The Art Of Scripting HTTP Requests Using Curl

Related:
curl man page
Manual
FAQ

Background

This document assumes that you are familiar with HTML and general networking.

The increasing amount of applications moving to the web has made "HTTP Scripting" more frequently requested and wanted. To be able to automatically extract information from the web, to fake users, to post or upload data to web servers are all important tasks today.

Curl is a command line tool for doing all sorts of URL manipulations and transfers, but this particular document will focus on how to use it when doing HTTP requests for fun and profit. This documents assumes that you know how to invoke `curl --help` or `curl --manual` to get basic information about it.

Curl is not written to do everything for you. It makes the requests, it gets the data, it sends data and it retrieves the information. You probably need to glue everything together using some kind of script language or repeated manual invokes.

The HTTP Protocol

HTTP is the protocol used to fetch data from web servers. It is a simple protocol that is built upon TCP/IP. The protocol also allows information to get sent to the server from the client using a few different methods, as will be shown here.

HTTP is plain ASCII text lines being sent by the client to a server to request a particular action, and then the server replies a few text lines before the actual requested content is sent to the client.

The client, curl, sends an HTTP request. The request contains a method (like GET, POST, HEAD etc), a number of request headers and sometimes a request body. The HTTP server responds with a status line (indicating if things went well), response headers and most often also a response body. The "body" part is the plain data you requested, like the actual HTML or the image etc.

See the Protocol

Using curl's option `--verbose` (`-v` as a short option) will display what kind of commands curl sends to the server, as well as a few other informational texts.

`--verbose` is the single most useful option when it comes to debug or even understand the curl<->server interaction.

Sometimes even `--verbose` is not enough. Then `--trace` and `--trace-ascii` offer even more details as they show **everything** curl sends and receives. Use it like this:

```
curl --trace-ascii debugdump.txt http://www.example.com/
```

See the Timing

Many times you may wonder what exactly is taking all the time, or you just want to know the amount of milliseconds between two points in a transfer. For those, and other similar situations, the `--`

trace-time option is what you need. It will prepend the time to each trace output line:

```
curl --trace-ascii d.txt --trace-time http://example.com/
```

See the Response

By default curl sends the response to stdout. You need to redirect it somewhere to avoid that, most often that is done with -o or -O.

URL

Spec

The Uniform Resource Locator format is how you specify the address of a particular resource on the Internet. You know these, you have seen URLs like https://curl.se or https://example.com a million times. RFC 3986 is the canonical spec. And yeah, the formal name is not URL, it is URI.

Host

The host name is usually resolved using DNS or your /etc/hosts file to an IP address and that is what curl will communicate with. Alternatively you specify the IP address directly in the URL instead of a name.

For development and other trying out situations, you can point to a different IP address for a host name than what would otherwise be used, by using curl's --resolve option:

```
curl --resolve www.example.org:80:127.0.0.1 http://www.example.org/
```

Port number

Each protocol curl supports operates on a default port number, be it over TCP or in some cases UDP. Normally you do not have to take that into consideration, but at times you run test servers on other ports or similar. Then you can specify the port number in the URL with a colon and a number immediately following the host name. Like when doing HTTP to port 1234:

```
curl http://www.example.org:1234/
```

The port number you specify in the URL is the number that the server uses to offer its services. Sometimes you may use a proxy, and then you may need to specify that proxy's port number separately from what curl needs to connect to the server. Like when using an HTTP proxy on port 4321:

```
curl --proxy http://proxy.example.org:4321 http://remote.example.org/
```

User name and password

Some services are setup to require HTTP authentication and then you need to provide name and password which is then transferred to the remote site in various ways depending on the exact authentication protocol used.

You can opt to either insert the user and password in the URL or you can provide them separately:

```
curl http://user:password@example.org/
```

or

```
curl -u user:password http://example.org/
```

You need to pay attention that this kind of HTTP authentication is not what is usually done and requested by user-oriented websites these days. They tend to use forms and cookies instead.

Path part

The path part is just sent off to the server to request that it sends back the associated response. The path is what is to the right side of the slash that follows the host name and possibly port number.

