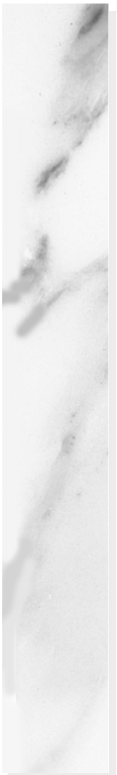
Handling the Client Request: HTTP Request Headers



Topics in This Chapter

- Reading HTTP request headers from servlets
- Building a table of all the request headers
- The purpose of each of the HTTP 1.1 request headers
- Reducing download times by compressing pages
- Restricting access with password-protected servlets

Chapter 4

One of the keys to creating effective servlets is understanding how to manipulate the HyperText Transfer Protocol (HTTP). Getting a thorough grasp of this protocol is not an esoteric, theoretical topic, but rather a practical issue that can have an immediate impact on the performance and usability of your servlets. This chapter discusses the HTTP information that is sent from the browser to the server in the form of request headers. It explains each of the HTTP 1.1 request headers, summarizing how and why they would be used in a servlet. The chapter also includes three detailed examples: listing all request headers sent by the browser, reducing download time by encoding the Web page with gzip when appropriate, and establishing password-based access control for servlets.

Note that HTTP request headers are distinct from the form 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 submitting a book-search request to a servlet at `http://www.somebookstore.com/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 /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/4.7 [en] (Win98; U)
```

4.1 Reading Request Headers from Servlets

Reading headers is straightforward; just call the `getHeader` method of `HttpServletRequest`, which returns a `String` if the specified header was supplied on this request, `null` otherwise. Header names are not case sensitive. So, for example, `request.getHeader("Connection")` and `request.getHeader("connection")` are interchangeable.

Although `getHeader` is the general-purpose way to read incoming headers, there are a couple of headers that are so commonly used that they have special access methods in `HttpServletRequest`. I'll list them here, and remember that Appendix A (Servlet and JSP Quick Reference) gives a separate syntax 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 more in Chapter 8 (Handling Cookies).
- **`getAuthType` and `getRemoteUser`**
  The `getAuthType` and `getRemoteUser` methods break the `Authorization` header into its component pieces. Use of the `Authorization` header is illustrated in Section 4.5 (Restricting Access to Web Pages).
- **`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 header 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 4.2 (Printing All 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. If a header name is repeated in the request, version 2.1 servlets cannot access the later values without reading the raw input stream, since `getHeader` returns the value of the first occurrence of the header only. In version 2.2, however, `getHeaders` returns 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, also by means of methods in `HttpServletRequest`.

- **`getMethod`**
  The `getMethod` method returns the main request method (normally `GET` or `POST`, but things 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`, `getRequestURI` would return `/servlet/search.BookSearch`.
- **`getProtocol`**
  Lastly, 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.

4.2 Printing All Headers

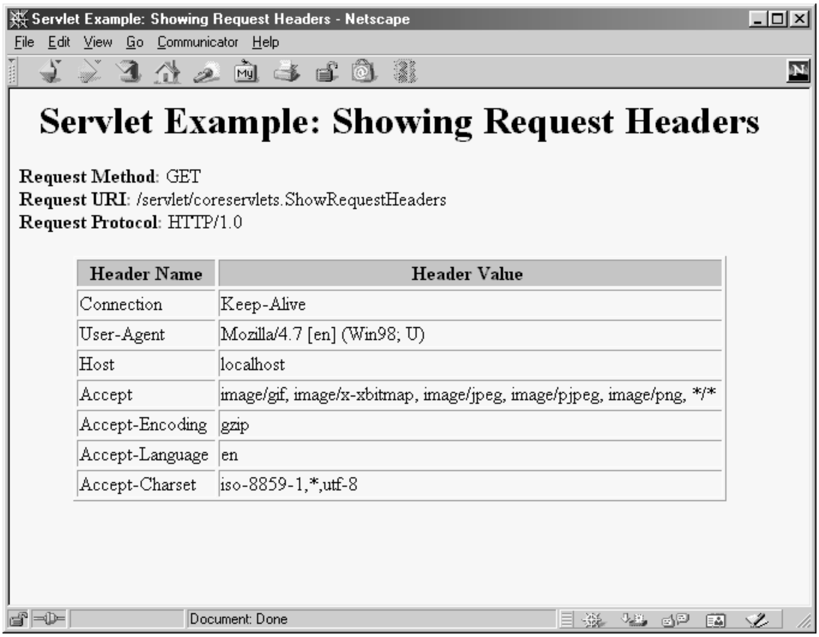
Listing 4.1 shows a servlet that simply creates a table of all the headers it receives, along with their associated values. It also prints out the three components of the main request line (method, URI, and protocol). Figures 4–1 and 4–2 show typical results with Netscape and Internet Explorer.

Listing 4.1 ShowRequestHeaders.java

