How the web works: HTTP and CGI explained

Introduction

About this tutorial

This is an attempt to give a basic understanding of how the web works and was written because I saw so many articles on various news groups that plainly showed that people needed to learn this. Not knowing of any one place to find this info, I decided to collect it into an article. Hope you find it useful!

It covers the HTTP protocol, which is used to transmit and receive web pages, as well as some server workings and scripting technologies. It is assumed that you already know how to make web pages and preferably some HTML as well.. It is also assumed that you have some basic knowledge of URLs. (A *URL* is the address of a document, what you need to be able to get hold of the document.)

I'm not entirely happy with this, so feedback would be very welcome. If you're not really a technical person and this tutorial leaves you puzzled or does not answer all your questions I'd very much like to hear about it. Corrections and opinions are also welcome.

Some background

When you browse the web the situation is basically this: you sit at your computer and want to see a document somewhere on the web, to which you have the URL.

Since the document you want to read is somewhere else in the world and probably very far away from you some more details are needed to make it available to you. The first detail is your browser. You start it up and type the URL into it (at least you tell the browser somehow where you want to go, perhaps by clicking on a link).

However, the picture is still not complete, as the browser can't read the document directly from the disk where it's stored if that disk is on another continent. So for you to be able to read the document the computer that contains the document must run a web server. A *web server* is a just a computer program that listens for requests from browsers and then execute them.

So what happens next is that the browser contacts the server and requests that the server deliver the document to it. The server then gives a response which contains the document and the browser happily displays this to the user. The server also tells the browser what kind of document this is (HTML file, PDF file, ZIP file etc) and the browser then shows the document with the program it was configured to use for this kind of document.

The browser will display HTML documents directly, and if there are references to images, Java applets, sound clips etc in it and the browser has been set up to display these it will request these also from the servers on which they reside. (Usually the same server as the document, but not always.) It's worth noting that these will be separate requests, and add additional load to the server and network. When the user follows another link the whole sequence starts anew.

These requests and responses are issued in a special language called *HTTP*, which is short for HyperText Transfer Protocol. What this article basically does is describe how this works. Other common protocols that work in similar ways are FTP and Gopher, but there are also protocols that work in completely different ways. None of these are covered here, sorry. (There is a link to some more details about FTP in the references.)

It's worth noting that HTTP only defines what the browser and web server say to each other, not how they communicate. The actual work of moving bits and bytes back and forth across the network is done by TCP and IP, which are also used by FTP and Gopher (as well as most other internet protocols).

When you continue, note that any software program that does the same as a web browser (ie: retrieve documents from servers) is called a *client* in network terminology and a *user agent* in web terminology. Also note that the server is properly the server program, and not the computer on which the server is an application program. (Sometimes called the *server machine*.)

What happens when I follow a link?

Step 1: Parsing the URL

The first thing the browser has to do is to look at the URL of the new document to find out how to get hold of the new document. Most URLs have this basic form: "protocol://server/request-URI". The protocol part describes how to tell the server which document the you want and how to retrieve it. The server part tells the browser which server to contact, and the request-URI is the name used by the web server to identify the document. (I use the term request-URI since it's the one used by the HTTP standard, and I can't think of anything else that is general enough to not be misleading.)

Step 2: Sending the request

Usually, the protocol is "http". To retrieve a document via HTTP the browser transmits the following request to the server: "GET /request-URI HTTP/version", where version tells the server which HTTP version is used. (Usually, the browser includes some more information as well. The details are covered later.)

One important point here is that this request string is all the server ever sees. So the server doesn't care if the request came from a browser, a link checker, a validator, a search engine robot or if you typed it in manually. It just performs the request and returns the result.

Step 3: The server response

When the server receives the HTTP request it locates the appropriate document and returns it. However, an HTTP response is required to have a particular form. It must look like this:

```
HTTP/[VER] [CODE] [TEXT]
Field1: Value1
Field2: Value2

...Document content here...
```

The first line shows the HTTP version used, followed by a three-digit number (the HTTP status code) and a reason phrase meant for humans. Usually the code is 200 (which basically means that all is well) and the phrase "OK". The first line is followed by some lines called the header, which contains information about the document. The header ends with a blank line, followed by the document content. This is a typical header:

```
HTTP/1.0 200 OK
Server: Netscape-Communications/1.1
Date: Tuesday, 25-Nov-97 01:22:04 GMT
```

```
Last-modified: Thursday, 20-Nov-97 10:44:53 GMT
Content-length: 6372
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
...followed by document content...
```

We see from the first line that the request was successful. The second line is optional and tells us that the server runs the Netscape Communications web server, version 1.1. We then get what the server thinks is the current date and when the document was modified last, followed by the size of the document in bytes and the most important field: "Content-type".

