OVERVIEW OF SERVLETS AND JAVASERVER PAGES

Topics in This Chapter

- What servlets are
- When and why you would use servlets
- What JavaServer Pages are
- When and why you would use JSP
- Obtaining the servlet and JSP software
- Software installation and setup

Chapter 1

This chapter gives a quick overview of servlets and JavaServer Pages (JSP), outlining the major advantages of each. It then summarizes how to obtain and configure the software you need to write servlets and develop JSP documents.

1.1 Servlets

Servlets are Java technology's answer to Common Gateway Interface (CGI) programming. They are programs that run on a Web server, acting as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server. Their job is to:

1. **Read any data sent by the user.**
   This data is usually entered in a form on a Web page, but could also come from a Java applet or a custom HTTP client program.
2. **Look up any other information about the request that is embedded in the HTTP request.**
   This information includes details about browser capabilities, cookies, the host name of the requesting client, and so forth.

3. **Generate the results.**
   This process may require talking to a database, executing an RMI or CORBA call, invoking a legacy application, or computing the response directly.
4. **Format the results inside a document.**
   In most cases, this involves embedding the information inside an HTML page.
5. **Set the appropriate HTTP response parameters.**
   This means telling the browser what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.
6. **Send the document back to the client.**
   This document may be sent in text format (HTML), binary format (GIF images), or even in a compressed format like gzip that is layered on top of some other underlying format.

Many client requests can be satisfied by returning pre-built documents, and these requests would be handled by the server without invoking servlets. In many cases, however, a static result is not sufficient, and a page needs to be generated for each request. There are a number of reasons why Web pages need to be built on-the-fly like this:

- **The Web page is based on data submitted by the user.**
  For instance, the results page from search engines and order-confirmation pages at on-line stores are specific to particular user requests.
- **The Web page is derived from data that changes frequently.**
  For example, a weather report or news headlines page might build the page dynamically, perhaps returning a previously built page if it is still up to date.
- **The Web page uses information from corporate databases or other server-side sources.**
  For example, an e-commerce site could use a servlet to build a Web page that lists the current price and availability of each item that is for sale.

In principle, servlets are not restricted to Web or application servers that handle HTTP requests, but can be used for other types of servers as well. For example, servlets could be embedded in mail or FTP servers to extend their functionality. In practice, however, this use of servlets has not caught on, and I'll only be discussing HTTP servlets.

1.2 The Advantages of Servlets Over "Traditional" CGI

Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.

Efficient

With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays running and handles each request using a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous requests to the same CGI program, the code for the CGI program is loaded into memory *N* times. With servlets, however, there would be *N* threads but only a single copy of the servlet class. Finally, when a CGI program finishes handling a request, the program terminates. This makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data. Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between requests.

Convenient

Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities. Besides, you already know the Java programming language. Why learn Perl too? You're already convinced that Java technology makes for more reliable and reusable code than does C++. Why go back to C++ for server-side programming?

Powerful

Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI. Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API. Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance. Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

Portable

Servlets are written in the Java programming language and follow a standard API. Consequently, servlets written for, say, I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft Internet Information Server (IIS), IBM WebSphere, or StarNine WebStar. For example, virtually all of the servlets and JSP pages in this book were executed on Sun's Java Web Server, Apache Tomcat and Sun's JavaServer Web Development Kit (JSWDK) with *no* changes whatsoever in the code. Many were tested on BEA WebLogic and IBM WebSphere as well. In fact, servlets are supported directly or by a plug-in on virtually *every* major Web server. They are now part of the Java 2 Platform, Enterprise Edition (J2EE; see `http://java.sun.com/j2ee/`), so industry support for servlets is becoming even more pervasive.

Secure

One of the main sources of vulnerabilities in traditional CGI programs stems from the fact that they are often executed by general-purpose operating system shells. So the CGI programmer has to be very careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. This is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries. A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a

100-element array then write into the 999th "element," which is really some random part of program memory. So programmers who forget to do this check themselves open their system up to deliberate or accidental buffer overflow attacks. Servlets suffer from neither of these problems. Even if a servlet executes a remote system call to invoke a program on the local operating system, it does not use a shell to do so. And of course array bounds checking and other memory protection features are a central part of the Java programming language.

