1 Other HTTP Request Methods

1.1 The HEAD Method

A HEAD request is just like a GET request, except it asks the server to return the response
headers only, and not the actual resource (i.e. no message body). This is useful to check
characteristics of a resource without actually downloading it, thus saving bandwidth. Use
HEAD when you don't actually need a file's contents.

The response to a HEAD request must never contain a message body, just the status line and
headers.

1.2 The POST Method

A POST request is used to send data to the server to be processed in some way, like by a CGI
script. A POST request is different from a GET request in the following ways:

There’s a block of data sent with the request, in the message body. There are usually extra
headers to describe this message body, like Content-Type: and Content-Length:. The
request URI is not a resource to retrieve; it’s usually a program to handle the data you’re
sending. The HTTP response is normally program output, not a static file. The most com-
mon use of POST, by far, is to submit HTML form data to CGI scripts. In this case, the
Content-Type: header is usually

    application/x-www-form-urlencoded

and the Content-Length: header gives the length of the URL-encoded form data (see Sec-
tion 2 for how to encoding URLs). The CGI script receives the message body through its
standard input, and decodes it. Here’s a typical form submission, using POST:

    POST /path/script.cgi HTTP/1.0
    From: frog@jmarshall.com
    User-Agent: HTTPTool/1.0
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 32

    home=Cosby&favorite+flavor=flies
You can use a POST request to send whatever data you want, not just form submissions. Just make sure the sender and the receiving program agree on the format.

The GET method can also be used to submit forms. The form data is URL-encoded and appended to the request URI. Here are more details.

If you’re writing HTTP servers that support CGI scripts, you should read the NCSA’s CGI Definition if you haven’t already, especially which environment variables you need to pass to the scripts.

2  URL-encoding

HTML form data is usually URL-encoded to package it in a GET or POST submission. In a nutshell, here’s how you URL-encode the name-value pairs of the form data:

1. Convert all “unsafe” characters in the names and values to “%xx”, where “xx” is the ascii value of the character, in hex. “Unsafe” characters include =, &, %, +, non-printable characters, and any others you want to encode. For simplicity, you might encode all non-alphanumeric characters.
2. Change all spaces to plusses.
3. String the names and values together with = and &, like

   name1=value1&name2=value2&name3=value3

This string is your message body for POST submissions, or the query string for GET submissions. For example, if a form has a field called name that’s set to Lucy, and a field called neighbors that’s set to Fred & Ethel, the URL-encoded form data would be

   name=Lucy&neighbors=Fred+%26+Ethel

with a length of 34. Technically, the term “URL-encoding” refers only to step 1 above; it’s defined in RFC 2396, section 2.4. Commonly, the term refers to this entire process of packaging form data into one long string.

3  More Headers

3.1 Referer: Header

The Referer: header is used for specifying the page that contained the link that lead to the request. It allows you to see who is pointing to you. Web servers can save the value of this
header for each request in the logs, so that a site can get a sense of where they are getting hits from.

Note: Referer is misspelled in the specification, but we’re stuck with it forever.

3.2 Accept: Header

The Accept: header specifies the MIME types the client can handle, for example image/png. The server can send whatever MIME type, and the client will do the best it can. Some browsers just send */* and sort it out on the client side when the data arrives.

3.3 Accept-Encoding: Header

By the Accept-Encoding: header, the client can indicate which encoding they can deal with. For instance,

    Accept-Encoding: gzip

where gzip encoding means that the client can process gzip-compressed HTML files thus the server may compress those files first before responding so that a significant bandwidth savings may be obtained, and better performance is achieved.

4 Virtual Hosts

Suppose you want to host 20 web sites on one machine. Hosting a web site is not very demanding, especially for plain web pages. You typically run out of networking bandwidth before you run out of CPU, memory, etc. Therefore, using one machine for many sites is common.

Suppose you want http://foo.com/, http://bar.com/, etc., all on the one machine. Each of them is called a virtual host. The problem is that how the server could know which virtual host it is for when it gets a request GET / HTTP/1.0. The solution is that with HTTP/1.1, the client includes a Host: header in the request, specifying which host it is visiting.

5 Persistent Connections

Consider a typical web page with one body of HTML and 20 little GIF buttons. Obviously with HTTP/1.0’s request-response-close style, 21 separate connections are required to transfer
these files. Each TCP and HTTP connection takes time to set up so this style is extremely inefficient. One solution to this problem is to allow one single HTTP connection to serve multiple request-response sessions before it is closed. HTTP/1.1 adds Connection: header for this purpose.

In more details, one of the following is added to the request:

- Connection: keep-alive: client request to keep connection open.
- Connection: close: client request that the server close connection after response (like HTTP 1.0).

With HTTP 1.1, keep-alive is the default choice. The server indicates if it is closing the connection with Connection: close in the response header.

What’s more, with the so-called persistent connections, HTTP/1.1 allows clients to send multiple requests on a same connection, even before receiving responses. This can be a great speedup, avoiding the latency*2 cost of a typical request/response system by not waiting for the response. This mechanism is similar to the way CPUs process instructions in a pipelining style.

### 6 Chunked Transfer Encoding

As we mentioned before, the Content-length: header is used to tell the length of message body in the responses. However this requirement is costly for server since sometimes it cannot determine that value in advance and has to wait before sending data. The solution to it is to send data back in chunks. A chunked message body contains a series of chunks, followed by a line with "0" (zero), followed by optional footers (just like headers), and a blank line. Each chunk consists of two parts:

- a line with the size of the chunk data, in hex, possibly followed by a semicolon and extra parameters you can ignore (none are currently standard), and ending with CRLF.
- the data itself, followed by CRLF.

So a chunked response might look like the following:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
```
With HTTP/1.1, a client must be capable of getting the response back chunked. For this reason, it’s easier to use HTTP/1.0 for simple web client software.

### 7 Example: EchoHttpServer

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

public class EchoHttpServer {
    public static void main(String args[]) {
        try{
            ServerSocket serverSocket = new ServerSocket(PORT);
            Socket socket = null;
            BufferedReader reader = null;
            PrintWriter writer = null;
            String request = null;
            String response = null;
            String line = null;

            while ((socket=serverSocket.accept()) != null) {
                reader = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
                writer = new PrintWriter(socket.getOutputStream());

                request = "";

                while (true) {
                    line = reader.readLine();
                    request += line + EOL;
                    if (line.length() == 0) {
                        break;
                    }
                }
            }
        }
    }
}
```

Request from Netscape Communicator 4.8 on Windows

GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.8 [en] (Windows NT 5.0; U)
Pragma: no-cache
Host: localhost:8181
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Request from Internet Explorer 6.0 on Windows

GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: localhost:8181
Connection: Keep-Alive

Request from Internet Explorer 6.0 on Windows

GET / HTTP/1.1
Host: localhost:8181
User-Agent: Mozilla/5.0 (Windows; U; en-US) Gecko/20031007 Firebird/0.7
Accept: text/xml,text/html;q=0.2, */*;q=0.1
Accept-Language: en-us, en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8;q=0.7, *;q=0.7
Keep-Alive: 300
Connection: keep-alive