

Everything curl

README

Everything curl is an extensive guide for all things curl. The project, the command-line tool, the library, how everything started and how it came to be the useful tool it is today. It explains how we work on developing it further, what it takes to use it, how you can contribute with code or bug reports and why millions of existing users use it.

This book is meant to be interesting and useful to both casual readers and somewhat more experienced developers. It offers something for everyone to pick and choose from.

Do not try to read it from front to back. Read the chapters or content you are curious about and flip back and forth as you see fit.

I hope to run this book project as I do all other projects I work on: in the open, completely free to download and read. I want it to be free for anyone to comment on, and available for everyone to contribute to and help out with.

Send your bug reports, pull requests or critiques to me and I will improve this book accordingly.

This book will never be finished. I intend to keep working on it. While I may at some point consider it fairly complete, covering most aspects of the project (even if only that seems like an insurmountable goal), the curl project will continue to move so there will always be things to update in the book as well.

This book project started at the end of September 2015.

Site

<https://everythingcurl.dev> is the home of this book. It features accessible links to read the book online in a web version, or download a PDF version for offline reading. Unfortunately, the previously provided ebook formats are no longer provided by gitbook.com that we use to produce the book.

(The "export PDF" option menu is only visible if the used web browser is wide enough.)

Content

All book content is hosted on GitHub in the <https://github.com/bagder/everything-curl> repository.

Author

With the hope of becoming just a co-author of this material, I am Daniel Stenberg. I founded the curl project and I am a developer at heart—for fun and profit. I live and work in Stockholm, Sweden.

All there is to know about Daniel can be found on daniel.haxx.se.

Contribute

If you find mistakes, omissions, errors or blatant lies in this document, please send us a refreshed version of the affected paragraph and we will amend and update. We give credits to and recognize everyone who helps out!

Preferably, you could submit [errors](#) or [pull requests](#) on the book's GitHub page.

Contributors

Lots of people have reported bugs, improved sections or otherwise helped making this book the success it is. These friends include the following:

AaronChen0 on github, alawvt on github, Amin Khoshnood, amnkh on github, Anders Roxell, Angad Gill, Aris (Karim) Merchant, auktis on github, Ben Bodenmiller Ben Peachey, bookofportals on github, Bruno Baguette, Carlton Gibson, Chris DeLuca, Citizen Esosa, Dan Fandrich, Daniel Brown, Daniel Sabsay, David Piano, DrDoom74 at GitHub, Emil Hessman, enachos71 on github, ethomag on github, Fabian Keil, faterer on github, Frank Dana, Frank Hassanabad, Gautham B A, Geir Hauge, Harry Wright, Helena Udd, Hubert Lin, i-ky on github, infinnovation-dev on GitHub, Jay Ottinger, Jay Satiro, Jeroen Ooms, Johan Wigert, John Simpson, JohnCoconut on github, Jonas Forsberg, Josh Vanderhook, JoylfBam5, KJM on github, knorr3 on github, lowttl on github, Luca Niccoli, Manuel on github, Marius Žilėnas, Mark Koester, Martin van den Nieuwelaar, mehandes on github, Michael Kaufmann, Ms2ger, Nick Travers, Nicolas Brassard, Oscar on github, Oskar Köök, Patrik Lundin, RekGRpth on github, Ryan McQuen, Saravanan Musuwathi Kesavan, Senthil Kumaran, Shusen Liu, Sonia Hamilton, Spiros Georgaras, Stephen, Steve Holme, Stian Hvatum, strupo on github, Viktor Szakats, Vitaliy T, Wayne Lai, Wieland Hoffmann, 谭九鼎

License

This document is licensed under the [Creative Commons Attribution 4.0 license](#).

How to read this book

Here is an overview of the main sections of this book and what they cover.

1. The cURL project

Project things. How it started, how we work and how often releases are made.

2. Network and protocols

What exactly are networks and protocols?

3. Install curl

How and where to get and install curl.

4. Source code

A description of the curl source tree and how the layout of the code is and works.

5. Build curl

How to build curl and libcurl from source.

6. Command line basics

Start at the beginning. How do you use curl from a command line?

7. Using curl

Going deeper, looking at things you do with curl the command line tool.

8. HTTP with curl

Digging deeper on HTTP specific actions to do with the curl command line tool.

9. FTP with curl

Learn FTP specific operations in this chapter.

10. Using libcurl

How libcurl works and how you use it when writing your own applications with it.

11. HTTP with libcurl

A closer look at doing HTTP specific things with libcurl.

12. Bindings for libcurl

A casual overview of some of the most popular libcurl bindings and how similar they are to the libcurl C API.

13. libcurl internals

Under the hood it works like this...

14. Index

The index.

The cURL project





curl logo

A funny detail about Open Source projects is that they are called "projects", as if they were somehow limited in time or ever can get done. The cURL "project" is a number of loosely coupled individual volunteers working on writing software together with a common mission: to do reliable data transfers with Internet protocols, as Open Source.

How it started

Back in 1996, [Daniel Stenberg](#) was writing an IRC bot in his spare time, an automated program that would offer services for the participants in a chatroom dedicated to the Amiga computer (#amiga on the IRC network EFnet). He came to think that it would be fun to get some updated currency rates and have his bot offer a service online for the chat room users to get current exchange rates, to ask the bot "please exchange 200 USD into SEK" or similar.

In order to have the provided exchange rates as accurate as possible, the bot would download the rates daily from a website that was hosting them. A small tool to download data over HTTP was needed for this task. A quick look-around at the time had Daniel find a tiny tool named httpget (written by the Brazilian developer Rafael Sagula). It did the job, almost, just needed a few little tweaks here and there.

Rafael released HttpGet 0.1 on November 11, 1996 and already in the next release, called 0.2 released in December that year, Daniel had his first changes included. Soon after that, Daniel had taken over maintenance of the few hundred lines of code it was.

HttpGet 1.0 was released on April 8 1997 with brand new HTTP proxy support.

We soon found and fixed support for getting currencies over GOPHER. Once FTP download support was added, the name of the project was changed and urlget 2.0 was released in August 1997. The HTTP-only days were already passed.

The project slowly grew bigger. When upload capabilities were added and the name once again was misleading, a second name change was made and on March 20, 1998 curl 4 was released. (The version numbering from the previous names was kept.)

We consider **March 20 1998** to be curl's birthday.

The name

Naming things is hard.

The tool was about uploading and downloading data specified with a URL. It was a client-side program (the

'c'), a URL client, and would show the data (by default). So 'c' for Client and URL: **cURL**.

