Handling the Client Request: HTTP Request Headers

Topics in This Chapter

- Reading HTTP request headers
- Building a table of all the request headers
- Understanding the various request headers
- Reducing download times by compressing pages
- Differentiating among types of browsers
- Customizing pages according to how users got there
- Accessing the standard CGI variables

Chapter 5

One of the keys to creating effective servlets is understanding how to manipulate the HyperText Transfer Protocol (HTTP). Thoroughly understanding this protocol is not an esoteric, theoretical concept, but rather a practical issue that can have an immediate impact on the performance and usability of your servlets. This section discusses the HTTP information that is sent from the browser to the server in the form of request headers. It explains the most important HTTP 1.1 request headers, summarizing how and why they would be used in a servlet. As we see later, request headers are read and applied the same way in JSP pages as they are in servlets.

Note that HTTP request headers are distinct from the form (query) data discussed in the previous chapter. Form data results directly from user input and is sent as part of the URL for `GET` requests and on a separate line for `POST` requests. Request headers, on the other hand, are indirectly set by the browser and are sent immediately following the initial `GET` or `POST` request line. For instance, the following example shows an HTTP request that might result from a user submitting a book-search request to a servlet at http://www.somebookstore.com/servlet/Search. The request includes the headers `Accept`, `Accept-Encoding`, `Connection`, `Cookie`, `Host`, `Referer`, and `User-Agent`, all of which might be important to the operation of the servlet, but none of which can be derived from the form data or deduced automatically: the servlet needs to explicitly read the request headers to make use of this information.

```
GET /servlet/Search?keywords=servlets+jsp HTTP/1.1
Accept: image/gif, image/jpg, */*
Accept-Encoding: gzip
Connection: Keep-Alive
Cookie: userID=id456578
```

```
Host: www.somebookstore.com
Referer: http://www.somebookstore.com/findbooks.html
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

5.1 Reading Request Headers

Reading headers is straightforward; just call the `getHeader` method of `HttpServletRequest` with the name of the header. This call returns a `String` if the specified header was supplied in the current request, `null` otherwise. In HTTP 1.0, all request headers are optional; in HTTP 1.1, only `Host` is required. So, always check for `null` before using a request header.

Core Approach

Always check that the result of `request.getHeader` is non-`null` before using it.

Header names are not case sensitive. So, for example, `request.getHeader("Connection")` is interchangeable with `request.getHeader("connection")`.

Although `getHeader` is the general-purpose way to read incoming headers, a few headers are so commonly used that they have special access methods in `HttpServletRequest`. Following is a summary.

- **`getCookies`**
  The `getCookies` method returns the contents of the `Cookie` header, parsed and stored in an array of `Cookie` objects. This method is discussed in more detail in Chapter 8 (Handling Cookies).
- **`getAuthType` and `getRemoteUser`**
  The `getAuthType` and `getRemoteUser` methods break the `Authorization` header into its component pieces.
- **`getContentLength`**
  The `getContentLength` method returns the value of the `Content-Length` header (as an `int`).
- **`getContentType`**
  The `getContentType` method returns the value of the `Content-Type` header (as a `String`).

- **`getDateHeader` and `getIntHeader`**
  The `getDateHeader` and `getIntHeader` methods read the specified headers and then convert them to `Date` and `int` values, respectively.
- **`getHeaderNames`**
  Rather than looking up one particular header, you can use the `getHeaderNames` method to get an `Enumeration` of all header names received on this particular request. This capability is illustrated in Section 5.2 (Making a Table of All Request Headers).
- **`getHeaders`**
  In most cases, each header name appears only once in the request. Occasionally, however, a header can appear multiple times, with each occurrence listing a separate value. `Accept-Language` is one such example. You can use `getHeaders` to obtain an `Enumeration` of the values of all occurrences of the header.

Finally, in addition to looking up the request headers, you can get information on the main request line itself (i.e., the first line in the example request just shown), also by means of methods in `HttpServletRequest`. Here is a summary of the four main methods.

- **`getMethod`**
  The `getMethod` method returns the main request method (normally, `GET` or `POST`, but methods like `HEAD`, `PUT`, and `DELETE` are possible).
- **`getRequestURI`**
  The `getRequestURI` method returns the part of the URL that comes after the host and port but before the form data. For example, for a URL of http://randomhost.com/servlet/search.BookSearch?subject=jsp, `getRequestURI` would return `"/servlet/search.BookSearch"`.
- **`getQueryString`**
  The `getQueryString` method returns the form data. For example, with http://randomhost.com/servlet/search.BookSearch?subject=jsp, `getQueryString` would return `"subject=jsp"`.
- **`getProtocol`**
  The `getProtocol` method returns the third part of the request line, which is generally `HTTP/1.0` or `HTTP/1.1`. Servlets should usually check `getProtocol` before specifying *response* headers (Chapter 7) that are specific to HTTP 1.1.

5.2 Making a Table of All Request Headers

Listing 5.1 shows a servlet that simply creates a table of all the headers it receives, along with their associated values. It accomplishes this task by calling `request.getHeaderNames` to obtain an `Enumeration` of headers in the current request. It then loops down the `Enumeration`, puts the header name in the left table cell, and puts the result of `getHeader` in the right table cell. Recall that `Enumeration` is a standard interface in Java; it is in the `java.util` package and contains just two methods: `hasMoreElements` and `nextElement`.

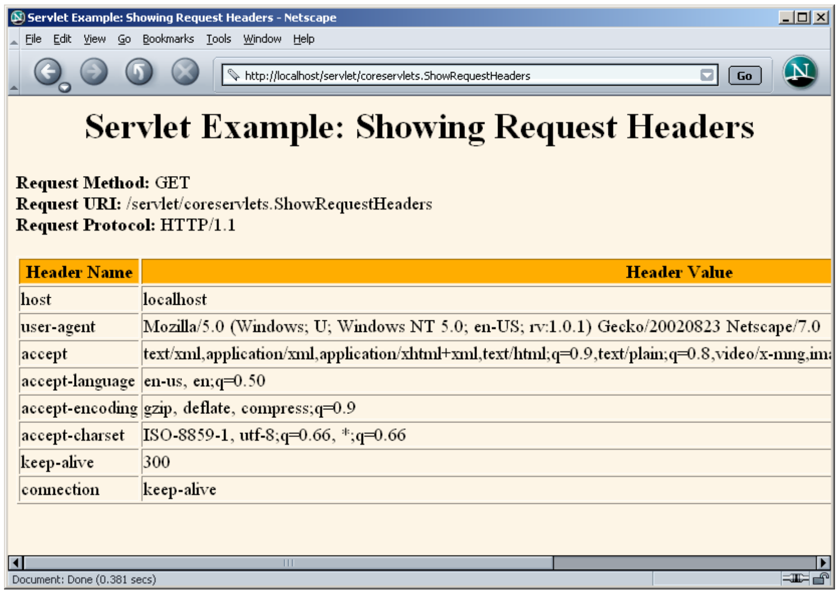
The servlet also prints three components of the main request line (method, URI, and protocol). Figures 5–1 and 5–2 show typical results with Netscape and Internet Explorer.

Listing 5.1 ShowRequestHeaders.java