The content-type field is used by the browser to tell which format the document it receives is in. HTML is identified with "text/html", ordinary text with "text/plain", a GIF is "image/gif" and so on. The advantage of this is that the URL can have any ending and the browser will still get it right.

An important concept here is that to the browser, the server works as a black box. Ie: the browser requests a specific document and the document is either returned or an error message is returned. How the server produces the document remains unknown to the browser. This means that the server can read it from a file, run a program that generates it, compile it by parsing some kind of command file or (very unlikely, but in principle possible) have it dictated by the server administrator via speech recognition software. This gives the server administrator great freedom to experiment with different kinds of services as the users don't care (or even know) how pages are produced.

What the server does

When the server is set up it is usually configured to use a directory somewhere on disk as its root directory and that there be a default file name (say "index.html") for each directory. This means that if you ask the server for the file "/" (as in "http://www.domain.tld/") you'll get the file index.html in the server root directory. Usually, asking for "/foo/bar.html" will give you the bar.html file from the foo directory directly beneath the server root.

Usually, that is. The server can be set up to map "/foo/" into some other directory elsewhere on disk or even to use server-side programs to answer all requests that ask for that directory. The server does not even have to map requests onto a directory structure at all, but can use some other scheme.

HTTP versions

So far there are three versions of HTTP. The first one was HTTP/0.9, which was truly primitive and never really specified in any standard. This was corrected by HTTP/1.0, which was issued as a standard in RFC 1945. (See references). HTTP/1.0 is the version of HTTP that is in common use today (usually with some 1.1 extensions), while HTTP/0.9 is rarely, if ever, used by browsers. (Some simpler HTTP clients still use it since they don't need the later extensions.)

RFC 2068 describes HTTP/1.1, which extends and improves HTTP/1.0 in a number of areas. Very few browsers support it (MSIE 4.0 is the only one known to the author), but servers are beginning to do so.

The major differences are a some extensions in HTTP/1.1 for authoring documents online via HTTP and a feature that lets clients request that the connection be kept open after a request so that it does not have to be reestablished for the next request. This can save some waiting and server load if several requests have to be issued quickly.

This document describes HTTP/1.0, except some sections that cover the HTTP/1.1 extensions. Those will be explicitly labeled.

The request sent by the client

The shape of a request

Basically, all requests look like this:

```
[METH] [REQUEST-URI] HTTP/[VER]
[fieldname1]: [field-value1]
[fieldname2]: [field-value2]

[request body, if any]
```

The METH (for request method) gives the request method used, of which there are several, and which all do different things. The above example used GET, but below some more are explained. The REQUEST-URI is the identifier of the document on the server, such as "/index.html" or whatever. VER is the HTTP version, like in the response. The header fields are also the same as in the server response.

The request body is only used for requests that transfer data to the server, such as POST and PUT. (Described below.)

GETting a document

There are several request types, with the most common one being GET. A GET request basically means "send me this document" and looks like this: "GET document_path HTTP/version". (Like it was described above.) For the URL "http://www.yahoo.com/" the document_path would be "/", and for "http://www.w3.org/Talks/General.html" it is "/Talks/General.html".

However, this first line is not the only thing a user agent (*UA*) usually sends, although it's the only thing that's really necessary. The UA can include a number of *header fields* in the request to give the server more information. These fields have the form "fieldname: value" and are all put on separate lines after the first request line.

Some of the header fields that can be used with GET are:

User-Agent
: This is a string identifying the user agent. An English version of Netscape 4.03 running under Windows NT would send "Mozilla/4.03 [en] (WinNT; I ;Nav)". (Mozilla is the old name for Netscape. See the references for more details.)

Referer
: The referer field (yes, it's misspelled in the standard) tells the server where the user came from, which is very useful for logging and keeping track of who links to ones pages.

If-Modified-Since
: If a browser already has a version of the document in its cache it can include this field and set it to the time it retrieved that version. The server can then check if the document has been modified since the browser last downloaded it and send it again if necessary. The whole point is of course that if the document *hasn't* changed, then the server can just say so and save some waiting and network traffic.

