Chapter 16

The HyperText Transfer Protocol (HTTP) is the language used by Web browsers to communicate with WWW servers. An understanding of this protocol is valuable to Web programmers for several reasons. First, CGI programs (Chapter 18) depend on information sent in HTTP requests to determine what action to take, and can manipulate HTTP responses to carry out these actions. Second, JavaScript programs (Chapter 19) can read and manipulate HTTP headers (including “cookies”) in order to customize the look of a Web page. Third, browsers are not the only type of HTTP client, and a knowledge of HTTP is critical to writing link checkers, off-line browsers, Web site indexers, WWW spiders, and the like. Finally, although a commercial-quality HTTP server is a significant undertaking, a special-purpose server (Section 15.7) is not necessarily difficult, and can be useful in many situations. Such a server must support the HTTP protocol correctly if it is to communicate with existing browsers.

As of 1997, most servers and browsers support HTTP 1.0, which is the version described in this chapter. Version 3 of Netscape Navigator and Internet Explorer both use HTTP 1.0. However, HTTP 1.1 is starting to gain support, and some of its key features are discussed here as well. The official protocol specifications can be found at the following sites:

HTTP 1.0
http://ds.internic.net/rfc/rfc1945.txt

HTTP 1.1
http://www.w3.org/pub/WWW/Protocols/
16.1 Communicating with an HTTP Server

HTTP is a pretty simple protocol. The client first opens a socket connection to an HTTP server, which by default is on port 80. It then issues a command (usually a request for a document) by sending a request line, some optional request headers, and a blank line. In some cases, data for server-side programs is included after the blank line. The HTTP server then sends a response line, some response headers, a blank line, and a document, then closes the connection. Each of these components is summarized as follows and covered in more detail in the following sections.

The Client Request

When contacting a server, an HTTP client first issues a request line, typically using the GET, HEAD, or POST methods. It then sends zero or more HTTP request header lines and a blank line. In the case of GET and HEAD, the server then returns the results. In the case of POST requests, the client can send additional data after the blank line. A request is in the following format:

Request-Method Document-Address HTTP/Version

The first thing sent by the client is the request line, which consists of a request method (e.g., GET), the address of a resource, and, in the case of HTTP 1.0, the string "HTTP/1.0". The request methods are discussed in Section 16.2, "The HTTP Request Line."

Request-Header I: Value I
... 
Request-HeaderN: ValueN

The next thing sent by the client is a series of zero or more HTTP request headers, each with an associated value. The request headers are discussed in Section 16.3.

Blank Line

Finally, the client ends the request by sending a blank line. In the case of some types of requests (e.g., POST), additional data can be sent after the blank line.

The Server Response

The response consists of a status line, one or more HTTP response header lines, a blank line, and then the requested document.

HTTP/Version Status-Code Message

The first thing returned by the server is a status line consisting (for HTTP version 1.0) of the string "HTTP/1.0", followed by a numeric status code and a short message describing the status code. The status codes are explained in Section 16.4, "The HTTP Response Status Line."

Response-Header I: Value I
... 
Response-HeaderN: ValueN

The second thing returned by the server is a series of one or more HTTP response headers, each followed by an associated value. Most of the headers are optional, but at the very least the server should send a Content-Type header identifying the type of data that follows. Section 16.5 describes the response headers.

Blank Line

After the final response header, a blank line is sent.

Response Document

Finally, most responses will end with a document in the format identified by the Content-Type header. However, some types of requests (e.g., HEAD) result in just the status line and the response headers. The response document is discussed in Section 16.7.

An Example Interaction

Figure 16-1 shows a typical interaction with the HTTP server at JavaSoft's main Web site. You can experiment with the request methods and headers and see responses from Web servers at various popular Web sites by opening an interactive socket connection using a telnet client. For example, on Unix, "telnet some.random.host.com 80" will connect to the HTTP port of some.random.host.com and let you type a request and see the response. A telnet client comes bundled with Unix (typically in /usr/bin/telnet) and Windows 95 (in C:\Windows) and NT systems, and is widely available separately for Macs. However, using telnet to talk to HTTP servers is inconvenient on Windows and Macintosh systems because many HTML docu-
ments end lines with a newline (linefeed) character. This causes the document body to be misaligned, because most non-Unix telnet packages expect a carriage return, not just a newline. Furthermore, the fact that the HTTP server closes the connection when finished often makes it difficult to see more than the last screenful of data. Consequently, I present WebClient, a simple graphical interface to HTTP servers used in Figure 16-1 and throughout the chapter. This client reads the request line and headers from the user, then sends them to the server along with a blank line, displaying the result in a scrolling text area. Downloading is performed in a separate thread so that the user can interrupt the download of long documents. The code for WebClient is given in Listing 16.1, with supporting classes given in Listings 16.2 and 16.3. It also makes use of the NetworkClient class of Chapter 15, "Client-Server Programming in Java" and the QuittableFrame class of Chapter 11 "Windows."

![Web Client](image)

**Figure 16-1** A conversation with JavaSoft's Web server shows a typical request and response.

### Listing 16.1 WebClient.java

```java
import java.awt.*;
import java.util.*;

public class WebClient extends QuittableFrame
  implements Runnable, Interruptable {

  public static void main(String[] args) {
    WebClient wc = new WebClient("Web Client");
    wc.resize(600, 700);
    wc.show();
  }

  private LabeledTextField hostField, portField, requestLineField;
  private TextArea requestHeadersArea, resultArea;
  private String host, requestLine;
  private int port;
  private String[] requestHeaders = new String[30];
  private Button submitButton, interruptButton;
  private boolean interrupted = false;

  public WebClient(String title) {
    super(title);
    setBackground(Color.lightGray);
    setLayout(new BorderLayout(5, 30));
    int fontSize = 14;
    Font labelFont = new Font("TimesRoman", Font.BOLD, fontSize);
    Font headingFont = new Font("Helvetica", Font.BOLD, fontSize+4);
    Font textFont = new Font("Courier", Font.BOLD, fontSize-2);
    Panel inputPanel = new Panel();
    inputPanel.setLayout(new BorderLayout());
    Panel labelPanel = new Panel();
    labelPanel.setLayout(new GridLayout(4,1));
    hostField = new LabeledTextField("Host:", labelFont, 30, textFont);
    portField = new LabeledTextField("Port:", labelFont, "80", 5, textFont);
  }
```

continued
Chapter 16  The HyperText Transfer Protocol

Listing 16.1 WebClient.java (continued)

```java
requestLineField =
    new LabeledTextField("Request Line:",
    labelFont,
    "GET / HTTP/1.0",
    50,
    textFont);
labelPanel.add(hostField);
lablePanel.add(portField);
lablePanel.add(requestLineField);
Label requestHeadersLabel =
    new Label("Request Headers:");
requestHeadersLabel.setFont(labelFont);
labelPanel.add(requestHeadersLabel);
inputPanel.add("North", labelPanel);
requestHeadersArea = new TextArea(5, 80);
requestHeadersArea.setFont(textFont);
inputPanel.add("Center", requestHeadersArea);
Panel buttonPanel = new Panel();
submitButton = new Button("Submit Request");
submitButton.setFont(labelFont);
buttonPanel.add(submitButton);
inputPanel.add("South", buttonPanel);
Panel resultPanel = new Panel();
resultPanel.setLayout(new BorderLayout());
Label resultLabel =
    new Label("Results", Label.CENTER);
resultLabel.setFont(headingFont);
resultPanel.add("North", resultLabel);
resultArea = new TextArea();
resultArea.setFont(textFont);
resultPanel.add("Center", resultArea);
Panel interruptPanel = new Panel();
interruptButton = new Button("Interrupt Download");
interruptButton.setFont(labelFont);
interruptPanel.add(interruptButton);
resultPanel.add("South", interruptPanel);
```

