of request processing using Java-based server logic is also presented in this chapter. JSP plays a vital role in this evolution. This role, along with how JSP assists in Web request processing, will be discussed. This overview serves as a foundation upon which to build new JSP concepts and to introduce new JSP features in later chapters.

Every chapter in this book contains hands-on JSP coding examples. Starting from this very first chapter you will be working immediately with JSP coding. This chapter shows in detail how to set up JSP code on your own Windows-based PC or Linux/UNIX workstation.

In particular, this chapter:

- ❑ Provides a historical review of the Web technology evolution that leads to JSP
- ❑ Discusses why JSP is needed
- ❑ Reveals how JSP works
- ❑ Shows where to download chapter code examples and the JSP Project examples
- ❑ Shows where to download a server for executing JSP code on your PC or workstation
- ❑ Reveals how to set up the open-source Tomcat server for running your JSP code

Creating Applications for the Internet

Before looking at a server that supports JSP, think of what happens under the hood when you use your browser to access a Web site. It's likely you're using one of the popular Web browsers, such as Netscape, Microsoft Internet Explorer, Firefox, Konquerer, or Opera. Figure 1-1 illustrates the sequence of events that occurs when the browser accesses a URL.

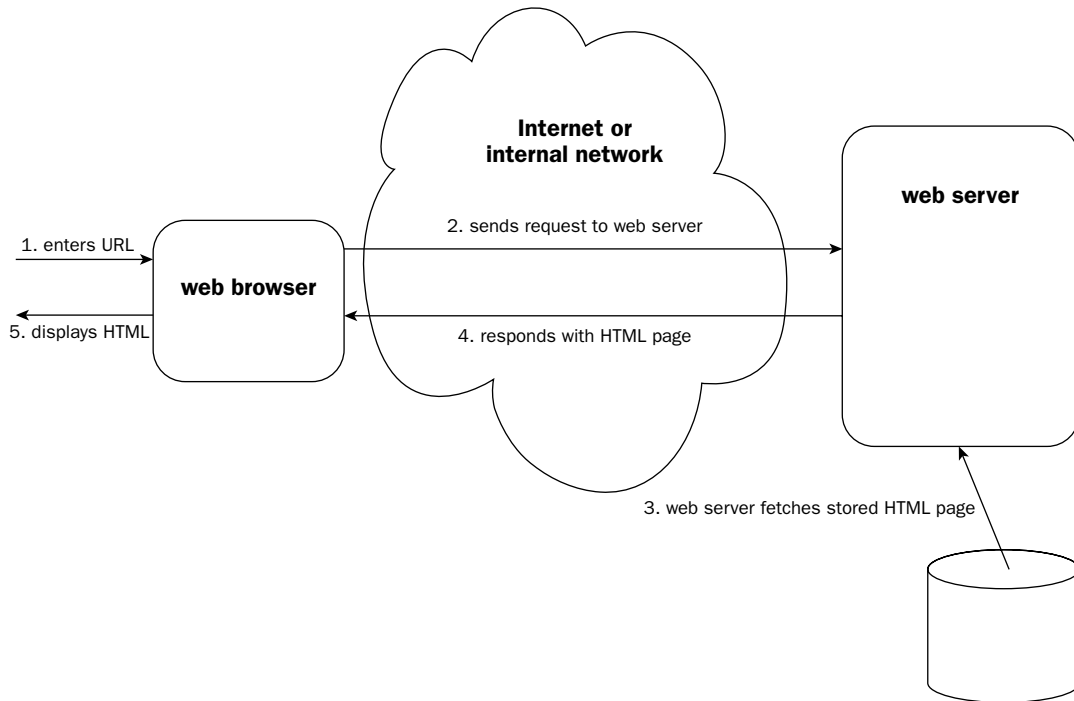


Figure 1-1: Web browser accessing a URL

The following steps correspond numerically with the numbered steps in Figure 1-1:

1. Enter the URL of a Web page into your browser. This URL tells the browser to contact a specific machine on the Internet.
2. The browser then sends the request to the specified machine on the Internet. The machine specified runs a piece of software called a *Web server*. The Web server receives the request and examines it. Popular Web servers include Apache, Microsoft Internet Information Services (IIS), Netscape Enterprise Server, Sun Java System Web Server (formerly Sun ONE), Oracle HTTP Server, and Zeus Web Server.
3. Depending on the request received, the Web server retrieves from its storage a Web page encoded in HTML.
4. The page acquired in Step 3 is passed back to the requesting browser as a response.
5. The browser, after receiving the response Web page, displays it to the user.

Of course, the Web page can contain graphical elements such as GIF files (the browser will have to issue additional requests to the server for these files), as well as hyperlinks to other URLs that the user can click.

HTML (Hypertext Markup Language) is the standard format in which Web pages are coded. HTML files are text-based files that can be edited in any text editor. An HTML page consists of tagged sections such as a header and a body, and formatted layout elements such as paragraphs and tables. All browsers understand HTML and will display (called *rendering* in HTML lingo) the page according to the formatting tags. For more information on HTML, check out *Beginning Web Programming with HTML, XHTML, and CSS* (Wrox Press; ISBN 0-7645-7078-1).

In the preceding process, the browser talks to the Web server over the Internet. This conversation is carried out via a standard network protocol. This particular protocol is appropriately called *HTTP* (*Hypertext Transfer Protocol*). HTTP is built on top of TCP/IP, the protocol suite that ties all the computers in the Internet together.

Limitations of the basic Web server model

The basic function of the Web server restricts it to serve a finite number of static Web pages. The content of each page remains the same. There is no easy way to show information that may change, such as today's weather, the latest news, or the current product list offered by an online store. A new set of static pages needs to be created to show new information.

Creating new static pages for every minute change of the underlying information is tedious. Obviously, a lot of time and effort could be saved if there were some way for the server to automatically generate portions of the HTML page. Doing so would eliminate the need to repeatedly create new static pages as information changes. This generation should happen dynamically when the request is processed. For example, it could generate the portion of the HTML page that displays the current date and time.

Internet software engineers quickly turned their attention to the Common Gateway Interface (CGI) to provide this dynamic generation capability.

Dynamic HTML generation via CGI

CGI provides a way to execute a program on the server side. This may be on the same machine running the Web server, or it may be on another machine connected to it. The CGI program's output is the HTML page that will be sent back to the Web browser for display. Figure 1-2 illustrates basic CGI operations.

- 1.** First the browser is instructed to access a URL. You may be entering this URL by hand. More likely, a CGI URL is accessed after you fill out an online form or click a hyperlink on a page already displayed. For example, the URL may be `http://www.wrox.com/beginjsp/ch1test.cgi`. This URL tells the browser to contact a specific machine on the Internet called `www.wrox.com`.
- 2.** The browser then sends the request to the specified machine on the Internet. This is identical to the non-CGI case in Figure 1-1. In addition to the machine, the URL also specifies a specific CGI program location. The portion of the URL that specifies the location is `beginjsp/ch1test.cgi`. The Web server examines the incoming request's URL and forwards the incoming request to the specified CGI program (`ch1test.cgi`).

Chapter 1

3. The CGI program is executed on the server side.
4. The CGI program's output is captured by the Web server.
5. This CGI program's output is passed back to the requesting Web browser using the HTTP protocol.
6. The client Web browser finally displays the output from the CGI program as an HTML page.