Nothing more was needed so the name was selected and we never looked back again.

Later on, someone suggested that curl could actually be a clever "recursive acronym" (where the first letter in the acronym refers back to the same word): "Curl URL Request Library".

While that is awesome, it was actually not the original thought. We wish we were that clever...

There are and were other projects using the name curl in various ways, but we were not aware of them by the time our curl came to be.

Pronunciation

Most of us pronounce "curl" with an initial k sound, just like the English word curl. It rhymes with words like girl and earl. Merriam Webster has a [short WAV file](#) to help.

Confusions and mix-ups

Soon after our curl was created another "curl" appeared that created a programming language. That curl still [exists](#).

Several libcurl bindings for various programming languages use the term "curl" or "CURL" in part or completely to describe their bindings. Sometimes you will find users talking about curl but referring to neither the command-line tool nor the library that is made by this project.

Used as a verb

'To curl something' is sometimes used as a reference to use a non-browser tool to download a file or resource from a URL.

What does curl do?

cURL is a project and its primary purpose and focus is to make two products:

- curl, the command-line tool
- libcurl the transfer library with a C API

Both the tool and the library do Internet transfers for resources specified as URLs using Internet protocols.

Everything and anything that is related to Internet protocol transfers can be considered curl's business.

Things that are not related to that should be avoided and be left for other projects and products.

It could be important to also consider that curl and libcurl try to avoid handling the actual data that is transferred. It has, for example, no knowledge about HTML or anything else of the content that is popular to transfer over HTTP, but it knows all about how to transfer such data over HTTP.

Both products are frequently used not only to drive thousands or millions of scripts and applications for an Internet connected world, but they are also widely used for server testing, protocol fiddling and trying out new things.

The library is used in every imaginable sort of embedded device where Internet transfers are needed: car infotainment, televisions, Blu-Ray players, set-top boxes, printers, routers, game systems, etc.

Command line tool

Running curl from the command line was natural and Daniel never considered anything else than that it would output data on stdout, to the terminal, by default. The "everything is a pipe" mantra of standard Unix philosophy was something Daniel believed in. curl is like 'cat' or one of the other Unix tools; it sends data to stdout to make it easy to chain together with other tools to do what you want. That is also why virtually all curl options that allow reading from a file or writing to a file, also have the ability to select doing it to stdout or from stdin.

Following the Unix style of how command-line tools work, there was also never any question about whether curl should support multiple URLs on the command line.

The command-line tool is designed to work perfectly from scripts or other automatic means. It does not feature any other GUI or UI other than mere text in and text out.

The library

While the command-line tool came first, the network engine was ripped out and converted into a library during the year 2000 and the concepts we still have today were introduced with libcurl 7.1 in August 2000. Since then, the command line tool has been a thin layer of logic to make a tool around the library that does all the heavy lifting.

libcurl is designed and meant to be available for anyone who wants to add client-side file transfer capabilities to their software, on any platform, any architecture and for any purpose. libcurl is also extremely liberally licensed to avoid that becoming an obstacle.

libcurl is written in traditional and conservative C. Where other languages are preferred, people have created libcurl [bindings](#) for them.

Project communication

CURL is an Open Source project consisting of voluntary members from all over the world, living and working in a large number of the world's time zones. To make such a setup actually work, communication and openness is key. We keep all communication public and we use open communication channels. Most discussions are held on mailing lists, we use bug trackers where all issues are discussed and handled with full insight for everyone who cares to look.

It is important to realize that we are all jointly taking care of the project, we fix problems and we add features. Sometimes a regular contributor grows bored and fades away, sometimes a new eager contributor steps out from the shadows and starts helping out more. To keep this ship going forward as well as possible, it is important that we maintain open discussions and that is one of the reasons why we frown upon users who take discussions privately or try to email individual team members about development issues, questions, debugging or whatever.

In this day, mailing lists may be considered the old style of communication — no fancy web forums or similar. Using a mailing list is therefore becoming an art that is not practiced everywhere and may be a bit strange and unusual to you. But fear not. It is just about sending emails to an address that then sends that email out to all the subscribers. Our mailing lists have at most a few thousand subscribers. If you are mailing for the first time, it might be good to read a few old mails first to get to learn the culture and what's considered good practice.

The mailing lists and the bug tracker have changed hosting providers a few times and there are reasons to suspect it might happen again in the future. It is just the kind of thing that happens to a project that lives for a long time.

A few users also hang out on IRC in the #curl channel on [libera.chat](#).

Mailing list etiquette

Like many communities and subcultures, we have developed guidelines and rules of what we think is the right way to behave and how to communicate on the mailing lists. The [curl mailing list etiquette](#) follows the style of traditional Open Source projects.

Do not mail a single individual

Many people send one question directly to one person. One person gets many mails, and there is only one person who can give you a reply. The question may be something that other people also want to ask. These other people have no way to read the reply but to ask the one person the question. The one person consequently gets overloaded with mail.

If you really want to contact an individual and perhaps pay for his or her services, by all means go ahead, but if it's just another curl question, take it to a suitable list instead.

Reply or new mail

Please do not reply to an existing message as a shortcut to post a message to the lists.

Many mail programs and web archivers use information within mails to keep them together as "threads", as collections of posts that discuss a certain subject. If you do not intend to reply on the same or similar subject, do not just hit reply on an existing mail and change subject; create a new mail.

Reply to the list

When replying to a message from the list, make sure that you do "group reply" or "reply to all", and not just reply to the author of the single mail you reply to.

We are actively discouraging replying back to just a single person privately. Keep follow-ups on discussions on the list.

Use a sensible subject

Please use a subject of the mail that makes sense and that is related to the contents of your mail. It makes it a lot easier to find your mail afterwards and it makes it easier to track mail threads and topics.

Do not top-post

If you reply to a message, do not use top-posting. Top-posting is when you write the new text at the top of a mail and you insert the previous quoted mail conversation below. It forces users to read the mail in a backwards order to properly understand it.

This is why top posting is so bad:

- 1 A: Because it messes up the order in which people normally read text.
- 2 Q: Why is top-posting such a bad thing?
- 3 A: Top-posting.
- 4 Q: What is the most annoying thing in email?

Apart from the screwed-up read order (especially when mixed together in a thread when someone responds using the mandated bottom-posting style), it also makes it impossible to quote only parts of the original mail.