16.1 Communicating with an HTTP Server

Listing 16.1 WebClient.java (continued)

```java
public boolean action(Event event, Object object) {
    if (event.target == submitButton) {
        Thread downloader = new Thread(this);
        downloader.start();
        return (true);
    } else if (event.target == interruptButton) {
        interrupted = true;
        return (true);
    } else
        return (false);
}

public void run() {
    interrupted = false;
    if (checkArgs())
        HttpClient client = new HttpClient(host,
            port,
            requestLine,
            requestHeaders,
            resultArea,
            this);
}

public boolean interrupted() {
    return (interrupted);
}

private boolean checkArgs() {
    host = hostField.getTextField().getText();
    if (host.length() == 0) {
        report("Missing hostname");
        return (false);
    }
    String portString =
        portField.getTextField().getText();
    if (portString.length() == 0) {
        report("Missing port number");
        return (false);
    }
    return (true);
}
```

continued
Chapter 16 The HyperText Transfer Protocol

Listing 16.1 WebClient.java (continued)

```java
try {
    port = Integer.parseInt(portString);
} catch (NumberFormatException nfe) {
    report("Illegal port number: " + portString);
    return(false);
}
requestLine =
    requestLineField.getTextField().getText();
if (requestLine.length() == 0) {
    report('Missing request line');
    return(false);
}
getRequestHeaders();
return(true);

private void report(String s) {
    resultArea.setText(s);
}

private void getRequestHeaders() {
    for (int i=0; i<requestHeaders.length; i++)
        requestHeaders[i] = null;
    int headerNum = 0;
    String header =
        requestHeadersArea.getText();
    StringTokenizer tok =
        new StringTokenizer(header, "\r\n");
    while (tok.hasMoreTokens())
        requestHeaders[headerNum++] = tok.nextToken();
}

protected void handleConnection(Socket uriSocket)
    throws IOException {
        try {
            SocketUtil s = new SocketUtil(uriSocket);
            PrintStream out = s.getPrintStream();
            DataInputStream in = s.getDataStream();
            outputArea.setText("\n");
            out.println(requestLine);
            for(int i=0; i<requestHeaders.length; i++) {
                if (requestHeaders[i] == null)
                    break;
                else
                    out.println(requestHeaders[i]);
            }
            out.println();
            String line;
            while ((line = in.readLine()) != null &&
                !app.interrupted())
                outputArea.appendText(line + "\n");
            if (app.interrupted())
                outputArea.appendText("--- Download Interrupted ----");
        }
    }
```
Chapter 16 The HyperText Transfer Protocol

Listing 16.2 HttpClient.java (continued)

```java
} catch (Exception e) {
    outputArea.setText("Error: "+e);
}

private boolean checkHost(String host) {
    try {
        InetAddress.getByName(host);
        return(true);
    } catch (UnknownHostException uhe) {
        outputArea.setText("Bogus host:" + host);
        return(false);
    }
}
```

Listing 16.3 Interruptable.java

```java
public interface Interruptable {
    public boolean interrupted();
}
```

16.2 The HTTP Request Line

An HTTP request consists of a request line, an optional number of request headers, a blank line, and, in the case of POST, some additional data. The request line is of the form:

```
Request-Method URI HTTP/Version
```

The request method is a single word taken from the options described later in this section. The URI (Uniform Resource Identifier) is simply the requested URL with the protocol portion (e.g., http://), hostname (e.g., andreesen-pc.netscape.com), and port number (e.g., the "5555" in http://gates-mac.microsoft.com:5555) stripped off. For instance, if a user requests http://www.javasoft.com/~gosling/cobol-users-guide.html, the URI sent by the browser is /~gosling/cobol-users-guide.html.

Technically, the HTTP version could be something other than HTTP/1.0, but currently this is usually used. GET, HEAD, and POST are the only HTTP 1.0 request methods. However, as more servers start to support HTTP 1.1, PUT, DELETE, OPTIONS, and TRACE may become more widely used. Some examples:

```
GET / HTTP/1.0
GET /~toy/ms-job-application.pdf HTTP/1.0
POST /cgi-bin/search HTTP/1.0
HEAD /reports/notes.html HTTP/1.0
```

HTTP 1.0 Request Methods

The following sections summarize the three standard HTTP 1.0 request methods and the most important HTTP 1.1 additions.

GET
This is the method used by browsers for normal document requests. For instance, if the user requests the URL http://www.javasoft.com/, the browser will connect to port 80 of www.javasoft.com and issue "GET / HTTP/1.0", as shown earlier in Figure 16-1. Data for CGI programs can be attached to the URL after a question mark ("?"); this is discussed in more detail in Chapter 17, "CGI Programming and Beyond—The Client Side."

HEAD
This request method has identical syntax to GET, but the server only returns the response headers, omitting the requested document itself. It is used by link verifiers or by clients that want to discover the file size, modification date, or server version without actually retrieving the document. Figure 16-2 gives an example.

POST
With GET and HEAD, the URI portion of the request line can contain data intended for a server-side program. This is done by attaching the data to the URI after a question mark ("?"). With POST, however, the data is sent on a separate line rather than attached to the URI. A client would send the request line, a Content-Length line, any additional HTTP headers, a blank line, then the data the CGI program is expecting. For example, Figure 16-3 shows a simple data input form that uses POST to send the data in the textfield to the mini-HTTP server created
in Section 15.7. Figure 16-4 shows the request line, HTTP headers, blank line, and data sent when the form is submitted. The CGI program can use the Content-Length line to determine how much data to read, or can be a little less cautious, assume that the POST request was generated by an HTML form, and read just a single line. Creating HTML forms is discussed in detail in Chapter 17, "CGI Programming and Beyond—The Client Side," and methods for processing POST data in CGI programs are covered in Chapter 18, "CGI Programming and Beyond—The Server Side."

**PUT**

This header is part of HTTP 1.1, and is not yet widely supported. It lets clients supply a document and request the server to store it at the specified URI.

**DELETE**

This HTTP 1.1 header requests the server to remove a specified URL.