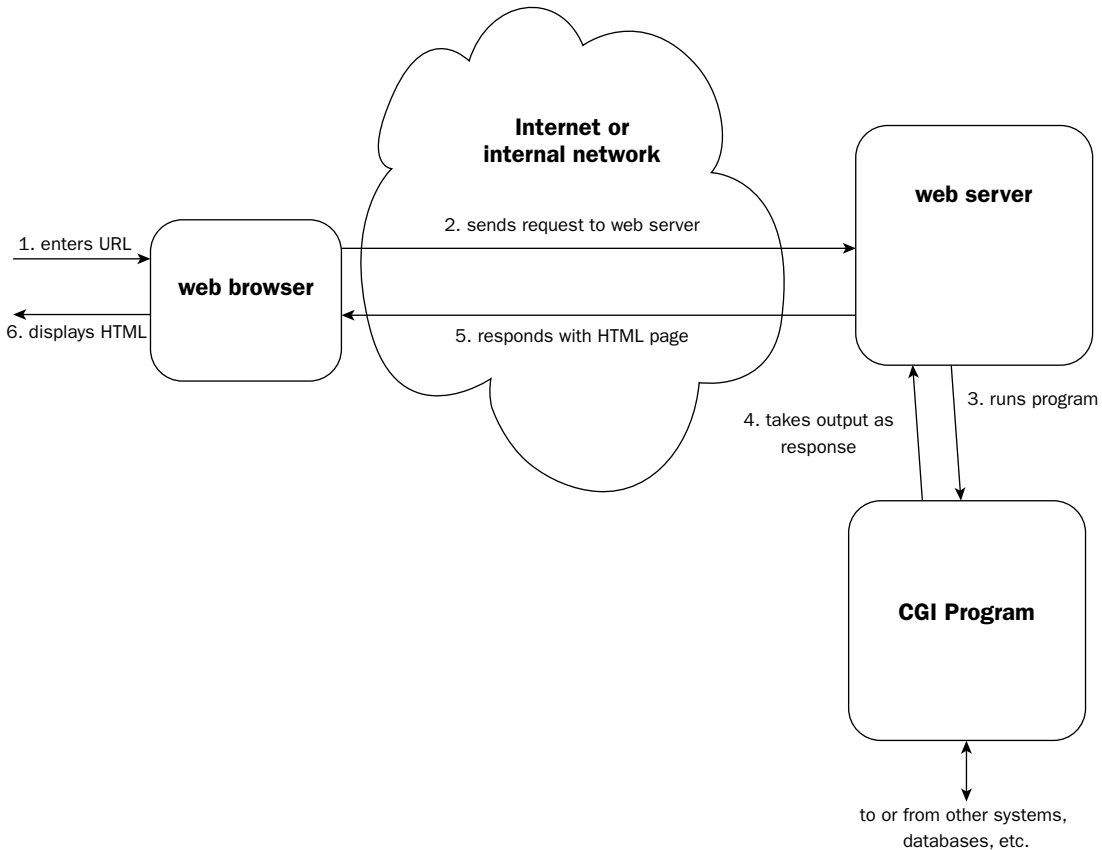


Figure 1-2: Basic CGI operations

The CGI program can be programmed using any computer programming language (the most popular CGI languages are Perl or a shell script, but C, C++, or Java can also be used). The output that the CGI program generates is not limited to a finite set of static pages. Furthermore, the CGI program can access additional resources in generating the output. For example, a CGI program displaying the available quantity for a product sold over the Internet may access a relational database containing current inventory information.

Elegant as CGI seems to be initially, it has major shortcomings. These shortcomings, described in the following section, are evident when a single CGI program is accessed by many users.

Shortcomings of CGI

The major shortcomings of basic CGI are as follows:

- ❑ The overhead of starting an operating system process for each incoming request
- ❑ The overhead of loading and running a program for each incoming request
- ❑ The need for tedious and repetitive coding to handle the network protocol and request decoding

The first two operations consume a large number of CPU cycles and memory. Because both operations must be performed for each incoming request, a server machine can be overloaded if too many requests arrive in a short period of time. Because each CGI program is independent of the other, and often reflect incompatible programming languages, it is not possible to share code used in networking and decoding.

The Java programming language can be used to create CGI programs. Unfortunately, using the Java programming language for CGI amplifies some of the shortcomings of basic CGI described in the preceding list. Early attempts to use the Java programming language to create CGI programs created servers that were extremely slow and inefficient, and crashed frequently due to overloading.

Java-based CGI is not suitable for handling CGI requests because Java is inherently an interpreted language. Because of this, a very large program called the Java Virtual Machine (JVM) must be started to handle an incoming request. The cycle of starting a system process, then a JVM within the process, and then running the Java CGI code within the JVM just for processing a single request is very expensive in terms of both computing cycles and resources. Worse, the whole process needs to be repeated for each incoming request. When compared to the time it takes to process the request and generate the output, this overhead can be significant. If the server needs to handle many incoming requests, the overhead can overwhelm the system.

Improving Java-based CGI: servlets

The Java-based CGI scenario can be improved if the overhead can be eliminated. If some way exists to process all incoming requests by initially starting a single operating system process with a single JVM image, the overhead can be eliminated.

Because the Java platform can load new classes during runtime dynamically, this capability can be used to load new Java code (classes) to handle incoming requests. In other words, a server-side process is started once and loaded with the JVM once, but additional classes are loaded by the JVM to process incoming requests. This is significantly more efficient. In this scenario, the following can be observed:

- ❑ The overhead of starting an operating system process for each request is eliminated.
- ❑ The overhead of loading a JVM for each request is eliminated.
- ❑ Java classes are loaded by the JVM to process incoming requests; if more than one request requires the same processing, the already loaded class can be used to handle it, eliminating even the class loading overhead for all but the first request.
- ❑ Code that handles the networking protocol and decodes incoming requests can be shared by all the dynamically loaded request processing Java classes.

The Java Web Server from Sun Microsystems released in the late 1990s worked exactly in this way. This Web server, written in the Java programming language, started up a JVM to handle all incoming requests and in turn loaded additional Java classes as needed to process specific requests.

In order to ensure that the Java classes loaded for handling requests do not step over one another, or stop the entire Web server altogether, a coding standard was created, which these classes must follow. This standard is called the *Java Servlets API (Application Programming Interface)*. The dynamically loaded classes for processing are known as *servlets*.

The portion of code that manages the loading, unloading, reloading, and execution of servlets is called a *servlet container*. Servlets may need to be unloaded if the system is running out of memory and some servlets have not been used for a very long time. A servlet may need to be reloaded if its code has been modified since it was last used.

There are many different ways to configure the Web server and servlet container. The next section discusses the most popular scenarios.

Web server configurations: integrating servlet containers

Depending on the Web server and servlet container used, very different configurations are possible. Some popular configurations include the following:

- ❑ The same JVM runs the Web server as well as the servlets; in this case, the Web server is coded in Java, as is also the case with the Java Web Server mentioned earlier. This is often called the *standalone configuration*. Figure 1-3a illustrates the standalone configuration.
- ❑ The Web server is not written in the Java programming language, but it starts a JVM within the same operating system process; in this case, the information is passed directly from the Web server into the JVM hosting the servlet container. (Some versions of both the Apache Web server and Microsoft IIS can work in this way.) This is often called the *in-process configuration*. Figure 1-3b illustrates the in-process configuration.
- ❑ The Web server is not written in the Java programming language and runs in a separate operating system process from the servlet container; in this case, the Web server passes the request to the servlet container, using either a local network or operating-system-specific interprocess communications mechanism (a typical configuration for the Apache or Microsoft's IIS Web server). This is often called the *independent configuration* or *networked configuration*. Figure 1-3c illustrates this configuration.

The first two configurations in the preceding list have the advantage that the JVM runs within the same OS process as the Web server. This enables rapid transfer of request information and processing output to and from the CGI code. Conversely, if the servlet container or one of its servlets crashes, the entire Web server may crash because they are in the same process.

The third configuration is less efficient when it comes to the transfer of request data between the Web server and the servlet container. However, the servlet container can crash and restart without affecting the operation of the Web server. This form creates a more robust system for handling Web requests.

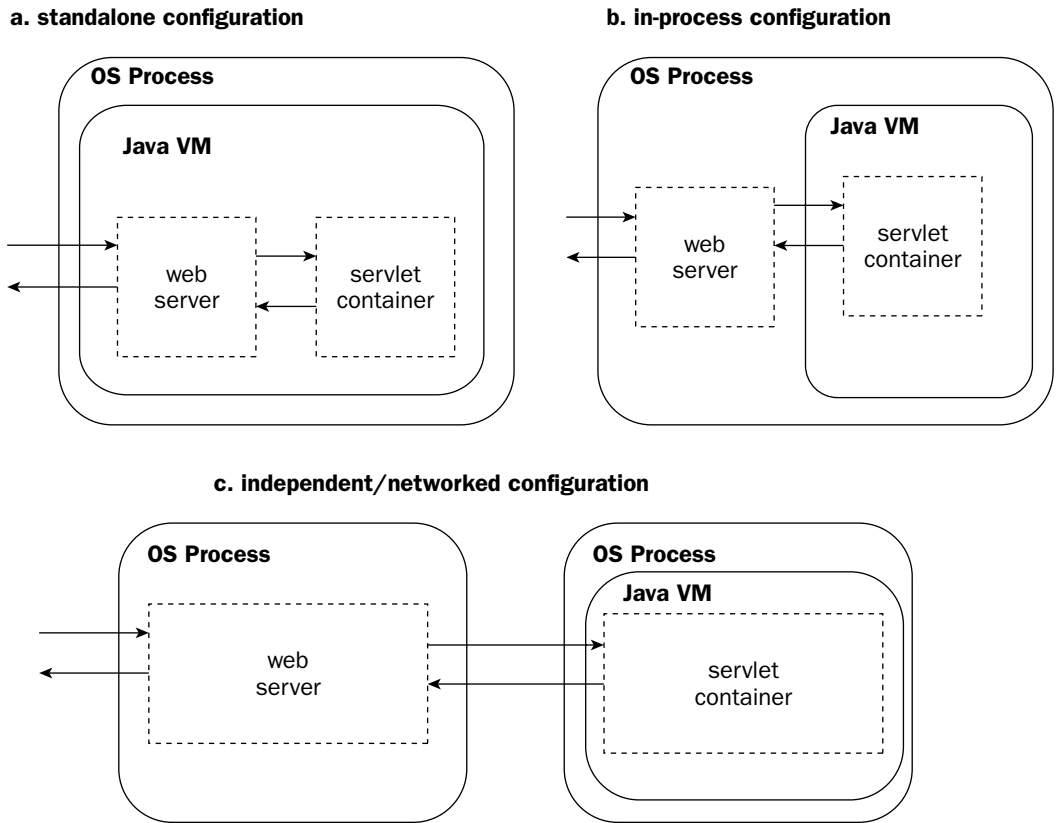


Figure 1-3: Web server and servlet container configurations

Each of the three configurations described exhibit superior performance characteristics when compared to the basic CGI operation introduced earlier.