From
: This header field is a spammers dream come true: it is supposed to contain the email address of whoever controls the user agent. Very few, if any, browsers use it, partly because of the threat from spammers. However, web robots should use it, so that webmasters can contact the people responsible for the robot should it misbehave.

Authorization
: This field can hold username and password if the document in question requires authorization to be accessed.

To put all these pieces together: this is a typical GET request, as issued by my browser (Opera):

```
GET / HTTP/1.0
User-Agent: Mozilla/3.0 (compatible; Opera/3.0; Windows 95/NT4)
Accept: */*
Host: birk105.studby.uio.no:81
```

HEAD: checking documents

One may somtimes want to see the headers returned by the server for a particular document, without actually downloading the document. This is exactly what the HEAD request method provides. HEAD looks and works exactly like GET, only with the difference that the server only returns the headers and not the document content.

This is very useful for programs like link checkers, people who want to see the response headers (to see what server is used or to verify that they are correct) and many other kinds of uses.

Playing web browser

You can actually play web browser yourself and write HTTP requests directly to web servers. This can be done by telnetting to port 80, writing the request and hitting enter twice, like this:

```
larsga - tyrfing>telnet www.w3.org 80
Trying 18.23.0.23...
Connected to www.w3.org.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 17 Feb 1998 22:24:53 GMT
Server: Apache/1.2.5
Last-Modified: Wed, 11 Feb 1998 18:22:22 GMT
ETag: "2c3136-23c1-34e1ec5e"
Content-Length: 9153
Accept-Ranges: bytes
Connection: close
Content-Type: text/html; charset=ISO-8859-1

Connection closed by foreign host.
larsga - tyrfing>
```

However, this works best under Unix as the Windows telnet clients I've used are not very suitable for this (and hard to set up so that it works). Instead you can use HTTPTest, a CGI script I've linked to in the references.

The response returned by the server

Outline

What the server returns consists of a line with the status code, a list of header fields, a blank line and then the requested document, if it is returned at all. Sort of like this:

```
HTTP/1.0 code text
Field1: Value1
Field2: Value2

...Document content here...
```

The status codes

The status codes are all three-digit numbers that are grouped by the first digit into 5 groups. The reason phrases given with the status codes below are just suggestions. Server can return any reason phrase they wish.

1xx: Informational

No 1xx status codes are defined, and they are reserved for experimental purposes only.

2xx: Successful

Means that the request was processed successfully.

200 OK
: Means that the server did whatever the client wanted it to, and all is well.

Others
: The rest of the 2xx status codes are mainly meant for script processing and are not often used.

3xx: Redirection

Means that the resource is somewhere else and that the client should try again at a new address.

301 Moved permanently
: The resource the client requested is somewhere else, and the client should go there to get it. Any links or other references to this resource should be updated.

302 Moved temporarily
: This means the same as the 301 response, but links should now not be updated, since the resource may be moved again in the future.

304 Not modified
: This response can be returned if the client used the if-modified-since header field and the resource has not been modified since the given time. Simply means that the cached version should be displayed for the user.

4xx: Client error

Means that the client screwed up somehow, usually by asking for something it should not have asked for.

400: Bad request
: The request sent by the client didn't have the correct syntax.
401: Unauthorized
: Means that the client is not allowed to access the resource. This may change if the client retries with an authorization header.
403: Forbidden
: The client is not allowed to access the resource and authorization will not help.
404: Not found
: Seen this one before? :) It means that the server has not heard of the resource and has no further clues as to what the client should do about it. In other words: dead link.

5xx: Server error

This means that the server screwed up or that it couldn't do as the client requested.

500: Internal server error
: Something went wrong inside the server.
501: Not implemented
: The request method is not supported by the server.
503: Service unavailable
: This sometimes happens if the server is too heavily loaded and cannot service the request. Usually, the solution is for the client to wait a while and try again.

The response header fields

These are the header fields a server can return in response to a request.

Location
: This tells the user agent where the resource it requested can be found. The value is just the URL of the new resource.
Server
: This tells the user agent which web server is used. Nearly all web servers return this header, although some leave it out.
Content-length
: This gives the size of the resource, in bytes.
Content-type
: This describes the file format of the resource.
Content-encoding
: This means that the resource has been coded in some way and must be decoded before use.
Expires
: This field can be set for data that are updated at a known time (for instance if they are generated by a script). It is used to prevent browsers from caching the resource beyond the given date.
Last-modified
: This tells the browser when the resource was last modified. Can be useful for mirroring, update notification etc.