**OPTIONS**

This HTTP 1.1 header requests information about what communication options are available for the specified URL. If the URI is replaced by an asterisk ("*"), the request is to be interpreted as asking for options available from the server in general.

**TRACE**

This HTTP 1.1 header requests that the server return the attached document unchanged; it is used for debugging purposes.
16.3 HTTP Request Headers

After the initial request line (the request method, document URI, and "HTTP/1.0"), the client can send several optional headers that provide additional information. For instance, Figure 16-5 uses the simple HTTP server developed in Section 15.7 to echo back the request line and headers sent by Netscape 3.01 when requesting a particular document.

Accept
This specifies the MIME types that the client is willing to accept, in order of preference. See Table 16.1 in Section 16.5 for more information on MIME types. Multiple options can be given on a single line separated by commas, or placed in separate headers. This information is available to the server or to CGI programs to decide what type of data to return. For instance, in Figure 16-5, Netscape Navigator specifies a preference for GIF images, X11 bitmap files, JPEG images, progressive JPEG images, and any other MIME type, in that order. The Accept header can be omitted or

Accept: */*

can be used to indicate that the client will accept anything.

Authorization: auth-scheme: credentials

When a requested document requires authentication, usually indicated by a WWW-Authenticate response header (see Figure 16-14), the authentication information can be supplied in an Authorization header. The most common authorization scheme is BASIC, which is followed by username:password encoded in base64. In the base64 encoding scheme, the 8-bit encoding of each set of three characters is
Appendixed, then broken into four 6-bit sections, each used as the value of one of 64 possible characters (A-Z, a-z, 0-9, +, /, in that order). Padding at the end is indicated with the "=" character. For more details on base64 encoding, see Section 5.2 of RFC 1521, available in plain text at http://ds.internic.net/rfc/rfc1521.txt or at http://ds.internic.net/rfc/rfc1521.ps in PostScript. Note that base64 encoding does not provide protection from an attacker that can intercept network traffic, as the encoding can be easily reversed. Secure Sockets Layer (SSL) would be used in a case needing this level of security, although it should be noted that for users connecting directly through an Internet Service Provider, the difficulty of attackers snooping on the network is roughly akin to that of tapping your phone line. However, many corporate and University sites are vulnerable to users running "packet sniffers" on the local network. SSL provides protection from such attackers as well as from more sophisticated attackers who might have access to ISP systems or telephone lines. It is a standard that allows compliant browsers and servers to talk to each other using encrypted data streams. This is known as "secure HTTP," and is generally indicated by "https" instead of "http" in the URL. SSL is supported by both Internet Explorer and Netscape Navigator, and by a variety of commercial HTTP servers. For more information on SSL, see http://www.netscape.com/assist/security/ssl/

For a quick summary of cryptographic techniques, see Section 16.8, "An Overview of Public-Key Cryptography."

Core Security

Base64 encoding is not intended to provide cryptographic protection. If you need to restrict access to Web sites and to protect against network eavesdropping, you will need to purchase a server that supports Secure Sockets Layer.

16.3 HTTP Request Headers

Establish the connection. Several servers support an experimental version of persistent connections, whereby the connection is not closed between requests associated with a single Web page. A widely adopted extension to HTTP 1.0 lets clients supply the Connection: header with Keep-Alive to indicate that this should be used if available. For instance, both Netscape Navigator (Figure 16-5) and Microsoft Internet Explorer (Figure 16-8) use this header in all requests.

Core Note

In HTTP 1.1, persistent connections are the default, and will be used unless the client supplies a header of Connection: close.

Content-Length

This header is generally associated with server responses, but it must be supplied for POST requests. For instance, in the POST example of Figure 16-4, a Content-Length line is included to indicate that 19 characters will be sent on the data line that follows the HTTP request headers and the blank line.

Cookie: nameI=valueI; ... ; nameN=valueN

This header is used by browsers supporting Netscape's persistent cookie specification. It is used to return information supplied via a Set-Cookie header in a previous connection. See Section 16.6 for more details on cookies.

From

This gives an e-mail address of a person responsible for the request. It is sometimes used by Web indexing programs ("robots" or "spiders") to supply a point of contact for problems. For instance, some early spiders used a depth-first algorithm for exploring the Web, resulting in unacceptably heavy loads on servers. However, regular requests from browsers do not include this header; the name and e-mail address of the user are deliberately kept private.

Host

The Host line gives the host and port as listed in the original URL. Technically, this is an HTTP 1.1 header (in fact, it is required in 1.1), but it is widely used by earlier clients. For instance, even version 2.02 of Netscape Navigator supplies this header.
If-Modified-Since

This header is used with GET to conditionally retrieve updated versions of files. The file will only be sent if its modification date is later than the date specified, otherwise a 304 ("Not Modified") response will be sent. The modification date is indicated by the Last-Modified response header (see Section 16.5, "HTTP Response Headers"). The date should be of the form "Weekday, Day Month Year Hr:Min:Sec GMT." For example, Figure 16-6 shows a request for the HTTP reference page at the World Wide Web Consortium.

HTTP/1.0 304 Not Modified
Server: CERN/3.0
Date: Wed. 11 Jun 1997 20:40:14 GMT
Content-Type: text/html
Content-Length: 22589

HTTP/1.0 200 OK
Server: CERN/3.0
Date: Wed. 11 Jun 1997 20:34:51 GMT
Content-Type: text/html
Content-Length: 22589

Figure 16-6 Response from the W3C server includes a Last-Modified header.

Note the time given in the Last-Modified header in Figure 16-6. Now, let's look at a similar request that does a conditional GET, asking for the page only if it has been modified after this. The response to this request omits the file, returning a "Not Modified" status line instead, as illustrated in Figure 16-7.

HTTP/1.0 304 Not Modified
Server: CERN/3.0
Date: Wed. 11 Jun 1997 16:00:00 GMT
Content-Type: text/html
Content-Length: 22589

Figure 16-7 Response from the W3C server for a conditional GET includes a Not Modified response when the file is older than the specified date.

Pragma: no-cache

This directs the server to return a fresh document even if it is a proxy with a locally cached copy. In HTTP 1.1, the Cache-Control: no-cache header can be used instead.

Referer

This specifies the URL of the page, if any, that contained the cross-reference that sent the client to the current document. This information is also normally recorded by servers, and can be used to find out what
WWW pages contain links to your pages. For instance, on Unix with the NCSA server, this may be in:

```
/usr/local/etc/httpd/logs/referer_log
```

You can have some fun with people who search this regularly by connecting to the HTTP server with WebClient and specifying some unusual URLs such as:

```
http://ceo.microsoft.com/~gates/favorite-pages.html
```

or

```
http://most-wanted.fbi.gov/suspects.html
```

User-Agent

This specifies the type of browser. Vendors can choose the format used to fill in this header, but it is often similar to:

```
Vendor/version (platform; encryption; os)
```

For instance, Netscape 3.01 sends:

```
User-Agent: Mozilla/3.01 (X11; U; SunOS 5.4 sun4m)
```

for the US-encryption version on a Sun platform and

```
User-Agent: Mozilla/3.01 (Win95; I)
```

for the international-encryption version on a Windows 95 platform. Interestingly, Internet Explorer 3.01 sends a "Mozilla" (Netscape) identification:

```
User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows 95)
```

This indicates that it is compatible with JavaScript version 1.0, which was supported in Netscape 2.0. This information is often logged by the server. For instance, on Unix with the NCSA server:

```
/usr/local/etc/httpd/logs/agent_log
```

typically records this information for each connection.