Inexpensive

There are a number of free or very inexpensive Web servers available that are good for "personal" use or low-volume Web sites. However, with the major exception of Apache, which is free, most commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter its cost, adding servlet support to it (if it doesn't come preconfigured to support servlets) costs very little extra. This is in contrast to many of the other CGI alternatives, which require a significant initial investment to purchase a proprietary package.

1.3 JavaServer Pages

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content from servlets. Many Web pages that are built by CGI programs are primarily static, with the parts that change limited to a few small locations. For example, the initial page at most on-line stores is the same for all visitors, except for a small welcome message giving the visitor's name if it is known. But most CGI variations, including servlets, make you generate the entire page via your program, even though most of it is always the same. JSP lets you create the two parts separately. Listing 1.1 gives an example. Most of the page consists of regular HTML, which is passed to the visitor unchanged. Parts that are generated dynamically are marked with special HTML-like tags and mixed right into the page.

Listing 1.1 A sample JSP page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Welcome to Our Store</TITLE></HEAD>
<BODY>
<H1>Welcome to Our Store</H1>
<SMALL>Welcome,
<!-- User name is "New User" for first-time visitors -->
<%= Utils.getUserNameFromCookie(request) %>
To access your account settings, click
<A HREF="Account-Settings.html">here.</A></SMALL>
<P>
Regular HTML for all the rest of the on-line store's Web page.
</BODY>
</HTML>
```

1.4 The Advantages of JSP

JSP has a number of advantages over many of its alternatives. Here are a few of them.

Versus Active Server Pages (ASP)

ASP is a competing technology from Microsoft. The advantages of JSP are twofold. First, the dynamic part is written in Java, not VBScript or another ASP-specific language, so it is more powerful and better suited to complex applications that require reusable components. Second, JSP is portable to other operating systems and Web servers; you aren't locked into Windows NT/2000 and IIS. You could make the same argument when comparing JSP to ColdFusion; with JSP you can use Java and are not tied to a particular server product.

Versus PHP

PHP is a free, open-source HTML-embedded scripting language that is somewhat similar to both ASP and JSP. The advantage of JSP is that the dynamic part is written in Java, which you probably already know, which already has an extensive API for networking, database access, distributed objects, and the like, whereas PHP requires learning an entirely new language.

Versus Pure Servlets

JSP doesn't provide any capabilities that couldn't in principle be accomplished with a servlet. In fact, JSP documents are automatically translated into servlets behind the scenes. But it is more convenient to write (and to modify!) regular HTML than to have a zillion `println` statements that generate the HTML. Plus, by separating the presentation from the content, you can put different people on different tasks: your Web page design experts can build the HTML using familiar tools and leave places for your servlet programmers to insert the dynamic content.

Versus Server-Side Includes (SSI)

SSI is a widely supported technology for inserting externally defined pieces into a static Web page. JSP is better because you have a richer set of tools for building that external piece and have more options regarding the stage of the HTTP response at which the piece actually gets inserted. Besides, SSI is really intended only for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

Versus JavaScript

JavaScript, which is completely distinct from the Java programming language, is normally used to generate HTML dynamically on the *client*, building parts of the Web page as the browser loads the document. This is a useful capability but only handles situations where the dynamic information is based on the client's environment. With the exception of cookies, the HTTP request data is not available to client-side JavaScript routines. And, since JavaScript lacks routines for network programming, JavaScript code on the client cannot access server-side resources like databases, catalogs, pricing information, and the like. JavaScript can also be used on the server, most notably on Netscape servers and as a scripting language for IIS. Java is far more powerful, flexible, reliable, and portable.

Versus Static HTML

Regular HTML, of course, cannot contain dynamic information, so static HTML pages cannot be based upon user input or server-side data sources. JSP is so easy and convenient that it is quite reasonable to augment HTML pages that only benefit slightly by the insertion of dynamic data. Previously, the difficulty of using dynamic data precluded its use in all but the most valuable instances.

1.5 Installation and Setup

Before you can get started, you have to download the software you need and configure your system to take advantage of it. Here's an outline of the steps involved. Please note, however, that although your servlet code will follow a standard API, there is no standard for downloading and configuring Web or application servers. Thus, unlike most sections of this book, the methods described here vary significantly from server to server, and the examples in this section should be taken only as representative samples. Check your server's documentation for authoritative instructions.

Obtain Servlet and JSP Software