Fetch a page

GET

The simplest and most common request/operation made using HTTP is to GET a URL. The URL could itself refer to a web page, an image or a file. The client issues a GET request to the server and receives the document it asked for. If you issue the command line

```
curl https://curl.se
```

you get a web page returned in your terminal window. The entire HTML document that that URL holds.

All HTTP replies contain a set of response headers that are normally hidden, use curl's `--include` (`-i`) option to display them as well as the rest of the document.

HEAD

You can ask the remote server for ONLY the headers by using the `--head` (`-I`) option which will make curl issue a HEAD request. In some special cases servers deny the HEAD method while others still work, which is a particular kind of annoyance.

The HEAD method is defined and made so that the server returns the headers exactly the way it would do for a GET, but without a body. It means that you may see a `Content-Length:` in the response headers, but there must not be an actual body in the HEAD response.

Multiple URLs in a single command line

A single curl command line may involve one or many URLs. The most common case is probably to just use one, but you can specify any amount of URLs. Yes any. No limits. You will then get requests repeated over and over for all the given URLs.

Example, send two GET requests:

```
curl http://url1.example.com http://url2.example.com
```

If you use `--data` to POST to the URL, using multiple URLs means that you send that same POST to all the given URLs.

Example, send two POSTs:

```
curl --data name=curl http://url1.example.com http://url2.example.com
```

Multiple HTTP methods in a single command line

Sometimes you need to operate on several URLs in a single command line and do different HTTP methods on each. For this, you will enjoy the `--next` option. It is basically a separator that separates a bunch of options from the next. All the URLs before `--next` will get the same method and will get all the POST data merged into one.

When curl reaches the --next on the command line, it will sort of reset the method and the POST data and allow a new set.

Perhaps this is best shown with a few examples. To send first a HEAD and then a GET:

```
curl -I http://example.com --next http://example.com
```

To first send a POST and then a GET:

```
curl -d score=10 http://example.com/post.cgi --next http://example.com/results.html
```

HTML forms

Forms explained

Forms are the general way a website can present an HTML page with fields for the user to enter data in, and then press some kind of 'OK' or 'Submit' button to get that data sent to the server. The server then typically uses the posted data to decide how to act. Like using the entered words to search in a database, or to add the info in a bug tracking system, display the entered address on a map or using the info as a login-prompt verifying that the user is allowed to see what it is about to see.

Of course there has to be some kind of program on the server end to receive the data you send. You cannot just invent something out of the air.

GET

A GET-form uses the method GET, as specified in HTML like:

```
<form method="GET" action="junk.cgi">
  <input type=text name="birthyear">
  <input type=submit name=press value="OK">
</form>
```

In your favorite browser, this form will appear with a text box to fill in and a press-button labeled "OK". If you fill in '1905' and press the OK button, your browser will then create a new URL to get for you. The URL will get junk.cgi?birthyear=1905&press=OK appended to the path part of the previous URL.

If the original form was seen on the page www.example.com/when/birth.html, the second page you will get will become www.example.com/when/junk.cgi?birthyear=1905&press=OK.

Most search engines work this way.

To make curl do the GET form post for you, just enter the expected created URL:

```
curl "http://www.example.com/when/junk.cgi?birthyear=1905&press=OK"
```

POST

The GET method makes all input field names get displayed in the URL field of your browser. That is generally a good thing when you want to be able to bookmark that page with your given data, but it is an obvious disadvantage if you entered secret information in one of the fields or if there are a large amount of fields creating a long and unreadable URL.

The HTTP protocol then offers the POST method. This way the client sends the data separated from the URL and thus you will not see any of it in the URL address field.

The form would look similar to the previous one:

```
<form method="POST" action="junk.cgi">
  <input type=text name="birthyear">
  <input type=submit name=press value=" OK ">
</form>
```

And to use curl to post this form with the same data filled in as before, we could do it like:

```
curl --data "birthyear=1905&press=%20OK%20" http://www.example.com/when/junk.cgi
```

This kind of POST will use the Content-Type application/x-www-form-urlencoded and is the most widely used POST kind.

The data you send to the server MUST already be properly encoded, curl will not do that for you. For example, if you want the data to contain a space, you need to replace that space with %20, etc. Failing to comply with this will most likely cause your data to be received wrongly and messed up.

