An Overview of Servlet and JSP Technology

Topics in This Chapter

- Understanding the role of servlets
- Building Web pages dynamically
- Looking at servlet code
- Evaluating servlets vs. other technologies
- Understanding the role of JSP

Chapter 1

Servlet and JSP technology has become the technology of choice for developing online stores, interactive Web applications, and other dynamic Web sites. Why? This chapter gives a high-level overview of the technology and some of the reasons for its popularity. Later chapters provide specific details on programming techniques.

1.1 A Servlet's Job

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 1–1.

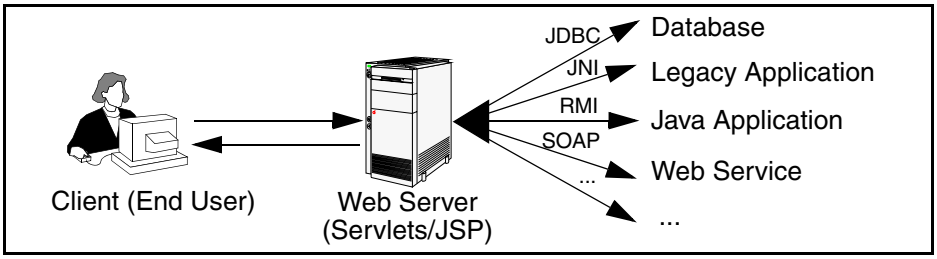

Figure 1–1 The role of Web middleware.

1. **Read the explicit data sent by the client.**
   The end user normally enters this data in an HTML form on a Web page. However, the data could also come from an applet or a custom HTTP client program. Chapter 4 discusses how servlets read this data.
2. **Read the implicit HTTP request data sent by the browser.**
   Figure 1–1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really *two* varieties of data: the explicit data that the end user enters in a form and the behind-the-scenes HTTP information. Both varieties are critical. The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth; it is discussed in Chapter 5.
3. **Generate the results.**
   This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly. Your real data may be in a relational database. Fine. But your database probably doesn't speak HTTP or return results in HTML, so the Web browser can't talk directly to the database. Even if it could, for security reasons, you probably would not want it to. The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.
4. **Send the explicit data (i.e., the document) to the client.**
   This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format. But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.
5. **Send the implicit HTTP response data.**
   Figure 1–1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client. But, there are really *two* varieties of data sent: the document itself and the behind-the-scenes HTTP information. Again, both varieties are critical to effective development. Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks. These tasks are discussed in Chapters 6 and 7.

1.2 Why Build Web Pages Dynamically?

After Marty wrote the first edition of *Core Servlets and JavaServer Pages*, various of his non-software-savvy friends and relations would ask him what his book was about. Marty would launch into a long, technical discussion of Java, object-oriented programming, and HTTP, only to see their eyes immediately glaze over. Finally, in exasperation, they would ask, "Oh, so your book is about how to make Web pages, right?"

"Well, no," the answer would be, "They are about how to make *programs* that make Web pages."

"Huh? Why wait until the client requests the page and then have a program build the result? Why not just build the Web page ahead of time?"

Yes, many client requests can be satisfied by prebuilt documents, and the server would handle these requests 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:

- **The Web page is based on data sent by the client.**
  For instance, the results page from search engines and order-confirmation pages at online stores are specific to particular user requests. You don't know what to display until you read the data that the user submits. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page based on a cookie value.
- **The Web page is derived from data that changes frequently.**
  If the page changes for every request, then you certainly need to build the response at request time. If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it. The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.
- **The Web page uses information from corporate databases or other server-side sources.**
  If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet. Imagine using an applet by itself for a search engine site:

> "Downloading 50 terabyte applet, please wait!" Obviously, that is silly; you need to talk to the database. Going from the client to the Web tier to the database (a three-tier approach) instead of from an applet directly to a database (a two-tier approach) provides increased flexibility and security with little or no performance penalty. After all, the database call is usually the rate-limiting step, so going through the Web server does not slow things down. In fact, a three-tier approach is often faster because the middle tier can perform caching and connection pooling.

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 FTP or mail servers to extend their functionality. And, a servlet API for SIP (Session Initiation Protocol) servers was recently standardized (see http://jcp.org/en/jsr/detail?id=116). In practice, however, this use of servlets has not caught on, and we'll only be discussing HTTP servlets.

1.3 A Quick Peek at Servlet Code

Now, this is hardly the time to delve into the depths of servlet syntax. Don't worry, you'll get plenty of that throughout the book. But it is worthwhile to take a quick look at a simple servlet, just to get a feel for the basic level of complexity.

Listing 1.1 shows a simple servlet that outputs a small HTML page to the client. Figure 1–2 shows the result.

The code is explained in detail in Chapter 3 (Servlet Basics), but for now, just notice four points:

- **It is regular Java code.** There are new APIs, but no new syntax.
- **It has unfamiliar import statements.** The servlet and JSP APIs are not part of the Java 2 Platform, Standard Edition (J2SE); they are a separate specification (and are also part of the Java 2 Platform, Enterprise Edition—J2EE).
- **It extends a standard class (`HttpServlet`).** Servlets provide a rich infrastructure for dealing with HTTP.
- **It overrides the `doGet` method.** Servlets have different methods to respond to different types of HTTP commands.