Optimizing servlet development: JavaServer Pages

Servlets can be viewed as an efficient way of performing CGI operations using the Java programming language. Java classes representing the servlet are dynamically loaded by the servlet container when they are needed. Bear in mind, however, that each servlet can be composed of several compiled Java classes. This means that a programmer will first write the Java code, then compile it, and then register and execute (or *deploy*) the code via the servlet container. Should there be any need to modify the servlet, the Java source code must be modified, recompiled, and then re-deployed via the servlet container. Figure 1-4 illustrates the steps in the servlet development process.

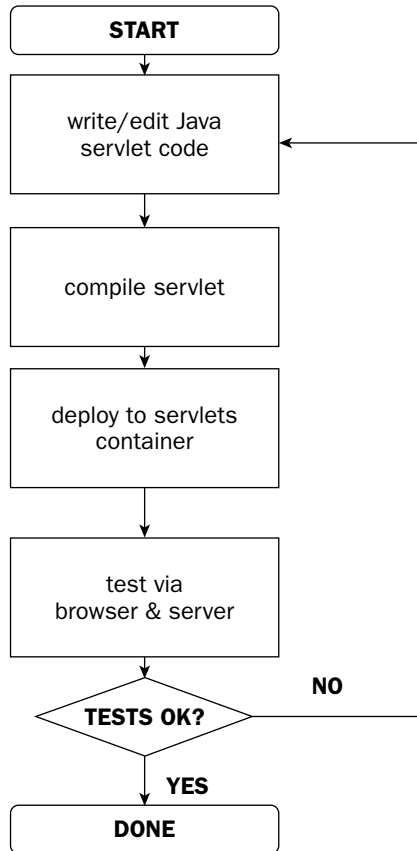


Figure 1-4: Steps in servlet development

Refer back to the basic CGI mechanism presented earlier, and it is evident that the main purpose of most servlets is to generate an HTML page. As a result, you may see servlet Java source code that looks like the following:

```
public class HelloWorldExample extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        String msg = "Hello, world!";
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>JSP 2.0 Hello World</title>");
        out.println("</head>");
        out.println("<body>");
        out.println(msg);
    }
}
```



```
        out.println("</body>");
        out.println("</html>");
    }
}
```

Aside from the servlet programming details, note the repeated use of the `PrintWriter.println()` method to generate the output HTML page. You can see that every time you need to make a minor change to the HTML, the Java source code must be modified, recompiled, and re-deployed. This created a lot of tedious work for early servlet developers. In fact, this process hopelessly bound together the work of Web page design with the work of server logic design and made the programmer responsible for both!

JavaServer Pages (or *JSP*) was introduced to solve these problems. Specifically, JSP provides the following benefits:

- ❑ A templating mechanism whereby Java-based logic can be embedded within HTML pages
- ❑ Automatic detection and recompilation whenever the JSP is changed

When using JSPs, it is not necessary to write or compile any code in Java programming language. The development cycle can be very quick. Modifications to the JSP can be viewed immediately because the *JSP container* (or *JSP engine*, or *JSP runner*, as it is sometimes called) will automatically recompile the JSP.

Unlike servlets, JSPs are not written in the Java programming language (although some JSPs may contain embedded Java coding). Instead, they are text-based templates. This is best illustrated with an example. The following JSP will display the same output as the servlet presented previously:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
  <head>
    <title>JSP 2.0 Hello World</title>
  </head>
  <body>
    <tags:helloWorld/>
  </body>
</html>
```

Note that the preceding JSP is essentially an HTML page with some special markup tags. JSPs are typically stored in source files with a `.jsp` file extension. When the JSP is accessed by an incoming request, the JSP container will parse the special tags and replace them with dynamic output. For example, the preceding JSP may produce the following output (just as the previous servlet) after processing by the JSP container. The examples later in this chapter explain in detail how this JSP works.

```
<html>
  <head>
    <title>JSP 2.0 Hello World</title>
  </head>
  <body>
    Hello, world!
  </body>
</html>
```

In most cases, the JSP container is actually a servlet, enabling JSPs to work within a basic servlet container. However, this detail is not important to the operation of JSP, and need not concern the beginning JSP developer.

It should be clear from this discussion that JSP greatly simplifies the development and construction of server-side Java-based CGI. JSP enables the rapid development of server-side applications that create dynamic output. There is one additional benefit to JSP that may not be immediately evident. Take a second look at the JSP template. Note that although the JSP tags are embedded in the HTML file, the general HTML page structure is still completely intact. In fact, it is possible to have professional Web page designers (typically graphic design professionals with little or no programming skills) design the template. This enables the task of Web site graphics and layout design to be decoupled from the application development task. The page designer can work on the HTML portion of the template, while the JSP developer can work on the tags used and their placement.

Now is a good time to set up a system and try some actual JSP code.

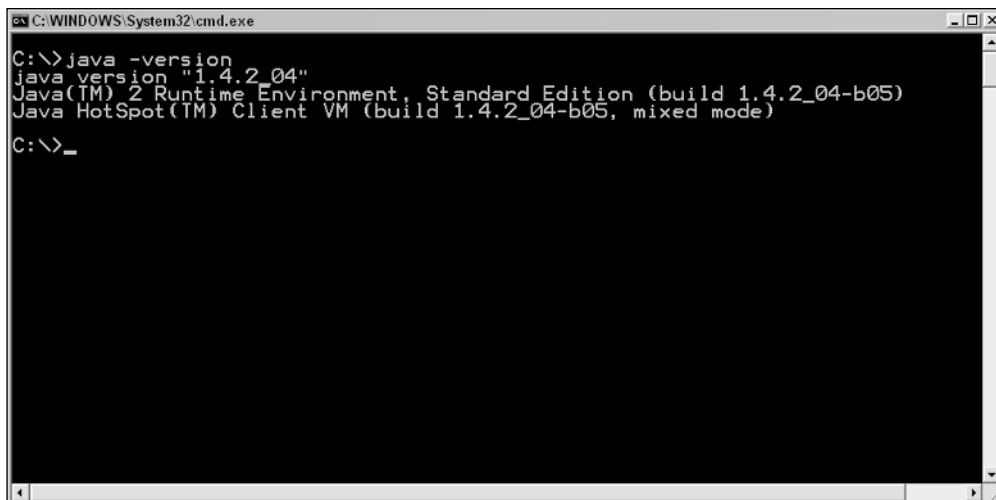
Try It Out Getting Tomcat Up and Running

This section provides step-by-step download and setup instructions for a servlet container and JSP engine called Tomcat. Instructions are provided for both a Windows-based system and a Linux or UNIX-based operating system, such as Red Hat Linux or FreeBSD.

As with all Try It Out sections, this section focuses on the hands-on aspects. If you come across questions during the procedure, it is likely that they are answered in the next "How it Works" section. You may want to read ahead if you cannot wait for the answer.

Checking your system for Java

Before downloading Tomcat, you should ascertain that you have the Java platform installed and running. This book requires version 1.4.2 or later. To determine if you have Java platform installed and running, type in the following command: On either Windows or a Linux/UNIX system, open a command prompt and type in the `java -version` command. Figure 1-5 shows Java version 1.4.2_04-b05 running on a Windows system



