Web Server Java -- Servlets and JSP

by Ian F. Darwin

This excerpt is Chapter 18 from Java Cookbook: Solutions and Examples for Java Developers, published in June 2001 by O'Reilly.

This chapter covers Web Server Java, but you won't find anything about writing CGI programs in Java here. Although it would be entirely possible to do so, it would not be efficient. The whole notion of CGI programs is pretty much passe. Every time a CGI program is invoked, the web server has to create a new heavyweight process in which to run it; this is inefficient. If it's interpreted in Java, the program has to be translated into machine code each time; this is even more inefficient.

Today's trend is toward building functionality into the web server: Microsoft ASP, PHP3, Java servlets, and JavaServer Pages? (JSP) are examples of this. None of these normally requires a separate process to be created for each request; the Java-based solutions run in a thread (see Chapter 24) inside the web server, and the Java bytecode need only be translated into machine code once in a long while, assuming a just-in-time (JIT) runtime system. Naturally, this book concentrates on the Java solutions.

We'll use two examples in this chapter. Consider the task of displaying a web page with five randomly chosen integer numbers (lottery players love this sort of thing). The Java code you need is simple:

```
// Part of file netweb/servlets_jsp/FiveInts.java
Random r = new Random(  );
for (int i=0; i<5; i++)
    System.out.println(r.nextInt(  ));
```

But of course you can't just run that and save its output into an HTML file because you want each person seeing the page to get a different set of numbers. If you wanted to mix that into a web page, you'd have to write code to `println()` a bit of HTML. This would be a Java servlet.

The servlet code could get messy, however, since you'd have to escape double quotes inside strings. Worse, if the webmaster wanted to change the

It has been said of Sun that when they copy something,
HTML, he'd have to approach the programmer's sanctified source code and plead to have it changed. Imagine if you could give the webmaster a page containing a bit of HTML and the Java code you need, and have it magically compiled into Java whenever the HTML was changed. Imagine no longer, says the marketer, for that capability is here now, with JavaServer Pages.

The second example is a dictionary (list of terms); I'll present this both as a servlet and as a JSP.

I won't talk about how you get your servlet engine installed, nor exactly how you install your servlet. If you don't already have a servlet engine, though, I'd recommend downloading Tomcat from http://jakarta.apache.org/. Tomcat is the official reference implementation--so designated by Sun--for the servlet and JSP standard. It is also (as you can infer from the URL) the official servlet engine for the ever-popular Apache web server.

Chapter sections

First Servlet: Generating an HTML Page
Servlets: Processing Form Parameters
Cookies
Session Tracking
Generating PDF from a Servlet
HTML Meets Java: JSP
JSP Include/Forward
JavaServer Pages Using a Servlet
Simplifying Your JSP with a JavaBean
JSP Syntax Summary
Program: CookieCutter
Program: JabaDot Web News Portal

First Servlet: Generating an HTML Page

Problem:
You want a servlet to present some information to the user.

Solution:
Override the HttpServlet method service(), or doGet() doPost().

Discussion
The abstract class javax.servlet.Servlet is designed for those who wish to structure an entire web server around the servlet notion. For example, in Sun's Java Web Server, there is a servlet
subclass for handling plain HTML pages, another for processing CGI programs, and so on. Unless you are writing your own web server, you will probably not extend from this class, but rather its subclass HttpServlet, in the package javax.servlet.http. This class has a method:

```java
public void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException;
```

The service method is passed two arguments, request and response. The request contains all the information about the request from the browser, including its input stream should you need to read data. The response argument contains information to get the response back to the browser, including the output stream to write your response back to the user.

But the web has several HTTP methods for passing data into a web page. Unimportant for plain HTML pages, this distinction becomes of interest when processing forms, i.e., web pages with fill-in-the-blank or choice items. Briefly, the GET method of HTTP is used to pass all the form data appended to the URL. GET URLs look like this, for example:

```
http://www.acmewidgets.com/cgi-bin/ordercgi?productId=123456
```

They have the advantage that the user can bookmark them, avoiding having to fill in the form multiple times. But there is a limit of about 1KB on the overall length of the URL. Since this must be a single string, there is an encoding that allows spaces, tabs, colons, and other characters to be presented as two hexadecimal digits; `%20` is the character hexadecimal 20, or the ASCII space character. The POST method, by contrast, passes any parameters as input on the socket connection, after the HTTP headers.

The default implementation of the `service()` method in the `HttpServlet` class figures out which method was used to invoke the servlet. It dispatches to the correct method: `doGet()` if a GET request, `doPost()` if a POST request, etc., passing along the `request` and `response` arguments. So while you can, in theory, override the `service()` method, it's more common (and officially recommended) to override either `doGet()`, `doPost()`, or both.

The simplest `HttpServlet` is something like Example 18-1.

---

**Example 18-1: HelloServlet.java**

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

/** Simple Hello World Servlet */
public class HelloServlet extends HttpServlet{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
        PrintWriter out = response.getWriter(  );
        response.setContentType("text/html");
        out.println("<H1>Hello from a Servlet</h2>");
        out.println("<P>This servlet ran at ");
        out.println(new Date().toString(  ));
        out.println("<P>Courtesy of HelloServlet.java 1.2 ");
    }
}
```

http://www.onjava.com/lpt/a/1000

29.9.2005
The program will give output resembling Figure 18-1.

![Hello from a servlet](http://www.onjava.com/lpt/a/1000)

Figure 18-1. Hello from a servlet

You can do much more with servlets. Suppose you wanted to print a dictionary -- a list of terms and their meanings--from within a servlet. The code would be pretty much as it was in Figure 18-1, except that you’d need a `doGet()` method instead of a `doPost()` method. Example 18-2 is the code for `TermsServlet`.

Example 18-2: `TermsServlet.java`

```java
/** A Servlet to list the dictionary terms. */
public class TermsServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        PrintWriter out = resp.getWriter();
        out.println("<HTML>");
        out.println("<TITLE>Ian Darwin's Computer Terms and Acronyms</TITLE>");
        out.println("<BODY>");
        out.println("<H1>Ian Darwin's Computer Terms and Acronyms</H1>");
        out.println("<TABLE BORDER=2>");
        out.println("<TR><TH>Term</TH><TH>Meaning</TH>");
        TermsAccessor tax = new TermsAccessor("terms.txt");
        Iterator e = tax.iterator();
        while (e.hasNext()) {
            Term t = (Term)e.next();
            out.println("<TR><TD>");
            out.println(t.term);
            out.println("</TD><TD>");
            out.println(t.definition);
            out.println("</TD></TR>");
        }
        out.println("</TABLE>");
    }
}
```

Debugging Tip for Servlets

Several servlet engines (e.g., Allaire JRun) generate a lot of very small log files spread over many different directories. It is worth investing the time to learn where your particular servlet engine records stack traces, standard error and output, and other messages.

See also recipe 16.5, which shows how a servlet or other server component can communicate with a network-based logging tool.
Servlets: Processing Form Parameters

Problem
You want to process the data from an HTML form in a servlet.

Solution
Use the request object's getParameter( ) method.

Discussion
Each uniquely named INPUT element in the FORM on the HTML page makes an entry in the request object's list of parameters. These can be obtained as an enumeration, but more commonly you request just one. Figure 18-2 shows a simple form that asks you how many random numbers you want generated, and makes up that many for you.

When I type the number 8 into the field and press the "Get Yours" button, I see the screen shot in Figure 18-3.
How does it work? The program obviously consists of both an HTML page and a Java servlet. The HTML page appears in Example 18-3; notice the FORM entry and the INPUT field.

Example 18-3: IntsServlet.htm