```
package coreservlets;

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

/** Shows all the request headers sent on this
 *  particular 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";
    out.println(ServletUtilities.headWithTitle(title) +
                "<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>" +
                request.getProtocol() + "<BR><BR>\n" +
```

Listing 4.1 ShowRequestHeaders.java (continued)

```
                "<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>");
  }

  /** Let the same servlet handle both GET and POST. */

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

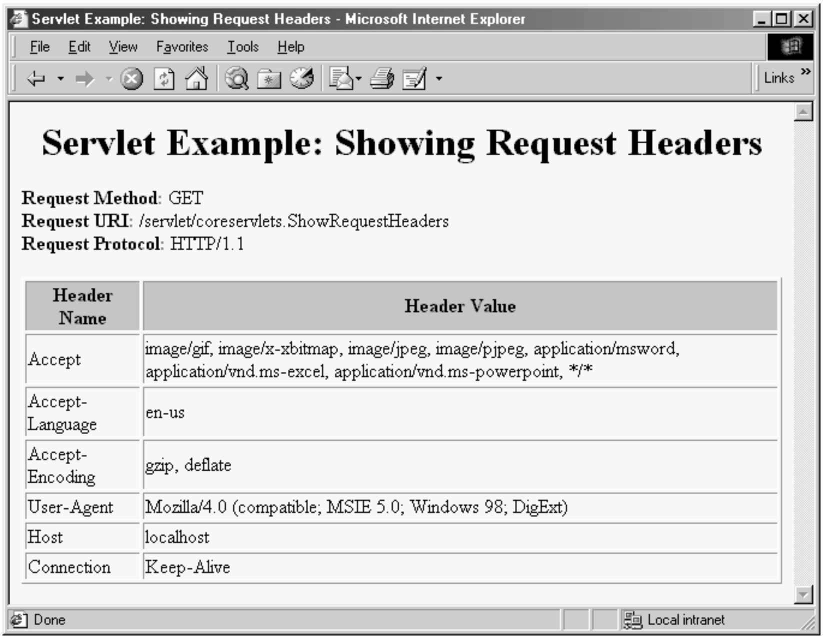

Servlet Example: Showing Request Headers - Netscape
File Edit View Go Communicator Help
Document: Done

Servlet Example: Showing Request Headers

Request Method: GET
Request URI: /servlet/coreservlets.ShowRequestHeaders
Request Protocol: HTTP/1.0

Header Name	Header Value
Connection	Keep-Alive
User-Agent	Mozilla/4.7 [en] (Win98; U)
Host	localhost
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding	gzip
Accept-Language	en
Accept-Charset	iso-8859-1,*,utf-8

Figure 4–1 Request headers sent by Netscape 4.7 on Windows 98.


Servlet Example: Showing Request Headers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links »
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/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint, */*
Accept-Language	en-us
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
Host	localhost
Connection	Keep-Alive

Figure 4–2 Request headers sent by Internet Explorer 5 on Windows 98.