Recent curl versions can in fact url-encode POST data for you, like this:

```
curl --data-urlencode "name=I am Daniel" http://www.example.com
```

If you repeat --data several times on the command line, curl will concatenate all the given data pieces - and put a & symbol between each data segment.

File Upload POST

Back in late 1995 they defined an additional way to post data over HTTP. It is documented in the RFC 1867, why this method sometimes is referred to as RFC1867-posting.

This method is mainly designed to better support file uploads. A form that allows a user to upload a file could be written like this in HTML:

```
<form method="POST" enctype='multipart/form-data' action="upload.cgi">
  <input type=file name=upload>
  <input type=submit name=press value="OK">
</form>
```

This clearly shows that the Content-Type about to be sent is multipart/form-data.

To post to a form like this with curl, you enter a command line like:

```
curl --form upload=@localfilename --form press=OK [URL]
```

Hidden Fields

A common way for HTML based applications to pass state information between pages is to add hidden fields to the forms. Hidden fields are already filled in, they are not displayed to the user and they get passed along just as all the other fields.

A similar example form with one visible field, one hidden field and one submit button could look like:

```
<form method="POST" action="foobar.cgi">
  <input type=text name="birthyear">
  <input type=hidden name="person" value="daniel">
  <input type=submit name="press" value="OK">
</form>
```

To POST this with curl, you will not have to think about if the fields are hidden or not. To curl they are all the same:

```
curl --data "birthyear=1905&press=OK&person=daniel" [URL]
```

Figure Out What A POST Looks Like

When you are about to fill in a form and send it to a server by using curl instead of a browser, you are of course interested in sending a POST exactly the way your browser does.

An easy way to get to see this, is to save the HTML page with the form on your local disk, modify the 'method' to a GET, and press the submit button (you could also change the action URL if you want to).

You will then clearly see the data get appended to the URL, separated with a ?-letter as GET forms are supposed to.

HTTP upload

PUT

Perhaps the best way to upload data to an HTTP server is to use PUT. Then again, this of course requires that someone put a program or script on the server end that knows how to receive an HTTP PUT stream.

Put a file to an HTTP server with curl:

```
curl --upload-file uploadfile http://www.example.com/receive.cgi
```

HTTP Authentication

Basic Authentication

HTTP Authentication is the ability to tell the server your username and password so that it can verify that you are allowed to do the request you are doing. The Basic authentication used in HTTP (which is the type curl uses by default) is **plain text** based, which means it sends username and password only slightly obfuscated, but still fully readable by anyone that sniffs on the network between you and the remote server.

To tell curl to use a user and password for authentication:

```
curl --user name:password http://www.example.com
```

Other Authentication

The site might require a different authentication method (check the headers returned by the server), and then `--ntlm`, `--digest`, `--negotiate` or even `--anyauth` might be options that suit you.

Proxy Authentication

Sometimes your HTTP access is only available through the use of an HTTP proxy. This seems to be especially common at various companies. An HTTP proxy may require its own user and password to allow the client to get through to the Internet. To specify those with curl, run something like:

```
curl --proxy-user proxyuser:proxypassword curl.se
```

If your proxy requires the authentication to be done using the NTLM method, use `--proxy-ntlm`, if it requires Digest use `--proxy-digest`.

If you use any one of these user+password options but leave out the password part, curl will prompt for the password interactively.

Hiding credentials

Do note that when a program is run, its parameters might be possible to see when listing the running processes of the system. Thus, other users may be able to watch your passwords if you pass them as plain command line options. There are ways to circumvent this.

It is worth noting that while this is how HTTP Authentication works, many websites will not use this concept when they provide logins etc. See the Web Login chapter further below for more details on that.

More HTTP Headers

Referer

An HTTP request may include a 'referer' field (yes it is misspelled), which can be used to tell from which URL the client got to this particular resource. Some programs/scripts check the referer field of requests to verify that this was not arriving from an external site or an unknown page. While this is a stupid way to check something so easily forged, many scripts still do it. Using curl, you can put anything you want in the referer-field and thus more easily be able to fool the server into serving your request.