```
C:\WINDOWS\System32\cmd.exe
C:\>java -version
java version "1.4.2_04"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_04-b05)
Java HotSpot(TM) Client VM (build 1.4.2_04-b05, mixed mode)
C:\>_
```

Figure 1-5: Verifying the Java platform version on a Windows system

If you do not get the Java version information, contact your system administrator to obtain the configuration for running Java programs. If it is your own machine, install the latest JDK. After the JDK is installed you should ensure that the `JAVA_HOME` environment variable is set to the Java installation directory, and that the `bin` directory under `JAVA_HOME` is added to your `PATH` environment variable.

Downloading Tomcat

You need to determine the latest version of Tomcat before downloading it. The latest version of the Tomcat container can be determined by checking the following URL:

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

Figure 1-6 shows a recent version of this page. Notice the table in the center of the page. This table indicates that 5.0.28 is the latest version that supports JSP Standard 2.0 (and Servlet standard 2.4). This book will make extensive use of features in JSP 2.0.

The ongoing updates of Tomcat server will transition to the Tomcat 5.5.x version. The configuration, deployment procedures, and level of support for JSP are identical between Tomcat 5.0.28 and Tomcat 5.5.x. All of the examples in this book will work with Tomcat 5.5.x.

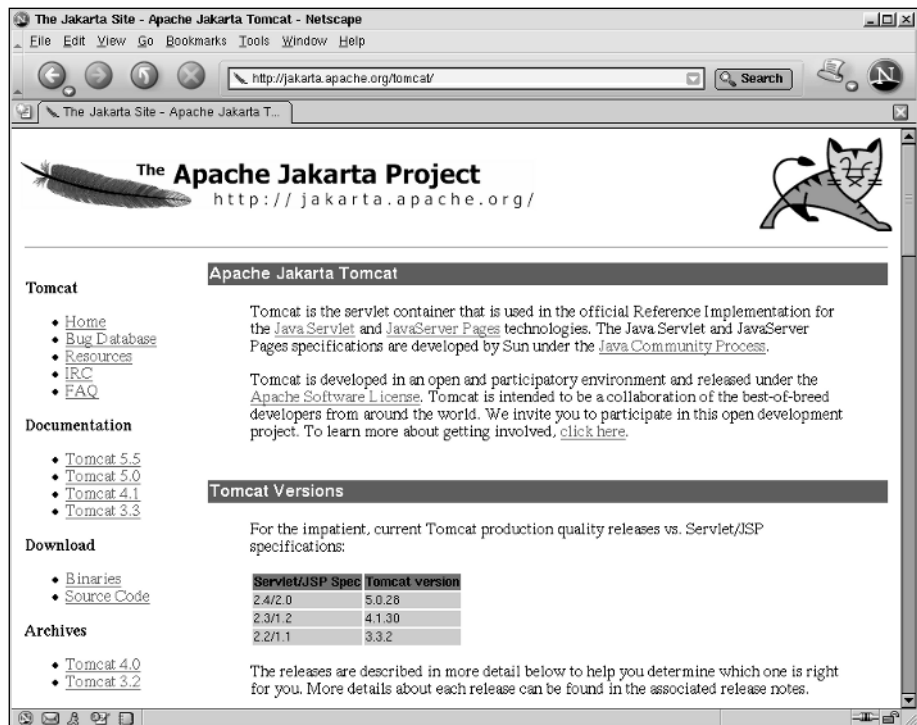


Figure 1-6: Tomcat's welcome page

Click the Binaries link under the Download heading on the left side of the Tomcat index page. Figure 1-7 shows this page.

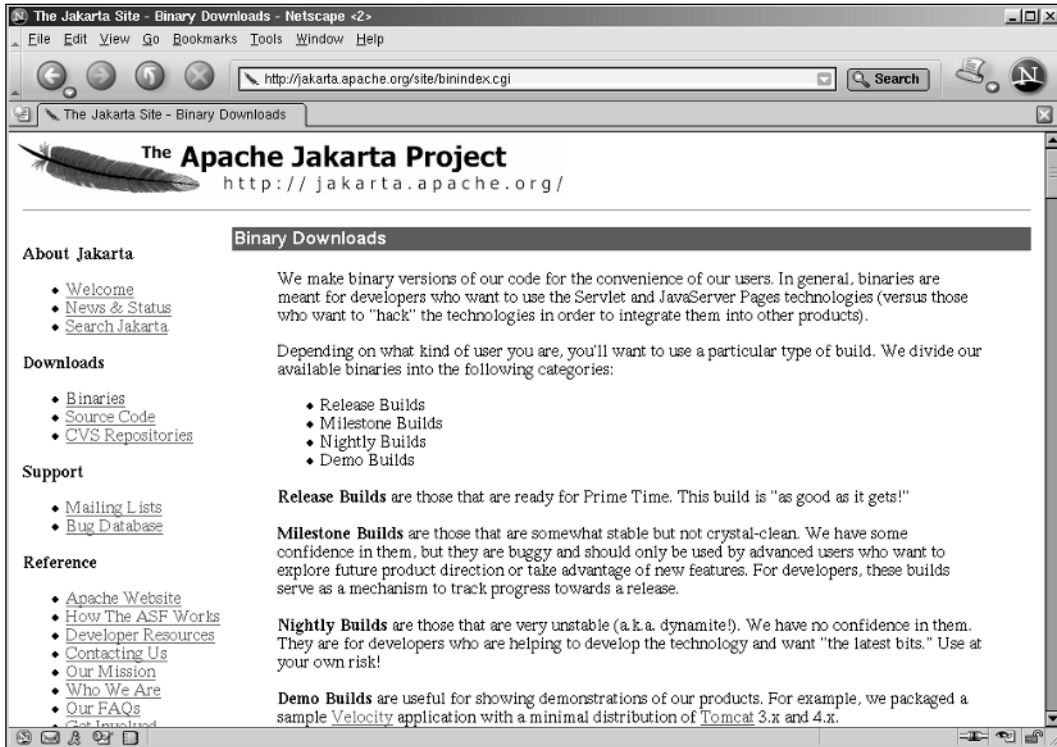


Figure 1-7: Tomcat's Binary Downloads page

From the Binary Downloads page, you will need to scroll down to the bottom and locate the latest Tomcat 5 version. In our case, it is Tomcat 5.0.28.

Now you have a choice of downloads:

- If you are working with Linux or another UNIX operating system, click and download the `.tar.gz` file (`jakarta-tomcat-5.0.28.tar.gz` in our case).
- If you are working with a Windows operating system, click and download the `.exe` file (`jakarta-tomcat-5.0.28.exe` in our case).

Installing Tomcat

After downloading, you need to extract the Tomcat installation on a Linux system. The following code shows how this is done:

```
tar zxvf jakarta-tomcat-5.0.28.tar.gz
```

This will extract Tomcat 5 into a subdirectory called `jakarta-tomcat-5.0.28` under the current directory. You may wish to rename this directory to one with a shorter name.

This directory will be referred to as <Tomcat Installation Directory> in future chapters.

This completes the Linux-based installation.

On a Windows system, execute the downloaded file (by selecting Open or by double-clicking the downloaded `jakarta-tomcat-5.0.28.exe` file). This will start the wizard-based windows installer. Keep in mind the following during installation:

- ❑ When the installer asks you to choose components, choose the Normal install, which is the default.
- ❑ When the installer asks you to choose an installation location, change the default folder to `c:\tomcat5` or a similar directory of your choice.

This will be referred to as the <Tomcat Installation Directory> in future chapters.

- ❑ When the installer asks you to enter configuration options, enter the administrator login username of **tomcat**, and choose a password that you can remember for the administrator user. The discussion following assumes that you used the password **tomcat**.

Accept the default for all other prompts during installation. Tomcat 5 will start up at the end of the installation.

You can also install on Windows using the ZIP file version of Tomcat. In this case, the installation is similar to that on UNIX/Linux, as all you have to do is to extract out the ZIP file in a directory of choice.

Adding an administrative user and password on a Linux system

If you are on a Linux/UNIX system, or have installed Tomcat using the ZIP file, and not the Windows installer executable, you will need to add the administrative username and password manually.

To do so, go to the `<Tomcat Installation Directory>/conf` directory. Using a text editor, make the following highlighted modifications to the file called `tomcat_users.xml`:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user username="tomcat" password="tomcat" roles="tomcat,manager"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
</tomcat-users>
```

The manager role has been added to the user named `tomcat`. This will add the user `tomcat` to the *access control list* for the Manager utility. This utility can be used to add applications to a running Tomcat server; the “Deploying Chapter Examples” section shows how this is done. An access control list specifies which users are allowed to use the utility, and which password needs to be entered.

Save the changes to the `tomcat_users.xml` file.

Starting and shutting down Tomcat 5

On a Windows system, you should see the *Apache Process Runner* (Tomcat Launcher) running on your system tray (lower right-hand corner), with a green arrow indicating that Tomcat 5 is running. Alternatively, you can always start Tomcat 5 by clicking the Start button and choosing Programs (or All Programs with Windows XP) ⇨ Apache Tomcat 5 ⇨ Start Tomcat.

To shut down Tomcat 5 on a Windows system, right-click the Apache Process Runner on your system tray and select Shutdown ⇨ Tomcat5. Tomcat will be shut down and the Apache Process Runner will disappear from the system tray.

To start the Tomcat 5 server on a Linux/UNIX installation, change directory to the *<Tomcat Installation Directory>/bin*. From the shell, run the `startup.sh` script via the following command:

```
sh startup.sh
```

In the same directory is a `shutdown.sh` script. To shut down the Tomcat 5 server on a Linux system, issue the following command:

```
sh shutdown.sh
```

*Note that even on Windows systems, there are two batch files, `startup.bat` and `shutdown.bat`, in the *<Tomcat Installation Directory>/bin* directory. These files can be used from a command prompt with the proper environment setup to start or shut down Tomcat 5. If you wish to use these command-line batch files, make sure you have the environmental variable `JAVA_HOME` set to point to the installation directory of your Java SDK.*

Verifying your Tomcat 5 installation

To verify that your version of Tomcat is installed properly, try to start a browser and access the following URL:

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

Figure 1-8 shows the resulting HTML page that you should see in your browser: the Tomcat 5 welcome page.

How It Works

Tomcat, often referred to as the *Tomcat server*, is an open-source servlet container and JSP engine. Both the servlet API and JSP specification have undergone many revisions. At this time, multiple versions of the servlet API and JSP specifications are in use. Very large bodies of applications have been written on these different versions. While each new version brings many new features and improvements, older versions tend to be well tested and more stable — two features greatly appreciated by business application developers.