4.3 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 presents each of the possible HTTP 1.1 request headers along with a brief summary of how servlets can make use of them. The following sections give more detailed examples.

Note that HTTP 1.1 supports a superset of the headers permitted in HTTP 1.0. For additional details on these headers, see the HTTP 1.1 specification, given in RFC 2616. There are a number of places the official RFCs are archived on-line; your best bet is to start at `http://www.rfc-editor.org/` to get a current list of the archive sites.

Accept

This header specifies the MIME types that the browser or other client 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 only a few browsers support PNG. If you had images in both formats, a servlet could call `request.getHeader("Accept")`, check for `image/png`, and if it finds it, use *`xxx`*`.png` filenames in all the `IMG` elements it generates. Otherwise it would just use *`xxx`*`.gif`.

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

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 it receives this header, the server is free to encode the page by using the format 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 totally 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 standard possibilities.

Compressing pages before returning them is a very valuable service because the decoding time is likely to be small compared to the savings in transmission time. See Section 4.4 (Sending Compressed Web Pages) for an example where compression reduces download times by a factor of 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.

Authorization
This header is used by clients to identify themselves when accessing password-protected Web pages. See Section 4.5 (Restricting Access to Web Pages) for an example.

Cache-Control
This header can be used by the client to specify a number of options for how pages should be cached by proxy servers. The request header is usually ignored by servlets, but the `Cache-Control` *response* header can be valuable to indicate that a page is constantly changing and shouldn't be cached. See Chapter 7 (Generating the Server Response: HTTP Response Headers) for details.

Connection
This header tells whether or not the client can handle persistent HTTP connections. These let the client or other browser retrieve multiple files (e.g., an HTML file and several associated images) with a single socket connection, 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, 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 help from the server to handle persistent connections. Consequently, the servlet's job is just to make it *possible* for the server to use persistent connections, which is done by sending a `Content-Length` response header. Section 7.4 (Using Persistent HTTP Connections) has a detailed example.

Content-Length
This header is only applicable 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 3, "Handling the Client Request: Form Data"), you are unlikely to use this header explicitly.

Content-Type

Although this header is usually used in responses *from* the server, it can also be part of client requests when the client attaches a document as the `POST` data or when making `PUT` requests. You can access this header with the shorthand `getContentType` method of `HttpServletRequest`.

Cookie

This header is used to return cookies to servers that previously sent them to the browser. 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 very widely supported, including in both Netscape and Internet Explorer.

Expect

This rarely used header lets the client tell the server what kinds of behaviors it expects. The one standard value for this header, `100-continue`, is sent by a browser that will be sending an attached document and wants to know if the server will accept it. The server should send a status code of either 100 (`Continue`) or 417 (`Expectation Failed`) in such a case. For more details on HTTP status codes, see Chapter 6 (Generating the Server Response: HTTP Status Codes).

From

This header gives the e-mail address of the person responsible for the HTTP request. Browsers do not send this header, but Web spiders (robots) often set it as a courtesy to help identify the source of server overloading or repeated improper requests.

Host

Browsers are required to specify this header, which indicates the host and port as given in the *original* URL. Due to request forwarding and machines that have multiple hostnames, 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-Match

This rarely used header applies primarily to `PUT` requests. The client can supply a list of entity tags as returned by the `ETag` response header, and the operation is performed only if one of them matches.

If-Modified-Since

This header indicates that the client wants the page only if it has been changed after the specified date. This option is very 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. An illustration is given in Section 2.8 (An Example Using Servlet Initialization and Page Modification Dates).

If-None-Match

This header is like `If-Match`, except that the operation should be performed only if *no* entity tags match.

If-Range

This rarely used header lets a client that has a partial copy of a document ask for either the parts it is missing (if unchanged) or an entire new document (if it has changed since a specified date).