```
package coreservlets;

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

/** Shows all the request headers sent on the current request. */

public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Servlet Example: Showing Request Headers";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<B>Request Method: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Request URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Request Protocol: </B>" +
```

Listing 5.1 ShowRequestHeaders.java *(continued)*

```
                    request.getProtocol() + "<BR><BR>\n" +
                    "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                    "<TR BGCOLOR=\"#FFAD00\">\n" +
                    "<TH>Header Name<TH>Header Value");
      Enumeration headerNames = request.getHeaderNames();
      while(headerNames.hasMoreElements()) {
        String headerName = (String)headerNames.nextElement();
        out.println("<TR><TD>" + headerName);
        out.println("    <TD>" + request.getHeader(headerName));
      }
      out.println("</TABLE>\n</BODY></HTML>");
    }

    /** Since this servlet is for debugging, have it
     *  handle GET and POST identically.
     */

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
      doGet(request, response);
    }
  }
```

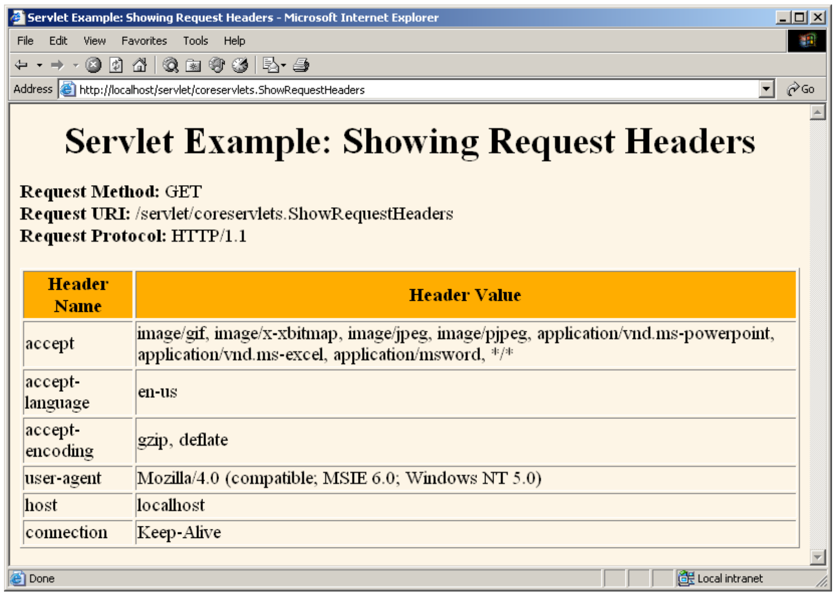

Servlet Example: Showing Request Headers - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.ShowRequestHeaders
Go
Document: Done (0.381 secs)

Servlet Example: Showing Request Headers

Request Method: GET
Request URI: /servlet/coreservlets.ShowRequestHeaders
Request Protocol: HTTP/1.1

Header Name	Header Value
host	localhost
user-agent	Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.0.1) Gecko/20020823 Netscape/7.0
accept	text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,im
accept-language	en-us, en;q=0.50
accept-encoding	gzip, deflate, compress;q=0.9
accept-charset	ISO-8859-1, utf-8;q=0.66, *;q=0.66
keep-alive	300
connection	keep-alive

Figure 5–1 Request headers sent by Netscape 7 on Windows 2000.


Servlet Example: Showing Request Headers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.ShowRequestHeaders
Go
Done
Local intranet

Servlet Example: Showing Request Headers

Request Method: GET
Request URI: /servlet/coreservlets.ShowRequestHeaders
Request Protocol: HTTP/1.1

Header Name	Header Value
accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
accept-language	en-us
accept-encoding	gzip, deflate
user-agent	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
host	localhost
connection	Keep-Alive

Figure 5–2 Request headers sent by Internet Explorer 6 on Windows 2000.

5.3 Understanding HTTP 1.1 Request Headers

Access to the request headers permits servlets to perform a number of optimizations and to provide a number of features not otherwise possible. This section summarizes the headers most often used by servlets; for additional details on these and other headers, see the HTTP 1.1 specification, given in RFC 2616. The official RFCs are archived in a number of places; your best bet is to start at http://www.rfc-editor.org/ to get a current list of the archive sites. Note that HTTP 1.1 supports a superset of the headers permitted in HTTP 1.0.

Accept

This header specifies the MIME types that the browser or other clients can handle. A servlet that can return a resource in more than one format can examine the `Accept` header to decide which format to use. For example, images in PNG format have some compression advantages over those in GIF, but not all browsers support PNG. If you have images in both formats, your servlet can call `request.getHeader("Accept")`, check for `image/png`, and if it finds a match, use *blah*.png filenames in all the `IMG` elements it generates. Otherwise, it would just use *blah*.gif.

See Table 7.1 in Section 7.2 (Understanding HTTP 1.1 Response Headers) for the names and meanings of the common MIME types.

Note that Internet Explorer 5 and 6 have a bug whereby the `Accept` header is sent improperly when you reload a page. It is sent properly in the original request, however.

Accept-Charset

This header indicates the character sets (e.g., ISO-8859-1) the browser can use.

Accept-Encoding

This header designates the types of encodings that the client knows how to handle. If the server receives this header, it is free to encode the page by using one of the formats specified (usually to reduce transmission time), sending the `Content-Encoding` response header to indicate that it has done so. This encoding type is completely distinct from the MIME type of the actual document (as specified in the `Content-Type` response header), since this encoding is reversed *before* the browser decides what to do with the content. On the other hand, using an encoding the browser doesn't understand results in incomprehensible pages. Consequently, it is critical that you explicitly check the `Accept-Encoding` header before using any type of content encoding. Values of `gzip` or `compress` are the two most common possibilities.

Compressing pages before returning them is a valuable service because the cost of decoding is likely to be small compared with the savings in transmission time. See Section 5.4 in which gzip compression is used to reduce download times by a factor of more than 10.

Accept-Language

This header specifies the client's preferred languages in case the servlet can produce results in more than one language. The value of the header should be one of the standard language codes such as `en`, `en-us`, `da`, etc. See RFC 1766 for details (start at http://www.rfc-editor.org/ to get a current list of the RFC archive sites).

Authorization

This header is used by clients to identify themselves when accessing password-protected Web pages. For details, see the chapters on Web application security in Volume 2 of this book.

Connection

This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or other browser to retrieve multiple files (e.g., an HTML file and several associated images) with a single socket connection, thus saving the overhead of negotiating several independent connections. With an HTTP 1.1 request, persistent connections are the default, and the client must specify a value of `close` for this header to use old-style connections. In HTTP 1.0, a value of `Keep-Alive` means that persistent connections should be used.

Each HTTP request results in a new invocation of a servlet (i.e., a thread calling the servlet's `service` and `do`*`Xxx`* methods), regardless of whether the request is a separate connection. That is, the server invokes the servlet only after the server has already read the HTTP request. This means that servlets need to cooperate with the server to handle persistent connections. Consequently, the servlet's job is just to make it *possible* for the server to use persistent connections; the servlet does so by setting the `Content-Length` response header. For details, see Chapter 7 (Generating the Server Response: HTTP Response Headers).

Content-Length

This header is applicable only to `POST` requests and gives the size of the `POST` data in bytes. Rather than calling `request.getIntHeader("Content-Length")`, you can simply use `request.getContentLength()`. However, since servlets take care of reading the form data for you (see Chapter 4), you rarely use this header explicitly.

Cookie

This header returns cookies to servers that previously sent them to the browser. Never read this header directly because doing so would require cumbersome low-level parsing; use `request.getCookies` instead. For details, see Chapter 8 (Handling Cookies). Technically, `Cookie` is not part of HTTP 1.1. It was originally a Netscape extension but is now widely supported, including in both Netscape and Internet Explorer.

Host

