Java Tip 41: POSTing via Java revisited

Learn how to display the HTML document returned by the Web server

Summary
In *JavaWorld*'s August issue, we ran a Java Tip on "POSTing from applets." Not only was the tip popular with readers, but it spawned a number of questions. This new tip on POSTing via Java attempts to satisfy your curiosity. We show you how to send POST requests to Web servers and display the response in an applet. *(2,000 words)*

By John D. Mitchell

Whew! Our previous tip on POSTing from applets generated a slew of reader questions. The predominant question was "How do I display the HTML document returned by the POST CGI-bin handler on the Web server?". In this tip, we explore the answers to that question and delve into some cool server-side Java issues.

Note: This tip assumes an understanding on the part of the reader of the basic problems and issues of POSTing via Java. If you're not familiar with these concepts, refer to Java Tip 34.

So, just how do we display the results of a POST from an applet? Well, there are four answers to that question. In order of increasing pain, these are:

- We can't.
- Don't post.
- Use a bean.
- Cheat.

As discussed in Java Tip 34, the current browser security managers will not allow an applet to display applet-generated HTML in a browser; the browser only lets us point it toward URLs that it will display on our behalf. This situation is just so unsatisfying!

We can side-step the POST results display restriction by -- surprise! -- *not* using POST. Instead, we can encode some information in the URL that we provide to the `showDocument()` method. That information will get passed on to the the Web server as parameters to the GET request. Unfortunately, this does have some drawbacks: Only limited amounts of data can be transferred -- plus, the URL is munged in the process. So it’s pretty ugly. We’ll see an example of how to code this a bit later.

Just recently, Sun’s JavaSoft division released an HTML renderer bean. (There are a couple of other commercial offerings.) So, it’s possible to use the bean as part of your applet and have it display the pages. What are the drawbacks? Size, compatibility, and cost. The bean certainly isn’t small, it requires a browser supporting beans, and it isn’t free. Of course, we could all spend time writing our own rendering component, but that’s silly.

The fun and productive solution to the problem is to cheat. In this particular case, we cheat by requiring the collusion of the server-side code (for example, the CGI-bin script) with our applet. The basic idea is simple: Combine the use of POST with a subsequent GET. The process is as follows:

1. The applet POSTs information to the server just as before.
2. The server uses the POST information to generate HTML.
3. The server *saves* the HTML to a file on the Web server.
4. The server returns a magical key to the applet.
5. The applet encodes the key into a URL back to the server.
6. The applet instructs the browser to display a Web page by using the generated URL in a `showDocument()` call.
7. The server accepts the GET request and extracts the magic key parameter.
8. The server retrieves the file associated with the magic key.
9. The server returns the HTML contents from the file to the browser.
10. The browser displays the HTML contents.

This back-and-forth process is definitely more complicated than the other solutions, but it works today across a wide combination of clients and servers. The drawbacks to this process stem from needing to perform multiple HTTP requests to fulfill a single, complete transaction. We must retain "state" information across the multiple requests to be able to keep track of what’s going on (recall that HTTP is a stateless request/response protocol). Robustly managing the requisite state information can be quite challenging. As John Ousterhout, the father of the Tcl scripting language and the Sprite distributed operating system, has said: "State is the second worst thing in distributed computing. No, state is the worst."

The server part is the most complicated piece, so let’s look at the applet first. :-) There are only a few differences between this applet and the Happy applet in the previous Java Tip 34. POSTing to the server is identical, but we have to modify the reading of the response from the server:

```
input = new DataInputStream (urlConn.getInputStream ());

String str = null;
String firstLine = null;
while (null != ((str = input.readLine())))
     {
     if (null == firstLine)
          firstLine = str;

     System.out.println (str);
     textArea.appendText (str + "\n");
     }

input.close ();
```

By fiat, the server returns a *magic key* as the first line. The magic key is the piece of state information that is used to uniquely identify which transaction the applet is involved in with this server. If there are any problems in processing the POST request, the server notifies the applet that this is the case by returning the string "nil" followed by a textual description of the problem. The only thing the applet needs to do now is build the URL and call `showDocument()` to display the HTML:

```
if (null != firstLine)
    {
    url = new URL ("http://" +
                   ((getCodeBase()).getHost()).toString() +
                   "/poster?" + firstLine);
    (getAppletContext()).showDocument (url, "_blank");
    }
```

Be sure to note that the URL parameter must be URL-encoded. In that snippet, we only have to add the question mark to separate the base URL from the passed parameters, since the magic key from the server is already safely encoded.