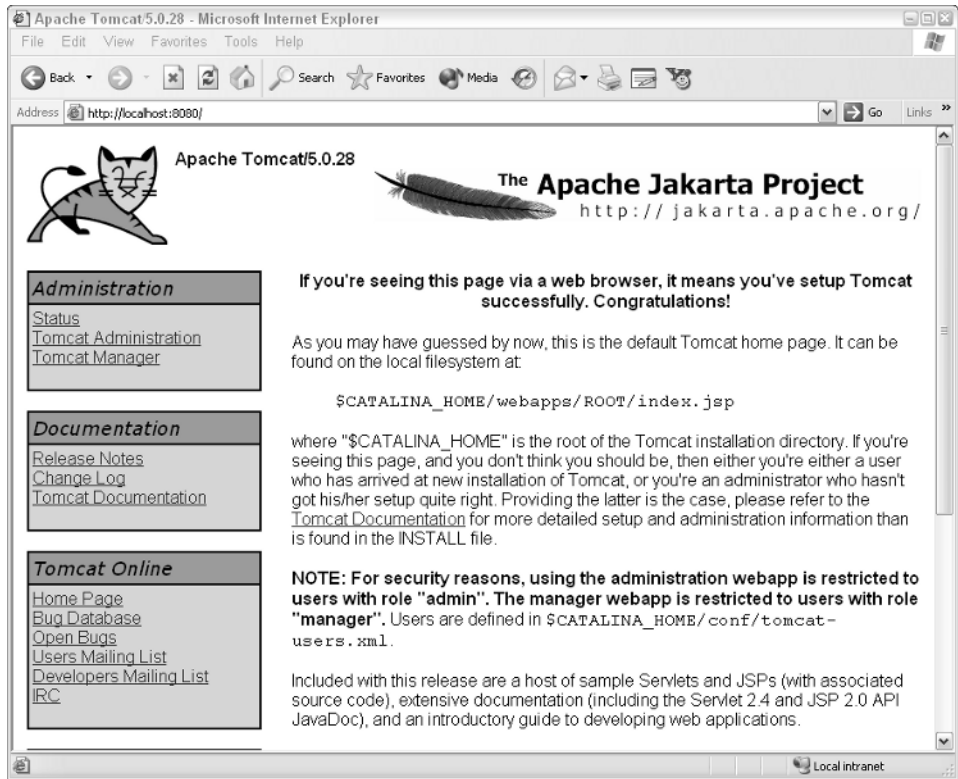


Figure 1-8: The Tomcat 5 server welcome page

The Tomcat server is the *reference implementation* for the servlet API and JSP specifications. This means that every facet of the API specification is implemented and validated against Tomcat. Because there are multiple versions of these specifications, there are also multiple versions of Tomcat servers. The following table shows the correspondence between the Tomcat server and servlet API and JSP specifications.

Tomcat Version	Servlet API Version, JSP Spec Version
3.x.x	2.2, 1.1
4.x.x	2.3, 1.2
5.x.x	2.4, 2.0

The servlet API and JSP specification designers aim to create, as much as possible, new versions that are *backward compatible*. This means that applications based on older versions should be able to run in newer version containers. However, in practice, this is not always possible. With new versions, there are often APIs that are *deprecated* (no longer supported) and new APIs and formats that must be utilized.

Chapter 1

For example, this book focuses on the *JSP 2.0* specification. The JSP engine for this specification is part of the Tomcat 5.x.x series of servers. Products of most commercial vendors have all updated to this version of the JSP specification.

When you first start up the Tomcat server, it is operating in *standalone mode*. In this mode, Tomcat is functioning in accordance with the first configuration presented earlier (in Figure 1-3a). Tomcat has a built-in Web server that is used to handle the incoming request from your browser. Along with the Web server, the JSP engine is also started. However, the servicing of this initial welcome page does not involve the JSP engine.

The JSP page that you see in your browser is stored under the `<Tomcat Installation Directory>/webapps/ROOT` directory. You may want to examine the source code of this file. Note that even though the page being accessed is a JSP page, there are no JSP tags (for dynamic elements) within the file. In other words, it is mainly a static HTML file. A static HTML file with no dynamic element can be a JSP file.

Try It Out An Initial JSP Experience

Having verified that Tomcat 5 is correctly installed, it is time to try out some JSP coding. The initial `index.jsp` that is displayed is not too exciting; it has no dynamically generated element. Using the same browser, access the following URL:

```
http://localhost:8080/jsp-examples/jsp2/tagfiles/hello.jsp
```

This time, the page that you see should be similar to the one shown in Figure 1-9, which shows the `hello.jsp` example from the Tomcat 5 distribution.

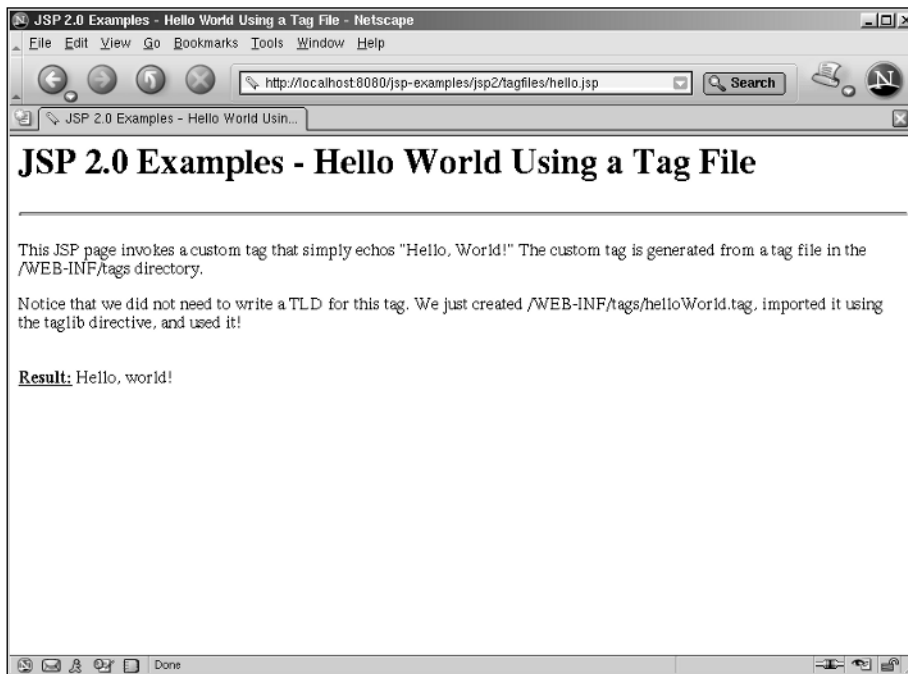


Figure 1-9: The `hello.jsp` example from the Tomcat 5 distribution

You can see the source code to this JSP by accessing the following URL:

```
http://localhost:8080/jsp-examples/jsp2/tagfiles/hello.jsp.html
```

The content is represented by the following code:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
  <head>
    <title>JSP 2.0 Examples - Hello World Using a Tag File</title>
  </head>
  <body>
    <h1>JSP 2.0 Examples - Hello World Using a Tag File</h1>
    <hr>
    <p>This JSP page invokes a custom tag that simply echos "Hello, World!"
    The custom tag is generated from a tag file in the /WEB-INF/tags
    directory.</p>
    <p>Notice that we did not need to write a TLD for this tag. We just
    created /WEB-INF/tags/helloWorld.tag, imported it using the taglib
    directive, and used it!</p>
    <br>
    <b><u>Result:</u></b>
    <tags:helloWorld/>
  </body>
</html>
```

In Figure 1-9, the “Hello, World!” message after the “Result:” label is dynamically generated via the `<tags:helloWorld/>` tag in the preceding code.

To see how JSP is automatically and dynamically compiled, make some modifications to the `hello.jsp` file. You will find the `hello.jsp` file in the `<Tomcat Installation Directory>/webapps/jsp-examples/tagfiles` directory. Using a text editor, make the following highlighted modifications and then save the resulting file in the same directory named `hello2.jsp`:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
  <head>
    <title>Presenting JSP 2.0</title>
  </head>
  <body>
    <h1>My First JSP 2.0 Template</h1>
    <hr>
    <p>All I want to say is <tags:helloWorld/></p>
  </body>
</html>
```

Now use your browser to load the new JSP file using the following URL:

```
http://localhost:8080/jsp-examples/jsp2/tagfiles/hello2.jsp
```

The resulting HTML output is shown in Figure 1-10.