Your first step is to download software that implements the Java Servlet 2.1 or 2.2 and JavaServer Pages 1.0 or 1.1 specifications. If you are using an up-to-date Web or application server, there is a good chance that it already has everything you need. Check your server documentation or see the latest list of servers that support servlets at `http://java.sun.com/products/servlet/industry.html`. Although you'll eventually want to deploy in a commercial-quality server, when first learning it is useful to have a free system that you can install on your desktop machine for development and testing purposes. Here are some of the most popular options:

- **Apache Tomcat.**
  Tomcat is the official reference implementation of the servlet 2.2 and JSP 1.1 specifications. It can be used as a small stand-alone server for testing servlets and JSP pages, or can be integrated into the Apache Web server. However, many other servers have announced upcoming support, so these specifications will be

covered in detail throughout this book. Tomcat, like Apache itself, is free. However, also like Apache (which is very fast, highly reliable, but a bit hard to configure and install), Tomcat requires significantly more effort to set up than do the commercial servlet engines. For details, see `http://jakarta.apache.org/`.

- **JavaServer Web Development Kit (JSWDK).**
  The JSWDK is the official reference implementation of the servlet 2.1 and JSP 1.0 specifications. It is used as a small stand-alone server for testing servlets and JSP pages before they are deployed to a full Web server that supports these technologies. It is free and reliable, but takes quite a bit of effort to install and configure. For details, see `http://java.sun.com/products/servlet/download.html`.
- **Allaire JRun.**
  JRun is a servlet and JSP engine that can be plugged into Netscape Enterprise or FastTrack servers, IIS, Microsoft Personal Web Server, older versions of Apache, O'Reilly's WebSite, or StarNine WebSTAR. A limited version that supports up to five simultaneous connections is available for free; the commercial version removes this restriction and adds capabilities like a remote administration console. For details, see `http://www.allaire.com/products/jrun/`.
- **New Atlanta's ServletExec.** ServletExec is a servlet and JSP engine that can be plugged into most popular Web servers for Solaris, Windows, MacOS, HP-UX and Linux. You can download and use it for free, but many of the advanced features and administration utilities are disabled until you purchase a license. For details, see `http://newatlanta.com/`.
- **LiteWebServer (LWS) from Gefion Software.**
  LWS is a small free Web server derived from Tomcat that supports servlets version 2.2 and JSP 1.1. Gefion also has a free plug-in called WAICoolRunner that adds servlet 2.2 and JSP 1.1 support to Netscape FastTrack and Enterprise servers. For details, see http://www.gefionsoftware.com/.
- **Sun's Java Web Server.**
  This server is written entirely in Java and was one of the first Web servers to fully support the servlet 2.1 and JSP 1.0 specifications. Although it is no longer under active development because Sun is concentrating on the Netscape/I-Planet server, it is still a popular choice for learning

servlets and JSP. For a free trial version, see `http://www.sun.com/software/jwebserver/try/`. For a free non-expiring version for teaching purposes at academic institutions, see `http://freeware.thesphere.com/`.

Bookmark or Install the Servlet and JSP API Documentation

Just as no serious programmer should develop general-purpose Java applications without access to the JDK 1.1 or 1.2 API documentation, no serious programmer should develop servlets or JSP pages without access to the API for classes in the `javax.servlet` packages. Here is a summary of where to find the API:

- **`http://java.sun.com/products/jsp/download.html`**
  This site lets you download either the 2.1/1.0 API or the 2.2/1.1 API to your local system. You may have to download the entire reference implementation and then extract the documentation.
- **`http://java.sun.com/products/servlet/2.2/javadoc/`**
  This site lets you browse the servlet 2.2 API on-line.
- **`http://www.java.sun.com/j2ee/j2sdkee/techdocs/api/`**
  This address lets you browse the complete API for the Java 2 Platform, Enterprise Edition (J2EE), which includes the servlet 2.2 and JSP 1.1 packages.

If Sun or Apache place any new additions on-line (e.g., a place to browse the 2.1/1.0 API), they will be listed under Chapter 1 in the book source archive at `http://www.coreservlets.com/`.

Identify the Classes to the Java Compiler

Once you've obtained the necessary software, you need to tell the Java compiler (`javac`) where to find the servlet and JSP class files when it compiles your servlets. Check the documentation of your particular package for definitive details, but the necessary class files are usually in the `lib` subdirectory of the server's installation directory, with the servlet classes in `servlet.jar` and the JSP classes in `jsp.jar`, `jspengine.jar`, or `jasper.jar`. There are a couple of different ways to tell `javac` about these classes, the easiest of which is to put the JAR files in your `CLASSPATH`. If you've never dealt with the `CLASSPATH` before, it is the variable that specifies where `javac` looks for classes when compiling. If the variable is unspecified, `javac` looks in the current directory and the standard system libraries. If you set `CLASSPATH` yourself, be sure to include "`.`", signifying the current directory.

Following is a brief summary of how to set the environment variable on a couple of different platforms. Assume *`dir`* is the directory in which the servlet and JSP classes are found.

Unix (C Shell)