```html
<html>
  <head><title>Random Numbers Page</title></head>
  <body bgcolor="white">
    <h1>Random Numbers Page</h1>
    <p>This site will let you pick some random numbers for Lottery, lucky number or other purposes, all electronically.</p>
    <form method="post" action="/servlets/IntsServlet">
      <h4>How Many Numbers Do You Want Today?</h4>
      <input name="howmany" size=2> (default is 5)<br>
      <input type="submit" value="Get YOURS!">
    </form>
  </body>
</html>
```

Example 18-4 shows the Java for the servlet. Watch for the use of `getParameter( )`.

Example 18-4: IntsServlet.java

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

public class IntsServlet extends HttpServlet {
  protected final int DEFAULT_NUMBER = 5;

  /** Called when the form is filled in by the user. */
```
public void doPost(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    // The usual HTML setup stuff.
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<BODY BGCOLOR="white">");
    // HTML for this page
    out.println("<TITLE>Your Personal Random Numbers</TITLE>" );
    out.println("<H1>Your Personal Random Numbers</h2>");
    out.println("<P>Here are your personal random numbers," );
    out.println("<A HREF="http://java.sun.com">Java</A> program." );
    out.println("<OL>" );
    // Figure out how many numbers to print.
    int n = DEFAULT_NUMBER;
    String num=req.getParameter("howmany");
    if (num != null && num.length( ) != 0) {
        try {
            n = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            out.println("<P>I didn't think much of ");
            out.println(num);
            out.println(" as a number.</P>" );
        }
    }
    // Now actually generate some random numbers.
    Random r = new Random( );
    for (int i=0; i<n; i++) {
        out.print("<LI>");
        out.println(r.nextInt(49));  // for Lotto 6/49
    }
    out.println("</OL>" );
    // Print a break and a back link.
    out.println("<HR></HR>" );
    out.println("<A HREF="index.html">Back to main Page</A>" );
    out.println("</HTML>" );
}

See Also: The online source includes OrderServlet, a slightly longer example.

Cookies

Problem:
You want the client (the browser) to remember some bit of information for you.

Solution:
Bake a cookie, and serve it to the client along with your response.

Discussion
Cookies were invented by Netscape as a debugging technique, but have since become ubiquitous: all modern browsers, including MSIE, and text browsers such as Lynx accept and store them. A cookie is, at heart, a small piece of text--a name and value pair--that the server side generates and sends to the client. The browser remembers them (nontransient cookies are stored to your hard disk; Netscape creates a file called cookies or cookies.txt, for example). The browser then sends them back to the server on any subsequent visit to a page from the same site. The Cookie class is part of the javax.servlet.http package, so any servlet implementation will include it. The constructor is passed a name and value, but there are other parameters you can set. Most important is the expiry time, which is in seconds from the time you first send it. The default is -1; if the value is negative, the cookie is not saved to disk; it becomes a "transient cookie" that exists only until the browser exits and is then forgotten. For cookies that are stored to disk, the expiry time is converted to a base of January 1, 1970, the beginning of Unix time and of the modern computing era.

When the browser visits a site that has sent it a cookie or cookies, it returns all of them as part of the HTTP headers. You retrieve them all (as an array) using the get_cookies() method, and iterate through them looking for the one you want.

```java
for (int i=0; i<mySiteCookies.length; i++) {
    Cookie c = mySiteCookies[i];
    if (c.getName().equals(name-you're-looking-for)) {
        someString = c.getValue();
        break;
    }
}
```

Suppose you want the user to pick a favorite color for the servlet to use as the background color for pages from that point on. Let's use a name of prefs.bgcolor for the color-coding cookie. The main servlet is CookieServlet, which checks for the cookie. If it was not set previously, it jumps off to an HTML page, which will eventually return here via another servlet. On the other hand, if the color cookie was previously set, CookieServlet (shown in Example 18-5) displays the welcome page with the user's color set.

**Example 18-5: CookieServlet.java**

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

/** Simple Cookie-based Page Color Display servlet demo. */
public class CookieServlet extends HttpServlet {
    /** The preferences cookie name */
    protected final static String PREFS_BGCOLOR = "prefs.bgcolor";
    /** Where to go if we have not yet been customized. */
    protected final static String CUSTOMIZER = "/ColorCustomize.html";
    /** The user's chosen color, if any */
    protected String faveColor = null;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        // Go through all the cookies we have, looking for a faveColor.
        Cookie[] mySiteCookies = request.getCookies();
        for (int i=0; i<mySiteCookies.length; i++) {
            Cookie c = mySiteCookies[i];
            if (c.getName().equals(PREFS_BGCOLOR)) {
```
faveColor = c.getValue();
break;
}

// if we did not find a faveColor in a cookie,
// punt to customization servlet to bake one up for us.
if (faveColor == null) {
    ServletContext sc = getServletContext();

    // Requires Servlet API 2.1 or later!
    RequestDispatcher rd =
        sc.getRequestDispatcher(CUSTOMIZER);
    rd.forward(request, response);

    // Do it the old way
    response.sendRedirect(CUSTOMIZER);
}

// OK, we have a color, so we can do the page.
PrintWriter out = response.getWriter();
response.setContentType("text/html");

out.println("<html><title>A Custom-Colored Page</title>");
out.print("<body bgcolor="");
out.print(faveColor);
out.println(">");
out.println("<P>Welcome! We hope you like your colored page!</P>");
out.println("</body></html>");
out.flush();
}

If the user has not yet set a color customization cookie, the CookieServlet passes control (by sending an HTTP redirect in the old API, or by use of a ServletDispatcher under the Servlet API 1.2 or later) to this HTML page.

<BODY BGCOLOR="pink">
<H1>Please choose a color</H1>
<FORM ACTION="/servlet/ColorCustServlet" METHOD=GET>
<SELECT NAME="color_name">
    <OPTION VALUE="green">Green</>
    <OPTION VALUE="white" SELECTED>White</>
    <OPTION VALUE="gray">Grey</>
</SELECT>
<INPUT TYPE="submit" VALUE="OK">
</FORM>

Finally, the HTML page will jump to the customization servlet (Example 18-6), which contains the code shown here to save the user's preference as a cookie, and then return to the CookieServlet by sending an HTTP "redirect," causing the browser to load the specified replacement page.

Example 18-6: ColorCustServlet.java

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
/** Color customization servlet */
public class ColorCustServlet extends HttpServlet {

    protected final static String DEFAULT_COLOR = "white";
    protected String faveColor = DEFAULT_COLOR;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            String cand=request.getParameter("color_name");
            if (cand != null) {
                faveColor = cand;
                Cookie c = new Cookie(CookieServlet.PREFS_BGCOLOR, faveColor);
                c.setMaxAge(60*60*24*365);
                response.addCookie(c);
            }
            response.sendRedirect("/servlet/CookieServlet");
        }
}

Of course, there are issues to consider when using cookies. Some users disable cookies out of justifiable fear that web sites will use them for gathering more information than a person might want to have known. In this case, our servlet would keep coming back to the customization page. It should probably have a warning to the effect that "cookies must be enabled to view this site." Or you could use other techniques, such as session tracking (see Session Tracking).

And realistically, you probably want to keep more than one preference item for a user. If you let them set the screen background, you also need to set the text color, for example. It's probably better to keep the preferences in a database on the server side, and just set a token that identifies the user (possibly the database primary key). Even then, remember that cookies can be altered! See Program: CookieCutter for a program to allow modification of the cookies stored on your hard drive.

Session Tracking

Problem:
You want to keep track of one user across several servlet invocations within the same browser session.

Solution:
Use an HttpSession object.

Discussion

HTTP was designed to be a stateless protocol: you would connect to a server, download a laboratory report, and that would be the end of it. Then people started getting clever, and began using it for interactive applications. For such purposes as a shopping cart in an online mall, and tracking answers during an online quiz or moves in an online game, the notion of an HTTP session has evolved to keep track of a particular browser. Sessions can be identified either by use of a cookie (see Cookies) or by a Session Identifier that is added to the URL. In either case the session ends when the user's browser program exits, but will otherwise stick around for a long time (there is probably a major denial-of-service attack hidden in here, so beware).

Using a session is fairly simple within the Servlet API. You request the HttpSession object from
the `HttpRequest` that is passed into your `service( )` or `doGet( )/doPost( )` method. The session object behaves rather like a `Hashtable` except that the method names are `putValue( )` and `getValue( )`. This allows you to store an arbitrary number of objects in the session and retrieve them later.

This program uses an `HttpSession` to keep track of a user's responses during a quiz about Java. There are some 20 categories; once you pick a category, you can answer all the multiple-choice questions in that topic. The first question looks like Figure 18-4.

![Figure 18-4. Quiz servlet starting.](http://www.onjava.com/lpt/a/1000)

After you've answered a few questions, it may look like Figure 18-5.

![Figure 18-5. Quiz servlet several questions later.](http://www.onjava.com/lpt/a/1000)

At the end of the quiz, you'll see the total number of questions that you answered correctly.
The Exam object (an object containing all the questions and answers, along with the number of correct answers) is loaded using an XamDataAccessor (the code for these two classes is not shown) and stored in a Progress object. Progress, an inner class inside the servlet, is a tiny data structure used to monitor your progress through one quiz. When you change topics, the Progress object is discarded and a new one created. The bulk of the code in Example 18-7 is taken up in checking and tracking your answers and in generating the HTML to show the results of your previous question (if any), as well as the question and possible answers for the current question.

Example 18-7: DoTestServlet.java