UA-Pixels, UA-Color, UA-OS, UA-CPU

These nonstandard headers are sent by Internet Explorer to indicate the screen size, color depth, operating system, and CPU type used by the browser (i.e., "User Agent," "UA") machine. Figure 16–8 gives an example. These values can be used by CGI programs to customize the types of page they build for users.

```
User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows 95)
```

Nonstandard Headers

In addition to the standard HTTP 1.0 request headers, many clients transmit proprietary and/or HTTP 1.1 headers. For instance, Netscape 3.01 (Figure 16–5) and Internet Explorer 3.01 (Figure 16–8) send the common but non-HTTP 1.0 Connection and Host headers, Internet Explorer sends proprietary UA-xxx and Accept-Language headers, and Mosaic (Figure 16–9) sends the nonstandard Extension header.
Chapter 16  The HyperText Transfer Protocol

Threaded Echo Server 1.0 Results - NCSA Mosaic

Threaded Echo Server 1.0 Results
Here is the request line and request headers sent by your browser:

GET /foo/bar.html HTTP/1.0
Accept: image/x-pct-type
Accept: image/jpeg-type
Accept: text/x-pct
Accept: image/jpeg
Accept: video/x-sgi
Accept: audio/x-adt
Accept: audio/x-mpeg
Accept: image/gif
Accept: text/plain
Accept: text/html
Accept: */ *
User-Agent: NCSA Mosaic/3.0.0 (Windows 95)
Extension: Security/Digest

Figure 16-9 HTTP request headers sent by NCSA Mosaic 3.0 on Windows 95.

16.4 The HTTP Response Status Line

After receiving a valid request (request line, optional headers, blank line), the server returns a status line, one or more response headers, a blank line, and the associated document. The status line is of the following form:

HTTP/Version Status-Code Message

Currently, the version is usually 1.0. A status code between 200 and 299 indicates success, a value between 300 and 399 indicates the file has moved, a value between 400 and 499 indicates an error by the client, and a value between 500 and 599 indicates a server error. There are only a few predefined codes in HTTP 1.0, but if the browser receives an unrecognized code, it can use the general range to determine what action to take. For instance, a value of 444 should be treated as an error, even though 444 is not a standard HTTP 1.0 status code. The message is a very short string describing the type of error. Some examples include:

HTTP/1.0 200 OK
HTTP/1.0 301 Moved Permanently
HTTP/1.0 404 Not Found

HTTP 1.0 Status Codes

The predefined HTTP 1.0 status codes are listed in the following paragraphs, along with the usual accompanying message. Although the status codes are standardized, the messages can vary slightly from server to server.

200 OK

This response indicates that the request was successful. If the request method is GET or POST, the resultant document is included after the headers. If HEAD is used, the response includes the headers only. Figure 16-10 shows an example.

Figure 16-10 A response of 200 indicates a successful request.
Chapter 16  The HyperText Transfer Protocol

201 Created
The 201 response applies to POST requests only; it indicates that a new resource was created as a result of the request. The URL of the new resource is given in the Location header.

202 Accepted
This status code is rarely used, but is intended to indicate that the request has been accepted, but results are not yet available. The body of the response should contain more information in such a case.

204 No Content
A response of 204 is generated when the server has fulfilled the request but has no data to return. Browsers should continue to display the document that generated the request in such a case, rather than updating their document view. This code is sometimes used by CGI programs that have no useful information to return, such as when the user clicks on an unused portion of an image map.

300 Multiple Choices
300 means that the requested resource is available at one or more locations. For GET or POST requests, the body of the response should contain a list of the possible locations in a way that lets the user select one. If the server has a preferred choice, the URL should be supplied in the Location header.

301 Moved Permanently
301 signifies that the requested resource has been assigned a new URL, and that all future requests should use the new location. This new URL should be supplied in the Location header, and for requests other than HEAD, the body of the response should contain a short explanation and a link to the new document. Most browsers will automatically redirect GET requests to the URL specified in the Location header, but should not do so for POST requests without user confirmation. Note that URLs that omit the trailing slash for directories generate either this response (301) or the similar 302, depending on the server. That is, this response is generated for requests such as

http://foo.bar.com/some/directory
instead of the correct
http://foo.bar.com/some/directory/

For instance, Figure 16-11 shows the 301 response received when requesting

http://www.apl.jhu.edu/~hall/java

301 and 302 responses require the browser to read the new URL out of the Location field, then to make a second connection to retrieve the proper document. This can require significant extra time if the network connection is slow, so you should be careful to explicitly include the trailing slash in hypertext links you create.

Figure 16-11 A response of 301 means the requested document has permanently moved to the specified location.
302 Moved Temporarily
302 signifies that the requested resource has been temporarily assigned a new URL, but that all future requests should use the original URL. Other than the interpretation of where to go for future requests, it is used identically to the 301 code.

Figure 16-12 A response of 302 means the requested document has moved temporarily to the specified location.

304 Not Modified
If the user requests a document via GET and uses an If-Modified-Since header, then if the document is older than the specified date, a 304 response should be returned. The browser might use its cached copy in such a case.

400 Bad Request
A response of 400 indicates that the server could not understand the request due to a syntax error. For example, Figure 16-13 shows the results when a request line of "A B C D" is sent to the Web server at www.ibm.com.

Figure 16-13 A response of 400 indicates a syntax error in the request.

401 Unauthorized
401 specifies that the request did not have the proper authorization, either because the Authorization header was missing, or because the data in it was incorrect. The response should also include a WWW-Authenticate header (see Section 16.2, "The HTTP Request Line"). For example, Figure 16-14 shows the headers returned by Netscape's site when their members-only documentation page is requested.
Chapter 16  The HyperText Transfer Protocol

403 Forbidden
403 is used to indicate that the user cannot access the requested resource, regardless of authorization. It is often used when the file or directory permissions of the resource do not permit access, or when access to a resource is restricted to certain hosts.

404 Not Found
404 indicates that no resource was found at the specified address. It is symptomatic of the dreaded “dead link” problem, where a Web page needs constant monitoring and updating to be sure that links to other pages aren’t “dead” (or “broken”) because the filename or host machine of the hypertext reference has changed. This is the most common error code. See Figure 16-15 for an example.