In HTTP 1.1, browsers and other clients are *required* to specify this header, which indicates the host and port as given in the original URL. Because of the widespread use of virtual hosting (one computer handling Web sites for multiple domain names), it is quite possible that the server could not otherwise determine this information. This header is not new in HTTP 1.1, but in HTTP 1.0 it was optional, not required.

If-Modified-Since

This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a 304 (`Not Modified`) header if no newer result is available. This option is useful because it lets browsers cache documents and reload them over the network only when they've changed. However, servlets don't need to deal directly with this header. Instead, they should just implement the `getLastModified` method to have the system handle modification dates automatically. For an example, see the lottery numbers servlet in Section 3.6 (The Servlet Life Cycle).

If-Unmodified-Since

This header is the reverse of `If-Modified-Since`; it specifies that the operation should succeed only if the document is older than the specified date. Typically, `If-Modified-Since` is used for `GET` requests ("give me the document only if it is newer than my cached version"), whereas `If-Unmodified-Since` is used for `PUT` requests ("update this document only if nobody else has changed it since I generated it"). This header is new in HTTP 1.1.

Referer

This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the `Referer` header when the browser requests Web page 2. Most major browsers set this header, so it is a useful way of tracking where requests come from. This capability is helpful for tracking advertisers who refer people to your site, for slightly changing content depending on the referring site, for identifying when users first enter your application, or simply for keeping track of where your traffic comes from. In the last case, most people rely on Web server log files, since the `Referer` is typically recorded there. Although the `Referer` header is useful, don't rely too heavily on it since it can easily be spoofed by a custom client. Also, note that, owing to a spelling mistake by one of the original HTTP authors, this header is `Referer`, not the expected `Referrer`.

Finally, note that some browsers (Opera), ad filters (Web Washer), and personal firewalls (Norton) screen out this header. Besides, even in normal situations, the header is only set when the user follows a link. So, be sure to follow the approach you should be using with all headers anyhow: check for `null` before using the header.

See Section 5.6 (Changing the Page According to How the User Got There) for details and an example.

User-Agent

This header identifies the browser or other client making the request and can be used to return different content to different types of browsers. Be wary of this use when dealing only with Web browsers; relying on a hard-coded list of browser versions and associated features can make for unreliable and hard-to-modify servlet code. Whenever possible, use something specific in the HTTP headers instead. For example, instead of trying to remember which browsers support gzip on which platforms, simply check the `Accept-Encoding` header.

However, the `User-Agent` header is quite useful for distinguishing among different *categories* of client. For example, Japanese developers might see whether the `User-Agent` is an Imode cell phone (in which case they would redirect to a chtml page), a Skynet cell phone (in which case they would redirect to a wml page), or a Web browser (in which case they would generate regular HTML).

Most Internet Explorer versions list a "Mozilla" (Netscape) version first in their `User-Agent` line, with the real browser version listed parenthetically. The Opera browser does the same thing. This deliberate misidentification is done for compatibility with JavaScript; JavaScript developers often use the `User-Agent` header to determine which JavaScript features are supported. So, if you want to differentiate Netscape from Internet Explorer, you have to check for the string "MSIE" or something more specific, not just the string "Mozilla." Also note that this header can be easily spoofed, a fact that calls into question the reliability of sites that use this header to "show" market penetration of various browser versions.

See Section 5.5 (Differentiating Among Different Browser Types) for details and an example.

5.4 Sending Compressed Web Pages

Gzip is a text compression scheme that can dramatically reduce the size of HTML (or plain text) pages. Most recent browsers know how to handle gzipped content, so the server can compress the document and send the smaller document over the network, after which the browser will automatically reverse the compression (no user action required) and treat the result in the normal manner. Sending such compressed content can be a real time saver since the time required to compress the document on the server and then uncompress it on the client is typically dwarfed by the time saved in download time, especially when dialup connections are used.

I CALCULATED THE TOTAL TIME THAT HUMANS HAVE WAITED FOR WEB PAGES TO LOAD...
IT CANCELS OUT ALL THE PRODUCTIVITY GAINS OF THE INFORMATION AGE.
SOMETIMES I THINK THE WEB IS A BIG PLOT TO KEEP PEOPLE LIKE ME AWAY FROM NORMAL SOCIETY.
UH-OH, HE'S ON TO ME.
E-mail: SCOTTADAMS@AOL.COM
© 1997 United Feature Syndicate, Inc.
5/5/97

DILBERT reprinted by permission of United Feature Syndicate, Inc.

However, although most recent browsers support this capability, not all do. If you send gzipped content to browsers that don't support this capability, the browsers will not be able to display the page at all. Fortunately, browsers that support this feature indicate that they do so by setting the `Accept-Encoding` request header. Browsers that support content encoding include most versions of Netscape for Unix, most versions of Internet Explorer for Windows, and Netscape 4.7 and later for Windows. Earlier Netscape versions on Windows and Internet Explorer on non-Windows platforms generally do not support content encoding.

Listing 5.2 shows a servlet that checks the `Accept-Encoding` header, sending a compressed Web page to clients that support gzip encoding (as determined by the `isGzipSupported` method of Listing 5.3) and sending a regular Web page to those that don't. The result (see Figure 5–3) yielded a compression of over *300*-fold and a speedup of more than a factor of *10* when a dialup connection was used. In repeated tests with Netscape and Internet Explorer on a 28.8K modem connection, the compressed page averaged less than 5 seconds to completely download, whereas the uncompressed page consistently took more than 50 seconds. Results were less dramatic with faster connections, but the improvement was still significant. Gzip compression is such a useful technique that we later present a filter that lets you apply gzip compression to designated servlets or JSP pages without changing the actual code of the individual resources. For details, see the chapter on servlet and JSP filters in Volume 2 of this book.

Core Tip

Gzip compression can dramatically reduce the download time of long text pages.

Implementing compression is straightforward since support for the gzip format is built in to the Java programming language by classes in `java.util.zip`. The servlet first checks the `Accept-Encoding` header to see if it contains an entry for gzip. If so, it uses a `PrintWriter` wrapped around a `GZIPOutputStream` and specifies `gzip` as the value of the `Content-Encoding` response header. If gzip is not supported, the servlet uses the normal `PrintWriter` and omits the `Content-Encoding` header. To make it easy to compare regular and compressed performance with the same browser, we also added a feature whereby we can suppress compression by including `?disableGzip` at the end of the URL.

Listing 5.2 LongServlet.java