```
setenv CLASSPATH .:dir/servlet.jar:dir/jspengine.jar
```

Add `:$CLASSPATH` to the end of the `setenv` line if your `CLASSPATH` is already set and you want to add more to it, not replace it. Note that on Unix systems you use forward slashes to separate directories within an entry and colons to separate entries, whereas you use backward slashes and semicolons on Windows. To make this setting permanent, you would typically put this statement in your `.cshrc` file.

Windows

```
set CLASSPATH=.;dir\servlet.jar;dir\jspengine.jar
```

Add `;%CLASSPATH%` to the end of the above line if your `CLASSPATH` is already set and you want to add more to it, not replace it. Note that on Windows you use backward slashes to separate directories within an entry and semicolons to separate entries, while you use forward slashes and colons on Unix. To make this setting permanent on Windows 95/98, you'd typically put this statement in your `autoexec.bat` file. On Windows NT or 2000, you would go to the Start menu, select Settings, select Control Panel, select System, select Environment, then enter the variable and value.

Package the Classes

As you'll see in the next chapter, you probably want to put your servlets into packages to avoid name conflicts with servlets other people write for the same Web or application server. In that case, you may find it convenient to add the top-level directory of your package hierarchy to the `CLASSPATH` as well. See Section 2.4 (Packaging Servlets) for details.

Configure the Server

Before you start the server, you may want to designate parameters like the port on which it listens, the directories in which it looks for HTML files, and so forth. This process is totally server-specific, and for commercial-quality Web servers should be clearly documented in the installation notes. However, with the small stand-alone servers that Apache and Sun provide as reference implementations of the servlet 2.2/JSP 1.1 specs (Apache Tomcat) or 2.1/1.0 specs (Sun JSWDK), there are a number of important but poorly documented settings that I'll describe here.

Port Number

Tomcat and the JSWDK both use a nonstandard port by default in order to avoid conflicts with existing Web servers. If you use one of these products for initial development and testing, and don't have another Web server running, you will probably find it convenient to switch to 80, the standard HTTP port number. With Tomcat 3.0, do so by editing `install_dir/server.xml`, changing 8080 to 80 in the line

```
<ContextManager port="8080" hostName="" inet="">
```

With the JSWDK 1.0.1, edit the `install_dir/webserver.xml` file and replace 8080 with 80 in the line

```
port NMTOKEN "8080"
```

The Java Web Server 2.0 also uses a non-standard port. To change it, use the remote administration interface, available by visiting `http://somehostname:9090/`, where *somehostname* is replaced by either the real name of the host running the server or by `localhost` if the server is running on the local machine.

JAVA_HOME Setting

If you use JDK 1.2 or 1.3 with Tomcat or the JSWDK, you must set the `JAVA_HOME` environment variable to refer to the JDK installation directory. This setting is unnecessary with JDK 1.1. The easiest way to specify this variable is to insert a line that sets it into the top of the `startup` (Tomcat) or `startserver` (JSWDK) script. For example, here's the top of the modified version of `startup.bat` and `startserver.bat` that I use:

```
rem Marty Hall: added JAVA_HOME setting below
set JAVA_HOME=C:\jdk1.2.2
```

DOS Memory Setting

If you start Tomcat or the JSWDK server from Windows 95 or 98, you probably have to modify the amount of memory DOS allocates for environment variables. To do this, start a fresh DOS window, click on the MS-DOS icon in the top-left corner of the window, and select `Properties`. From there, choose the `Memory` tab, go to the `Initial Environment` setting, and change the value from `Auto` to 2816. This configuration only needs to be done once.

Tomcat 3.0 CR/LF Settings

The first releases of Tomcat suffered from a serious problem: the text files were saved in Unix format (where the end of line is marked with a linefeed), not Windows format (where the end of the line is marked with a carriage return/linefeed pair). As a result, the startup and shutdown scripts failed on Windows. You can determine if your version suffers from this problem by opening `install_dir/startup.bat` in Notepad; if it appears normal you have a patched version. If the file appears to be one long jumbled line, then quit Notepad and open and immediately save the following files using Wordpad (*not* Notepad):

- *`install_dir`*`/startup.bat`
- *`install_dir`*`/tomcat.bat`
- *`install_dir`*`/shutdown.bat`
- *`install_dir`*`/tomcatEnv.bat`
- *`install_dir`*`/webpages/WEB-INF/web.xml`
- *`install_dir`*`/examples/WEB-INF/web.xml`

Start the Server

To start one of the “real” Web servers, check its documentation. In many cases, starting it involves executing a command called `httpd` either from the command line or by instructing the operating system to do so automatically when the system is first booted.

With Tomcat 3.0, you start the server by executing a script called `startup` in the main installation directory. With the JSWDK 1.0.1, you execute a similar script called `startserver`.

Compile and Install Your Servlets

Once you've properly set your `CLASSPATH`, as described earlier in this section, just use "`javac ServletName.java`" to compile a servlet. The resultant class file needs to go in a location that the server knows to check during execution. As you might expect, this location varies from server to server. Following is a quick summary of the locations used by the latest releases of Tomcat, the JSWDK, and the Java Web Server. In all three cases, assume `install_dir` is the server's main installation directory.

Tomcat

- **`install_dir/webpages/WEB-INF/classes`**
  Standard location for servlet classes.
- **`install_dir/classes`**
  Alternate location for servlet classes.
- **`install_dir/lib`**
  Location for JAR files containing classes.

Tomcat 3.1

Just before this book went to press, Apache released a beta version of Tomcat 3.1. If there is a final version of this version available when you go to download Tomcat, you should use it. Here is the new directory organization that Tomcat 3.1 uses:

- **`install_dir/webapps/ROOT/WEB-INF/classes`**
  Standard location for servlet classes.
- **`install_dir/classes`**
  Alternate location for servlet classes.
- **`install_dir/lib`**
  Location for JAR files containing classes.

The JSWDK

- **`install_dir/webpages/WEB-INF/servlets`**
  Standard location for servlet classes.
- **`install_dir/classes`**
  Alternate location for servlet classes.
- **`install_dir/lib`**
  Location for JAR files containing classes.

Java Web Server 2.0

- **`install_dir/servlets`**
  Location for frequently changing servlet classes. The server automatically detects when servlets in this directory change, and reloads them if necessary. This is in contrast to Tomcat and the JSWDK, where you have to restart the server when a servlet that is already in server memory changes. Most commercial servers have an option similar to this auto-reloading feature.
- **`install_dir/classes`**
  Location for infrequently changing servlet classes.
- **`install_dir/lib`**
  Location for JAR files containing classes.

I realize that this sounds a bit overwhelming. Don't worry, I'll walk you through the process with a couple of different servers when I introduce some real servlet code in the next chapter.