Now that we have the applet part taken care of, let's dive into the server. The server-side code in the earlier Java Tip on POSTing was a traditional CGI-bin script written in Perl. Perl is a fine solution, but wouldn't you rather write server-side Java? We can write CGI-bin scripts in Java (see the Resources section), but there's a much better solution: Java that is run as part of the Web server itself. This type of server-side Java is known as a *servlet*. The solution we're presenting here will be a servlet written to the Java Servlet API -- although you can implement the same solution using CGI-bin scripts in Perl, Tcl, Java, or whatever.

Note that an introduction to servlet programming and administration is beyond the scope of this article; we'll be covering only the main issues that directly relate to the POSTing solution.

The PosterServlet code contains a fair number of comments to guide your walk-through. There's lots of error handling and extra checking to handle the plethora of possible problems, some denial-of-service attacks, and so on, but mostly you can ignore that stuff. (We'll discuss security issues in more depth later.) The servlet is written to the Java 1.1.x APIs (whereas the applet code is Java 1.0.2 code).

The `doPost()` method handles the POST request -- that is, it takes care of the first three server responsibilities: It generates an HTML document based on the POSTed information, saves the document to a temporary disk file, and returns a magic key to the applet that identifies the HTML document and is suitable for directly embedding into a subsequent GET request.

Here's the core of the code. (See the actual source file for the complete code.) Implementation note: The magic key actually is the name of the file that generated the HTML document for that transaction. The name is generated using the `java.util.Random` class to generate a Long value.

```
protected void doPost (HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException
    {
    response.setContentType ("text/plain");

    // Build the output filename.
    String fileName = (new Long (randomizer.nextLong())).toString();
    File file = null;
```

```
try
    {
    file = new File (posterTempDir + File.separator +
                     fileName + posterTempExt);
    }
catch (Exception e)
    {
    sendPostFailure (response,
                     "Unable to build output file path!");
    return;
    }

// Open the output file.
PrintWriter output = null;
try
    {
    output = new PrintWriter
        ( new BufferedWriter
          ( new FileWriter (file)));
    }
catch (IOException e)
    {
    sendPostFailure (response, "Unable to open output file!");
    return;
    }

output.println ("<html>");
output.print ("<head><title>Poster Servlet Generated Output");
output.println ("</title></head>");
output.println ("<body>");

// Now, loop through the request headers and spew them to the file.
String headerName = null;
Enumeration headers = request.getHeaderNames();
if (headers.hasMoreElements())
    {
    output.println ("<h1>CGI headers:</h1><hr>");
    output.println ("<ul>");
    while (headers.hasMoreElements())
        {
        headerName = (String) headers.nextElement();
        output.print ("<li><b>");
        output.print (headerName);
        output.print (" = ");
        output.print (request.getHeader (headerName));
        output.println ("</b></li><br>");
        }
    output.println ("</ul><hr><br>");
    }

// Deal with the POST contents.
if (0 < request.getContentLength())
    {
    String line = null;

    // Translate all of those incoming bytes to characters.
    BufferedReader in = new BufferedReader
        ( new InputStreamReader (request.getInputStream()));

    output.println ("<h1>POST contents:<h1><hr>");
    output.println ("<p><pre>");
```

```
        // Read each line of input and spew it to the output file.
        HttpUtils httpUtils = new HttpUtils();
        try
            {
            while (null != (line = in.readLine()))
                {
                try
                    {
                    Hashtable data = httpUtils.parseQueryString (line);
                    String keyName = null;
                    Enumeration keys = data.keys();

                    while (keys.hasMoreElements())
                        {
                        String[] values = null;

                        keyName = (String) keys.nextElement();
                        output.print (keyName);

                        values = (String[]) data.get (keyName);

                        output.print (" =");
                        if (1 < values.length)
                            output.println ("");
                        for (int i = 0; i < values.length; i++)
                            {
                            output.print ("\t");
                            output.println (values[i]);
                            }
                        }
                    output.println();
                    }
                catch (IllegalArgumentException e)
                    {
                    output.print (line);
                    }
                }
            }
        catch (IOException e)
            {
            output.println ("\n</pre><br><p><b>");
            output.println ("Unable to read entire POST contents!");
            output.println ("</b></p><br><pre>\n");
            }

        output.println ("</pre></p><hr>");
        }

    // Tidy up the output file.
    output.println ("</body></html>");
    output.flush();
    output.close();

    // Build the POST response.
    ServletOutputStream out = response.getOutputStream();
    out.println (fileName);
    out.println ("That is the magic value to return as a URL " +
                 "parameter in a showDocument() call to\n" +
                 "get the data generated by this POST call.");
    }       // End of doPost().
```