When you reply to a mail you let the mail client insert the previous mail quoted. Then you put the cursor on the first line of the mail and you move down through the mail, deleting all parts of the quotes that do not add context for your comments. When you want to add a comment you do so, inline, right after the quotes that relate to your comment. Then you continue downwards again.

When most of the quotes have been removed and you have added your own words, you are done.

HTML is not for mails

Please switch off those HTML encoded messages. You can mail all those funny mails to your friends. We speak plain text mails.

Quoting

Quote as little as possible. Just enough to provide the context you cannot leave out. A lengthy description can be found [here](#).

Digest

We allow subscribers to subscribe to the "digest" version of the mailing lists. A digest is a collection of mails lumped together in one single mail.

Should you decide to reply to a mail sent out as a digest, there are two things you **MUST** consider if you really really cannot subscribe normally instead:

Cut off all mails and chatter that is not related to the mail you want to reply to.

Change the subject name to something sensible and related to the subject, preferably even the actual subject of the single mail you wanted to reply to.

Please tell us how you solved the problem

Many people mail questions to the list, people spend some of their time and make an effort in providing good answers to these questions.

If you are the one who asks, please consider responding once more in case one of the hints was what solved your problems. Those who write answers feel good to know that they provided a good answer and that you fixed the problem. Far too often, the person who asked the question is never heard of again, and we never get to know if he/she is gone because the problem was solved or perhaps because the problem was unsolvable!

Getting the solution posted also helps other users that experience the same problem(s). They get to see (possibly in the web archives) that the suggested fixes actually has helped at least one person.

Mailing lists

Some of the most important mailing lists are...

curl-users

The main mailing list for users and developers of the curl command-line tool, for questions and help around curl concepts, command-line options, the protocols curl can speak or even related tools. We tend to move development issues or more advanced bug fixes discussions over to curl-library instead, since libcurl is the engine that drives most of curl.

See [curl-users](#)

curl-library

The main development list, and also for users of libcurl. We discuss how to use libcurl in applications as well as development of libcurl itself. You will find lots of questions on libcurl behavior, debugging and documentation issues.

See [curl-library](#)

curl-announce

This mailing list only gets announcements about new releases and security problems—nothing else. This one is for those who want a more casual feed of information from the project.

See [curl-announce](#)

Reporting bugs

Reporting bugs

The development team does a lot of testing. We have a whole test suite that is run frequently every day on numerous platforms in order to exercise all code and make sure everything works as supposed.

Still, there are times when things are not working the way they should. Then we appreciate getting those problems reported.

A bug is a problem

Any problem can be considered a bug. A weirdly phrased wording in the manual that prevents you from understanding something is a bug. A surprising side effect of combining multiple options can be a bug—or perhaps it should be better documented? Perhaps the option does not do at all what you expected it to? That is a problem and we should fix it.

Problems must be known to get fixed

This may sound easy and uncomplicated but is a fundamental truth in our and other projects. Just because it is an old project and have thousands of users does not mean that the development team knows about the problem you just fell over. Maybe users have not paid enough attention to details like you, or perhaps it just never triggered for anyone else.

We rely on users experiencing problems to report them. We need to learn the problems exist so that we can fix them.

Fixing the problems

Software engineering is, to a large degree, about fixing problems. To fix a problem a developer needs to understand how to repeat it and to do that the debugging person needs to be told what set of circumstances that made the problem trigger.

A good bug report

A good report explains what happened and what you thought was going to happen. Tell us exactly what versions of the different components you used and take us step by step through what you do to get the problem.

After you submit a bug report, you can expect there to be follow-up questions or perhaps requests that you try out various things and tasks in order for the developer to be able to narrow down the suspects and make sure your problem is being cornered in properly.

A bug report that is submitted but is abandoned by the submitter risks getting closed if the developer fails to understand it, fails to reproduce it or faces other problems when working on it. Do not abandon your report!

Report curl bugs in the [curl bug tracker on GitHub](#)!

Testing

Testing software thoroughly and properly is a lot of work. Testing software that runs on dozens of operating systems and dozens of CPU architectures, with server implementations with their own sets of bugs and interpretations of the specs, is even more work.

The curl project has a test suite that iterates over all existing test cases, runs the test and verifies that the outcome is the correct one and that no other problem happened, like a memory leak or something fishy in the protocol layer.

The test suite is meant to be possible to run after you have built curl yourself and there are a fair number of volunteers who also help out by running the test suite automatically a few times per day to make sure the latest commits get a run. This way, we discover the worst flaws pretty soon after they were introduced.

We do not test everything and even when we try to test things there will always be subtle details that get through and that we, sometimes years after the fact, figure out were wrong.

Due to the nature of different systems and funny use cases on the Internet, eventually some of the best

testing is done by users when they run the code to perform their own use cases.

Another limiting factor with the test suite is that the test setup itself is less portable than curl and libcurl so there are in fact platforms where curl runs fine but the test suite cannot execute at all.

Commercial support

Commercial support for curl and libcurl is offered and provided by Daniel Stenberg (the curl founder) through the company [wolfSSL](#).

wolfSSL offers world-wide commercial support on curl done by the masters of curl. wolfSSL handles curl customization, ports to new operating systems, feature development, patch maintenance, bug fixing, upstreaming, training, code reviews of libcurl API use, security scanning your curl use and more - with several different support options from basic up to full 24/7 support. With guaranteed response times.

See [the support page](#) for contact details.

Releases

Releases

A release in the curl project means packaging up all the source code that is in the master branch of the code repository, signing the package, tagging the point in the code repository, and then putting it up on the website for the world to download.

It is one single source code archive for all platforms curl can run on. It is the one and only package for both curl and libcurl.

We never ship any curl or libcurl *binaries* from the project with one exception: we host official curl binaries built for Windows users. All the other packaged binaries that are provided with operating systems or on other download sites are done by gracious volunteers outside of the project.

As of several years back, we make an effort to do our releases on an eight week cycle and unless some really serious and urgent problem shows up we stick to this schedule. We release on a Wednesday, and then again a Wednesday eight weeks later and so it continues. Non-stop.

For every release we tag the source code in the repository with "curl-release version" and we update the [changelog](#).

We had done 204 curl releases by November 2021. The entire release history and changelog is available in our [curl release log](#).

Daily snapshots

Every single change to the source code is committed and pushed to the source code repository. This repository is hosted on [github.com](#) and is using git these days (but has not always been this way). When building curl off the repository, there are a few things you need to generate and setup that sometimes cause people some problems or just friction. To help with that, we provide daily snapshots.