Use curl to set the referer field with:

```
curl --referer http://www.example.come http://www.example.com
```

User Agent

Similar to the referer field, all HTTP requests may set the User-Agent field. It names what user agent (client) that is being used. Many applications use this information to decide how to display pages. Silly web programmers try to make different pages for users of different browsers to make them look the best possible for their particular browsers. They usually also do different kinds of JavaScript etc.

At times, you will see that getting a page with curl will not return the same page that you see when getting the page with your browser. Then you know it is time to set the User Agent field to fool the server into thinking you are one of those browsers.

To make curl look like Internet Explorer 5 on a Windows 2000 box:

```
curl --user-agent "Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)" [URL]
```

Or why not look like you are using Netscape 4.73 on an old Linux box:

```
curl --user-agent "Mozilla/4.73 [en] (X11; U; Linux 2.2.15 i686)" [URL]
```

Redirects

Location header

When a resource is requested from a server, the reply from the server may include a hint about where the browser should go next to find this page, or a new page keeping newly generated output. The header that tells the browser to redirect is Location:.

Curl does not follow Location: headers by default, but will simply display such pages in the same manner it displays all HTTP replies. It does however feature an option that will make it attempt to follow the Location: pointers.

To tell curl to follow a Location:

```
curl --location http://www.example.com
```

If you use curl to POST to a site that immediately redirects you to another page, you can safely use --location (-L) and --data/--form together. Curl will only use POST in the first request, and then revert to GET in the following operations.

Other redirects

Browsers typically support at least two other ways of redirects that curl does not: first the html may contain a meta refresh tag that asks the browser to load a specific URL after a set number of seconds, or it may use JavaScript to do it.

Cookies

Cookie Basics

The way the web browsers do "client side state control" is by using cookies. Cookies are just names with associated contents. The cookies are sent to the client by the server. The server tells the client for what path and host name it wants the cookie sent back, and it also sends an expiration date and a few more properties.

When a client communicates with a server with a name and path as previously specified in a received cookie, the client sends back the cookies and their contents to the server, unless of course they are expired.

Many applications and servers use this method to connect a series of requests into a single logical session. To be able to use curl in such occasions, we must be able to record and send back cookies the way the web application expects them. The same way browsers deal with them.

Cookie options

The simplest way to send a few cookies to the server when getting a page with curl is to add them on the command line like:

```
curl --cookie "name=Daniel" http://www.example.com
```

Cookies are sent as common HTTP headers. This is practical as it allows curl to record cookies simply by recording headers. Record cookies with curl by using the --dump-header (-D) option like:

```
curl --dump-header headers_and_cookies http://www.example.com
```

(Take note that the --cookie-jar option described below is a better way to store cookies.)

Curl has a full blown cookie parsing engine built-in that comes in use if you want to reconnect to a server and use cookies that were stored from a previous connection (or hand-crafted manually to fool the server into believing you had a previous connection). To use previously stored cookies, you run curl like:

```
curl --cookie stored_cookies_in_file http://www.example.com
```

Curl's "cookie engine" gets enabled when you use the --cookie option. If you only want curl to understand received cookies, use --cookie with a file that does not exist. Example, if you want to

let curl understand cookies from a page and follow a location (and thus possibly send back cookies it received), you can invoke it like:

```
curl --cookie nada --location http://www.example.com
```

Curl has the ability to read and write cookie files that use the same file format that Netscape and Mozilla once used. It is a convenient way to share cookies between scripts or invokes. The --cookie (-b) switch automatically detects if a given file is such a cookie file and parses it, and by using the --cookie-jar (-c) option you will make curl write a new cookie file at the end of an operation:

```
curl --cookie cookies.txt --cookie-jar newcookies.txt http://www.example.com
```

HTTPS

HTTPS is HTTP secure

There are a few ways to do secure HTTP transfers. By far the most common protocol for doing this is what is generally known as HTTPS, HTTP over SSL. SSL encrypts all the data that is sent and received over the network and thus makes it harder for attackers to spy on sensitive information.

SSL (or TLS as the current version of the standard is called) offers a set of advanced features to do secure transfers over HTTP.