500 Internal Server Error
500 is used to designate an unspecified error in the server. It is often the result of CGI programs that crash or return incorrectly formatted headers. How to write CGI programs is discussed in Chapter 18, “CGI Programming and Beyond—The Server Side.” For now, note that most CGI programs mimic an HTTP server by generating an HTTP Content-Type line, a blank line, and then a document matching the Content-Type. Listing 16.4 gives an example using the Unix Bourne shell. This example illustrates a common error where whitespace (a single SPACE character, in this case) is generated instead of the blank line, in violation of the HTTP specification. Figure 16-16 shows the result.

Figure 16-14  A response of 401 indicates a password-protected site.

Figure 16-15  A response of 404 indicates a missing document.
Chapter 16 The HyperText Transfer Protocol

Listing 16.4 An Incorrect CGI Program that Doesn't Generate a Blank Line

```
#!/bin/sh

echo 'Content-Type: text/plain'
echo ''
echo 'Oops! Sent a SPACE instead of a blank line'
```

Figure 16-16 A response of 500 could indicate a misconfigured server or an incorrect CGI program.

501 Not Implemented
A 501 response means that the server does not support the functionality needed to fulfill the request. For instance, it cannot handle the request method (POST), or it received an unrecognized method (MUNGE).

502 Bad Gateway
502 is used by servers that act as proxies or gateways; it indicates that the initial server got an invalid response from a remote server.

503 Service Unavailable
503 indicates that the server cannot respond due to maintenance or overloading. When getting this response, the user should reconnect at a later time.

16.5 HTTP Response Headers

In addition to the status line (HTTP/1.0 Status-Code Message), there are a number of other headers that can be sent by the server in response to a request. Most are optional, but a Content-Type header should be included in all responses so that the client knows what to do with the document.

**Allow**
This header lists the set of request methods (GET, POST, and so forth) permitted for the specified resource.

**Content-Encoding**
Content-Encoding describes the decoding mechanism that must be used to obtain the MIME media type specified in the Content-Type header. The predefined HTTP 1.0 types are “x-gzip” (GNU zip) and “x-compress” (Unix “compress”).

**Content-Length**
This specifies the number of bytes contained in the file.

**Content-Type**
This specifies the MIME type and subtype of the entity being sent (or, in the case of HEAD requests, of the entity requested but not sent) so that the browser knows what to do with the result. For instance, Content-Type: text/html indicates an HTML document. The subtype will begin with “x-” if it is not officially registered, and some media types can be specified either way (e.g., audio/midi or audio/x-midi) because the status changed from unofficial to official after many browsers were released. MIME types are described in RFC 1521, MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies, available in plain text at http://ds.internic.net/rfc/rfc1521.txt
16.5 HTTP Response Headers

or at

http://ds.internic.net/rfc/rfc1521.ps

in PostScript. The most important types for Web clients are summarized in Table 16.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/octet-stream</td>
<td>Unrecognized or binary data</td>
</tr>
<tr>
<td>application/pdf</td>
<td>Acrobat (.pdf) file</td>
</tr>
<tr>
<td>application/postscript</td>
<td>PostScript file</td>
</tr>
<tr>
<td>audio/basic</td>
<td>Sound file in .au or .snd format</td>
</tr>
<tr>
<td>audio/x-aiff</td>
<td>AIF sound file</td>
</tr>
<tr>
<td>audio/x-wav</td>
<td>WAV sound file</td>
</tr>
<tr>
<td>audio/midi</td>
<td>MIDI sound file</td>
</tr>
<tr>
<td>text/html</td>
<td>HTML document</td>
</tr>
<tr>
<td>text/plain</td>
<td>Plain text</td>
</tr>
<tr>
<td>image/gif</td>
<td>GIF image</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>JPEG image</td>
</tr>
<tr>
<td>image/x-xbitmap</td>
<td>X Window bitmap image</td>
</tr>
<tr>
<td>video/mpeg</td>
<td>MPEG video clip</td>
</tr>
<tr>
<td>video/quicktime</td>
<td>QuickTime video clip</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>The Date header designates the current time, in Greenwich Mean Time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expires</td>
<td>This specifies the time and date after which the information should be considered invalid. The document should not be cached after this date.</td>
</tr>
<tr>
<td>Last-Modified</td>
<td>Last-Modified gives the time and date at which the document was last changed. The client can supply a date in the If-Modified-Since request header in order to prevent downloading of documents that were modified prior to that date. See Figures 16-6 and 16-7 (Section 16.3, &quot;HTTP Request Headers&quot;) for an example.</td>
</tr>
<tr>
<td>Location</td>
<td>This gives the new location of the requested document. It is usually supplied in conjunction with responses with a status code of 201 (Created), 300 (Multiple Choices), 301 (Moved Permanently), or 302 (Moved Temporarily).</td>
</tr>
<tr>
<td>Server</td>
<td>This header supplies descriptive information about the HTTP server. It usually contains vendor and version information. For instance, the Server lines of Figures 16-10 through 16-15 show six different HTTP servers used by various sites.</td>
</tr>
<tr>
<td>Set-Cookie</td>
<td>This header contains a name/value pair to be stored by browsers such as Netscape Navigator and Microsoft Internet Explorer that support the Netscape persistent &quot;cookie&quot; specification. Optionally, it can contain expires, path, domain, and secure attributes. Until the cookie expires, it will be transmitted in the Cookie header in future requests to the same URL or to URLs in the specified path and domain. See Section 16.6 for more details on cookies.</td>
</tr>
<tr>
<td>WWW-Authenticate</td>
<td>This header is required for 401 (Unauthorized) responses. It gives an authorization type and realm that the client has to supply in an Authorization header in order to gain access to the document. For example: WWW-Authenticate: BASIC realm=&quot;Executive-Branch&quot;</td>
</tr>
</tbody>
</table>
Chapter 17  CGI Programming and Beyond—The Client Side

**SELECTED**

If present, SELECTED specifies that the particular menu item shown is selected when the page is first loaded.

**VALUE**

VALUE gives the value to be associated with the NAME of the SELECT menu if the current option is selected. This is not the text that is displayed to the user; that is specified by separate text listed after the OPTION tag.

The following example creates a menu of standard personality types. Because only a single selection is allowed and because no visible SIZE is specified, it is displayed as a combo box, shown in Figures 17-23 and 17-24. If the entry "Hacker (Best)" is active when the form is submitted, then "Type2=Hacker" is sent to the CGI program. Notice that it is the VALUE field, not the descriptive text, that is transmitted.

```html
<p>
Choose Your Type

<SELECT NAME="Type2">
  <OPTION VALUE="Geek" SELECTED>Geek (Good)
  <OPTION VALUE="Nerd">Nerd (Better)
  <OPTION VALUE="Hacker">Hacker (Best)
</SELECT>
</p>
```

![Figure 17-23 A SELECT element displayed as a combo box.](image)

The second example shows a SELECT element rendered as a list box. If more than one entry is active when the form is submitted, then more than one value is sent, listed as separate entries (repeating the NAME). For instance, in the example shown in Figure 17-25, "Language=CPP&Language=Java" gets added to the data being sent to the server.

```html
<p>
Choose the languages you know:<BR>

<SELECT NAME="Language" MULTIPLE SIZE=5>
  <OPTION VALUE="C">C
  <OPTION VALUE="CPP">C++
  <OPTION VALUE="Lisp">Lisp
  <OPTION VALUE="Smalltalk">Smalltalk
  <OPTION VALUE="Java" SELECTED>Java
</SELECT>
</p>
```

![Figure 17-25 A SELECT element with MULTIPLE or SIZE specified results in a list box.](image)

### 17.4 ISINDEX

ISINDEX is not a FORM element, but can be used to make simple textual input fields for CGI programming. Due to its limited flexibility, it is not recommended for most CGI applications. It can appear in either the HEAD or the BODY of the document.