The daily snapshots are generated daily (clever naming, right?) as if a release had been made at that point. It produces a package of all sources code and all files that are normally part of a release and puts it in a package and uploads it to [this special place](#) to allow interested people to get the latest code to test, to experiment or whatever.

The snapshots are kept for around 20 days until deleted.

Security

Security is a primary concern for us in the curl project. We take it seriously and we work hard on providing secure and safe implementations of all protocols and related code. As soon as we get knowledge about a security related problem or just a suspected problem, we deal with it and we will attempt to provide a fix and security notice no later than in the next pending release.

We use a responsible disclosure policy, meaning that we prefer to discuss and work on security fixes out of the public eye and we alert the vendors on the openwall.org list a few days before we announce the problem and fix to the world. This, in an attempt to shorten the time span the bad guys can take advantage of a problem until a fixed version has been deployed.

Past security problems

During the years we have had our fair share of security related problems. We work hard on [documenting every problem](#) thoroughly with all details listed and clearly stated to aid users. Users of curl should be able to figure out what problems their particular curl versions and use cases are vulnerable to.

To help with this, we present [this waterfall chart](#) showing how all vulnerabilities affect which curl versions and we have this complete list of all known security problems since the birth of this project.

Trust

For a software to conquer the world, it needs to be trusted. It takes trust to build more trust and it can all be broken down really fast if the foundation is proven to have cracks.

In the curl project we build trust for our users in a few different ways:

1. We are completely transparent about everything. Every decision, every discussion as well as every line of code and every considered code change are always public and done in the open.
2. We work hard to write reliable code. We write test cases, we review code, we document best practices and we have a style guide that helps us keep code consistent.
3. We stick to promises and guarantees as much as possible. We do not break APIs and we do not abandon support for old systems.
4. Security is of utmost importance and we take every reported incident seriously and realize that we **must**

fix all known problems and we need to do it responsibly. We do our best to not endanger our users.

5. We act like adults. We can be silly and we can joke around, but we do it responsibly and we follow our [Code of Conduct](#). Everyone should be able to even trust us to behave.

Code of Conduct

As contributors and maintainers of this project, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, or religion.

Examples of unacceptable behavior by participants include the use of sexual language or imagery, derogatory comments or personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. Project maintainers who do not follow the Code of Conduct may be removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

Development

Development

We encourage everyone to participate in the development of curl and libcurl. We appreciate all the help we can get and while the main portion of this project is source code, there is a lot more than just coding and debugging help that is needed and useful.

We develop and discuss everything in the open, preferably on the mailing lists.

Source code on GitHub

The source code to curl and libcurl have also been provided and published publicly and it continues to be uploaded to the [main web site](#) for every release.

Since March 2010, the curl source code repository has been hosted on github.com. By being up to date with

the changes there, you can follow our day to day development closely.

The development team

Daniel Stenberg is the founder and self-proclaimed leader of the project. Everybody else that participates or contributes in the project has thus arrived at a later point. Some contributors worked for a while and then left again. Most contributors hang around only for a short while to get their bug fixed or feature merged or similar. Counting all contributors we know the names of, we have received help from more than 2,500 individuals.

The full list of people who ever did ten commits or more within a single year in the project are:

Alessandro Ghedini, Ben Greear, Benoit Neil, Bill Hoffman, Bill Nagel, Björn Stenberg, Brad Hards, Dan Fandrich, Daniel Gustafsson, Daniel Stenberg, Dominick Meglio, Emil Engler, Fabian Frank, Fabian Keil, Gergely Nagy, Gisle Vanem, Guenter Knauf, Harry Sintonen, Isaac Boukris, Jacob Hoffman-Andrews, Jakub Zakrzewski, James Housley, Jay Satiro, Jiri Hruska, Joe Mason, Johannes Schindelin, Josh Soref, Julien Chaffraix, Kamil Dudka, Marc Hoersken, Marcel Raad, Mark Salisbury, Marty Kuhrt, Max Dymond, Michael Kaufmann, Michael Osipov, Michal Marek, Nick Zitzmann, Nikos Mavrogiannopoulos, Patrick Monnerat, Peter Wu, Rikard Falkeborn, Ruslan Baratov, Simon Warta, Steinar H. Gunderson, Sterling Hughes, Steve Holme, Svyatoslav Mishyn, Tatsuhiro Tsujikawa, Tor Arntsen, Viktor Szakats, Yang Tse

Users of curl

Users of curl



ten billion installations

We use to say that there are around ten billion curl installations. It makes a good line to say but in reality we, of course, do not have any numbers that exact. We just estimate and guess based on observations and trends. It also depends on exactly what we consider "an installation". Let's elaborate.

Open Source

The project being Open Source and liberally licensed means that just about anyone can redistribute curl in source format or built into binary form.

Counting downloads

The curl command-line tool and the libcurl library are available for download for most operating systems via the curl website, they are provided via third party installers to a bunch and they come installed by default with even more operating systems. This makes counting downloads from the curl website completely inappropriate as a means of measurement.

Finding users

So, we cannot count downloads and anyone may redistribute it and nobody is forced to tell us they use curl. How can we figure out the numbers? How can we figure out the users? The answer is that we really cannot with any decent level of accuracy.

Instead we rely on witness reports, circumstantial evidence, on findings on the Internet, the occasional "about box" or license agreement mentioning curl or that authors ask for help and tell us about their use.

The curl license says users need to repeat it somewhere, like in the documentation, but that is not easy for us to find in many cases and it's also not easy for us to do anything about should they decide not to follow the small license requirement.

Command-line tool users

The command-line tool curl is widely used by programmers around the world in shell and batch scripts, to debug servers and to test out things. There is no doubt it is used by millions every day.

Embedded library

libcurl is what makes our project reach the really large volume of users. The ability to quickly and easily get client side file transfer abilities into your application is desirable for a lot of users, and then libcurl's great portability also helps: you can write more or less the same application on a wide variety of platforms and you can still keep using libcurl for transfers.

libcurl being written in C with no or just a few required dependencies also help to get it used in embedded systems.

libcurl is popularly used in smartphone operating systems, in car infotainment setups, in television sets, in set-top boxes, in audio and video equipment such as Blu-Ray players and higher-end receivers. It is often used in home routers and printers.