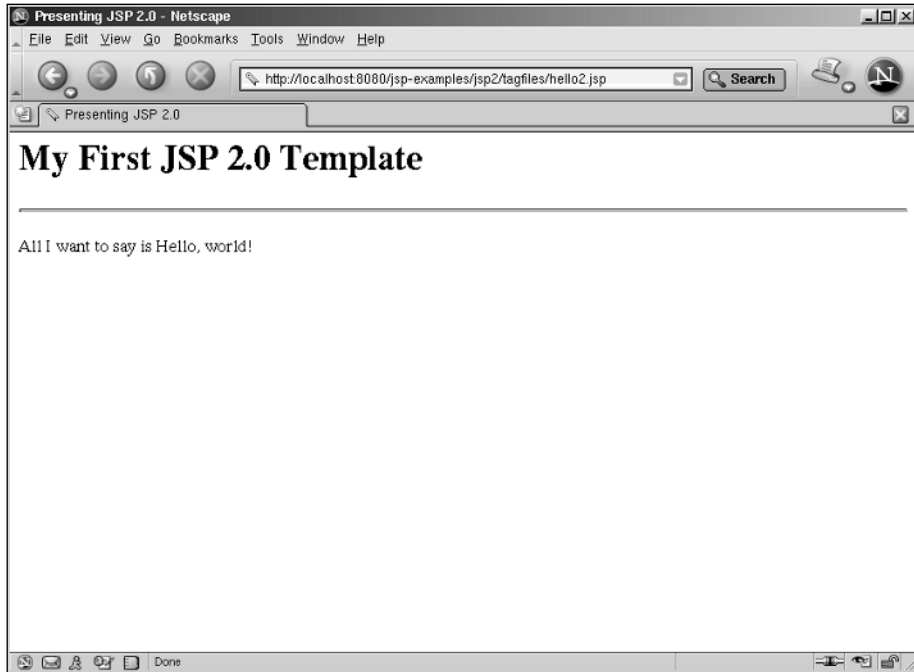


Figure 1-10: Customized hello2.jsp template

If you get an error when attempting to access `hello2.jsp`, your installation may not have set the required environment for compiling JSP pages. This can happen with certain versions of the Windows installer. To fix this, you need to stop Tomcat 5. Next, you need to copy the `tools.jar` file from the `<java SDK installation>/lib` directory to the `<tomcat 5 installation>/common/lib` directory. Start Tomcat 5 after copying this file and `hello2.jsp` should then display without any error.

Edit the `hello2.jsp` page again, this time making the following highlighted modifications, and save the file under the same name:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
  <head>
    <title>Presenting JSP 2.0</title>
  </head>
  <body>
    <h1>My First JSP 2.0 Template</h1>
    <hr>
    <p>I am so excited, I want to scream "<tags:helloWorld/> <tags:helloWorld/>
<tags:helloWorld/>"</p>
  </body>
</html>
```

In this case, the dynamically generated text is repeated. Reload the JSP page by clicking the Reload (or Refresh) button on your browser. This time, your page should look like the one shown in Figure 1-11.

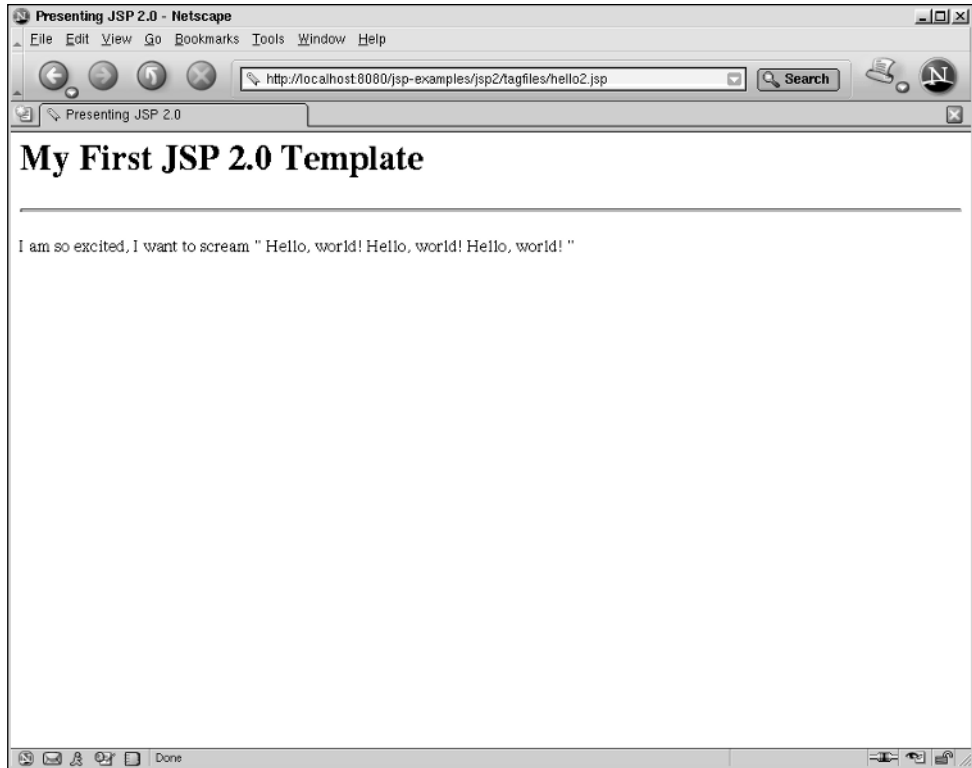


Figure 1-11: Customized hello2.jsp with repeated tag

How It Works

The Tomcat 5 distribution includes many JSP examples. Instead of creating a JSP from scratch, the preceding examples utilize one of the simple examples in the distribution.

The different JSP examples included with Tomcat are located under the *<Tomcat Installation Directory>/webapps/jsp-examples* directory. The preceding example is located under the *<Tomcat Installation Directory>/webapps/jsp-examples/jsp2/tagfiles* subdirectory.

Tags and taglib

This JSP makes use of a special custom tag that simply outputs the text "Hello World!" Custom JSP tags are collected in a *tag library*, or *taglib* for short. To notify the JSP container of the location of the custom tag, a special JSP tag (called the *taglib directive*) needs to be included:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
```

Chapter 1

This directive, explained in detail in Chapter 11, tells Tomcat 5 that the tags can be found in the `/WEB-INF/tags` directory. In this case, the path is relative to the `<Tomcat Installation Directory>/webapps/jsp-examples` directory. In addition, the preceding `taglib` directive also tells Tomcat that all of the tags in this library are prefixed by the word *tags*. This prefix is very useful in production code, where a single JSP page may use tags from many different tag libraries. By having this prefix, two libraries may have tags with the same name and Tomcat will not get them confused. The prefix that is added to the tag to distinguish which library it originates from is often called the *namespace*.

Within the JSP body, which is essentially an HTML page here, the tag is embedded as follows:

```
<tags:helloWorld/>
```

Note that this `helloWorld` tag has no body, as indicated by the empty notation `</>`, and has the namespace *tags*.

Wherever this `helloWorld` tag is placed within the JSP, the JSP container will substitute the dynamically generated text “Hello, World!” in its place. This was thoroughly tested in the Try It Out section earlier, where the tag was duplicated throughout the JSP page.

Using a tag that prints “Hello, World” all the time is rather boring. Bear in mind, however, that you can create the tag using the Java programming language to do most anything. For example, the tag could perform a lookup on the inventory database and display the remaining quantity of an item instead of printing “Hello, World”. Therefore, the craft of JSP programming revolves around the placement of JSP tags within a template. If you’re already familiar with HTML, learning JSP involves learning a set of new tags and how they can be applied on the pages that you design. For readers who want to create their own tags, Chapter 11, “Building Your Own Custom JSP Tag Library,” describes how to create your own JSP tags and tag libraries.

JSP’s rapid development cycle

In the preceding Try It Out section, the `hello2.jsp` page is checked for modification every time the URL containing the JSP is accessed. This should not be surprising. Each access from a browser is a separate request, and recall that the server-side CGI mechanism (in this case, the JSP container) handles each request independently.

Because each request is independent, the JSP container is given a chance to determine whether the JSP has been modified since it was last accessed. If the JSP container detects that the JSP has been modified, the page is recompiled before being used to handle an incoming request. This makes the JSP development cycle quite straightforward; just modify the JSP file and the changes take effect immediately.

Try It Out Deploying Chapter Examples

Now let’s take a look at how to deploy the examples from this book to our Tomcat container.

If you have not yet downloaded the chapter examples, you can download the latest version of the examples from www.wrox.com. Once at the site, simply locate the book's title (either by using the Search box or by using the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

After extracting the example code, you should have a directory structure similar to what is shown in Figure 1-12.

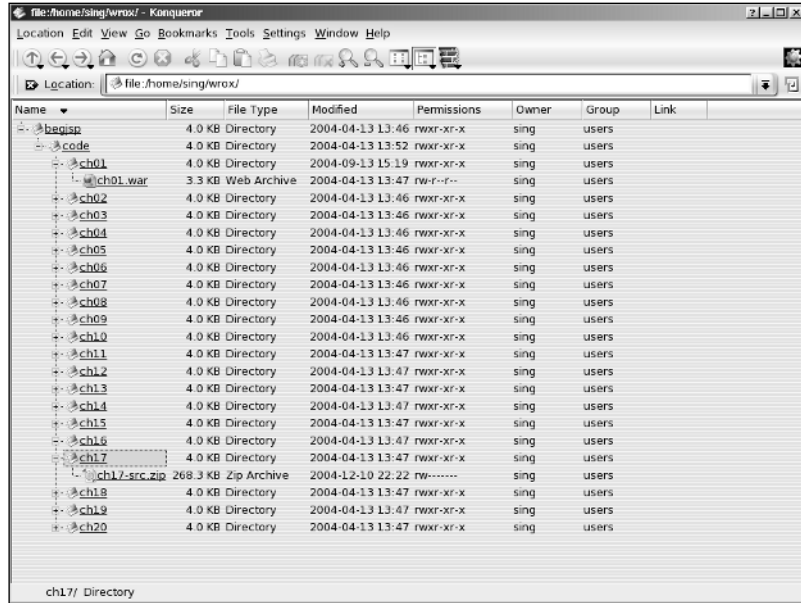


Figure 1-12: Directory structure of example code distribution

In Figure 1-12, all of the code and associated files for each chapter is stored under its own directory. For example, the code and associated files for this chapter is stored under the `ch01` subdirectory.