Curl supports encrypted fetches when built to use a TLS library and it can be built to use one out of a fairly large set of libraries - `curl -V` will show which one your curl was built to use (if any!). To get a page from an HTTPS server, simply run curl like:

```
curl https://secure.example.com
```

Certificates

In the HTTPS world, you use certificates to validate that you are the one you claim to be, as an addition to normal passwords. Curl supports client- side certificates. All certificates are locked with a pass phrase, which you need to enter before the certificate can be used by curl. The pass phrase can be specified on the command line or if not, entered interactively when curl queries for it. Use a certificate with curl on an HTTPS server like:

```
curl --cert mycert.pem https://secure.example.com
```

curl also tries to verify that the server is who it claims to be, by verifying the server's certificate against a locally stored CA cert bundle. Failing the verification will cause curl to deny the connection. You must then use `--insecure` (-k) in case you want to tell curl to ignore that the server cannot be verified.

More about server certificate verification and ca cert bundles can be read in the SSLCERTS document.

At times you may end up with your own CA cert store and then you can tell curl to use that to verify the server's certificate:

```
curl --cacert ca-bundle.pem https://example.com/
```

Custom Request Elements

Modify method and headers

Doing fancy stuff, you may need to add or change elements of a single curl request.

For example, you can change the POST method to PROPFIND and send the data as Content-Type: text/xml (instead of the default Content-Type) like this:

```
curl --data "<xml>" --header "Content-Type: text/xml"   --request PROPFIND example.cc
```

You can delete a default header by providing one without content. Like you can ruin the request by chopping off the Host: header:

```
curl --header "Host:" http://www.example.com
```

You can add headers the same way. Your server may want a Destination: header, and you can add it:

```
curl --header "Destination: http://nowhere" http://example.com
```

More on changed methods

It should be noted that curl selects which methods to use on its own depending on what action to ask for. -d will do POST, -I will do HEAD and so on. If you use the --request / -X option you can change the method keyword curl selects, but you will not modify curl's behavior. This means that if you for example use -d "data" to do a POST, you can modify the method to a PROPFIND with -X and curl will still think it sends a POST . You can change the normal GET to a POST method by simply adding -X POST in a command line like:

```
curl -X POST http://example.org/
```

... but curl will still think and act as if it sent a GET so it will not send any request body etc.

Web Login

Some login tricks

While not strictly just HTTP related, it still causes a lot of people problems so here's the executive run-down of how the vast majority of all login forms work and how to login to them using curl.

It can also be noted that to do this properly in an automated fashion, you will most certainly need to script things and do multiple curl invokes etc.

First, servers mostly use cookies to track the logged-in status of the client, so you will need to capture the cookies you receive in the responses. Then, many sites also set a special cookie on the login page (to make sure you got there through their login page) so you should make a habit of first getting the login-form page to capture the cookies set there.

Some web-based login systems feature various amounts of JavaScript, and sometimes they use such code to set or modify cookie contents. Possibly they do that to prevent programmed logins, like this manual describes how to... Anyway, if reading the code is not enough to let you repeat the behavior manually, capturing the HTTP requests done by your browsers and analyzing the sent cookies is usually a working method to work out how to shortcut the JavaScript need.

In the actual <form> tag for the login, lots of sites fill-in random/session or otherwise secretly generated hidden tags and you may need to first capture the HTML code for the login form and extract all the hidden fields to be able to do a proper login POST. Remember that the contents need to be URL encoded when sent in a normal POST.

Debug

Some debug tricks

Many times when you run curl on a site, you will notice that the site does not seem to respond the same way to your curl requests as it does to your browser's.

Then you need to start making your curl requests more similar to your browser's requests:

- Use the `--trace-ascii` option to store fully detailed logs of the requests for easier analyzing and better understanding
- Make sure you check for and use cookies when needed (both reading with `--cookie` and writing with `--cookie-jar`)
- Set user-agent (with `-A`) to one like a recent popular browser does
- Set referer (with `-E`) like it is set by the browser
- If you use POST, make sure you send all the fields and in the same order as the browser does it.

Check what the browsers do

A good helper to make sure you do this right, is the web browsers' developers tools that let you view all headers you send and receive (even when using HTTPS).

A more raw approach is to capture the HTTP traffic on the network with tools such as Wireshark or tcpdump and check what headers that were sent and received by the browser. (HTTPS forces you to use SSLKEYLOGFILE to do that.)