**HTML Element:** `<ISINDEX ...> (No End Tag)`

**Attributes:** PROMPT, ACTION (nonstandard)

This element produces a simple textfield for user input.
PROMPT

PROMPT specifies the advisory text to be displayed before the textfield. If it is left unspecified, then the browser chooses some default like 'This is a searchable index. Enter search keywords:'

ACTION

Unless the non-standard ACTION attribute is supplied (Internet Explorer only), when the user hits a carriage return in the textfield, the data in the textfield is sent to the same URL from which the original page was loaded. This means that in the absence of ACTION, the page containing the ISINDEX must itself be the output of a CGI program. The typical scenario involves the CGI program first checking if it received any data, then either generating the request page (no data received) or the results page (data received). In general, ISINDEX is a poor choice compared to HTML forms. However, the data it transmits does not contain an equals sign (=), and thus is easier to process (see Chapter 18 on the server side of CGI programming). So it is still occasionally used for simple interfaces. For instance, http://hoo­hoo.ncsa.uiuc.edu/cgi-bin/finger is a CGI program that provides an interface to the "finger" user-identification service. If a connection is made with no attached data, a data input page is created. Listing 17.4 shows the HTML generated for this page, with the result shown in Figure 17-26.

Listing 17.4 NCSA's input page for the "finger" service

```html
<TITLE>Finger Gateway</TITLE>
<H1>Finger Gateway</H1>

<ISINDEX>

This is a gateway to "finger". Type a user@host combination in your browser's search dialog.<p>

However, when the same URL is requested with accompanying data, either directly or by entering data in the textfield as above, the result page is returned instead, as shown in Figure 17-27.

Figure 17-26 Accessing the finger program with no attached data generates an input page that uses ISINDEX.

Figure 17-27 Accessing the finger program with attached data generates the results page.
The Common Gateway Interface lets Web pages communicate with databases or other programs on a system running an HTTP server. In CGI programming, the HTTP server is configured to treat URLs in certain locations specially. When the user requests a URL in such a reserved location, instead of treating it as a file whose contents should be returned, the HTTP server treats it as a program whose output should be returned. In Chapter 17 we discussed the client side of this process: how to collect input from HTML forms or via Java applets, and how to transmit that data to CGI programs. This chapter describes the server side of the process: how to write the programs that process data from CGI clients and return the results. Technically, "CGI" refers to the situation when the HTTP server starts an external program to process the data and generate results. However, the approach is very similar when the server is directly linked with databases or shared libraries, or when the server has an embedded interpreter or compiler for server-side programs in languages such as Java and JavaScript. These latter approaches are discussed at the end of this chapter.

18.1 The CGI Interaction Process

A CGI program should perform four basic steps, outlined as follows.

**Read the data**
Data can be sent by GET or by POST; the CGI program reads it differently in each case.
Chapter 18  CGI Programming and Beyond—The Server Side

18.2  Reading GET Data: The QUERY_STRING Variable

Before going further, we recommend that you get a simple program of this type working to be sure that you are using the proper directory, understand the interaction process, and have execute permissions set properly. If you get a “Server Error” (status 500, generally), this probably means that you are sending incorrect headers, perhaps by a spelling error in “Content-Type” or by generating whitespace such as a space character instead of a completely blank line. If you get a “Forbidden” (403) message, this may mean that your program does not have execute permissions set (use “chmod a+x program” on Unix to make a script executable). A “Not Found” (404) error often means that you have supplied the wrong name for the CGI program or misunderstood which directory your server uses for CGI programs.

18.2  Reading GET Data: The QUERY_STRING Variable

When the client requests a CGI URL of the form http://server/cgi-bin/program?data, the HTTP server calls “program” after first setting the QUERY_STRING environment variable to “data”. The program is responsible for reading the data out of the environment variable, parsing it to find the pieces of interest, then generating output that mimics an HTTP server. For instance, Listing 18.2 presents a simple Unix shell script that accepts a GET request and generates a plain-text page that describes the input it received.

```
#!/bin/bash
echo "Content-Type: text/plain"
echo ""
echo "QUERY STRING is '$QUERY_STRING'
```

If installed as an executable file named ShowData in http://www.apl.jhu.edu/cgi-instruct/hall, then GET requests get answered as illustrated in Figure 18-2.
The examples in these sections assume that the HTTP server is running on Unix, a popular choice for large Web sites. The exact details of getting data to the CGI program may vary on other platforms. For instance, some Windows-based servers use temporary files for input and output, rather than environment variables and standard input/output as in the original CGI standard. In general, however, the target program needs to be in a format that the host operating system considers to be executable.Popular languages for CGI programs include Perl, the Unix Bourne or Korn shells, Windows batch files, C, C++, Visual Basic, AppleScript, and, increasingly, Java. To do something interesting with the data in Java, the shell script can pass the data to a Java program to process it, as illustrated in Listing 18.3. This intermediate shell script (or .bat file if on Windows) is necessary to pass along the value of the QUERY_STRING variable even if your Java compiler generates stand-alone executables. This is because Java has no method to directly read environment variables due to the fact that this concept is foreign to some operating systems supported by Java. Note the use of the full path to the Java executable in this example, because the process owning the HTTP server may not have the same search path as the user writing the program.

### Listing 18.3 The CgiGet executable script

```bash
#!/bin/sh

/usr/local/JDK/bin/java CgiGet "$QUERY_STRING"
```

Once the shell script passes the data to CgiGet.java (Listing 18.4), the Java program can then generate an HTML page describing the results.

### Listing 18.4 CgiGet.java

```java
public class CgiGet {
    public static void main(String[] args) {
        CgiGet app = new CgiGet("CgiGet", args);
        app.printFile();
    }

    private String name;
    private String[] args;
    protected String type = "GET";

    public CgiGet(String name, String[] args) {
        this.name = name;
        this.args = args;
    }

    public void printFile() {
        printHeader(name);
        printBody(args);
        printTrailer();
    }

    protected void printHeader(String name) {
        System.out.println("Content-Type: text/html\n"
                + "" + "
        <HTML><HEAD><TITLE>The " + name + " Program</TITLE>" \n
```

```
Chapter 18  CGI Programming and Beyond—The Server Side

Now, if the shell script is installed as CgiGet on the same server as in the previous example, GET requests would result in an HTML document being returned instead of plain text, as illustrated in Figure 18-3.