Caching: agents between the server and client

The browser cache

You may have noticed that when you go back to a page you've looked at not too long before the page loads much quicker. That's because the browser stored a local copy of it when it was first downloaded. These local copies are kept in what's called a cache. Usually one sets a maximum size for the cache and a maximum caching time for documents.

This means that when a new page is visited it is stored in the cache, and if the cache is full (near the maximum size limit) some document that the browser considers unlikely to be visited again soon is deleted to make room. Also, if you go to a page that is stored in the cache the browser may find that you've set 7 days as a the maximum storage time and 8 days have now passed since the last visit, so the page needs to be reloaded.

Exactly how caches work differ between browsers, but this is the basic idea, and it's a good one because it saves both time for the user and network traffic. There are also some HTTP details involved, but they will be covered later.

Proxy caches

Browser caches are a nice feature, but when many users browse from the same site one usually ends up storing the same document in many different caches and refreshing it over and over for different uses. Clearly, this isn't optimal.

The solution is to let the users share a cache, and this is exactly what proxy caches are all about. Browsers still have their local caches, but HTTP requests for documents not in the browser cache are not sent to the server any more, instead they are sent to the proxy cache. If the proxy has the document in its cache it will just return the document (like the browser cache would), and if it doesn't it will submit the request on behalf of the browser, store the result and relay it to the browser.

So the proxy is really a common cache for a number of users and can reduce network traffic rather dramatically. It can also skew log-based statistics badly. :)

A more advanced solution than a single proxy cache is a hierarchy of proxy caches. Imagine a large ISP may have one proxy cache for each part of the country and set up each of the regional proxies to use a national proxy cache instead of going directly to the source web servers. This solution can reduce network traffic even further. More detail on this is linked to in the references.

Server-side programming

What is it and why do it?

Server-side scripts or programs are simply programs that are run on the web server in response to requests from the client. These scripts produce normal HTML (and sometimes HTTP headers as well) as output which is then fed back to the client as if the client had requested an ordinary page. In fact, there is no way for the client software to tell whether scripting has been used or not.

Technologies such as JavaScript, VBScript and Java applets all run in the client and so are not examples of this. There is a major difference between server-side and client-side scripting as the client and server are usually different computers. So if all the data the program needs are located on the server it may make sense to use server-side scripting instead of client-side. (There is also the problem that the client may not have a browser that supports the scripting technology or it may be turned off.) If the program and user need to interact often client-side scripting is probably best, to reduce the number of requests sent to the server.

So: in general, if the program needs a lot of data and infrequent interactions with the server server-side scripting is probably best. Applications that use less data and more interaction are best put in the client. Also, applications that gather data over time need to be on the server where the data file is kept.

An example of the first would be search engines like Altavista. Obviously it's not feasible to download all the documents Altavista has collected to search in them locally. An example of the last would be a simple board game. No data are needed and having to send a new request to the server for each move you make quickly gets tedious.

There is one kind of use that has been left out here: what do you do when you want a program to work on data with a lot of interaction with the user? There is no good solution right now, but there is one on the horizon called XML. (See the the references for more info.)

How it works

The details of how server-side scripting works vary widely with the technique used (and there are loads of them). However, some things remain the same. The web server receives a request just like any other, but notes that this URL does not map to a flat file, but instead somehow to a scripting area.

The server then starts the script, feeding it all the information contained in the request headers and URL. The script then runs and produces as its output the HTML and HTTP headers to be returned to the client, which the server takes care of.

CGI

CGI (Common Gateway Interface) is a way for web servers and server-side programs to interact. CGI is completely independent of programming language, operating system and web server. Currently it is the most common server-side programming technique and it's also supported by almost every web server in existence. Moreover, all servers implement it in (nearly) the same way, so that you can make a CGI script for one server and then distribute it to be run on any web server.

Like I wrote above, the server needs a way to know which URLs map to scripts and which URLs just map to ordinary HTML files. For CGI this is usually done by creating CGI directories on the server. This is done in the server setup and tells the server that all files in a particular top-level directory are CGI scripts (located somewhere on the disk) to be executed when requested. (The default directory is usually /cgi-bin/, so one can tell that URLs like this: http://www.varsity.edu/cgi-bin/search point to a CGI script. Note that the directory can be called anything.) Some servers can also be set up to not use CGI directories and instead require that all CGI programs have file names ending in .cgi.