```
package coreservlets;

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

/** Servlet with <B>long</B> output. Used to test
 *  the effect of the gzip compression.
 */

public class LongServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");

    // Change the definition of "out" depending on whether
    // or not gzip is supported.
    PrintWriter out;
    if (GzipUtilities.isGzipSupported(request) &&
        !GzipUtilities.isGzipDisabled(request)) {
      out = GzipUtilities.getGzipWriter(response);
      response.setHeader("Content-Encoding", "gzip");
    } else {
      out = response.getWriter();
    }

    // Once "out" has been assigned appropriately, the
    // rest of the page has no dependencies on the type
    // of writer being used.
```

Listing 5.2 LongServlet.java *(continued)*

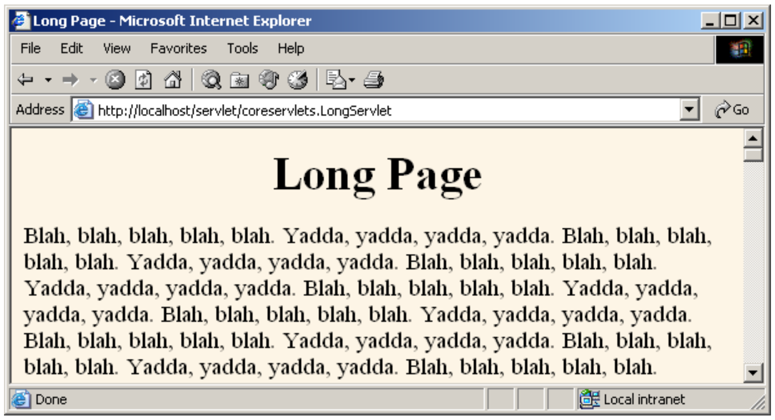
```
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Long Page";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n");
    String line = "Blah, blah, blah, blah, blah. " +
                  "Yadda, yadda, yadda, yadda.";
    for(int i=0; i<10000; i++) {
      out.println(line);
    }
    out.println("</BODY></HTML>");
    out.close(); // Needed for gzip; optional otherwise.
  }
}
```

Listing 5.3 GzipUtilities.java

```
package coreservlets;

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

/** Three small static utilities to assist with gzip encoding.
 *  <UL>
 *    <LI>isGzipSupported: does the browser support gzip?
 *    <LI>isGzipDisabled: has the user passed in a flag
 *        saying that gzip encoding should be disabled for
 *        this request? (Useful so that you can measure
 *        results with and without gzip on the same browser).
 *    <LI>getGzipWriter: return a gzipping PrintWriter.
 *  </UL>
 */

public class GzipUtilities {

  /** Does the client support gzip? */
```

Listing 5.3 GzipUtilities.java *(continued)*

```
  public static boolean isGzipSupported
      (HttpServletRequest request) {
    String encodings = request.getHeader("Accept-Encoding");
    return((encodings != null) &&
           (encodings.indexOf("gzip") != -1));
  }

  /** Has user disabled gzip (e.g., for benchmarking)? */

  public static boolean isGzipDisabled
      (HttpServletRequest request) {
    String flag = request.getParameter("disableGzip");
    return((flag != null) && (!flag.equalsIgnoreCase("false")));
  }