If-Unmodified-Since

This header is like `If-Modified-Since` in reverse, indicating 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").

Pragma

A `Pragma` header with a value of `no-cache` indicates that a servlet that is acting as a proxy should forward the request even if it has a local copy. The *only* standard value for this header is `no-cache`.

Proxy-Authorization

This header lets clients identify themselves to proxies that require it. Servlets typically ignore this header, using `Authorization` instead.

Range

This rarely used header lets a client that has a partial copy of a document ask for only the parts it is missing.

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. All major browsers set this header, so it is a useful way of tracking where requests came from. This capability is helpful for tracking advertisers who refer people to your site, for changing content slightly depending on the referring site, or simply for keeping track of where your traffic comes from. In the last case, most people simply rely on Web server log files, since the `Referer` is typically recorded there. Although it's useful, don't rely too heavily on the `Referer` header since it can be easily spoofed by a custom client. Finally, note that this header is `Referer`, not the expected `Referrer`, due to a spelling mistake by one of the original HTTP authors.

Upgrade

The `Upgrade` header lets the browser or other client specify a communication protocol it prefers over HTTP 1.1. If the server also supports that protocol, both the client and the server can switch protocols. This type of protocol negotiation is almost always performed before the servlet is invoked. Thus, servlets rarely care about this header.

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 usage, however; 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. Admittedly, this is not always possible, but when it is not, you should ask yourself if the browser-specific feature you are using really adds enough value to be worth the maintenance cost.

Most Internet Explorer versions list a "Mozilla" (Netscape) version first in their `User-Agent` line, with the real browser version listed parenthetically. This is done for compatibility with JavaScript, where the `User-Agent` header is sometimes used to determine which JavaScript features are supported. 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. Hmm, millions of dollars in marketing money riding on statistics that could be skewed by a custom client written in less than an hour, and I should take those numbers as accurate ones?

Via
This header is set by gateways and proxies to show the intermediate sites the request passed through.

Warning
This rarely used catchall header lets clients warn about caching or content transformation errors.

4.4 Sending Compressed Web Pages

Several recent browsers know how to handle gzipped content, automatically uncompressing documents that are marked with the `Content-Encoding` header and then treating the result as though it were the original document. Sending such compressed content can be a real timesaver, since the time required to compress the document on the server and then uncompress it on the client is typically dwarfed by the savings in download time, especially when dialup connections are used.

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

I CALCULATED THE TOTAL TIME THAT HUMANS HAVE WAITED FOR WEB PAGES TO LOAD...
E-mail: SCOTTADAMS@AOL.COM
S. Adams
IT CANCELS OUT ALL THE PRODUCTIVITY GAINS OF THE INFORMATION AGE.
© 1997 United Feature Syndicate, Inc.
5/5/97
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.

DILBERT reprinted by permission of United Syndicate, Inc.

Explorer on non-Windows platforms generally do not support content encoding. Fortunately, browsers that support this feature indicate that they do so by setting the `Accept-Encoding` request header. Listing 4.2 shows a servlet that checks this header, sending a compressed Web page to clients that support gzip encoding and sending a regular Web page to those that don't. The result showed a *tenfold* speedup for the compressed page when a dialup connection was used. In repeated tests with Netscape 4.7 and Internet Explorer 5.0 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.