A fair number of best-selling games are also using libcurl, on Windows and game consoles.



different devices, tool, applications and services that all run curl

In website backends

The libcurl binding for PHP was one of, if not the, first bindings for libcurl to really catch on and get used widely. It quickly got adopted as a default way for PHP users to transfer data and as it has now been in that position for over a decade and PHP has turned out to be a fairly popular technology on the Internet (recent numbers indicated that something like a quarter of all sites on the Internet uses PHP).

A few really high-demand sites are using PHP and are using libcurl in the backend. Facebook and Yahoo are two such sites.

Famous users

Nothing forces users to tell us they use curl or libcurl in their services or in the products. We usually only find out they do by accident, by reading about dialogues, documentation and license agreements. Of course some companies also just flat out tell us.

We used to collect names of companies and products on our website of users that use the project's products "in commercial environments". We did this mostly just to show-off to other big brands that if these other guys can build products that depend on us, maybe you can, too?

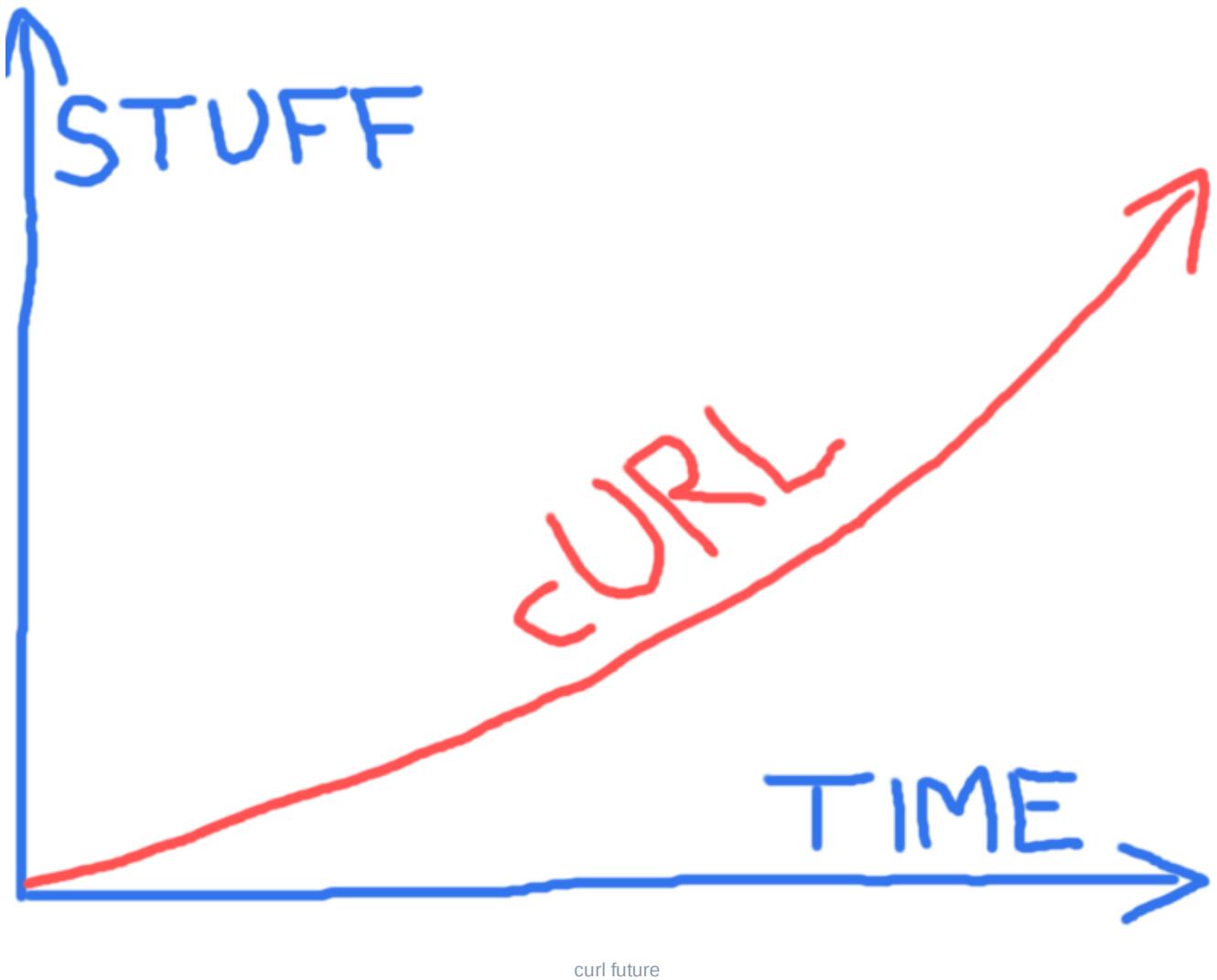
The list of companies contains hundreds of names, but extracting some of the larger or more well-known brands, here's a pretty good list that, of course, is only a small selection:

Adobe, Altera, AOL, Apple, AT&T, BBC, Blackberry, BMW, Bosch, Broadcom, Chevrolet, Cisco, Comcast, Facebook, Google, Hitachi, Honeywell, HP, Huawei, HTC, IBM, Intel, LG, Mazda, Mercedes-Benz, Microsoft, Motorola, NASA, Netflix, Nintendo, Oracle, Panasonic, Philips, Pioneer, RBS, Samsung, SanDisk, SAP, SAS Institute, SEB, Sharp, Siemens, Sony, Spotify, Sun, Swisscom, Tomtom, Toshiba, VMware, Xilinx, Yahoo, Yamaha

Famous high volume apps using curl

The Google Youtube app, the Google Photos app, Spotify, Instagram, Skype (on Android), bundled with iOS, Grand Theft Auto V, Fortnite.

Future



There is no slowdown in sight in curl's future, bugs reported, development pace or how Internet protocols are being developed or updated.

We are looking forward to support for more protocols, support for more features within the already supported protocols, and more and better APIs for libcurl to allow users to do transfers even better and faster.

The project casually maintains a [TODO](#) file holding a bunch of ideas that we could work on in the future. It also keeps a [KNOWN_BUGS](#) document with a list of known problems we would like to fix.

There is a [ROADMAP](#) document that describes some plans for the short-term that some of the active developers

thought they would work on next. Of course, we can not promise that we will always follow it perfectly.

We are highly dependent on developers to join in and work on what they want to get done, be it bug fixes or new features.

Network and protocols

Before diving in and talking about how to use curl to get things done, let's take a look at what all this networking is and how it works, using simplifications and some minor shortcuts to give an easy overview.

The basics are in the [networking simplified](#) chapter that tries to just draw a simple picture of what networking is from a curl perspective, and the [protocols](#) section which explains what exactly a "protocol" is and how that works.