Note that each directory contains a similarly named file with the `.war` extension. For example, a `ch01.war` file is under the `ch01` subdirectory. A WAR file is a Web ARchive file. It contains a deployable Web application in a format that is accepted by all standard complaint JSP/servlet containers (such as Tomcat). In other words, the WAR file contains JSP along with other applications components that can be loaded and executed immediately on a JSP/servlet container.

Some of the later chapters contain examples with large body of code. These examples require an automated tool, called Apache Ant, in the creation of the WAR file. Unlike the earlier chapters, the code for these chapters is distributed as zip files. For example, the source code for Chapter 17 is in `ch17-src.zip`.

Using a Tomcat utility called *Manager*, deploying a WAR file can be quite a simple process.

Authentication for Manager access

Now try to access the Manager utility using the following URL:

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

At this point, the Manager utility should ask you for your user ID and password. This process is known as *authentication*. Figure 1-13 illustrates a typical authentication screen that you may see.

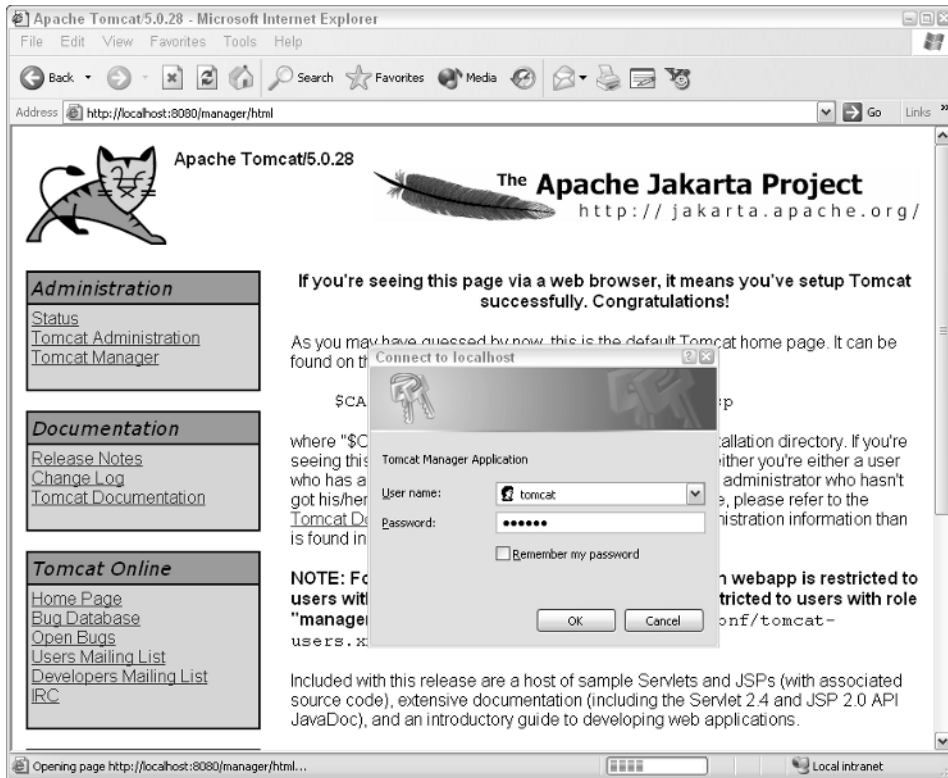


Figure 1-13: Tomcat Manager authentication screen

In the authentication screen, enter the username and password you have set up for administration during installation. Once successful, you will see the Manager utility's main screen, as shown in Figure 1-14.

In Figure 1-14 you can see the WAR files that are already deployed on the running Tomcat server. Now try to deploy the `ch01.war` file. Scroll down the page if you need to, until you see the box entitled WAR File to Deploy. Click the Browse button next to the Select WAR File to Upload prompt. Browse to the `ch01.war` file and select it. Next, click the Deploy button.

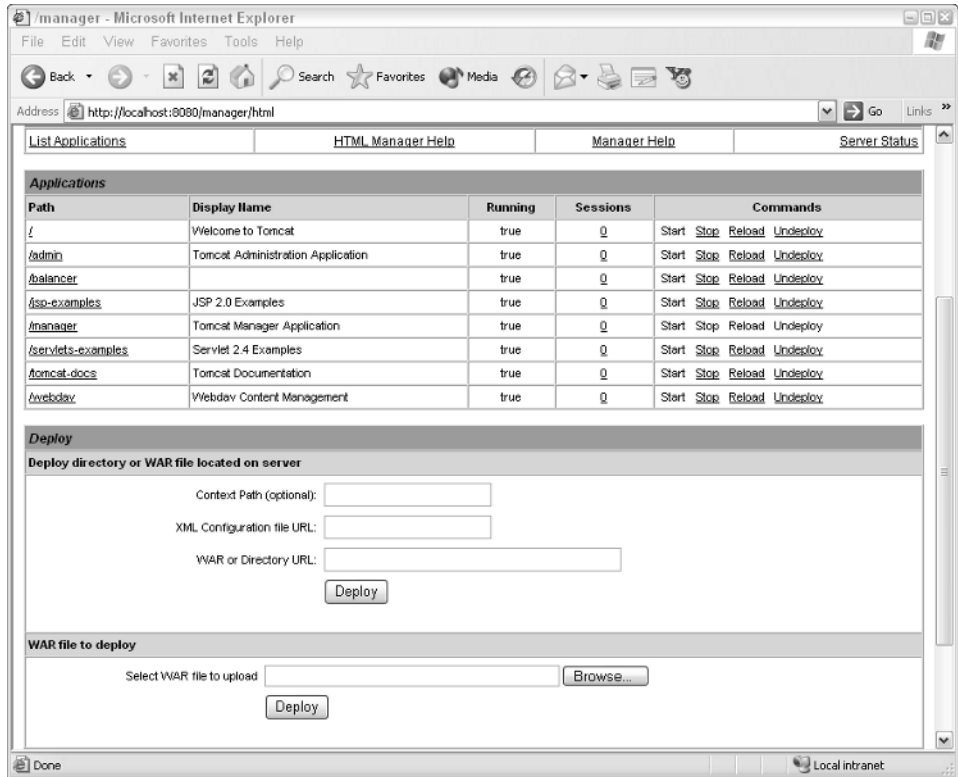


Figure 1-14: Main screen of Tomcat's Manager utility

This should start our final Chapter 1 example Web application. Looking at the list of Web applications deployed on the server, notice that the `ch01` Web application is now running on the server, as shown in Figure 1-15.

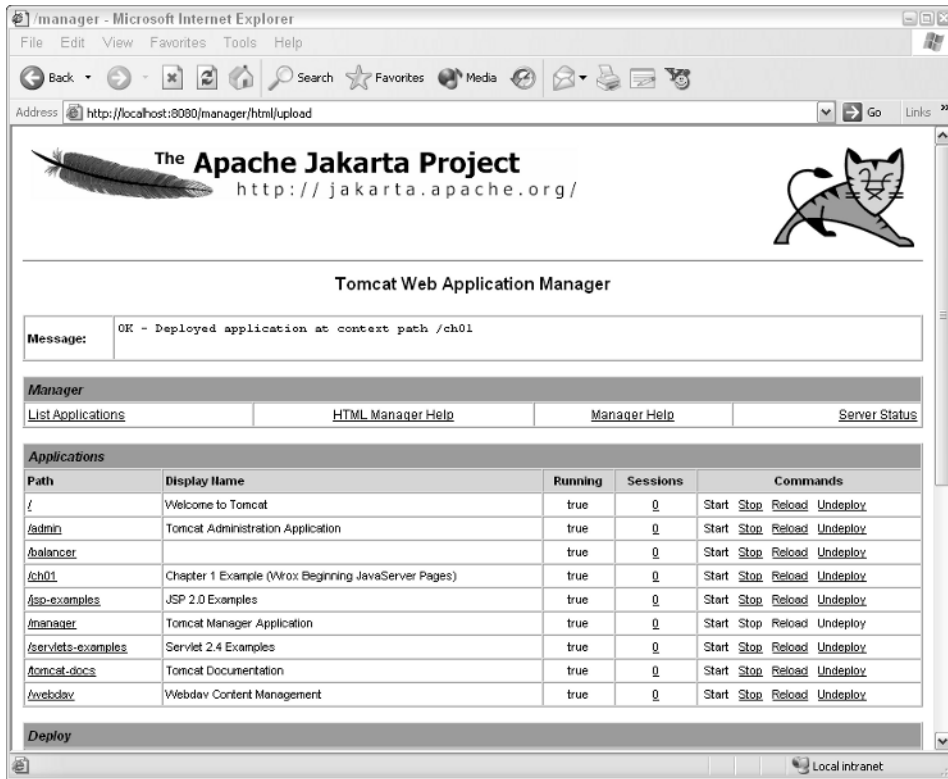


Figure 1-15: Successful deployment of the ch01.war example

You can try out the deployed ch01 example by accessing the following URL:

`http://localhost:8080/ch01/index.jsp`

The page displayed should be similar to what is shown in Figure 1-16.