  /** Return gzipping PrintWriter for response. */

  public static PrintWriter getGzipWriter
      (HttpServletResponse response) throws IOException {
    return(new PrintWriter
            (new GZIPOutputStream
              (response.getOutputStream())));
  }
}
```


Long Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.LongServlet
Go
Done
Local intranet

Long Page

Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah.

Figure 5–3 Since the Windows version of Internet Explorer 6 supports gzip, this page was sent gzipped over the network and automatically reconstituted by the browser, resulting in a large saving in download time.

5.5 Differentiating Among Different Browser Types

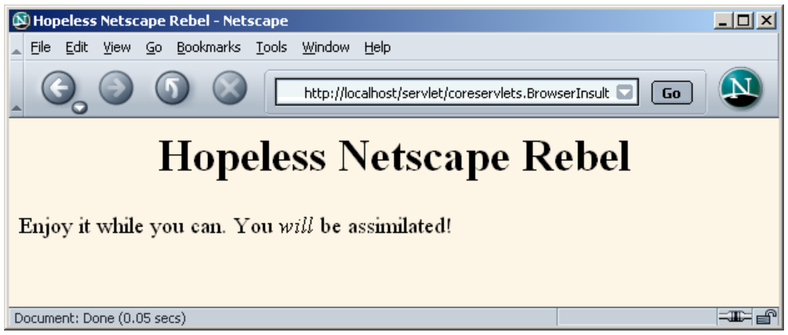
The `User-Agent` header identifies the specific browser that is making the request. Although use of this header appears straightforward at first glance, a few subtleties are involved:

- **Use `User-Agent` only when necessary.** Otherwise, you will have difficult-to-maintain code that consists of tables of browser versions and associated capabilities. For example, instead of remembering that the Windows version of Internet Explorer 5 supports gzip compression but the MacOS version doesn't, check the `Accept-Encoding` header. Instead of remembering which browsers support Java and which don't, use the `APPLET` tag with fallback code between `<APPLET>` and `</APPLET>`.
- **Check for `null`.** Sure, all major browser versions send the `User-Agent` header. But, the header is not *required* by the HTTP 1.1 specification, some browsers let you disable it (e.g., Opera), and custom clients (e.g., Web spiders or link verifiers) might not use the header at all. In fact, you should *always* check that the result of `request.getHeader` is non-`null` before trying to use it, regardless of which header you are dealing with.
- **To differentiate between Netscape and Internet Explorer, check for "MSIE," not "Mozilla."** Both Netscape and Internet Explorer say "Mozilla" at the beginning of the header, even though Mozilla is the Godzilla-like Netscape mascot. This characteristic is for compatibility with JavaScript.
- **Note that the header can be faked.** Some browsers let the user change the value of this header. Even if the browser didn't allow this, the user could always use a custom client. If a client fakes this header, the servlet cannot tell the difference.

Listing 5.4 shows a servlet that sends browser-specific insults to users. For the sake of simplicity, it assumes that Internet Explorer and Netscape are the only two browsers being used. Specifically, it assumes that any browser whose `User-Agent` contains "MSIE" is Internet Explorer and any whose `User-Agent` does not is Netscape. Figures 5–4 and 5–5 show the results.

Listing 5.4 BrowserInsult.java

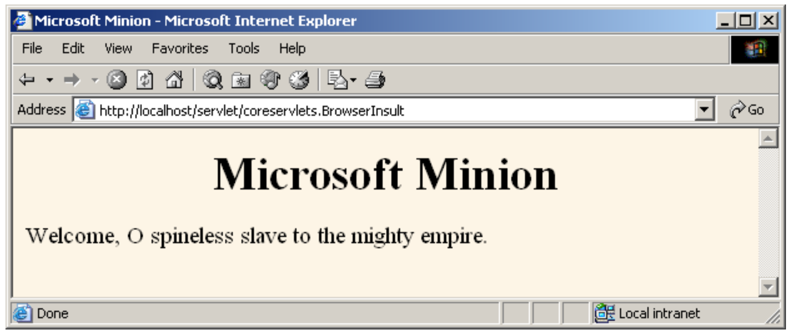
```
package coreservlets;

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

/** Servlet that gives browser-specific insult.
 *  Illustrates how to use the User-Agent
 *  header to tell browsers apart.
 */

public class BrowserInsult extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title, message;
    // Assume for simplicity that Netscape and IE are
    // the only two browsers.
    String userAgent = request.getHeader("User-Agent");
    if ((userAgent != null) &&
        (userAgent.indexOf("MSIE") != -1)) {
      title = "Microsoft Minion";
      message = "Welcome, O spineless slave to the " +
                "mighty empire.";
    } else {
      title = "Hopeless Netscape Rebel";
      message = "Enjoy it while you can. " +
                "You <I>will</I> be assimilated!";
    }
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                message + "\n" +
                "</BODY></HTML>");
  }
}
```


Hopeless Netscape Rebel - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.BrowserInsult
Go
Document: Done (0.05 secs)

Hopeless Netscape Rebel

Enjoy it while you can. You *will* be assimilated!

Figure 5–4 The `BrowserInsult` servlet as viewed by a Netscape user.


Microsoft Minion - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.BrowserInsult
Go
Done
Local intranet

Microsoft Minion

Welcome, O spineless slave to the mighty empire.

Figure 5–5 The `BrowserInsult` servlet as viewed by an Internet Explorer user.

5.6 Changing the Page According to How the User Got There

The `Referer` header designates the location of the page users were on when they clicked a link to get to the current page. If users simply type the address of a page, the browser sends no `Referer` at all and `request.getHeader("Referer")` returns `null`.

This header enables you to customize the page depending on how the user reached it. For example, you could use this header to do the following:

- Create a jobs/careers site that takes on the look and feel of the associated site that links to it.
- Change the content of a page depending on whether the link came from inside or outside the firewall. (Do not use this trick for secure applications, however; the `Referer` header, like all headers, is easily forged.)

- Supply links that take users back to the page they came from.
- Track the effectiveness of banner ads or record click-through rates from various different sites that display your ads.

Listing 5.5 shows a servlet that uses the `Referer` header to customize the image it displays. If the address of the referring page contains the string "JRun," the servlet displays the logo of Macromedia JRun. If the address contains the string "Resin," the servlet displays the logo of Caucho Resin. Otherwise, the servlet displays the logo of Apache Tomcat. The servlet also displays the address of the referring page.

Listing 5.6 shows the HTML pages used to link to the servlet. We created three *identical* pages named JRun-Referer.html, Resin-Referer.html, and Tomcat-Referer.html; the servlet uses the name of the referring page, not form data, to distinguish among the three. Recall that HTML pages are placed in the top-level directory of your Web application (or an arbitrary subdirectory thereof), whereas servlet code is placed in a subdirectory of WEB-INF/classes that matches the package name. So, for example, with Tomcat and the default Web application, the HTML pages are placed in *install_dir*/webapps/ROOT/request-headers/ and accessed with URLs of the form http://*hostname*/request-headers/*Xxx*-Referer.html.

Figures 5–6 through 5–9 show some representative results.

Listing 5.5 CustomizeImage.java

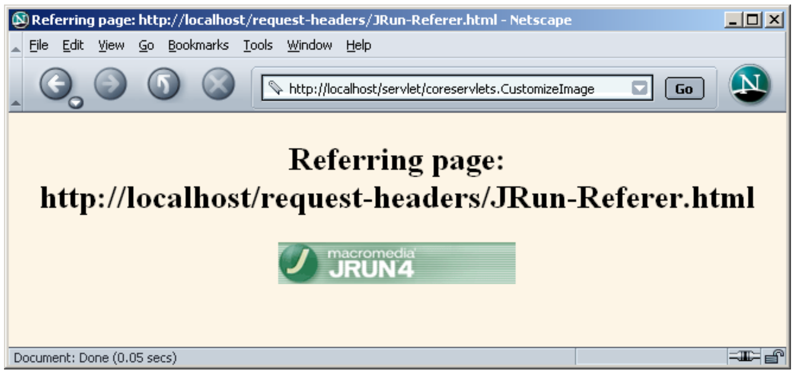
```
package coreservlets;