```java
/** A Java Servlet to administer the tests over the Web. *
* Saves exam and status session object to avoid having to reload it, *
* but also to keep the exam constant during a session! */
*
public class DoTestServlet extends HttpServlet {

/** Where to find the exams du jour */
protected final static String DIRECTORY =
"/home/ian/webs/daroadweb/quizzes-";
/** The body color */
protected final static String BGCOLOR = "white";

/** An Inner Class to track the student's progress */
class Progress {
    Exam exam;      // exam being taken
    boolean done;   // exam is finished.
    String category;  // name of exam, in effect
    int curQuest;    // Question number working on, 0-origin
    int correct;    // number gotten right on first try
}

/** Service is used to service each request. */
public void service(HttpServletRequest request,
        HttpServletResponse response) throws IOException, ServletException {

    PrintWriter out = response.getWriter( );
    HttpSession session;
    Progress progress;
    String reqCategory;

    // Set response type to HTML. Print the HTML header.
    response.setContentType("text/html");
    out.println("<HTML>");

    // Find the requested category
    reqCategory = request.getParameter("category");
    reqSubject = request.getParameter("subject");  // unix or java

    // Request the user's session, creating it if new.
    session = request.getSession(true);
    if (session.isNew( )) {
        // out.println("<B>NEW SESSION</B>");
        progress = new Progress( );
```
progress.category = reqCategory;
session.putValue("progress", progress);
} else {
    progress = (Progress) session.getValue("progress");
}

if (reqCategory != null && progress.category != null && !reqCategory.equals(progress.category)) {
    // CHANGE OF CATEGORIES
    // out.println("<B>NEW PROGRESS CUZ " +
    // reqCategory + " != " +progress.category + "</B>");
    progress = new Progress();
    progress.category = reqCategory;
    session.putValue("progress", progress);
}

if (progress.exam == null) {
    XamDataAccessor ls = new XamDataAccessor();
    try {
        progress.exam = ls.load(DIRECTORY + subject + "/" +
                               progress.category + ".xam");
    } catch (IOException ex) {
        eHandler(out, ex, "We had some problems loading that exam!");
    } catch (NullPointerException ex) {
        eHandler(out, ex, "Hmmm, that exam file seems to be corrupt!");
    }
}

// Now that we have "exam", use it to get Title.
out.print("<TITLE>Questions on ");
out.print(progress.exam.getCourseTitle( )); out.println("</TITLE>");
out.print("<BODY BGCOLOR=""); out.print(BGCOLOR); out.println("">");
out.print("<H1>");
out.print(progress.exam.getCourseTitle( ));
out.println("</h2>");

// Guard against reloading last page
if (progress.done) {
    out.println("<HR><a href="/quizzes/">Another Quiz?\</a>");
    out.flush();
    return;
}

// Are we asking a question, or marking it?
out.println("<P>");
String answer =request.getParameter("answer");
int theirAnswer = -1;
if (answer != null) {
    // MARK IT.
    Q q = progress.exam.getQuestion(progress.curQuest);
    theirAnswer = Integer.parseInt(answer);
    if (theirAnswer == q.getAns( )) {
        // WE HAVE A RIGHT ANSWER -- HURRAH!
        if (!q.tried) {
            out.println("<P><B>Right first try!</B>");
            progress.correct++;
        } else
            out.println("<P><B>Right. Knew you'd get it.</B>");
    q.tried = true;  // " Tried and true ...

    if (++progress.curQuest >= progress.exam.getNumQuestions( )) {
        out.println("<P>END OF EXAM.");
        if (progress.correct == progress.curQuest) {

out.println("<P><B>Awesome!</B> You got 100% right.");
} else {
    out.print("You got ");
    out.print(progress.correct);
    out.print(" correct out of ");
    out.print(progress.curQuest);
    out.println(".");
}
out.println("<HR><a href="/quizzes/">Another Quiz?</a> ");

// todo invalidate "progress" in case user retries
progress.done = true;

// Return, so we don't try to print the next question!
return;
}

// Progress?
out.print("<P>Question ");
out.print(progress.curQuest+1);
out.print(" of ");
out.print(progress.exam.getNumQuestions( ));
out.print(".");
if (progress.curQuest >= 2) {
    out.print(progress.correct);
    out.print(" correct out of ");
    out.print(progress.curQuest);
    out.print(" tried so far ");
    double pct = 100.0 * progress.correct / progress.curQuest;
    out.print((int) pct);
    out.println("%.");
}

// Now generate a form for the next (or same) question
out.print("<FORM ACTION=/servlet/DoTestServlet METHOD=POST> ");
out.print("<INPUT TYPE=hidden NAME=category VALUE=");
out.print(progress.category); out.println(">");
out.println("<HR>");
Q q = progress.exam.getQuestion(progress.curQuest);
out.println(q.getQText( ));
for (int j=0; j<q.getNumAnswers( ); j++) {
    out.print("<BR><INPUT TYPE=radio NAME=answer VALUE=");
    out.print(j);
    out.print(" ");
    if (j==theirAnswer)
        out.print(" CHECKED");
    out.print(">");
    out.print(q.getAnsText(j));
    out.println("</INPUT>");
}
out.println("<HR>");
out.println("<INPUT TYPE=SUBMIT VALUE="Mark it!">");
Generating PDF from a Servlet

Problem:
You want to make a printer-friendly document using a format like Adobe PDF.

Solution:
Use `response.setContentType("application/pdf")` and a third-party Java API that can generate PDF.

Discussion

Portable Document Format (PDF) is a file format created by Adobe Systems Inc. PDF gives you full control over how your document looks, much more so than HTML, XML, or even Java's printing routines (see Chapter 12). Adobe Acrobat is a set of programs for reading and writing PDF. Adobe itself does not publish a Java API for generating PDF from scratch, but it does publish the file format specification (*Adobe Portable File Format Specification*) and explicitly gives everyone permission to write software to generate and/or process PDF files. PDF is a good fit for processing by an object-oriented language like Java, as it's an object-based text format. As a result, there are several PDF APIs available for Java, both free and commercial:

- Sitraka/KL Group ([http://www.klg.com/](http://www.klg.com/)) has a PDF API as well as charting and other widgets, and JProbe, a leading tuning tool.

- StyleWriterEE (see [http://www.inetsoftcorp.com/](http://www.inetsoftcorp.com/)).

- PDFLib GmbH ([http://www.pdflib.com/pdflib/](http://www.pdflib.com/pdflib/)) produces PDFLib. PDFLib is mostly in C, with a Java wrapper; it also has bindings for several other popular languages. The source code is distributed, making it very cross-platform. It's free for noncommercial use; for commercial use, a small licensing fee is required.


- Finally, since I couldn't decide which of these alternatives to use, I just went ahead and wrote my own, SPDF.