### 18.3 The Standard CGI Environment Variables

In addition to the QUERY_STRING environment variable that stores the data that was appended to the URL, HTTP servers set several other variables containing information such as the host making the request, the port number used, and so forth. These variables can be accessed from the CGI programs written in Perl, C, C++, or scripting languages, or, with a one-line intermediate script, from Java.

#### Passing Variables as Named Parameters

In the CgiGet example in the previous section, we passed the query string data to the CgiGet program as a command-line argument. When supplying multiple variables, however, listing them all as command-line arguments makes maintenance difficult. This is because changes to the number or order of variables supplied requires changes to both the shell script (or .bat file, depending on the OS) and the Java program. A more convenient alternative is to supply the variables as properties, allowing the Java program to look them up by name. Entries that are passed to the Java interpreter or JIT compiler via "-Dname=value" can be accessed from within the program via `System.getProperty("name")`, returning value as a String. Listing 18.5 gives a simple example, with results shown in Listing 18.6.

#### Listing 18.5 PropertyTest.java

```java
public class PropertyTest {
    public static void main(String[] args) {
        System.out.println("Java: "+ System.getProperty("java") + \
            "CGI: "+ System.getProperty("cgi") + \
            "C++: "+ System.getProperty("cpp"));
    }
}
```

#### Listing 18.6 Results of the PropertyTest program

```
> java PropertyTest
Java: null
CGI: null
C++: null
```

```
> java -Djava=cool -Dcgi=hot -Dcpp=stale PropertyTest
Java: cool
CGI: hot
C++: stale
```

An interface to a Java-based CGI program would then invoke the program via

```
java -DVAR1="$VAR1" -DVAR2="$VAR2" ... CgiProgram
```

#### Summary of CGI Variables

The following list summarizes the standard variables available to CGI programs.

**AUTH_TYPE**
The AUTH_TYPE variable gives authentication method used by the server to validate a user. It is taken from the Auth-Scheme header.

**CONTENT_LENGTH**
This variable specifies the number of characters (bytes) contained in QUERY_STRING. It is always supplied for POST requests, but it is usually not supplied for requests that use the GET method.
18.3 The Standard CGI Environment Variables

CONTENT_TYPE
If supplied, this specifies the MIME type of the attached data. It is normally left unspecified.

DOCUMENT_ROOT
When a user accesses http://some.host/some.file, the HTTP server looks for some.file in a designated directory. The DOCUMENT_ROOT variable gives the pathname of that directory. This variable is omitted by some servers.

GATEWAY_INTERFACE
This gives the server's CGI version, e.g., "CGI/1.1".

HTTP_XXX
This set of variables gives the contents of standard HTTP headers. The name of the environment variable is derived from the header name with dashes changed to underscores and "HTTP_" attached. For instance, the contents of the Cookie header would be stored in the variable HTTP_COOKIE, Referer in HTTP_REFERER, User-Agent in HTTP_USER_AGENT, and so forth. See Section 16.3 (HTTP Request Headers) for an explanation of the standard headers. Note, however, that the server might not set variables for all headers, and some variables (e.g., HTTP_AUTHORIZATION from the Authorization header) should not be made available to CGI programs.

PATH_INFO
This gives a virtual pathname to be used by the CGI program. PATH_TRANSLATED gives the actual path this represents. It is supplied by the client by appending it to the URL, but without a leading "?". The data for QUERY_STRING can be supplied after this. For example, the value of PATH_INFO would be /virtual/path in both http://host.com/cgi-bin/program.cgi/virtual/path and http://host.com/cgi-bin/program.cgi/virtual/path?query_data. QUERY_STRING would be empty in the first case and be "query_data" in the second case.

PATH_TRANSLATED
This variable specifies the actual path of the file given in PATH_INFO. Note, however, that for security reasons many servers do not support this variable.

QUERY_STRING
This variable contains the data sent by the user by appending it to the URL after a "?". It will arrive URL-encoded if sent via HTML forms unless a nonstandard encoding type was used. See Section 17.2 (The HTML FORM Element) for details on changing the encoding type via the ENCTYPE attribute of FORM.

REMOTE_ADDR, REMOTE_HOST
These variables specify the IP address and fully qualified name of the requesting host. They can be used to restrict access to CGI resources to users from certain sites. However, this is not absolutely foolproof due to the risk of IP-spoofing by sophisticated attackers.

REMOTE_IDENT
This variable gives the name of the user on the remote system. It is supplied only if identification is supported by the client, which is rare.

REMOTE_USER
REMOTE_USER gives the user name supplied for authentication purposes. It is available only if authentication is supported and the CGI program requires authorization for access.

REQUEST_METHOD
The method by which the HTTP request was made, usually GET or POST.

SCRIPT_NAME
This is the URI of the CGI program (that is, the part of the URL after the protocol and hostname). For example, for the URL http://www.somewhere.com/cgi-bin/program.cgi, the value would be /cgi-bin/program.cgi.

SERVER_NAME
This is the hostname or IP address of the server.

SERVER_PORT
The gives the port on which the server is listening.
982 Chapter 18 CGI Programming and Beyond—The Server Side

18.4 CGI Command-Line Arguments

With GET requests in general, the data appended to the URL gets placed in the QUERY_STRING variable, and the application is responsible for reading this variable and parsing the data. However, there is a special case when the data supplied does not contain an equals sign (=). In such a case, in addition to being stored in QUERY_STRING, the data is separated at the plus signs (+) and then passed as command-line arguments to the CGI program. Note that this situation will never occur when using HTML forms to generate the CGI request, because forms that send any data always transmit at least one equals sign. So this only happens when using ISINDEX, Java applets, or generating the request by hand.

Core Note

HTML forms never generate CGI command-line arguments. You must use the QUERY_STRING variable to process form data.

For instance, suppose that the URL http://www.some-isp.com/cgi-bin/test?foo+bar+baz is requested. In such a case, the program test is called with foo, bar, and baz as arguments, and its results are returned to the client in the normal manner.

To use a Java program to process such a request, the executable shell script or batch file would pass the normal arguments to the Java program instead of the QUERY_STRING variable, as illustrated in Listing 18.7. In fact, on Java systems that generate stand-alone executables (e.g., Asymetrix' SuperCoder or Microsoft SDK), the shell script or .bat file can be omitted altogether, and the URL can refer directly to the executable Java program. This cannot be done in the general case because Java applications do not have a portable way to retrieve the value of environment variables.

Listing 18.7 The CgiCommandLine executable script

```bash
#!/bin/sh

local/JDK/bin/java CgiCommandLine "$@
```

The Java program would then read its arguments in the normal way, and generate output describing the results in the same manner as in the CgiGet example. For example, in Listing 18.8, the CgiCommandLine program separates each of the arguments in a separate entry in a bulleted (UL) list. Figure 18.4 shows the results.