CGI programs are just ordinary executable programs (or interpreted programs written in, say, Perl or Python, as long as the server knows how to start the program), so you can use just about any programming language you want. Before the CGI program is started the web server sets a number of environment variables that contain the information the web server received in the request. Examples of this are the IP address of the client, the headers of the request etc. Also, if the URL requested contained a ?, everything after the ? is put in an environment variable by itself.

This means that extra information about the request can be put into the URL in the link. One way this is often used is by multi-user hit counters to tell which user was hit this time. Thus, the user can insert an image on his/her page and have the SRC attribute be a link to the CGI script like this: SRC="http://stats.vendor.com/cgi-bin/counter.pl?username". Then the script can tell which user was hit and increment and display the correct count. (Ie: that of Peter and not Paul.)

The way the CGI returns its output (HTTP headers and HTML document) to the server is exceedingly simple: it writes it to standard out. In other words, in a Perl or Python script you just use the print statement. In C you use printf or some equivalent (C++ uses cout <<) while Java would use System.out.println.

More information on CGI is available in the references.

Other techniques

CGI is certainly not the only way to make server-side programs and has been much criticized for inefficiency. This last claim has some weight since the CGI program has to be loaded into memory and reexecuted from scratch each time it is requested.

A much faster alternative is programming to the server API itself. Ie: making a program that essentially becomes a part of the server process and uses an API exposed by the server. The problem with this technique is that the API is of course server-dependent and that if you use C/C++ (which is common) programming errors can crash the whole server.

The main advantage of server API programming is that it is much faster since when the request arrives the program is already loaded into memory together with whatever data it needs.

Some servers allow scripting in crash-proof languages. One example is AOLServer, which uses tcl. There are also modules available for servers like Apache, which let you do your server API programming in Perl or Python, which effectively removes the risk of programming errors crashing the server.

There are also lots and lots of proprietary (and non-proprietary) scripting languages and techniques for various web servers. Some of the best-known are ASP, MetaHTML and PHP3.

Submitting forms

The most common way for server-side programs to communicate with the web server is through ordinary HTML forms. The user fills in the form and hits the submit button, upon which the data are submitted to the server. If the form author specified that the data should be submitted via a GET request the form data are encoded into the URL, using the ? syntax I described above. The encoding used is in fact very simple. If the form consists of the fields name and email and the user fills them out as Joe and joe@hotmail.com the resulting URL looks like this: http://www.domain.tld/cgi-bin/script?name=joe&email=joe@hotmail.com.

If the data contain characters that are not allowed in URLs, these characters are URL-encoded. This basically means that the character (say ~) is replaced with a % followed by its two-digit ASCII number (say %7E). The details are available in RFC 1738 about URLs, which is linked to in the references.

POST: Pushing data to the server

GET is not the only way to submit data from a form, however. One can also use POST, in which case the request contains both headers and a body. (This is just like the response from the server.) The body is then the form data encoded just like they would be on the URL if one had used GET.

Primarily, POST should be used when the request causes a permanent change of state on the server (such as adding to a data list) and GET when this is not the case (like when doing a search).

If the data can be long (more than 256 characters) it is a bit risky to use GET as the URL can end up being snipped in transit. Some OSes don't allow environment variables to be longer than 256 characters, so the environment variable that holds the ?-part of the request may be silently truncated. This problem is avoided with POST as the data are then not pushed through an environment variable.

Some scripts that handle POST requests cause problems by using a 302 status code and Location header to redirect browsers to a confirmation page. However, according to the standard, this does not mean that the browser should retrieve the referenced page, but rather that it should resubmit the data to the new destination. See the references for details on this.

More details

Content negotiation

Imagine that you've just heard of the fancy new PNG image format, and want to use this for your images instead of the older GIF. However, GIF is supported by all browsers with some self-respect, while PNG is only supported by the most modern ones. So if you replace all your GIFs with PNGs, only a limited number of users will be able to actually see your images.

To avoid this one could set up a link to an imaginary location (lets call it /foo/imgbar). Then, when browsers request this image after seeing the location in an IMG element they will add an Accept header field to their request. (Most browsers do this with all requests.)