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

/** Servlet that displays referer-specific image. */

public class CustomizeImage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String referer = request.getHeader("Referer");
    if (referer == null) {
      referer = "<I>none</I>";
    }
    String title = "Referring page: " + referer;
    String imageName;
    if (contains(referer, "JRun")) {
      imageName = "jrun-powered.gif";
```

Listing 5.5 CustomizeImage.java *(continued)*

```
    } else if (contains(referer, "Resin")) {
      imageName = "resin-powered.gif";
    } else {
      imageName = "tomcat-powered.gif";
    }
    String imagePath = "../request-headers/images/" + imageName;
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER><H2>" + title + "</H2>\n" +
                "<IMG SRC=\"" + imagePath + "\">\n" +
                "</CENTER></BODY></HTML>");
  }

  private boolean contains(String mainString,
                           String subString) {
    return(mainString.indexOf(subString) != -1);
  }
}
```

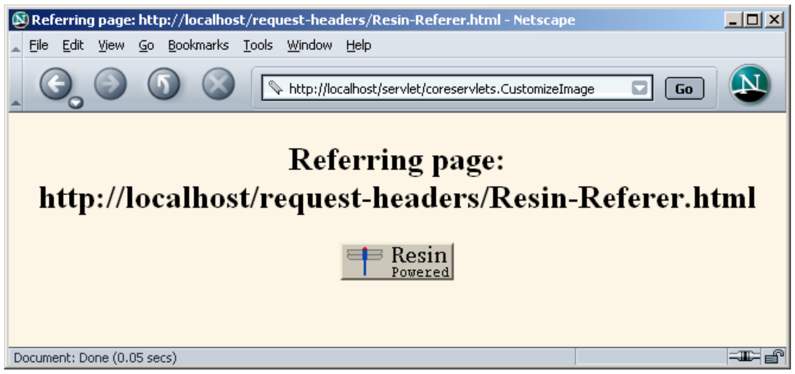
Listing 5.6 JRun-Referer.html (identical to Tomcat-Referer.html and Resin-Referer.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Referer Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Referer Test</H1>
Click <A HREF="/servlet/coreservlets.CustomizeImage">here</A>
to visit the servlet.
</BODY></HTML>
```


Referring page: http://localhost/request-headers/JRun-Referer.html - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.CustomizeImage
Go
macromedia JRUN4
Document: Done (0.05 secs)

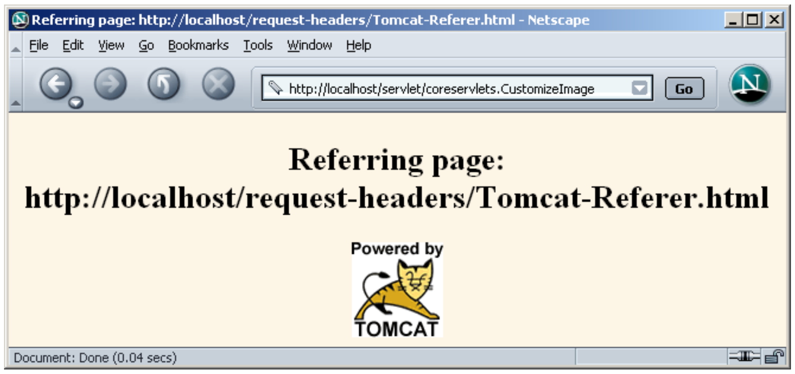
Referring page:
http://localhost/request-headers/JRun-Referer.html

Figure 5–6 The `CustomizeImage` servlet when the address of the referring page contains the string "JRun."


Referring page: http://localhost/request-headers/Resin-Referer.html - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.CustomizeImage
Go
Resin Powered
Document: Done (0.05 secs)

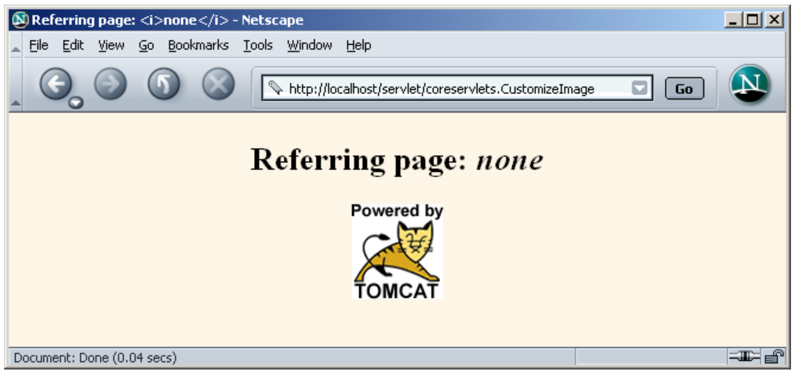
Referring page:
http://localhost/request-headers/Resin-Referer.html

Figure 5–7 The `CustomizeImage` servlet when the address of the referring page contains the string "Resin."