Like Perl, SPDF has several names. Perl on a good day is the Practical Extraction and Report Language, but on a bad day it's the Purely Eclectic Rubbish Lister. A true geek has to admire that kind of whimsy. SPDF can be the Simple PDF API, but it can also be the Stupid PDF API. Mostly the latter, I fear. Example 18-8 is a simple servlet that takes the user's name from the HTML form in Figure 18-6 and generates a custom-made shopping coupon with the customer's info imprinted into it, and a unique serial number (for which a Date object provides a cheap stand-in here) to prevent multiple uses of the coupon. I suspect there is a real window of opportunity for such coupons in conjunction with online web sites and large discount retail stores. Unfortunately, I'm too busy writing this book to exploit this marvelous opportunity, so I'll just release the source code to SPDF. If you get rich from it, send me some of the money, OK?

![PDF coupon servlet.](http://www.onjava.com/lpt/a/1000)

I'm not showing the source code for SPDF in this book, as the present version is pretty crude. No font support. No graphics. Single-page documents. It may be released, however; check out http://www.darwinsys.com/freeware/spdf.html if you're interested. Think of SPDF as the Standby PDF API, which you can use while you decide which of the other PDF APIs you really want to use.

When you click on the "Get Yours" button, the servlet is run, generating a PDF file and sending it back to the browser. My Unix version of Netscape tries to save it to disk since I don't have Acrobat loaded; the filename MyCoupon.pdf is provided by the Content-disposition header that I added to the response object. See Figure 18-7.
My test MS-Windows system's copy of Netscape has Acrobat installed, and will run Acrobat as a Netscape Plug-in to display it; see Figure 18-8.

The basic SPDF API uses a PDF object to represent one PDF file. The PDF object has methods to set various things, to add pages to the object (and Page has methods to add text strings, moveTo operations, and others), and finally to write the file. Example 18-8 is the servlet that responds to the coupon request shown in Figure 18-6.

Example 18-8: PDFCouponServlet.java

```java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.darwinsys.spdf.*;

/** Simple PDF-based Coupon Printer Servlet */
public class PDFCouponServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
```
HttpServletResponse response) throws IOException {

    PrintWriter out = response.getWriter();
    response.setContentType("application/pdf");

    // Tell browser to try to display inline, but if not,
    // to save under the given filename.
    response.setHeader("Content-disposition",
        "inline; filename=" + "MyCoupon.pdf");

    PDF p = new PDF(out);
    Page p1 = new Page(p);
    p1.add(new MoveTo(p, 100, 600));
    p1.add(new Text(p,
        "This coupon good for one free coffee in the student lounge."));
    String name = request.getParameter("name");
    if (name == null)
        name = "unknown user";
    p1.add(new Text(p,
        "Printed for the exclusive use of " + name));
    p1.add(new Text(p,
        "by Ian Darwin's PDFCoupon Servlet and DarwinSys SPDF software"));
    p1.add(new Text(p, "at " + new Date().toString(  )));
    p.add(p1);
    p.setAuthor("Ian F. Darwin");

    // Write the PDF file page
    p.writePDF();
}

Most of the Java PDF APIs are roughly similar. Example 18-9 is the Terms servlet rewritten using PDFLib to generate a fancier PDF document with the same information as the HTML version.

Example 18-9: TermsServletPDF.java

    import javax.servlet.*;
    import javax.servlet.http.*;
    import java.io.*;
    import java.util.*;
    import com.pdflib.*;

    /** Output the dictionary in fancy (?) PDF.
    * This version uses "PDFlib", from PDFLib.GmbH (www.pdflib.com).
    */
    public class TermsServletPDF extends HttpServlet {
        /** A printwriter for getting the response. */
        PrintWriter out;

        /** Handle the get request. */
        public void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException {

            try {

                out = new PrintWriter(response.getOutputStream());

                int font;
The end of the servlet in Example 18-10 demonstrates a way to provide a user-friendly wrapper around the occasional exception traceback. Method `warning()` can also be used to print a generic error message without a traceback by passing `null` as the exception argument.

Example 18-10: TermsServletPDF error handling

```java
/** Generic error handler. Must call before any use of "out" */
protected void warning(HttpServletResponse response, String error, Exception e) {
    response.setContentType("text/html");
    try {
        PrintWriter out = response.getWriter();
        out.println("<html><body><pre>");
        catch (IOException exc) {
            System.err.println("EGAD! IO error " + exc + " trying to tell user about " + error + " " + e);
        }
    }
} catch (Exception e) {
    warning(response, "pdflib error:", e);
}
```
return;
}
out.println("<H1>Error</h2>");
out.print("<P>Oh dear. You seem to have run across an error in ");
out.print("our dictionary formatter. We apologize for the inconvenience");
out.print("<P>Error message is ");
out.println(error);

if (e != null) {
    out.print("<P>Exception is: ");
    out.println(e.toString( ));
    out.print("Traceback is: ");
    out.print("<PRE>");
    e.printStackTrace(out);
    out.print("</PRE>");
}
System.out.print("DictionaryServletPDF: ");
System.out.println(error);
if (e != null) {
    System.out.println(e.toString( ));
}

---

**HTML Meets Java: JSP**

**Problem:**
You have a web page that could use a jolt of Java.

**Solution:**
Use the JavaServer Pages method of mixing HTML and Java.

**Discussion**

JavaServer Pages (JSP) shares some general syntax with Microsoft's ASP (Application Server Pages) and the free-software PHP (Programmable Hypertext Processor). They allow a mix of HTML and code; the code is executed on the server side, and the HTML plus the code results are printed as HTML. Because of Java's portability and JSP's full access to the entire Java API, JSP may be the most exciting web technology to come along since the online pizza demonstration. Example 18-11, for example, is the "five integers" code as a JSP.

---

**Example 18-11: fiveints.jsp**

http://www.onjava.com/lpt/a/1000 29.9.2005
Notice how much more compact this is than the servlet version in First Servlet: Generating an HTML Page. It should not surprise you to learn that JSPs are actually compiled into servlets, so most of what you know about servlets also applies to JSP. Let's look at another example that generates an HTML form and calls itself back when you activate the form, and also contains an HTML table to display the current month. Figure 18-9 and Example 18-12 show a JSP version of the CalendarPage program.

Example 18-12: CalendarPage.jsp