The accept header would then be set to "image/*, image/png", which would then mean: I prefer PNG images, but if you can't deliver that, then just send me any image you've got.

That would be in a standards-based world. We do, however, live in a decidedly non-standard world. Most browsers simply send the header "Accept: */*" which doesn't really tell the server anything at all. This may change in the future, but for now this is a nice, but useless feature.

Cookies

An inconvenient side of the HTTP protocol is that each request essentially stands on its own and is completely unrelated to any other request. This is inconvenient for scripting, because one may want to know what a user has done before this last request was issued. As long as plain HTTP is used there is really no way to know this. (There are some tricks, but they are ugly and expensive.)

To illustrate the problem: imagine a server that offers a lottery. People have to view ads to participate in the lottery, and those who offer the lottery don't want people to be able to just reload and reload until they win something. This can be done by not allowing subsequent visits from a single IP address (ie: computer) within a certain time interval. However, this causes problems as one doesn't really know if it's the same user.

People who dial up via modems to connect to the internet are usually given a new IP address from a pool of available addresses each time, which means that the same IP address may be given to two different users within an hour if the first one disconnects and another user is given the same IP later. (Also: on larger UNIX installations many people are usually using the same computer simultaneously from different terminals.)

The solution proposed by Netscape is to use magic strings called cookies. (The specification says they are called this "for no compelling reason", but the name has a long history.) The server returns a "Set-cookie" header that gives a cookie name, expiry time and some more info. When the user returns to the same URL (or some other URL, this can be specified by the server) the browser returns the cookie if it hasn't expired.

This way, our imaginary lottery could set a cookie when the user first tries the lottery and set it to expire when the user can return. The lottery script could then check if the cookie is delivered with the request and if so just tell the user to try again later. This would work just fine if browsers didn't allow users to turn off cookies...

Cookies can also be used to track the path of a user through a web site or to give pages a personalized look by remembering what the user has done before. Rather useful, but there are some privacy issues involved here.

Server logs

Most servers (if not all) create logs of their usage. This means that every time the server gets a request it will add a line in its log, which gives information about the request. Below an excerpt from an actual log:

```
rip.axis.se - - [04/Jan/1998:21:24:46 +0100] "HEAD /ftp/pub/software/ HTTP/1.0" 200 6312 - "Mozilla/4.04 [en] (WinNT; I)"
tide14.microsoft.com - - [04/Jan/1998:21:30:32 +0100] "GET /robots.txt HTTP/1.0" 304 158 - "Mozilla/4.0 (compatible; MSIE 4.0; MSIECrawler; Wi
microsnot.HIP.Berkeley.EDU - - [04/Jan/1998:22:28:21 +0100] "GET /cgi-bin/wwwbrowser.pl HTTP/1.0" 200 1445 "http://www.ifi.uio.no/~larsga/down
isdn69.ppp.uib.no - - [05/Jan/1998:00:13:53 +0100] "GET /download/RFCsearch.html HTTP/1.0" 200 2399 "http://www.kvarteret.uib.no/~pas/" "Mozil
isdn69.ppp.uib.no - - [05/Jan/1998:00:13:53 +0100] "GET /standard.css HTTP/1.0" 200 1064 - "Mozilla/4.04 [en] (Win95; I)"
```

This log is in the extended common log format, which is supported by most web servers. The first hit is from Netscape 4.04, the second from some robot version of MSIE 4.0, while three to five are again from Netscape 4.04. (Note that the MSIECrawler got a 304 response, which means that if used the If-modified-since header.)

A server log can be useful when debugging applications and scripts or the server setup. It can also be run through a log analyzer, which can create various kinds of usage reports. One should however be aware that these reports are not 100% accurate due to the use of caches.

A sample HTTP client

Just to illustrate, in case some are interested, here is a simple HTTP client written as a Python function that takes a host name and path as parameters and issues a GET request, printing the returned results. (This can be made even simpler by using the Python URL library, but that would make the example useless.)

```
# Simple Python function that issues an HTTP request

from socket import *

def http_req(server, path):

    # Creating a socket to connect and read from
    s=socket(AF_INET,SOCK_STREAM)

    # Finding server address (assuming port 80)
    adr=(gethostbyname(server),80)

    # Connecting to server
    s.connect(adr)

    # Sending request
    s.send("GET "+path+" HTTP/1.0\n\n")

    # Printing response
    resp=s.recv(1024)
    while resp!="":
        print resp
        resp=s.recv(1024)
```

Here is the server log entry that resulted from this call: `http_req("birk105.studby.uio.no","/")`

```
birk105.studby.uio.no - - [26/Jan/1998:12:01:51 +0100] "GET / HTTP/1.0" 200 2272 - -
```

The "- -"s at the end are the referrer and user-agent fields, which are empty because the request did not contain this information.