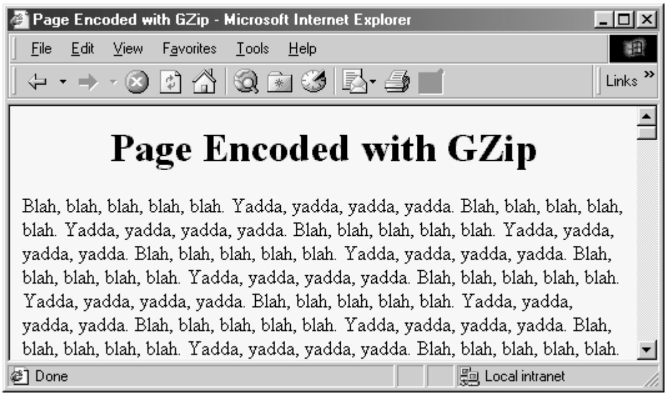
Core Tip

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

Implementing compression is straightforward since gzip format is built in to the Java programming languages via 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 `GZIPOutputStream` to generate the page, specifying `gzip` as the value of the `Content-Encoding` header. You must explicitly call `close` when using a `GZIPOutputStream`. If gzip is not supported, the servlet uses the normal `PrintWriter` to send the page. To make it easy to create benchmarks with a single browser, I also added a feature whereby compression could be suppressed by including `?encoding=none` at the end of the URL.

Listing 4.2 EncodedPage.java

```
package coreservlets;

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

/** Example showing benefits of gzipping pages to browsers
 *  that can handle gzip.
 */

public class EncodedPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    String encodings = request.getHeader("Accept-Encoding");
    String encodeFlag = request.getParameter("encoding");

    PrintWriter out;
    String title;
    if ((encodings != null) &&
        (encodings.indexOf("gzip") != -1) &&
        !"none".equals(encodeFlag)) {
      title = "Page Encoded with GZip";
      OutputStream out1 = response.getOutputStream();
      out = new PrintWriter(new GZIPOutputStream(out1), false);
      response.setHeader("Content-Encoding", "gzip");
    } else {
      title = "Unencoded Page";
      out = response.getWriter();
    }
    out.println(ServletUtilities.headWithTitle(title) +
                "<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();
  }
}
```


Page Encoded with GZip - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links »
Done
Local intranet

Page Encoded with GZip

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

Figure 4–3 Since the Windows version of Internet Explorer 5.0 supports gzip, this page was sent gzipped over the network and reconstituted by the browser, resulting in a large saving in download time.

4.5 Restricting Access to Web Pages

Many Web servers support standard mechanisms for limiting access to designated Web pages. These mechanisms can apply to static pages as well as those generated by servlets, so many authors use their server-specific mechanisms for restricting access to servlets. Furthermore, most users at e-commerce sites prefer to use regular HTML forms to provide authorization information since these forms are more familiar, can provide more explanatory information, and can ask for additional information beyond just a username and password. Once a servlet that uses form-based access grants initial access to a user, it would use session tracking to give the user access to other pages that require the same level of authorization. See Chapter 9 (Session Tracking) for more information.

Nevertheless, form-based access control requires more effort on the part of the servlet developer, and HTTP-based authorization is sufficient for many simple applications. Here’s a summary of the steps involved for “basic” authorization. There is also a slightly better variation called “digest” authorization, but among the major browsers, only Internet Explorer supports it.

1. Check whether there is an `Authorization` header. If there is no such header, go to Step 2. If there is, skip over the word “basic” and reverse the base64 encoding of the remaining part. This results in a string of the form `username:password`. Check the username and password against some stored set. If it matches, return the page. If not, go to Step 2.
2. Return a 401 (`Unauthorized`) response code and a header of the following form:
   `WWW-Authenticate: BASIC realm="some-name"`
   This response instructs the browser to pop up a dialog box telling the user to enter a name and password for `some-name`, then to reconnect with that username and password embedded in a single base64 string inside the `Authorization` header.