Networking simplified

Networking means communicating between two endpoints on the Internet. The Internet is just a bunch of interconnected machines (computers really), each using their own private addresses (called [IP addresses](#)). The addresses each machine have can be of different types and machines can even have temporary addresses. These computers are often called hosts.

The computer, tablet or phone you sit in front of is usually called "the client" and the machine out there somewhere that you want to exchange data with is called "the server". The main difference between the client and the server is in the roles they play here. There is nothing that prevents the roles from being reversed in a subsequent operation.

Which machine

When you want to initiate a transfer to one of the machines out there (a server), you usually do not know its IP addresses but instead you usually know its name. The name of the machine you will talk to is embedded in the URL that you work with when you use curl.

You might use a URL like `http://example.com/index.html`, which means you will connect to and communicate with the host named example.com.

Host name resolving

Once we know the host name, we need to figure out which IP addresses that host has so that we can contact it.

Converting the name to an IP address is often called 'name resolving'. The name is "resolved" to one or a set of addresses. This is usually done by a "DNS server", DNS being like a big lookup table that can convert

names to addresses—all the names on the Internet, really. Your computer normally already knows the address of a computer that runs the DNS server as that is part of setting up the network curl will therefore ask the DNS server: "Hello, please give me all the addresses for example.com", and the server responds with a list of them. Or in the case you spell the name wrong, it can answer back that the name does not exist.

Establish a connection

With a list of IP addresses for the host curl wants to contact, curl sends out a "connect request". The connection curl wants to establish is called TCP ([Transmission Control Protocol](#)) and it works similar to connecting an invisible string between two computers. Once established, it can be used to send a stream of data in both directions.

As curl gets a list of addresses for the host, it will actually traverse that list of addresses when connecting and in case one fails it will try to connect to the next one until either one works or they all fail.

Connects to "port numbers"

When connecting with TCP to a remote server, a client selects which port number to do that on. A port number is just a dedicated place for a particular service, which allows that same server to listen to other services on other port numbers at the same time.

Most common protocols have default port numbers that clients and servers use. For example, when using the `http://example.com/index.html` URL, that URL specifies a scheme called "HTTP" which tells the client that it should try TCP port number 80 on the server by default. The URL can optionally provide another, custom, port number but if nothing special is specified, it will use the default port for the scheme used in the URL.

TLS

After the TCP connection has been established, many transfers will require that both sides negotiate a better security level before continuing, and that is often TLS; Transport Layer Security. If that is used, the client and server will do a TLS handshake first and only continue further if that succeeds.

Transfer data

When the connecting "string" we call TCP is attached to the remote computer (and we have done the possible additional TLS handshake), there is an established connection between the two machines and that connection can then be used to exchange data. That communication is done using a "protocol", as discussed in the following chapter.

Traditionally, what is called a *download* is when data is transferred from a server to a client and inversely an *upload* is when data is transferred from the client to the server. The client is down here. The server is up there

Protocols

Protocol

The language used to ask for data to get sent—in either direction—is called **the protocol**. The protocol describes exactly how to ask the server for data, or to tell the server that there is data coming.

Protocols are typically defined by the IETF ([Internet Engineering Task Force](#)), which hosts RFC documents that describe exactly how each protocol works: how clients and servers are supposed to act and what to send and so on.

What protocols does curl support?

curl supports protocols that allow "data transfers" in either or both directions. We usually also restrict ourselves to protocols which have a "URI format" described in an RFC or at least is somewhat widely used, as curl works primarily with URLs (URIs really) as the input key that specifies the transfer.

The latest curl (as of this writing) supports these protocols:

DICT, FILE, FTP, FTPS, GOPHER, GOPHERS, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, MQTT, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET, TFTP

To complicate matters further, the protocols often exist in different versions or flavors as well.

What other protocols are there?

The world is full of protocols, both old and new. Old protocols get abandoned and dropped and new ones get introduced. There is never a state of stability but the situation changes from day to day and year to year. You can rest assured that there will be new protocols added in the list above in the future and that there will be new versions of the protocols already listed.

There are, of course, already other protocols in existence that curl does not yet support. We are open to supporting more protocols that suit the general curl paradigms, we just need developers to write the necessary code adjustments for them.

How are protocols developed?

Both new versions of existing protocols and entirely new protocols are usually developed by persons or teams that feel that the existing ones are not good enough. Something about them makes them not suitable for a particular use case or perhaps some new idea has popped up that could be applied to improve things.

Of course, nothing prevents anyone from developing a protocol entirely on their own at their own pleasure in their own backyard, but the major protocols are usually brought to the IETF at a fairly early stage where they are then discussed, refined, debated and polished and then eventually, ideally, turned into a published RFC document.

Software developers then read the RFC specifications and deploy their code in the world based on their interpretations of the words in those documents. It sometimes turn out that some of the specifications are subject to vastly different interpretations or sometimes the engineers are just lazy and ignore sound advice in the specs and deploy something that does not adhere. Writing software that interoperates with other implementations of the specifications can therefore end up being hard work.

How much do protocols change?

Like software, protocol specifications are frequently updated and new protocol versions are created.

Most protocols allow some level of extensibility which makes new extensions show up over time, extensions that make sense to support.

The interpretation of a protocol sometimes changes even if the spec remains the same.

The protocols mentioned in this chapter are all "Application Protocols", which means they are transferred over more lower level protocols, like TCP, UDP and TLS. They are also themselves protocols that change over time, get new features and get attacked so that new ways of handling security, etc., forces curl to adapt and change.

About adhering to standards and who's right

Generally, there are protocol specs that tell us how to send and receive data for specific protocols. The protocol specs we follow are RFCs put together and published by IETF.

Some protocols are not properly documented in a final RFC, like, for example, SFTP for which our implementation is based on an Internet-draft that is not even the last available one.

Protocols are, however, spoken by two parties and like in any given conversation, there are then two sides of understanding something or interpreting the given instructions in a spec. Also, lots of network software is written without the authors paying close attention to the spec so they end up taking some shortcuts, or perhaps they just interpreted the text differently. Sometimes even mistakes and bugs make software behave in ways that are not mandated by the spec and sometimes even downright forbidden in the specs.

In the curl project we use the published specs as rules on how to act until we learn anything else. If popular alternative implementations act differently than what we think the spec says and that alternative behavior is what works widely on the big Internet, then chances are we will change foot and instead decide to act like those others. If a server refuses to talk with us when we think we follow the spec but works fine when we bend the rules every so slightly, then we probably end up bending them exactly that way—if we can still work successfully with other implementations.