```html
<%@page import="java.util.*,java.text.*" %>

<head>
<title>Print a month page.</title>
<meta name="version"

```

Figure 18-9. CalendarPage.jsp in action.
Print a month page, for the Western calendar.

Author Ian F. Darwin, ian@darwinsys.com

HTML code:

```html
<html>
<body bgcolor="white">

<h2>Print a month page, for the Western calendar.</h2>

Author Ian F. Darwin, ian@darwinsys.com

<!-- First get the month and year from the form. -->

boolean yyok = false; // -1 is a valid year, use boolean
int yy = 0, mm = 0;
String yyString = request.getParameter("year");
if (yyString != null && yyString.length() > 0) {
    try {
        yy = Integer.parseInt(yyString);
        yyok = true;
    } catch (NumberFormatException e) {
        out.println("Year " + yyString + " invalid");
    }
}
Calendar c = Calendar.getInstance();
if (!yyok)
    yy = c.get(Calendar.YEAR);
String mmString = request.getParameter("month");
if (mmString == null) {
    mm = c.get(Calendar.MONTH);
} else {
    for (int i=0; i<months.length; i++)
        if (months[i].equals(mmString)) {
            mm = i;
            break;
        }
}
%

<form method=post action="CalendarPage.jsp">
    Month: <select name=month>
        <% for (int i=0; i<months.length; i++) {
            if (i==mm)
                out.print("<option selected>");
            else
                out.print("<option>");
            out.print(months[i]);
            out.println("</option>");
        }%>
    </select>
    Year (4-digit):
    <input type="text" size="5" name="year" value="<%= yy %>"></input>
    <input type=submit value="Display">
</form>

/** The names of the months */
String[] months = {
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"};

/** The days in each month. */
int dom[] = {
    31, 28, 31, 30, /* jan feb mar apr */
    31, 30, 31, 31, /* may jun jul aug */
    30, 31, 30, 31 /* sep oct nov dec */
};
%
```

http://www.onjava.com/lpt/a/1000

29.9.2005
/** The number of days to leave blank at the start of this month */
int leadGap = 0;
%
<table border=1>
<tr><th colspan=7><%= months[mm] %> <%= yy %></tr>
<tr><td>Su<td>Mo<td>Tu<td>We<td>Th<td>Fr<td>Sa</tr>
%
// Compute how much to leave before the first.  
// getDay( ) returns 0 for Sunday, which is just right.  
leadGap = calendar.get(Calendar.DAY_OF_WEEK)-1;

int daysInMonth = dom[mm];
if (calendar.isLeapYear(calendar.get(Calendar.YEAR)) && mm == 1)
++daysInMonth;
out.print("<tr>");

// Blank out the labels before 1st day of month
for (int i = 0; i < leadGap; i++) {
    out.print("<td>&nbsp;");
}

// Fill in numbers for the day of month.
for (int i = 1; i <= daysInMonth; i++) {
    out.print("<td>");
    out.print(i);
    out.print("</td>");

    if (((leadGap + i) % 7 == 0) {  // wrap if end of line.
        out.println("</tr>");
        out.print("<tr>");
    }
}
%
</tr>
</table>

For another example, Example 18-13 shows the list of terms and definitions from First Servlet: Generating an HTML Page done as a JSP.

---

**Example 18-13: terms.jsp**

```html
<HTML>
<HEAD>
    <TITLE>Ian Darwin's Computer Terms and Acronyms</TITLE>
  <%@ page import="java.io.*" %>
</HEAD>
<BODY BGColor=white>
<H1>Ian Darwin's Computer Terms and Acronyms</H1>
<TABLE BORDER=2>
<TR><TH>Term<TH>Meaning</TR>
```

---

http://www.onjava.com/lpt/a/1000 29.9.2005
JSP Include/Forward

Problem:
You want to write a "page-composite" JSP that includes other pages or passes control to another page.

Solution:
Use `<jsp:include>` or `<jsp:forward>.

Discussion

Suppose you have some common HTML code that you want to appear on every page, such as a navigator or header. You could copy it into each HTML and JSP file, but if it changed, you'd have to find all the files that used it and update each of them. It would be much easier to have one copy and include it everywhere you need it. Most web servers feature such a mechanism already (e.g., server-side includes). However, using JSP's mechanism has some advantages, such as the ability to attach objects to a request, a topic I'll explore in *JavaServer Pages Using a Servlet*.

The basic mechanism is simply to have `<jsp:include>` with a PAGE attribute naming the page to be included, and end with `</jsp:include>`. For convenience, you can put the `/` at the end of the opening tag and omit the closing tag. Much of this syntax is taken from XML namespaces (see Chapter 21). The FLUSH attribute is also required, and it must have the value TRUE; this is to remind you that, once you do an include, the contents of the output are actually written. Therefore, you can no longer do anything that involves sending HTTP headers, such as changing content type or transferring control using an HTTP redirect request. So a full JSP include might look like this:
<H2>News of the day</H2>
<jsp:include page="./news.jsp" flush="true" />

The `jsp:forward` request is similar to a `jsp:include`, but you don't get control back afterwards. The attribute `flush="true"` is required on some JSP engines (including the release of Tomcat at the time this book went to press) to remind you that once you do this include, you have committed your output (prior to the include, the output might be all in a buffer). Therefore, as I just stated, you can no longer do anything that might generate headers, including `setContentType()`, `sendRedirect()` and so on.

An alternate include mechanism is `<%@include file="filename"%>`. This mechanism is a bit more efficient (the inclusion is done at the time the JSP is being compiled), but is limited to including text files (the file is read, rather than being processed as an HTTP URL; so if you include, say, a CGI script, the contents of your CGI script are revealed in the JSP output: not useful!). The `jsp:include` can include a URL of any type (HTML, servlet, JSP, CGI, even PHP or ASP).

JavaServer Pages Using a Servlet

Problem:
It may seem that servlets and JSPs are mutually exclusive, but in fact they work well together. You can reduce the amount of Java coding in your JSP by passing control from a servlet to a JSP.

Solution:
Use the Model-View-Controller paradigm, and implement it using `ServletDispatcher().forward()`.

Discussion

Model-View-Controller is a paradigm for building programs that interact well with the user. The Model is an object or collection that represents your data; the View is what the user sees; and the Controller responds to user request. Think of a slide-show (presentation) program: you probably have a text view, a slide view, and a sorter view. Yet when you change the data in any view, all the other views are updated immediately. This is because MVC allows a single model to have multiple views attached to it. MVC provides the basis for most well-designed GUI applications.

Using the Model-View-Controller paradigm, a servlet can be the controller and the JSP can be the view. A servlet, for example, could receive the initial request from the form, interrogate a database based upon the query, construct a collection of objects matching the user's query, and forward it to a JSP to be displayed (the servlet can attach data it found to the request). A good example of this is a search page, which might have only a few (or even one) form parameters, so using a JSP with a bean to receive the results would be overkill. A better design is to have a servlet retrieve the form parameter and contact the search API or database. From there, it would retrieve a list of pages matching the query. It could package these into a `Vector` or `ArrayList`, attach this to the request, and forward it to a JSP for formatting.

The basic syntax of this is:

```java
ArrayList searchResultsList = // get from the query
RequestDispatcher disp;
disp = getServletContext().getRequestDispatcher("searchresults.jsp");
request.setAttribute("my.search.results", searchResultsList);
disp.forward(request, response);
```

This causes the servlet to pass the search results to the JSP. The JSP can retrieve the result set using
this code:

```java
ArrayList myList = (ArrayList) request.getAttribute("my.search.results");
```

You can then use a for loop to print the contents of the search request. Note that the URL in the `getRequestDispatcher()` call must be a call to the same web server, not to a server on a different port or machine.

## Simplifying Your JSP with a JavaBean

**Problem:**
You want to reduce the amount of Java coding in your JSP using a JavaBean component.

**Solution:**
Use `<jsp:useBean>` with the name of your bean.

**Discussion**

JavaBeans is Java's component technology, analogous to COM components on MS-Windows. Recipes and contain a formula for packaging certain Java classes as JavaBeans. While JavaBeans were originally introduced as client-side, GUI-builder-friendly components, there is nothing in the JavaBeans specification that limits their use to the client-side or GUI. In fact, it's fairly common to use JavaBean components with a JSP. It's also easy and useful, so let's see how to do it.

At the bare minimum, a JavaBean is an object that has a public no-argument constructor and follows the set/get paradigm. This means that there is regularity in the get and set methods. Consider a class, each instance of which represents one user account on a login-based web site. For the name, for example, the methods:

```java
public void setName(String name);
public String getName();
```

allow other classes full control over the "name" field in the class but with some degree of encapsulation; that is, the program doesn't have to know the actual name of the field (which might be `name`, or `myName`, or anything else suitable). Other programs can even get a list of your get/set methods using introspection. Example 18-14 is the full class file; as you can see, it is mostly concerned with these set and get methods.

---

**Example 18-14:** User.java, a class usable as a bean
/** Represents one logged in user */
public class User {

    protected String name;
    protected String passwd;
    protected String fullName;
    protected String email;
    protected String city;
    protected String prov;
    protected String country;

    protected boolean editPrivs = false;
    protected boolean adminPrivs = false;

    /** Construct a user with no data -- must be a no-argument */
    * constructor for use in jsp:useBean.
    */
    public User() {
    }

    /** Construct a user with just the name */
    public User(String n) {
        name = n;
    }

    /** Return the nickname. */
    public String getName() {
    }

    public void setName(String nick) {
        name = nick;
    }

    // The password is not public - no getPassword.

    /** Validate a given password against the user's. */
    public boolean checkPassword(String userInput) {
        return passwd.equals(userInput);
    }

    /** Set password */
    public void setPassword(String passwd) {
        this.passwd = passwd;
    }

    /** Get email */
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    // MANY SIMILAR STRING-BASED SET/GET METHODS OMITTED

    /** Get adminPrivs */
    public boolean isAdminPrivileged() {
        return adminPrivs;
    }

    /** Set adminPrivs */

}
The only methods that do anything other than set/get are the normal `toString()` and `isComplete()` (the latter returns true if all required fields have been set in the bean). If you guessed that this has something to do with validating required fields in an HTML form, give yourself a gold star.

We can use this bean in a JSP-based web page just by saying:

```jsp
<jsp:useBean id="myUserBean" scope="request" class="User">
</jsp:useBean>
```

This creates an instance of the class called `myUserBean`. However, at present it is blank; no fields have been set. To fill in the fields, we can either refer to the bean directly within scriptlets, or, more conveniently, we can use `<jsp:setProperty>` to pass a value from the HTML form directly into the bean! This can save us a great deal of coding.

Further, if all the names match up, such as an HTML parameter "name" in the form and a `setName(String)` method in the bean, the entire contents of the HTML form can be passed into a bean using `property="*"`!

```jsp
<jsp:setProperty name="myUserBean" property="*"/>
</jsp:useBean>
```

Now that the bean has been populated, we can check that it is complete by calling its `isComplete()` method. If it’s complete, we print a response, but if not, we direct the user to go back and fill out all the required fields:

```jsp
<% // Now see if they already filled in the form or not...
   if (!myUserBean.isComplete( )) {
   %>
   <TITLE>Welcome New User - Please fill in this form.</TITLE>
   <BODY BGCOLOR=White>
   <H1>Welcome New User - Please fill in this form.</H2>
   <FORM ACTION="name_of_this_page.jsp" METHOD=post>
   // Here we would output the form again, for them to try again.
   </FORM>
   <%
   } else {
   String nick = newUserBean.getName( );
   String fullname = newUserBean.getFullName( );
   // etc...
   // Give the user a welcome
   out.println("Welcome " + fullname);
   %>
```

You'll see the full version of this JSP in Program: JabaDot Web News Portal.

**See Also**

You can extract even more Java out of the JSP, making it look almost like pure HTML, by using Java custom tags. *Custom tags* (also called custom actions) are a new mechanism for reducing the amount of Java code that must be maintained in a JSP. They have the further advantage of looking syntactically just like elements brought in from an XML namespace, making them more palatable both to HTML editor software and to HTML editor personware. Their disadvantage is that to write them requires a greater investment of time than, say, servlets or JSP. However, you don't have to write them to use them; there are several good libraries of custom tags available, one from Tomcat and another from JRun. Sun is also working on a standard for a generic tag library. JSP tags are compiled classes, like applets or servlets, so any tag library from any vendor can be used with any conforming JSP engine. There are a couple of JSP custom tags in the source directory for the
JabaDot program in Program: JabaDot Web News Portal.

JSP Syntax Summary

Problem:
You can't remember all this post-HTML syntax.

Solution:
Use the Table.

Discussion

Table 18-1 summarizes the syntax of JavaServer Pages. As the title implies, it contains only the basics; a more complete syntax can be downloaded from http://java.sun.com/products/jsp/.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scriptlet</td>
<td>&lt;% ;%&gt;</td>
</tr>
<tr>
<td>Expression</td>
<td>&lt;%= expr %&gt;</td>
</tr>
<tr>
<td>Declaration</td>
<td>&lt;%! decls; %&gt;</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;jsp:include page=&quot;URL&quot; flush=true /&gt;</code></td>
</tr>
<tr>
<td>Forward</td>
<td><code>&lt;jsp:forward page=&quot;url&quot;/&gt;</code></td>
</tr>
<tr>
<td>Use bean</td>
<td><code>&lt;jsp:useBean .../&gt;</code></td>
</tr>
<tr>
<td>Set property</td>
<td><code>&lt;jsp:setProperty ... /&gt;</code></td>
</tr>
<tr>
<td>Page directive</td>
<td><code>&lt;%@ page ... %&gt;</code></td>
</tr>
<tr>
<td>Comment</td>
<td><code>&lt;!-- comment --&gt;</code></td>
</tr>
<tr>
<td>Hidden comment</td>
<td><code>&lt;%-- comment --&gt;%&gt;</code></td>
</tr>
</tbody>
</table>

Program: CookieCutter

CookieCutter is a little program I wrote that allows you to display, modify, and even delete
cookies. Since the banner-ad-tracking firm DoubleClick probably keeps a lot of information on your
browsing habits, you want to befuddle them. After all, they are using a tiny bit of storage on your
hard disk to rack up per-click profits, giving you nothing in return (directly, at least; obviously, ad
sponsorship keeps some web sites on the air). In Figure 18-10, I am editing the cookie to, umm,
"update" the personal identity cookie to an invalid number (a lot of 9's, and too many digits). A few
lines above that, you can see the \texttt{prefs.bgcolor} cookie that I set to "green."

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cookiecutter.png}
\caption{The CookieCutter display}
\end{figure}

I won't show the \texttt{CookieCutter} source code here as it doesn't really relate to web techniques (it's a
client-side application), but it's included in the source archive for the book. \texttt{CookieCutter} also
assumes your cookies are stored in the Netscape format; for the Microsoft Explorer format, you'll
have to change the file-reading and file-writing code.

\section*{Program: JabaDot Web News Portal}

Here is perhaps the most ambitious program developed in this book. It's the beginnings of a complete
"news portal" web site, similar to \url{http://www.slashdot.org/}, \url{http://www.deadly.org/}, or
\url{http://daily.daemonnews.org/}. However (and as you should expect!), the entire site is written in Java.
Or perhaps should I say "written in or by Java," since the JSP mechanism -- which is written entirely
in Java -- turns the JSP pages into Java servlets that get run on this site. The web site is shown in
Figure 18-11.
Like most portal sites, JabaDot allows some services (such as the current news items and of course the ubiquitous banner ads) without logging in, but requires a login for others. In this figure I am logged in as myself, so I have a list of all available services. The page that supports this view is index.jsp (Example 18-15), which contains a hodgepodge of HTML and Java code.

Example 18-15: index.jsp