If you care about the details, base64 encoding is explained in RFC 1521 (remember, to retrieve RFCs, start at `http://www.rfc-editor.org/` to get a current list of the RFC archive sites). However, there are probably only two things you need to know about it. First, it is not intended to provide security, as the encoding can be easily reversed. So, it does not obviate the need for SSL to thwart attackers who might be able to snoop on your network connection (no easy task unless they are on your local subnet). SSL, or Secure Sockets Layer, is a variation of HTTP where the entire stream is encrypted. It is supported by many commercial servers and is generally invoked by using `https` in the URL instead of `http`. Servlets can run on SSL servers just as easily as on standard servers, and the encryption and decryption is handled transparently before the servlets are invoked. The second point you should know about base64 encoding is that Sun provides the `sun.misc.BASE64Decoder` class, distributed with both JDK 1.1 and 1.2, to decode strings that were encoded with base64. Just be aware that classes in the `sun` package hierarchy are not part of the official language specification, and thus are not guaranteed to appear in all implementations. So, if you use this decoder class, make sure that you explicitly include the class file when you distribute your application.

Listing 4.3 presents a password-protected servlet. It is explicitly registered with the Web server under the name `SecretServlet`. The process for registering servlets varies from server to server, but Section 2.7 (An Example Using Initialization Parameters) gives details on the process for Tomcat, the JSWDK and the Java Web Server. The reason the servlet is registered is so that initialization parameters can be associated with it, since most servers don't let you set initialization parameters for servlets that are available merely by virtue of being in the `servlets` (or equivalent) directory. The initialization parameter gives the location of a Java `Properties` file that stores user names and passwords. If the security of the page was very important, you'd want to encrypt the passwords so that access to the `Properties` file would not equate to knowledge of the passwords.

In addition to reading the incoming `Authorization` header, the servlet specifies a status code of 401 and sets the outgoing `WWW-Authenticate` header. Status codes are discussed in detail in Chapter 6 (Generating the Server Response: HTTP Status Codes), but for now, just note that they convey high-level information to the browser and generally need to be set whenever the response is something other than the document requested. The most common way to set status codes is through the use of the `setStatus` method of `HttpServletResponse`, and you typically supply a constant instead of an explicit integer in order to make your code clearer and to prevent typographic errors.

`WWW-Authenticate` and other HTTP response headers are discussed in Chapter 7 (Generating the Server Response: HTTP Response Headers), but for now note that they convey auxiliary information to support the response specified by the status code, and they are commonly set through use of the `setHeader` method of `HttpServletResponse`.

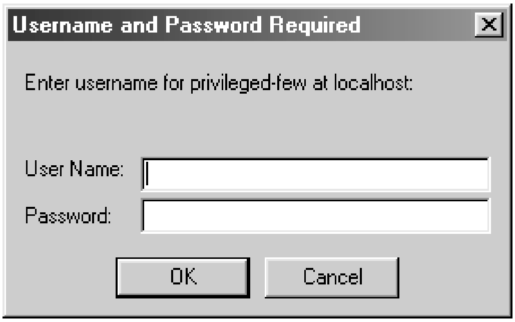
Figures 4–4, 4–5, and 4–6 show the result when a user first tries to access the page, after the user enters an unknown password, and after the user enters a known password. Listing 4.4 gives the program that built the simple password file.

Listing 4.3 ProtectedPage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Properties;
import sun.misc.BASE64Decoder;

/** Example of password-protected pages handled directly
 *  by servlets.
 */

public class ProtectedPage extends HttpServlet {
  private Properties passwords;
  private String passwordFile;

  /** Read the password file from the location specified
   *  by the passwordFile initialization parameter.
   */

  public void init(ServletConfig config)
      throws ServletException {
    super.init(config);
    try {
      passwordFile = config.getInitParameter("passwordFile");
      passwords = new Properties();
      passwords.load(new FileInputStream(passwordFile));
    } catch(IOException ioe) {}
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String authorization = request.getHeader("Authorization");
    if (authorization == null) {
      askForPassword(response);
    } else {
      String userInfo = authorization.substring(6).trim();
```

Listing 4.3 `ProtectedPage.java` **(continued)**