Referring page: http://localhost/request-headers/Tomcat-Referer.html - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.CustomizeImage
Go
Powered by
TOMCAT
Document: Done (0.04 secs)

Referring page:
http://localhost/request-headers/Tomcat-Referer.html

Figure 5–8 The `CustomizeImage` servlet when the address of the referring page contains neither "JRun" nor "Resin."


Referring page: <i>none</i> - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.CustomizeImage
Go

Referring page: *none*

Powered by
TOMCAT
Document: Done (0.04 secs)

Figure 5–9 The `CustomizeImage` servlet when the `Referer` header is missing. When using the `Referer` header, always handle the case in which the result of `getHeader` is `null`.

5.7 Accessing the Standard CGI Variables

If you come to servlets with a background in traditional Common Gateway Interface (CGI) programming, you are probably used to the idea of "CGI variables." These are a somewhat eclectic collection of information about the current request. Some are based on the HTTP request line and headers (e.g., form data), others are derived from the socket itself (e.g., the name and IP address of the requesting host), and still others are taken from server installation parameters (e.g., the mapping of URLs to actual paths).

Although it probably makes more sense to think of different sources of data (request data, server information, etc.) as distinct, experienced CGI programmers may find it useful to see the servlet equivalent of each of the CGI variables. If you don't have a background in traditional CGI, first, count your blessings; servlets are easier to use, more flexible, and more efficient than standard CGI. Second, just skim this section, noting the parts not directly related to the incoming HTTP request. In particular, observe that you can use `getServletContext().getRealPath` to map a URI (the part of the URL that comes after the host and port) to an actual path and that you can use `request.getRemoteHost` and `request.getRemoteAddress` to get the name and IP address of the client.

Servlet Equivalent of CGI Variables

For each standard CGI variable, this subsection summarizes its purpose and the means of accessing it from a servlet. Assume `request` is the `HttpServletRequest` supplied to the `doGet` and `doPost` methods.

AUTH_TYPE

If an `Authorization` header was supplied, this variable gives the scheme specified (`basic` or `digest`). Access it with `request.getAuthType()`.

CONTENT_LENGTH

For `POST` requests only, this variable stores the number of bytes of data sent, as given by the `Content-Length` request header. Technically, since the `CONTENT_LENGTH` CGI variable is a string, the servlet equivalent is `String.valueOf(request.getContentLength())` or `request.getHeader("Content-Length")`. You'll probably want to just call `request.getContentLength()`, which returns an `int`.

CONTENT_TYPE

`CONTENT_TYPE` designates the MIME type of attached data, if specified. See Table 7.1 in Section 7.2 (Understanding HTTP 1.1 Response Headers) for the names and meanings of the common MIME types. Access `CONTENT_TYPE` with `request.getContentType()`.

DOCUMENT_ROOT

The `DOCUMENT_ROOT` variable specifies the real directory corresponding to the URL `http://host/`. Access it with `getServletContext().getRealPath("/")`. Also, you can use `getServletContext().getRealPath` to map an arbitrary URI (i.e., URL suffix that comes after the hostname and port) to an actual path on the local machine.

HTTP_XXX_YYY

Variables of the form `HTTP_HEADER_NAME` are how CGI programs access arbitrary HTTP request headers. The `Cookie` header becomes `HTTP_COOKIE`, `User-Agent` becomes `HTTP_USER_AGENT`, `Referer` becomes `HTTP_REFERER`, and so forth. Servlets should just use `request.getHeader` or one of the shortcut methods described in Section 5.1 (Reading Request Headers).

PATH_INFO
This variable supplies any path information attached to the URL after the address of the servlet but before the query data. For example, with http://host/servlet/coreservlets.SomeServlet/foo/bar?baz=quux, the path information is `/foo/bar`. Since servlets, unlike standard CGI programs, can talk directly to the server, they don't need to treat path information specially. Path information could be sent as part of the regular form data and then translated by `getServletContext().getRealPath`. Access the value of `PATH_INFO` by using `request.getPathInfo()`.

PATH_TRANSLATED
`PATH_TRANSLATED` gives the path information mapped to a real path on the server. Again, with servlets there is no need to have a special case for path information, since a servlet can call `getServletContext().getRealPath` to translate partial URLs into real paths. This translation is not possible with standard CGI because the CGI program runs entirely separately from the server. Access this variable by means of `request.getPathTranslated()`.

QUERY_STRING
For `GET` requests, this variable gives the attached data as a single string with values still URL-encoded. You rarely want the raw data in servlets; instead, use `request.getParameter` to access individual parameters, as described in Section 5.1 (Reading Request Headers). However, if you do want the raw data, you can get it with `request.getQueryString()`.

REMOTE_ADDR
This variable designates the IP address of the client that made the request, as a `String` (e.g., `"198.137.241.30"`). Access it by calling `request.getRemoteAddr()`.

REMOTE_HOST
`REMOTE_HOST` indicates the fully qualified domain name (e.g., `whitehouse.gov`) of the client that made the request. The IP address is returned if the domain name cannot be determined. You can access this variable with `request.getRemoteHost()`.

REMOTE_USER
If an `Authorization` header was supplied and decoded by the server itself, the `REMOTE_USER` variable gives the user part, which is useful for session tracking in protected sites. Access it with `request.getRemoteUser()`. For decoding `Authorization` information directly in servlets, see the chapters on Web application security in Volume 2 of this book.

REQUEST_METHOD

This variable stipulates the HTTP request type, which is usually `GET` or `POST` but is occasionally `HEAD`, `PUT`, `DELETE`, `OPTIONS`, or `TRACE`. Servlets rarely need to look up `REQUEST_METHOD` explicitly, since each of the request types is typically handled by a different servlet method (`doGet`, `doPost`, etc.). An exception is `HEAD`, which is handled automatically by the `service` method returning whatever headers and status codes the `doGet` method would use. Access this variable by means of `request.getMethod()`.

SCRIPT_NAME

This variable specifies the path to the servlet, relative to the server's root directory. It can be accessed through `request.getServletPath()`.

SERVER_NAME

`SERVER_NAME` gives the host name of the server machine. It can be accessed by means of `request.getServerName()`.