Authentication

A server can be set up to disallow access to certain URLs unless the user can confirm his/her identity. This is usually done with a user-name/password combination which is specified in the setup, but there are other methods as well.

When such a page is requested the server generally returns a "401 Not authorized" status code as mentioned above. The browser will then usually prompt the user for a user name and password, which the user supplies (if it is known!). The browser then tries again, this time adding an "Authorization" header with the user name and password as the value.

If this is accepted by the server the resource is returned just like in an ordinary request. If not, the server again responds with a 401 status code.

Server-side HTML extensions

Some servers, such as the Roxen and MetaHTML servers, allow users to embed non-HTML commands within their HTML files. These commands are then processed when the file is requested and generates the HTML that is sent to the client. This can be used to tailor the page contents and/or to access a database to deliver contents from it.

What these languages have in common is usually that they are tied to a single server and that they produce a combination of ordinary HTTP headers and HTML contents as their output.

This is important because it means that the client does not notice what is going on at all, and so can use any browser.

Authoring/maintainance: HTTP extensions for this

HTTP/1.0 only defined the GET, HEAD and POST methods and for ordinary browsing this is enough. However, one may want to use HTTP to edit and maintain files directly on the server, instead of having to go through an FTP server as is common today. HTTP/1.1 adds a number of new methods for this.

These are:

PUT
: PUT uploads a new resource (file) to the server under the given URL. Exactly what happens on the server is not defined by the HTTP/1.1 spec, but authoring programs like Netscape Composer use PUT to upload a file to the web server and store it there. PUT requests should not be cached.

DELETE
: Well, it's obvious, isn't it? The DELETE method requests the server to delete the resource identified by the URL. On UNIX systems this can fail if the server is not allowed to delete the file by the file system.

META HTTP-EQUIV

Not all web servers are made in such a way that it is easy to set, say the Expires header field or some other header field you might want to set for one particular file. There is a way around that problem. In HTML there is an element called META that can be used to set HTTP header fields. It was really supposed to be parsed by the web server and inserted into the response headers, but very few servers actually do this, so browsers have started implementing it instead. It's not supported in all browsers, but it can still be of some help.

The usage is basically this: put it in the HEAD-element of the HTML file and make it look like this:

```
<META HTTP-EQUIV="header field name" CONTENT="field value">
```

The Host header field

Many web hotels let a single physical machine serve what to the user looks like several different servers. For instance, http://www.foo.com/, http://www.bar.com/ and http://www.baz.com/ could all very well be served from the same web server. Still, the user would see different pages by going to each of the different URLs. To enable this, an extension to the HTTP protocol was needed to let the web server know which of these different web servers the user wanted to access.

The solution is the Host header field. When the user requests http://www.bar.com/ the web server will receive a header set to "Host: www.bar.com" and thus know which top page to deliver. It will also usually create separate logs for the different virtual servers.

Answers to some common questions

Can I prevent people from seeing my HTML source?

No. Once you let people issue an HTTP request that retrieves your document they can do whatever they want with it. No HTML tag can stop this because one could just use a Python program like the one I included above and let it save the HTML to file without even looking at it. Nor is any such tag recognized by browsers.

Some people try obfuscating the HTML by putting all the markup on a single line, but this is easily fixed by running the HTML through a pretty-printer.

In short: give it up. There's no way to protect the information in your HTML and there's no reason to try unless you put vital information in hidden form controls, and you shouldn't do that anyway.

Can I prevent people from stealing my images?

The answer is the same for this as it is for HTML: you can't do it. You can watermark them and prove that they belong to you and enter comments that say this in clear text, but it still doesn't keep people from issuing an HTTP request and saving the image.

Files in format X are not displayed correctly, why not?

This is a rather common problem for some formats: you put up a file on your web server in a new format you've never used before and when you try to download it it comes up in your browser as plain text (where it looks like complete gobbledygook) instead of being saved, played or shown or whatever.