```jsp
<%@page errorPage="oops.jsp"%>
<HTML>
<TITLE>JabaDot - Java News For Ever(yone)</TITLE>
<P ALIGN=CENTER><jsp:include page="/servlet/AdRotator" flush="true"/></P>
<BODY BGCOLOR="#f0f0f0">
<%  HttpSession sess = request.getSession(true);
User user = (User)sess.getValue("jobadot.login");
%>
<TABLE>
<TD WIDTH=75% ALIGN="TOP">
<!-- Most of page, at left -->
<IMG SRC="logo.gif" ALIGN="LEFT"></IMG>
<br CLEAR="ALL">
<jsp:include page="/news.jsp" flush="true"/>
</TD>
<TD WIDTH=25% BGCOLOR="#00cc00" ALIGN="TOP">
<!-- Rest of page, at right -->
<FONT COLOR=WHITE NAME="Helvetica,Arial">% if (user == null) { %>
<FORM ACTION=login.jsp METHOD=POST>
  Name: <INPUT TYPE=TEXT SIZE=8 NAME=nick>
  Password: <INPUT TYPE=PASSWORD SIZE=8 NAME=pass>
  <INPUT TYPE=SUBMIT VALUE="Login" ALIGN= CENTER>
</FORM>
<jsp:include page="public_services.html" flush="true"/>
% } else { %>
Logged in as <%= user.getName( ) %> %>
</TD>
</TABLE>
</BODY>
</HTML>
```
As you can see, this code actually starts with a "page" tag (%@page) to specify an error handling page (the error page just prints the stack trace neatly, along with an apology). Then the output of the AdRotator servlet is included; the program just randomly selects a banner advertisement and outputs it as an HTML anchor around an IMG tag. Then I get the HttpSession object and, from that, the current User object, which is null if there is not a currently logged-in user. The User class was discussed when we talked about JavaBeans in JSPs (see Simplifying Your JSP with a JavaBean); it's used as an ordinary object in most of these JSPs, but as a bean in the newuser.jsp page, when the user has entered all the fields on the "Create an Account" page.

Then there's an HTML table, which basically divides the rest of the page into two large columns. The left side of the page is fairly wide and contains the news stories, their headlines, the submitter's name, the time, optionally a URL, and the text of the news article. A future version will allow the user to send comments on the stories; as Slashdot has demonstrated, this is an important part of "community building," part of the art of keeping people coming back to your web site so you can show them more banner ads. :-)

The navigator part is displayed differently depending on whether you are logged in or not. If you're not, it begins with a login form, then lists the few services that are publicly available as HTML anchors, with the unavailable services in italic text. If you are logged in, there is a full list of links and a logout page at the end.

Before you log in, you must create an account. The trick here is that we require the user to give a valid email address, which we'll use for various authentication purposes and, just possibly, to send them a monthly newsletter by email. To ensure that the user gives a valid email address, we email to them the URL from which they must download the password. Figure 18-12 shows the entry page for this. This form is processed by newuser.jsp.
Example 18-16 is the source for newuser.jsp. As mentioned previously, this gets a User object as a JavaBean (see Simplifying Your JSP with a JavaBean).

Example 18-16: newuser.jsp

```jsp
<%@page errorPage="oops.jsp" import="jabadot.*, java.io.*" %>
<%! java.util.Random r = new java.util.Random(); %>
<jsp:useBean id="newUserBean" scope="request" class="jabadot.User">
   <jsp:setProperty name="newUserBean" property="*"/>
</jsp:useBean>
<jsp:useBean id="mailBean" scope="request" class="jabadot.Mailer"/>
<html>
<%@include file="header.html" %>
<% User user = (User)session.getAttribute("jabadot.login");
   if (user != null) {
   %>
<TITLE>You're already logged on!</TITLE>
<H1>You are logged on!</H2>
<P>Please <A href="logout.jsp">log out</A> before
trying to create a new account. Thank you!</P>
   %>
   <% // Now see if they already filled in the form or not...
      if (!newUserBean.isComplete()) {
         // out.println("<!-- in new -->");
      %>
<TITLE>Welcome New User - Please fill in this form.</TITLE>
<BODY bgcolor=White>
<H1>Welcome New User - Please fill in this form.</H1>
<TABLE>
<TD>
<FORM ACTION="newuser.jsp" METHOD=post>
<table>!-- inner table so fields line up -->
<table>
<thead>
<tr>
<th>Field</th>
<th>Input Type</th>
<th>Size</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nickname</td>
<td>TEXT</td>
<td>10</td>
<td>(required)</td>
</tr>
<tr>
<td>Full name</td>
<td>TEXT</td>
<td>10</td>
<td>(required)</td>
</tr>
<tr>
<td>E-mail</td>
<td>TEXT</td>
<td>10</td>
<td>(required)</td>
</tr>
<tr>
<td>City</td>
<td>TEXT</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Province/State</td>
<td>TEXT</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>Select</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<% if (UserDB.getInstance().getUser(nick) != null) { %>
  <P>It seems that that user name is already in use! Please go back and pick another name.
  <% return; %>
<% } %>

<% String nick = newUserBean.getName(); %>
<% if (UserDB.getInstance().getUser(nick) != null) { %>
  <P>It seems that that user name is already in use! Please go back and pick another name.
  <% return; %>
<% } %>

<% String fullname = newUserBean.getFullName(); %>
<% String email = newUserBean.getEmail(); %>

<!-- Give the user a welcome -->
Welcome <%= fullname %>. We will mail you (at <%= email %> ) with a URL from which you can download your initial password.