Ultimately, it is a personal decision and up for discussion in every case where we think a spec and the real world do not align.

In the worst cases we introduce options to let application developers and curl users have the final say on what curl should do. I say worst because it is often really tough to ask users to make these decisions as it usually involves tricky details and weirdness going on and it is a lot to ask of users. We should always do our best to avoid pushing such protocol decisions to users.

curl protocols

curl supports about 26 protocols. We say "about" because it depends on how you count and what you consider to be distinctly different protocols.

DICT

DICT is a dictionary network protocol, it allows clients to ask dictionary servers about a meaning or explanation for words. See RFC 2229. Dict servers and clients use TCP port 2628.

FILE

FILE is not actually a "network" protocol. It is a URL scheme that allows you to tell curl to get a file from the local file system instead of getting it over the network from a remote server. See RFC 1738.

FTP

FTP stands for File Transfer Protocol and is an old (originates in the early 1970s) way to transfer files back and forth between a client and a server. See RFC 959. It has been extended greatly over the years. FTP servers and clients use TCP port 21 plus one more port, though the second one is usually dynamically established during communication.

See the external page [FTP vs HTTP](#) for how it differs to HTTP.

FTPS

FTPS stands for Secure File Transfer Protocol. It follows the tradition of appending an 'S' to the protocol name to signify that the protocol is done like normal FTP but with an added SSL/TLS security layer. See RFC 4217.

This protocol is problematic to use through firewalls and other network equipment.

GOPHER

Designed for "distributing, searching, and retrieving documents over the Internet", Gopher is somewhat of the grand father to HTTP as HTTP has mostly taken over completely for the same use cases. See RFC 1436. Gopher servers and clients use TCP port 70.

GOPHERS

Gopher over TLS. A recent extension to the old protocol.

HTTP

The Hypertext Transfer Protocol, HTTP, is the most widely used protocol for transferring data on the web and over the Internet. See RFC 7230 for HTTP/1.1 and RFC 7540 for [HTTP/2](#). HTTP servers and clients use TCP port 80.

HTTPS

Secure HTTP is HTTP done over an SSL/TLS connection. See RFC 2818. HTTPS servers and clients use TCP port 443, unless they speak [HTTP/3](#) which then uses QUIC and is done over UDP.

IMAP

The Internet Message Access Protocol, IMAP, is a protocol for accessing, controlling and "reading" email. See RFC 3501. IMAP servers and clients use TCP port 143. Whilst connections to the server start out as cleartext, SSL/TLS communication may be supported by the client explicitly requesting to upgrade the connection using the `STARTTLS` command. See RFC 2595.

IMAPS

Secure IMAP is IMAP done over an SSL/TLS connection. Such connections implicitly start out using SSL/TLS and as such servers and clients use TCP port 993 to communicate with each other. See RFC 8314.

LDAP

The Lightweight Directory Access Protocol, LDAP, is a protocol for accessing and maintaining distributed directory information. Basically a database lookup. See RFC 4511. LDAP servers and clients use TCP port 389.

LDAPS

Secure LDAP is LDAP done over an SSL/TLS connection.

MQTT

Message Queuing Telemetry Transport, MQTT, is a protocol commonly used in IoT systems for interchanging data mostly involving smaller devices. It is a so called "publish-subscribe" protocol.

POP3

The Post Office Protocol version 3 (POP3) is a protocol for retrieving email from a server. See RFC 1939. POP3 servers and clients use TCP port 110. Whilst connections to the server start out as cleartext, SSL/TLS communication may be supported by the client explicitly requesting to upgrade the connection using the STLS command. See RFC 2595.

POP3S

Secure POP3 is POP3 done over an SSL/TLS connection. Such connections implicitly start out using SSL/TLS and as such servers and clients use TCP port 995 to communicate with each other. See RFC 8314.

RTMP

The Real-Time Messaging Protocol (RTMP) is a protocol for streaming audio, video and data. RTMP servers and clients use TCP port 1935.

RTSP

The Real Time Streaming Protocol (RTSP) is a network control protocol to control streaming media servers. See RFC 2326. RTSP servers and clients use TCP and UDP port 554.

SCP

The Secure Copy (SCP) protocol is designed to copy files to and from a remote SSH server. SCP servers and clients use TCP port 22.

SFTP

The SSH File Transfer Protocol (SFTP) that provides file access, file transfer, and file management over a reliable data stream. SFTP servers and clients use TCP port 22.

SMB

The Server Message Block (SMB) protocol is also known as CIFS. It is an application-layer network protocol mainly used for providing shared access to files, printers, and serial ports and miscellaneous communications between nodes on a network. SMB servers and clients use TCP port 445.

SMBS

SMB done over TLS.

SMTP

The Simple Mail Transfer Protocol (SMTP) is a protocol for email transmission. See RFC 5321. SMTP servers and clients use TCP port 25. Whilst connections to the server start out as cleartext, SSL/TLS communication may be supported by the client explicitly requesting to upgrade the connection using the STARTTLS command. See RFC 3207.

SMTPS

Secure SMTP, is SMTP done over an SSL/TLS connection. Such connections implicitly start out using SSL/TLS and as such servers and clients use TCP port 465 to communicate with each other. See RFC 8314.

TELNET

TELNET is an application layer protocol used over networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. See RFC 854. TELNET servers and clients use TCP port 23.

TFTP

The Trivial File Transfer Protocol (TFTP) is a protocol for doing simple file transfers over UDP to get a file from or put a file onto a remote host. TFTP servers and clients use UDP port 69.

Install curl

curl is totally free, open and available. There are numerous ways to get it and install it for most operating systems and architecture. This section will give you some answers to start with, but will not be a complete reference.

Some operating systems ship curl by default. Some do not.

In addition, You can always download the source from curl.se or find binary packages to download from there.

- [Linux](#)
- [Windows](#)
- [macOS](#)
- [Docker](#)

Linux

Linux distributions come with "packager managers" that let you install software that they offer. Most Linux distributions offer curl and libcurl to be installed if they are not installed by default.

Ubuntu and Debian

`apt` is a tool to install prebuilt packages on Debian Linux and Ubuntu Linux distributions and derivatives.

To install the curl command-line tool, you usually just

```
1 apt install curl
```

...and that then makes sure the dependencies are installed and usually libcurl is then also installed as an individual package.