The `doGet()` processes the GET request. It takes care of extracting the magic key from the GET query, loading the HTML document from the associated disk file, and returning the HTML document:

```
protected void doGet (HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException
    {
    response.setContentType ("text/html");

    // Get the identifier the doPost() method gave the caller, if any.
    String fileName = request.getQueryString();

    // Build the input filename.
    File file = null;
    try
        {
        file = new File (posterTempDir + File.separator +
                         fileName + posterTempExt);
        }
    catch (Exception e)
        {
        sendGetFailure (response,
                        "Unable to get data file. Please start over!");
        return;
        }

    // Open the file.
    BufferedReader input = null;
    try
        {
        input = new BufferedReader ( new FileReader (file));
        }
    catch (FileNotFoundException e)
        {
        sendGetFailure (response,
                        "Unable to find data file. Please start over!");
        return;
        }

    // Read in the data file and spew it out.
    ServletOutputStream out = response.getOutputStream();
    String line = null;
    try
        {
        while (null != (line = input.readLine()))
            {
            out.println (line);
            }
        }
    catch (IOException e)
        {
        out.println ("\n<br><p><b>");
        out.println ("Unable to read entire data file contents!");
        out.println ("</b></p><br>\n");
        }

    // Close and delete the input file.
    input.close();
    try
        {
```

```
            if (!file.delete())
                {
                // Log the error for the SysAdmin to deal with.
                }
            }
        catch (SecurityException e)
            {
            }
        }       // End of doGet().
```

The last bit of code worth mentioning is the `init()` method. It initializes some global data and deletes any temporary files that have not yet been claimed by using the `clearTempDir()` method. Check out the source files for the definitions of those two methods (there's nothing interesting enough about them to show them here :-). Note that a more robust and secure solution would be to limit the time that a temporary file can live, and put a limit on the total number of files that can be alive at any given time. Adding some logic to deal with other resource saturation issues also would be prudent.

Speaking of security and robustness, the predominantly paranoid may want something a bit more, um, paranoid. Using SSL or some other cryptographic solution definitely would increase overall security.

Well, there you have it. We've colluded with the server to make it possible to have the browser display the results of a POST. ■

Also this month in JavaWorld
Go

About the author
Alternately as employee, consultant, and principal of his own company, John has invested the last ten years in developing cutting-edge computer software and advising other developers. John co-authored *Making Sense of Java: A Guide for Managers and the Rest of Us* and *Dummies 101: Java Programming* and has published articles in programming journals. In addition to writing the *Java Tips & Tricks* column for *JavaWorld*, he moderates the comp.lang.tcl.announce and comp.binaries.geos newsgroups. Reach John at john.mitchell@javaworld.com.

Resources

- PosterServlet.java http://www.javaworld.com/javatips/javatip41/PosterServlet.java
- Java Tip #34 http://www.javaworld.com/javatips/jw-javatip34.html
- W3C's Java web server, Jigsaw http://www.w3.org/Jigsaw/
- JavaSoft's Java Web Server http://jserv.javasoft.com/index.html
- Using Java for CGI-bin scripting http://www.javaworld.com/jw-01-1997/jw-01-cgiscripts.html