SERVER_PORT

This variable stores the port the server is listening on. Technically, the servlet equivalent is `String.valueOf(request.getServerPort())`, which returns a `String`. You'll usually just want `request.getServerPort()`, which returns an `int`.

SERVER_PROTOCOL

The `SERVER_PROTOCOL` variable indicates the protocol name and version used in the request line (e.g., `HTTP/1.0` or `HTTP/1.1`). Access it by calling `request.getProtocol()`.

SERVER_SOFTWARE

This variable gives identifying information about the Web server. Access it by means of `getServletContext().getServerInfo()`.

A Servlet That Shows the CGI Variables

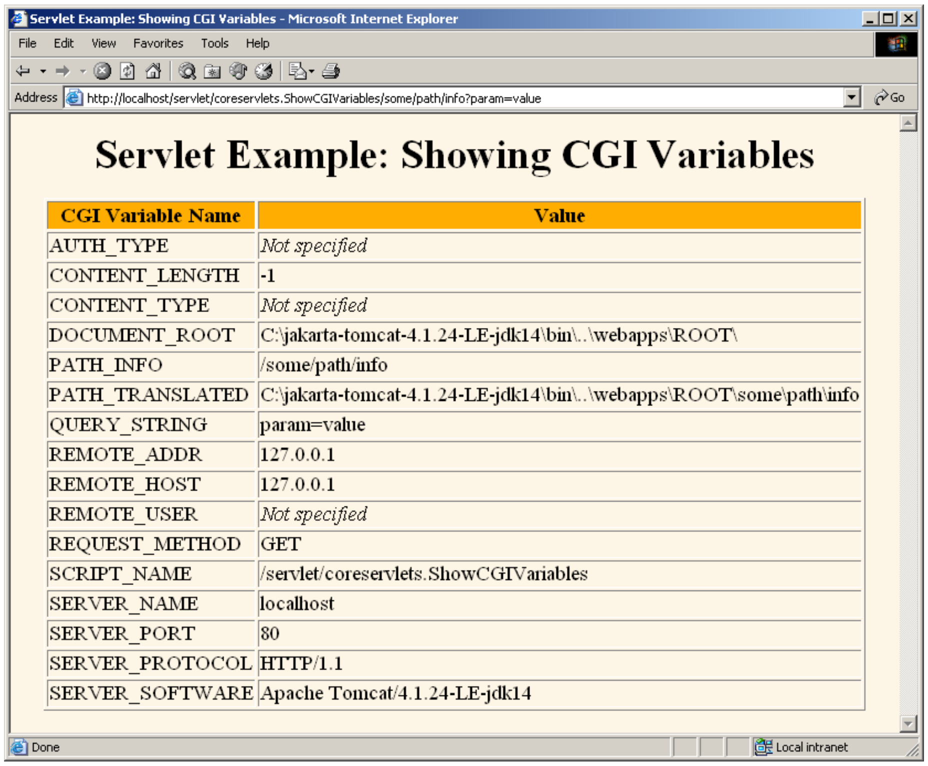
Listing 5.7 presents a servlet that creates a table showing the values of all the CGI variables other than `HTTP_XXX_YYY`, which are just the HTTP request headers described in Section 5.3. Figure 5–10 shows the result for a typical request.

Listing 5.7 ShowCGIVariables.java

```
package coreservlets;

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

/** Creates a table showing the current value of each
 *  of the standard CGI variables.
 */

public class ShowCGIVariables extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String[][] variables =
      { { "AUTH_TYPE", request.getAuthType() },
        { "CONTENT_LENGTH",
          String.valueOf(request.getContentLength()) },
        { "CONTENT_TYPE", request.getContentType() },
        { "DOCUMENT_ROOT",
          getServletContext().getRealPath("/") },
        { "PATH_INFO", request.getPathInfo() },
        { "PATH_TRANSLATED", request.getPathTranslated() },
        { "QUERY_STRING", request.getQueryString() },
        { "REMOTE_ADDR", request.getRemoteAddr() },
        { "REMOTE_HOST", request.getRemoteHost() },
        { "REMOTE_USER", request.getRemoteUser() },
        { "REQUEST_METHOD", request.getMethod() },
        { "SCRIPT_NAME", request.getServletPath() },
        { "SERVER_NAME", request.getServerName() },
        { "SERVER_PORT",
          String.valueOf(request.getServerPort()) },
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE",
          getServletContext().getServerInfo() }
      };
    String title = "Servlet Example: Showing CGI Variables";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
```

Listing 5.7 ShowCGIVariables.java *(continued)*

```
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + title + "</H1>\n" +
                "<TABLE BORDER=1>\n" +
                "  <TR BGCOLOR=\"#FFAD00\">\n" +
                "    <TH>CGI Variable Name<TH>Value");
    for(int i=0; i<variables.length; i++) {
      String varName = variables[i][0];
      String varValue = variables[i][1];
      if (varValue == null)
        varValue = "<I>Not specified</I>";
      out.println("  <TR><TD>" + varName + "<TD>" + varValue);
    }
    out.println("</TABLE></CENTER></BODY></HTML>");
  }

  /** POST and GET requests handled identically. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```


Servlet Example: Showing CGI Variables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.ShowCGIVariables/some/path/info?param=value
Go
Done
Local intranet

Servlet Example: Showing CGI Variables

CGI Variable Name	Value
AUTH_TYPE	*Not specified*
CONTENT_LENGTH	-1
CONTENT_TYPE	*Not specified*
DOCUMENT_ROOT	C:\jakarta-tomcat-4.1.24-LE-jdk14\bin\..\webapps\ROOT\
PATH_INFO	/some/path/info
PATH_TRANSLATED	C:\jakarta-tomcat-4.1.24-LE-jdk14\bin\..\webapps\ROOT\some\path\info
QUERY_STRING	param=value
REMOTE_ADDR	127.0.0.1
REMOTE_HOST	127.0.0.1
REMOTE_USER	*Not specified*
REQUEST_METHOD	GET
SCRIPT_NAME	/servlet/coreservlets.ShowCGIVariables
SERVER_NAME	localhost
SERVER_PORT	80
SERVER_PROTOCOL	HTTP/1.1
SERVER_SOFTWARE	Apache Tomcat/4.1.24-LE-jdk14

Figure 5–10 The standard CGI variables for a typical request.