```
      BASE64Decoder decoder = new BASE64Decoder();
      String nameAndPassword =
        new String(decoder.decodeBuffer(userInfo));
      int index = nameAndPassword.indexOf(":");
      String user = nameAndPassword.substring(0, index);
      String password = nameAndPassword.substring(index+1);
      String realPassword = passwords.getProperty(user);
      if ((realPassword != null) &&
          (realPassword.equals(password))) {
        String title = "Welcome to the Protected Page";
        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "Congratulations. You have accessed a\n" +
                    "highly proprietary company document.\n" +
                    "Shred or eat all hardcopies before\n" +
                    "going to bed tonight.\n" +
                    "</BODY></HTML>");
      } else {
        askForPassword(response);
      }
    }
  }

  // If no Authorization header was supplied in the request.

  private void askForPassword(HttpServletResponse response) {
    response.setStatus(response.SC_UNAUTHORIZED); // Ie 401
    response.setHeader("WWW-Authenticate",
                       "BASIC realm=\"privileged-few\"");
  }

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



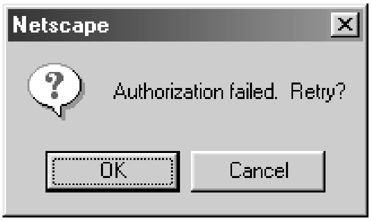
Username and Password Required

Enter username for privileged-few at localhost:

User Name:

Password:

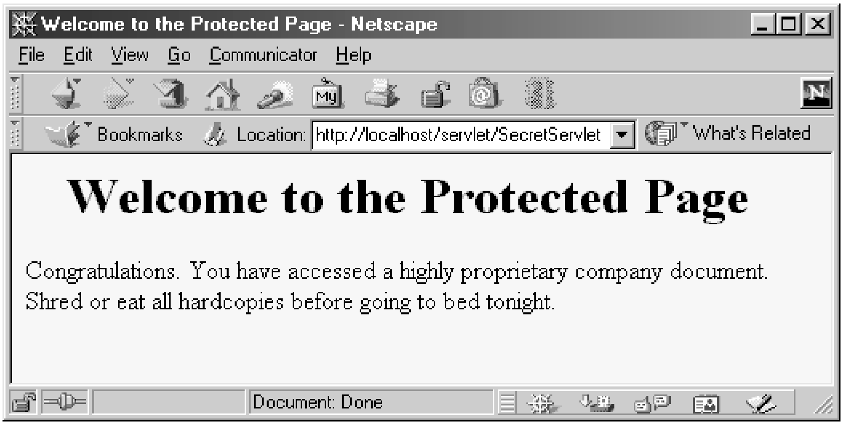
Figure 4–4 Initial result when accessing `SecretServlet` (the registered name for the `ProtectedPage` servlet).



Netscape

Authorization failed. Retry?

Figure 4–5 Result after entering incorrect name or password.



Welcome to the Protected Page - Netscape

Location: http://localhost/servlet/SecretServlet

Welcome to the Protected Page

Congratulations. You have accessed a highly proprietary company document. Shred or eat all hardcopies before going to bed tonight.

Document: Done

Figure 4–6 Result after entering known name and password.

Listing 4.4 `PasswordBuilder.java`

```
import java.util.*;
import java.io.*;

/** Application that writes a simple Java properties file
 *  containing usernames and associated passwords.
 */

public class PasswordBuilder {
  public static void main(String[] args) throws Exception {
    Properties passwords = new Properties();
    passwords.put("marty", "martypw");
    passwords.put("bj", "bjpw");
    passwords.put("lindsay", "lindsaypw");
    passwords.put("nathan", "nathanpw");
    // This location should *not* be Web-accessible.
    String passwordFile =
      "C:\\JavaWebServer2.0\\data\\passwords.properties";
    FileOutputStream out = new FileOutputStream(passwordFile);
    // Using JDK 1.1 for portability among all servlet
    // engines. In JDK 1.2, use "store" instead of "save"
    // to avoid deprecation warnings.
    passwords.save(out, "Passwords");
  }
}
```