Listing 18.8 CgiCommandLine.java

```java
public class CgiCommandLine extends CgiGet {
    public static void main(String[] args) {
        CgiCommandLine app
            = new CgiCommandLine("CgiCommandLine", args);
        app.printFile();
    }

    public CgiCommandLine(String name, String[] args) {
        super(name, args);
    }

    protected void printBody(String[] data) {
        System.out.println("Data supplied:
<UL>");
        for(int i=0; i<data.length; i++)
            System.out.println(" <LI>");
        System.out.println("</UL>");
    }
}
```
18.5 Handling ISINDEX

When working with HTML forms, a common mode of interaction is to have one URL be an HTML document that gathers the input and have another URL be the CGI program specified by the ACTION attribute of the form. For instance, the Web search service at http://www.lycos.com gathers the search strings, then contacts the search engine at http://www.lycos.com/cgi-bin/pursuit. The ISINDEX element is unusual in that the URL that is contacted when the user hits return in the textfield is always the current document's own URL. This means that

- ISINDEX forms are only useful in pages that are generated by CGI programs, and
- The CGI program needs to check the command-line arguments to determine whether to generate the original data-input page or the secondary-results page.

Furthermore, the data is not sent in the name1=value1&name2=value2 format of HTML forms, and thus can be delivered as command-line arguments to the destination program. This is even safe if the user enters a blank sign (\") explicitly, as it will be URLencoded to %20 before being transmitted.

For instance, Listings 18.9 and 18.10 contain a variation of the CgiCommandLine program that checks the number of command-line arguments before taking action. It generates a page with $ISINDEX ...$ if there are no arguments and creates a page with a list of the arguments if one or more arguments are supplied.

```
Listing 18.9  The IsIndex executable script
bin/sh
:~:/local/JDK/bin/java IsIndex "$@"
```

```
Listing 18.10  IsIndex.java

public class IsIndex extends CgiCommandLine {
  public static void main(String[] args) {
    IsIndex app = new IsIndex("IsIndex", args);
    app.printFile();
  }

  public IsIndex(String name, String[] args) {
    super(name, args);
  }

  protected void printBody(String[] args) {
    if (args.length > 0)
      super.printBody(args);
    else
      System.out.println
        ("Input is collected via the\n" +
        
        
        "<TT>ISINDEX</TT> element. Enter some data\n" +
        "and then hit RETURN.\n" +
        "<ISINDEX PROMPT="Data: ">");
  }
```

Furthermore, the data is not sent in the name1=value1&name2=value2 format of HTML forms, and thus can be delivered as command-line arguments to the destination program. This is even safe if the user enters a blank sign (\") explicitly, as it will be URLencoded to %20 before being transmitted.

For instance, Listings 18.9 and 18.10 contain a variation of the CgiCommandLine program that checks the number of command-line arguments before taking action. It generates a page with $ISINDEX ...$ if there are no arguments and creates a page with a list of the arguments if one or more arguments are supplied.

```
Listing 18.9  The IsIndex executable script
bin/sh
:~:/local/JDK/bin/java IsIndex "$@"
```

```
Listing 18.10  IsIndex.java

public class IsIndex extends CgiCommandLine {
  public static void main(String[] args) {
    IsIndex app = new IsIndex("IsIndex", args);
    app.printFile();
  }

  public IsIndex(String name, String[] args) {
    super(name, args);
  }

  protected void printBody(String[] args) {
    if (args.length > 0)
      super.printBody(args);
    else
      System.out.println
        ("Input is collected via the\n" +
        
        
        "<TT>ISINDEX</TT> element. Enter some data\n" +
        "and then hit RETURN.\n" +
        "<ISINDEX PROMPT="Data: ">");
  }
```

Furthermore, the data is not sent in the name1=value1&name2=value2 format of HTML forms, and thus can be delivered as command-line arguments to the destination program. This is even safe if the user enters a blank sign (\") explicitly, as it will be URLencoded to %20 before being transmitted.
18.6 Reading POST Data

When using the GET method to talk to CGI programs, the client data is appended to the URL of the server program, after a question mark (?). With GET, however, no information is attached to the URL itself. Instead, the client program sends a POST line, a Content-Length line, additional optional HTTP headers, a blank line, and then some data. This data (after the blank line) is available to the specified program as standard input. In principle, the client could send multiple lines of data in a variety of formats. In practice, however, HTML forms that have a METHOD of POST send a single line of data that is a long string of the same form as the data that is supplied in GET_REQUEST for GET requests (name1=value1&name2=value2...).

The POST method has several disadvantages. A Java applet cannot display output (using getAppletContext().showDocument) generated by a CGI program that uses POST, nor can it read data from a POST program on another than the applet’s home machine. Secondly, a POST request cannot be generated by the user simply by opening a particular URL in a WWW browser. On the other hand, POST has several advantages. First of all, because the data is being transmitted separately from the URL, there is no danger of having the data truncated due to a particular browser’s limit on URL length, nor, for ISINDEX data or applet data not containing an equals (=), of exceeding the command-line argument limit of the language implementing the CGI program. Secondly, POST can be used to send private information because the information sent does not show up in the URL, which is often displayed at the top of the browser. Thirdly, because Java programs can read from standard input but not access environment variables, an intermediate shell script or batch file can be omitted for CGI programs developed in Java with compilers that generate stand-alone executables. For instance, class files are automatically passed to Java on recent versions of Java, the Microsoft SDK can wrap class files inside executable files, and Asymmetric’s SuperCede compiler can generate executables that can run completely independent of a separate Java run-time system.

For example, Listings 18.11 and 18.12 show a POST program that reads in the standard input and returns a page showing that data.

Listing 18.11 The CgiPost executable script

```
bin/sh

./local/JDK/bin/java CgiPost
```
Listing 18.12 CgiPost.java

import java.io.*;

public class CgiPost extends CgiGet {
    public static void main(String[] args) {
        try {
            DataInputStream in = new DataInputStream(System.in);
            String data = in.readLine();
            CgiPost app = new CgiPost("CgiPost", data);
            app.printFile();
        } catch (IOException ioe) {
            System.out.println(
                "IOException reading POST data: " + ioe);
        }
    }
}

public CgiPost(String name, String[] args) {
    super(name, args);
    type = "POST";
}

Now, Listing 18.13 shows a simple HTML form that collects some data and sends it by POST to the CgiPost program. This input screen appears in Figure 18-7.

Listing 18.13 CgiPost.html

<!-- DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>Sending POST Data to a CGI Program</title>
</head>
<body>
<h1>Sending POST Data to a CGI Program</h1>
<form
    action="http://www.apl.jhu.edu/cgi-instruct/hall/CgiPost">
    Enter some data:
    <input type="text" name="TextField1"><br>
    Enter some more:
    <input type="text" name="TextField2"><br>
    <input type="submit" value="Send It">
</form>
</body>
</html>