The problem is probably that the server does not know what kind of file this is and signals it as Content-type: text/plain, which the browser happily displays as if it really were text. The solution is to configure the server so that it knows what kind of file this is and signals it correctly. The references link to the list of registered MIME types.

One thing to note here is that MSIE 3.0 (4.0?) does not honour the content-type given by the server, but instead tries to guess which format the file is in and treat it

accordingly. This is a blatant violation of the HTTP specification on the part of Microsoft, but it's not the only standard violation they've committed and there's nothing to do about it except to be aware of it.

How can I pass a parameter to a web page?

If the web page is a plain HTML file: forget it. HTML files are just displayed and nothing more is ever done with them, so the concept of a parameter just doesn't apply.

What you *can* do is to use server-side scripting the way it is described elsewhere in this document.

How can I prevent browsers from caching my page/script?

This is done by setting the correct HTTP headers. If the Expiration header is set to a date/time in the past the output from the request will not be cached. (Note that HTTP requires browsers to keep the result in the browser history, so that going back to a non-cached page does not cause a new request.)

The expiration time can be set with a server-script or possibly by configuring the server correctly.

Should I include a slash (/) at the end of my URLs?

If the URL points to a directory and you want the server to list the contents or the index.html file: yes. (Remember: the URL may point to a file without an extension or the server may not map it to directory/file structure at all.) If you follow a URL like this one: http://www.garshol.priv.no/download the server will notice that someone requested a file that doesn't exist, but there is a directory with the same name. The server will then give a 301 response redirecting the client to the URL http://www.garshol.priv.no/download/ which the client will then try and succeed. It is worth noting here that these two are actually different URLs, which is why the server cannot return the contents of the first URL directly. (In fact, one could well argue that it shouldn't offer the redirect at all.)

All this is invisible to the user, but the user will have to wait a little longer and the server will have to work harder. So the best thing is to include the slash and avoid extra network traffic and server load.

Appendices

Explanations of some technical terms

API
: An Application Programming Interface is an interface exposed by a program, part of an operating system or programming language to other programs, so that the programs that use the API can exploit the features of the program that exposes the API. One example of this would be the AWT windowing library of Java that exposes an API that can be used to write programs with graphical user interfaces. APIs are only used by programs, they are not user interfaces.

TCP/IP
: The IP protocol (IP is short for Internet Protocol) is the backbone of the internet, the foundation on which all else is built. To be a part of the internet a computer must support IP, which is what is used for all data transfer on the internet. TCP is another protocl (Transport Control Protocol) that extends IP with features useful for most higher-level protocols such as HTTP. (Lots of other protocols also use TCP: FTP, Gopher, SMTP, POP, IMAP, NNTP etc.) Some protocols use UDP instead of TCP.

Acknowledgements

In closing I'd like to thank the following people who have helped me with this tutorial:

- Jelks Cabaniss, who provided early feedback on form and contents as well as encouragement.
- Alan J. Flavell, for detailed and highly useful criticism.
- Jukka Korpela, for help with the markup in this document and extensive criticism on form and content, as well as a couple of useful links.
- Christian Nybø, for suggesting that I mention telnetting to servers and making HTTPTest a little more prominent.
- Harald Joerg, who found a number of typos, and suggested some improvements in the organization.
- Bjørn Borud, for some hints on organization as well as a typo fix.
- Ingrid Melve for some useful links.
- Darren Moore for suggesting that I define some technical terms.
- Felipe Wersen for pointing out a typo.

References

Important specifications and official pages

- The W3C pages on HTTP.
- RFC 1945, the specification of HTTP 1.0.
- RFC 2068, the specification of HTTP 1.1.
- RFC 1738, which describes URLs.
- The magic cookie specification, from Netscape.

Proxy caches

- Web caching architecture, a guide for system administrators who want to set up proxy caches.
- A Distributed Testbed for National Information Provisioning, a project to set up a national US-wide cache system.

Various

- The Mozilla Museum
- The registered MIME types, from IANA.
- HTTPTest. Try sending HTTP requests to various servers and see the responses.
- An overview of most web servers available.
- The POST redirect problem.
- About the use of the word 'cookie' in computing.
- More information about XML.
- About FTP URLs.
- A short Norwegian intro to HTTP.

Last update 12.Oct.99 22:56, by Lars M. Garshol.