Listing 1.1 HelloServlet.java

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

public class HelloServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello</H1>\n" +
                "</BODY></HTML>");
  }
}
```

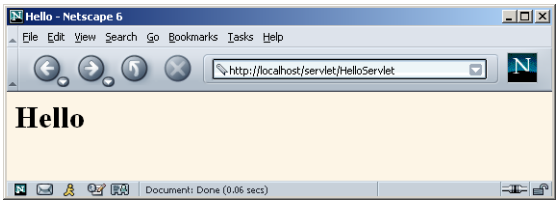

Figure 1–2 Result of `HelloServlet`.

1.4 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 with a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* 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 would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects. Finally, when a CGI program finishes handling a request, the program terminates. This approach 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 client 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. In CGI, you have to do much of this yourself. Besides, if 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 Visual Basic, VBScript, or C++. Why go back to those languages 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. Servlets are supported directly or by a plugin on virtually *every* major Web server. Consequently, servlets written for, say, Macromedia JRun can run virtually unchanged on Apache Tomcat, Microsoft Internet Information Server (with a separate plugin), IBM WebSphere, iPlanet Enterprise Server, Oracle9i AS, or StarNine WebStar. They are 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.

Inexpensive

A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites. Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success. This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package.

Price and portability are somewhat connected. For example, Marty tries to keep track of the countries of readers that send him questions by email. India was near the top of the list, probably #2 behind the U.S. Marty also taught one of his JSP and servlet training courses (see http://courses.coreservlets.com/) in Manila, and there was great interest in servlet and JSP technology there.

Now, why are India and the Philippines both so interested? We surmise that the answer is twofold. First, both countries have large pools of well-educated software developers. Second, both countries have (or had, at that time) highly unfavorable currency exchange rates against the U.S. dollar. So, buying a special-purpose Web server from a U.S. company consumed a large part of early project funds.

But, with servlets and JSP, they could start with a free server: Apache Tomcat (either standalone, embedded in the regular Apache Web server, or embedded in Microsoft IIS). Once the project starts to become successful, they could move to a server like Caucho Resin that had higher performance and easier administration but that is not free. But none of their servlets or JSP pages have to be rewritten. If their project becomes even larger, they might want to move to a distributed (clustered) environment. No problem: they could move to Macromedia JRun Professional, which supports distributed applications (Web farms). Again, none of their servlets or JSP pages have to be rewritten. If the project becomes quite large and complex, they might want to use Enterprise JavaBeans (EJB) to encapsulate their business logic. So, they might switch to BEA WebLogic or Oracle9i AS. Again, none of their servlets or JSP pages have to be rewritten. Finally, if their project becomes even bigger, they might move it off of their Linux box and onto an IBM mainframe running IBM WebSphere. But once again, none of their servlets or JSP pages have to be rewritten.

Secure

One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells. So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution 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 and then write into the 999th "element," which is really some random part of program memory. So, programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks.

Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with `Runtime.exec` or JNI) 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.

Mainstream

There are a lot of good technologies out there. But if vendors don't support them and developers don't know how to use them, what good are they? Servlet and JSP technology is supported by servers from Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, the World Wide Web Consortium (W3C), and many others. Several low-cost plugins add support to Microsoft IIS and Zeus as well. They run on Windows, Unix/Linux, MacOS, VMS, and IBM mainframe operating systems. They are the single most popular application of the Java programming language. They are arguably the most popular choice for developing medium to large Web applications. They are used by the airline industry (most United Airlines and Delta Airlines Web sites), e-commerce (ofoto.com), online banking (First USA Bank, Banco Popular de Puerto Rico), Web search engines/portals (excite.com), large financial sites (American Century Investments), and hundreds of other sites that you visit every day.

Of course, popularity alone is no proof of good technology. Numerous counter-examples abound. But our point is that you are not experimenting with a new and unproven technology when you work with server-side Java.

1.5 The Role of JSP

A somewhat oversimplified view of servlets is that they are Java programs with HTML embedded inside of them. A somewhat oversimplified view of JSP documents is that they are HTML pages with Java code embedded inside of them.

For example, compare the sample servlet shown earlier (Listing 1.1) with the JSP page shown below (Listing 1.2). They look totally different; the first looks mostly like a regular Java class, whereas the second looks mostly like a normal HTML page. The interesting thing is that, despite the huge apparent difference, behind the scenes they are the same. In fact, a JSP document is just another way of writing a servlet. JSP pages get translated into servlets, the servlets get compiled, and it is the servlets that run at request time.

Listing 1.2 Store.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Welcome to Our Store</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>Welcome to Our Store</H1>
<SMALL>Welcome,
<!-- User name is "New User" for first-time visitors -->
<%= coreservlets.Utils.getUserNameFromCookie(request) %>
To access your account settings, click
<A HREF="Account-Settings.html">here.</A></SMALL>
<P>
Regular HTML for  rest of online store's Web page
</BODY></HTML>
```

So, the question is, If JSP technology and servlet technology are essentially equivalent in power, does it matter which you use? The answer is, Yes, yes, yes! The issue is not power, but convenience, ease of use, and maintainability. For example, anything you can do in the Java programming language you could do in assembly language. Does this mean that it does not matter which you use? Hardly.

JSP is discussed in great detail starting in Chapter 10. But, it is worthwhile mentioning now how servlets and JSP fit together. JSP is focused on simplifying the creation and maintenance of the HTML. Servlets are best at invoking the business logic and performing complicated operations. A quick rule of thumb is that servlets are

best for tasks oriented toward *processing*, whereas JSP is best for tasks oriented toward *presentation*. For some requests, servlets are the right choice. For other requests, JSP is a better option. For still others, neither servlets alone nor JSP alone is best, and a combination of the two (see Chapter 15, "Integrating Servlets and JSP: The Model View Controller (MVC) Architecture") is best. But the point is that you need *both* servlets and JSP in your overall project: almost no project will consist entirely of servlets or entirely of JSP. You want both.

OK, enough talk. Move on to the next chapter and get started!