This JSP page displays information about this book.

How It Works

This Manager application enables you to deploy, start, stop, or undeploy server-side *Web applications*. A Web application is a deployable unit represented by the WAR file. When the `ch01.war` file is selected using the Browse button and the Deploy button is clicked, the file is uploaded from the local directory to the Tomcat server. The WAR file is placed into the server's `webapps` directory. Tomcat is programmed to scan for new Web applications in this directory. When Tomcat detects the newly uploaded `ch01.war` file, it will deploy the `ch01` Web application. This results in the retrieval of the WAR file from the `webapps` directory, creating a new `ch01` directory holding all of the application files (JSP, servlets, tags, descriptors, and so on).

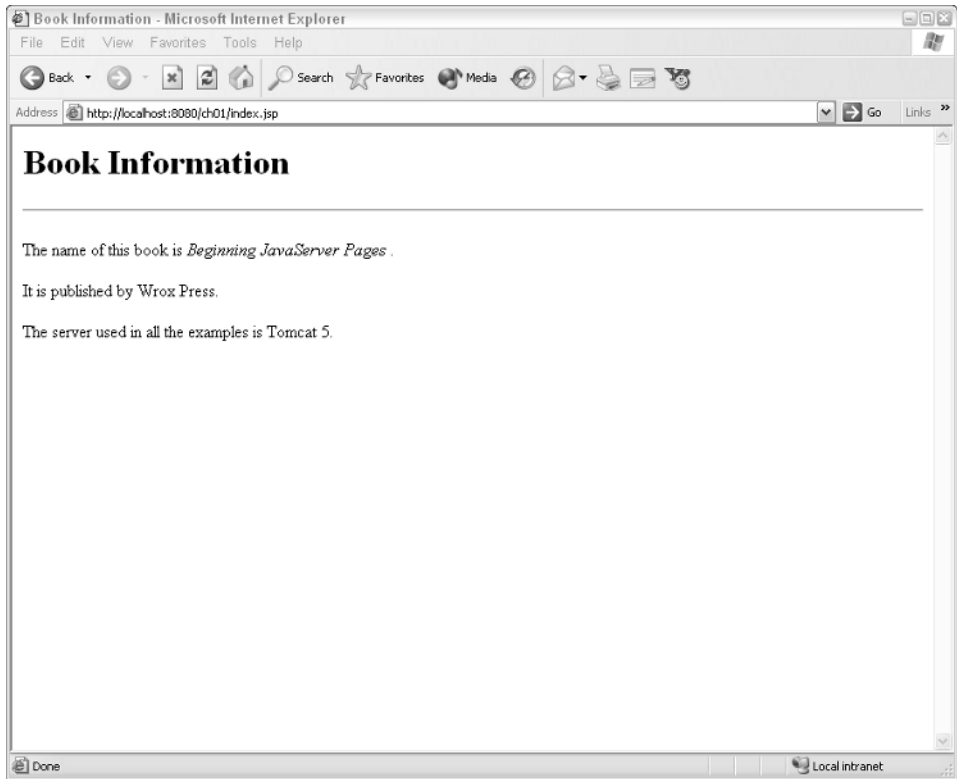


Figure 1-16: Book information presented by the ch01.war JSP example

The `index.jsp` file from `ch01.war` can be relocated to the `<Tomcat Installation Directory>/webapps/ch01` directory. Using a text editor to view this file, you should see the following:

```
<%@ taglib prefix="wroxtags" tagdir="/WEB-INF/tags" %>
<html>
  <head>
    <title>Book Information</title>
  </head>
  <body>
    <h1>Book Information</h1>
    <hr>
    <p>The name of this book is <i><wroxtags:bookTitle/></i>.</p>
    <p>It is published by <wroxtags:publisher/>.</p>
    <p>The server used in all the examples is <wroxtags:containerName/>.</p>
  </body>
</html>
```

Again, this is an HTML template with embedded JSP tags. The first tag at the top, the `taglib` directive, looks familiar. It is almost exactly the same as the one in `hello.jsp` from the second example. This directive associates the `wroxtags` namespace with the tag files in the `WEB-INF/tags` directory.

Chapter 1

If you take a look at the `<Tomcat Installation Directory>/webapps/ch01/WEB-INF/tags` directory you will see several `.tag` files there. Each of these files is associated with an available tag. The following table describes the tags that are used in `index.jsp`.

Tag Name	Description
<code>bookTitle</code>	Displays the title of the book
<code>publisher</code>	Displays the publisher for the book
<code>containerName</code>	Displays the name of the JSP container used in the book

These tags are embedded inline with the HTML within `index.jsp`. Note that the resulting JSP is independent of how these tags work. For example, if the tags were designed to look up the book information from a library database, the very same `index.jsp` file can be used to display the book information without any change. By editing the JSP file and modifying the HTML, it is easy to create the exact look required to present the information.

In other words, the presentation of the information, captured in this JSP, is completely independent of and separated from the information itself (and the method used to retrieve or synthesize this information — the tags). JSP technology is considered to be most suitable for use in the Web-based presentation of business information. Often, one will hear that “JSP is for presentation,” or that JSP is a *presentation-layer* technology.

Summary

This chapter provided an introduction to JSP technology, described its intimate relationship with servlets and CGI, and provided two actual hands-on examples that enabled you to work with the technology. You now have

- ❑ Examined the historical evolution that led to the development of JSP
- ❑ An appreciation of why JSP is necessary in addition to CGI and servlets
- ❑ An understanding of what JSP is and how it works
- ❑ An understanding of Java-based server-side Web request processing in general
- ❑ Downloaded and installed the Tomcat server for executing JSP code on your own PC or workstation
- ❑ An understanding of how to modify a JSP page to dynamically generate Web output

The next chapter explores the basic tags used in JSP programming.

Exercises

1. Modify the `index.jsp` file in the third example to display a page similar to the one shown in Figure 1-17. Make sure you use the tags to generate the book title, publisher, and container name information.



Figure 1-17: JSP output of Exercise 1

2. The `ch01.war` file contains a JSP example that displays information about this book, as well as the server used in the examples. Modify this JSP, and add JSP coding if necessary, to render a page that displays the information shown in Figure 1-18. The last line in the page should identify the browser that you are using, and will vary depending on whether you are using Internet Explorer, Netscape, or another browser. You should examine the available tags in the tag library carefully.

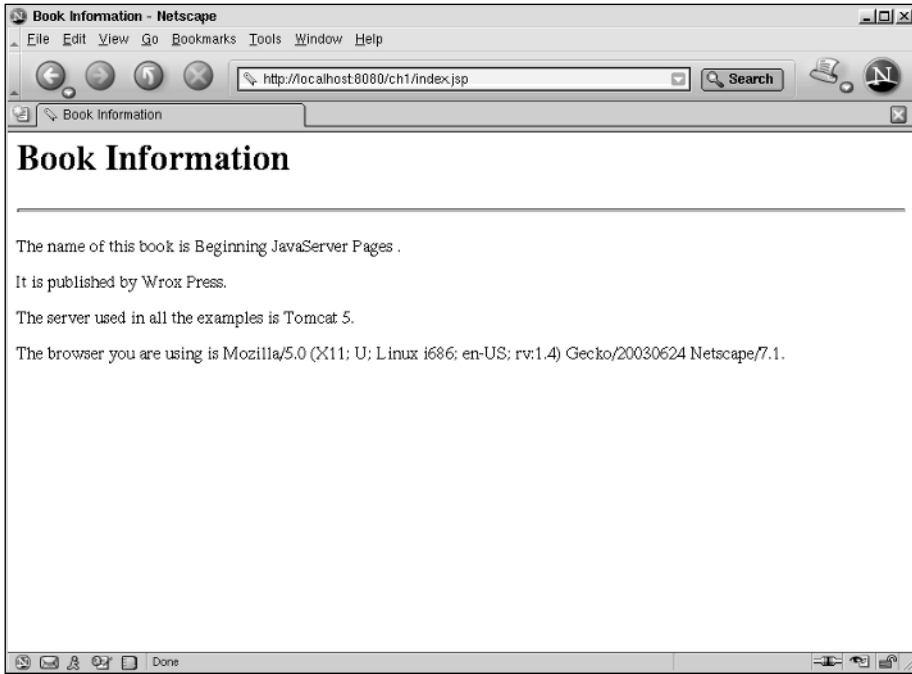


Figure 1-18: JSP output of Exercise 2