<% jsp:setProperty name="newUserBean" property="editPrivileged" value="false"/>
<% jsp:setProperty name="newUserBean" property="adminPrivileged" value="false"%>

<!-- Generate initial random password and store it in the User
String newPass = Password.getNext().toString();
newUserBean.setPassword(newPass);

// NOW add the user to the persistent database.
UserDB.getInstance().addUser(newUserBean);

// Create a temporary HTML file containing the full name
// and the new password, and mail the URL for it to the user.
// This will confirm that the user gave us a working email.
// NEVER show the nickname and the password together!
String tempDir = JDConstants.getProperty("jabadot.tmp_links_dir");
Once you create an account and read the email containing the link for the password, you can return to the site and log in normally. The login form is handled by `login.jsp`, shown in Example 18-17.

---

**Example 18-17: login.jsp**

```jsp
<%@page errorPage="oops.jsp" import="jabadot.*" %>
<HTML>
<%
    User user = (User)session.getAttribute("jabadot.login");
    if (user != null) {
        session.setAttribute("jabadot.message",
            "<H1>You're already logged on!</h2>"+
            "(as user " + user.getName() + "). Please" +
            "<a href="/logout.jsp">logout</a> if you wish to log in as a different user.");
        response.sendRedirect("/jabadot/");
    }

    String nick = request.getParameter("nick");
    String pass = request.getParameter("pass");
    if (nick == null || nick.length() == 0 ||
        pass == null || pass.length() == 0) {
        <!-- Must use jsp include not @include here since
tomcat complains if it sees the @ include twice.
    }
```
** Can't just include it once at beginning, since we
** do a redirect at the end of this jsp if successful.
-->
<jsp:include page="./header.html" flush="true" />
<TITLE>Missing name/password!</TITLE>
<BODY BGCOLOR=WHITE>
<H1>Missing name/password!</h2>
<P>Please enter both a name and a password in the form.
<%    return;
}%

User u = UserDB.getInstance().getUser(nick);
if (u == null || !u.checkPassword(pass)) {
<jsp:include page="./header.html" flush="true" />
<TITLE>Invalid name/password</TITLE>
<BODY BGCOLOR=WHITE>
<H1>Invalid name/password!</h2>
<P>We could not find that name and password combination.
Please try again if you have an account, else go create one.
<%    return;
}%

// Hallelujah! WE FINALLY GOT THIS ONE LOGGED IN.

session.setAttribute("jabadot.login", u); // login flag
//session.setAttribute("jabadot.ads", new AdServlet(  ));
session.setAttribute("jabadot.message","<H1>Welcome back, " + u.getFullName() + "</h2>"));

// For non-admin logins, provide a 3-hour timeout
if (!u.isAdminPrivileged()) {
    session.setMaxInactiveInterval(3600*3);
}

// Send Redirect back to top, so user sees just this in URL textfield.
response.sendRedirect("/jabadot/");
%

After ensuring that you're not already logged in, this page gets the username and password from the
HTML form, checks that both are present, looks up the name in the password database and, if found,
validates the password. If either the name or the password is wrong, I report a generic error (this is
deliberate security policy to avoid giving malicious users any more information than they already
have (This ancient advice comes from the early days of Unix; you'd be surprised how many sites still
don't get it). If you log in, I put the User object representing you into the HttpSession, set a little
greeting, and pass control to the main page via a redirect.

Whether logged in or not, you can send a general comment to the system's administrators via the
submit.jsp page. This simply generates the HTML form shown in Figure 18-13.
This form is processed by `comments.jsp`, shown in Example 18-18, when you press the "Submit Article" button.

**Example 18-18: comments.jsp**

```jsp
<%@page errorPage="oops.jsp" %>
<%@page import="jabadot.*, javax.mail.*" %>
<jsp:useBean id="mailBean" scope="request" class="jabadot.Mailer">
  <jsp:setProperty name="mailBean" property="*"/>
</jsp:useBean>
<%@include file="header.html" %>

User user = (User)session.getAttribute("jabadot.login");
mailBean.setFrom(JDConstants.getProperty("jabadot.mail_from"));
mailBean.setSubject("Comment from jabadot site");
mailBean.addTo(JDConstants.getProperty("jabadot.mail_comments_to"));
mailBean.setServer(JDConstants.getProperty("jabadot.mail.server.smtp"));
String message = request.getParameter("message");
if (message != null)
  mailBean.setBody(message);

// See if they already filled in the form or not...
if (mailBean.isComplete()) {
  try {
    mailBean.doSend();
    // Now attach a thank you note and send them to the index page
    session.setAttribute("jabadot.message",
      "<H1>Mail sent</h2><p>Your commentary has been sent to our chief" +
      " pickle.<b>Thank you.</b></p>"");
    response.sendRedirect("/jabadot/");
    // No return from sendRedirect
  } catch (MessagingException ex) {
    throw new IllegalArgumentException(ex.toString());
  }
}
// ELSE - mailbean is NOT complete, put up form.

<% include file="header.html" %>
```

Figure 18-13. Input form for comments.jsp.
Please send us your feedback on this site.

If (user != null) {
  <a href="mailto:<%= JDConstants.getProperty("jabadot.mail_comments_to") %>?subject=Comments about JabaDot">this <i>mailto</i> link</a>.
}

Name: <input type="text" name="name" size="15" value=""><br>
Email: <input type="text" name="from" size="15" value=""><br>
City: <input type="text" name="City" size="15" value=""><br>
State: <input type="text" name="State" size="15" value=""><br>
Country: <input type="text" name="country" size="15" value=""><br>

<textarea name="message" rows="8" cols="50" wrap="physical"> <%= message % </textarea>

<input type="submit" value="Send Comments"></form>

This page starts off like the first one. I particularly like the code that displays a mailto: URL only if the user is logged in. SPAM perpetrators (see Chapter 19) are notorious for automatically loading entire web sites just to look for mailto: URLs. This is a good way to fence these rodents out, since they normally won't go to the trouble of signing up for a service and providing a real (working) email address just to get one mailto: URL from your site. There are easier ways to find mailto:’s on other sites; hopefully the SPAM perps will go there. For extra fun, make up a unique email address for each user to send mail to, so if you do get spammed, you have an idea who might have done it.

**See Also**

There is more to servlets and JSPs than I’ve got room to tell you about. These technologies offer an interesting partitioning of code and functionality. The JSP can be concerned primarily with getting the input and displaying the results. A JSP can forward to a servlet, or can include or jump to any other local web resource, like an audio file. Servlets and JSP are primary parts of the Java Enterprise Edition, and are becoming very important web server technologies.

For an opposing view (that JSPs are the wrong solution to the wrong problem), surf on over to [http://www.servlets.com/](http://www.servlets.com/). For more information on servlets and JSPs, refer to the O'Reilly books *Java Servlet Programming* and *JavaServer Pages*. 

Return to [ONJava.com](http://www.onjava.com/lpt/a/1000).