If you want to build applications against libcurl, you need a development package installed to get the include headers and some additional documentation, etc. You can then select a libcurl with the TLS backend you

prefer:

```
1 apt install libcurl4-openssl-dev
```

or

```
1 apt install libcurl4-nss-dev
```

or

```
1 apt install libcurl4-gnutls-dev
```

Redhat and CentOS

With Redhat Linux and CentOS Linux derivatives, you use `yum` to install packages. Install the command-line tool with:

```
1 yum install curl
```

You install the libcurl development package (with include files and some docs, etc.) with this:

```
1 yum install libcurl-devel
```

nix

[Nix](#) is a package manager default to the NixOS distribution, but it can also be used on any Linux distribution.

In order to install command-line tool:

```
1 nix-env -i curl
```

Arch Linux

curl is located in the core repository of Arch Linux. This means it should be installed automatically if you follow the normal installation procedure.

If curl is not installed, Arch Linux uses `pacman` to install packages:

```
1 pacman -S curl
```

SUSE and openSUSE

With SUSE Linux and openSUSE Linux you use `zypper` to install packages. To install the curl command-line utility:

```
1 zypper install curl
```

In order to install the libcurl development package you run:

```
1 zypper install libcurl-devel
```

SUSE SLE Micro and openSUSE MicroOS

These versions of SUSE/openSUSE Linux are immutable OSes and have a read only root file system, to install packages you would use `transactional-update` instead of `zypper`. To install the curl command-line utility:

```
1 transactional-update pkg install curl
```

And to install the libcurl development package:

```
1 transactional-update pkg install libcurl-devel
```

Windows

Windows 10 comes with the curl tool bundled with the operating system since version 1804. If you have an older Windows version or just want to upgrade to the latest version shipped by the curl project, download the latest official curl release for Windows from curl.se/windows and install that.

There are several different ways to get curl and libcurl onto your Windows systems:

1. [MSYS2](#)
2. [vcpkg](#)

MSYS2

MSYS2 is a popular build system for Windows based on `mingw-w64` and includes both gcc and clang compilers. MSYS2 uses a package manager named `pacman` (a port from arch-linux) and has about 2000 precompiled `mingw-packages`. MSYS2 is designed to build standalone software: the binaries built with `mingw-w64` compilers do not depend on MSYS2 itself.¹¹

Get curl and libcurl on MSYS2

Current information about the `mingw-w64-curl` package can be found on the msys2 website: <https://packages.msys2.org/base/mingw-w64-curl>. Here we can also find installation instructions for the various available flavors. For example to install the default x64 binary for curl we run:

```
1 pacman -Sy mingw-w64-x86_64-curl
```

This package contains both the `curl` command line tool as well as libcurl headers and shared libraries. The default `curl` packages are built with the OpenSSL backend and hence depend on `mingw-w64-x86_64-openssl`. There are also `mingw-w64-x86_64-curl-gnutls` and `mingw-w64-x86_64-curl-gnutls` packages, refer to the [msys2 website](#) for more details.

Just like on Linux, we can use `pkg-config` to query the flags needed to build against libcurl. Start msys2 using the mingw64 shell (which automatically sets the path to include `/mingw64`) and run:

```
1 pkg-config --cflags libcurl
2 # -IC:/msys64/mingw64/include
3
4 pkg-config --libs libcurl
5 # -LC:/msys64/mingw64/lib -lcurl
```

The pacman package manager installs precompiled binaries. Next up we explain how to use pacman to build curl locally, for example to customize the configuration.

Building libcurl on MSYS2

Building packages with pacman is almost just as simple as installing. The entire process is contained in the `PKGBUILD` file from the `mingw-w64-curl` package. We can easily modify the file to rebuild the package ourselves.

If we start with a clean msys2 installation, we first want to install some build tools, like `autoconf`, `patch` and `git`. Start the msys2 shell and run:

```
1 # Sync the repositories
2 pacman -Syu
3
4 # Install git, autoconf, patch, etc
5 pacman -S git base-devel
```

```
7 # Install GCC for x86_64
8 pacman -S mingw-w64-x86_64-toolchain
```

Now clone the `mingw-packages` repository and go to the `mingw-w64-curl` package:

```
1 git clone https://github.com/msys2/MINGW-packages
2 cd MINGW-packages/mingw-w64-curl
```

This directory contains the PKGBUILD file and patches that are used for building curl. Have a look at the PKGBUILD file to see what is going on. Now to compile it, we can do:

```
1 makepkg-mingw --syncdeps --skippgpcheck
```

That is it. The `--syncdeps` parameter will automatically check and prompt to install dependencies of `mingw-w64-curl` if these are not yet installed. Once the process is complete you will have 3 new files in the current directory, for example:

- `pacman -U mingw-w64-x86_64-curl-7.80.0-1-any.pkg.tar.zst`
- `pacman -U mingw-w64-x86_64-curl-gnutls-7.80.0-1-any.pkg.tar.zst`
- `pacman -U mingw-w64-x86_64-curl-winssl-7.80.0-1-any.pkg.tar.zst`

Use the `pacman -u` command to install such a local package file:

```
1 pacman -U mingw-w64-x86_64-curl-winssl-7.80.0-1-any.pkg.tar.zst
```

Have a look at the [msys2 docs](#) or join the [gitter](#) to learn more about building with pacman and msys2!

[^1]: Be careful not to confuse the [mingw-package](#) `mingw-w64-curl` with the [msys-packages](#) `curl` and `curl-devel`. The latter are part of msys2 environment itself (e.g. to support pacman downloads), but not suitable for redistribution. To build redistributable software that does not depend on MSYS2 itself, you always need `mingw-w64-...` packages and toolchains.

vcpkg

[Vcpkg](#) helps you manage C and C++ libraries on Windows, Linux and MacOS.

There is no "curl" package on vcpkg, only libcurl.

Install libcurl

macOS

macOS comes with the curl tool bundled with the operating system since many years. If you want to upgrade to the latest version shipped by the curl project, we recommend installing [homebrew](#) (a macOS software package manager) and then install the curl package from them:

```
1 brew install curl
```

Get libcurl for macOS

A popular and easy way to get libcurl installed for macOS is to use [homebrew](#) and then run this simple command in a terminal:

```
1 brew install libcurl
```

Docker

The docker image is hosted at <https://hub.docker.com/r/curlimages/curl>

You can run the latest version of curl with the following command:

```
1 docker run -it --rm curlimages/curl www.example.com
```

Running curl seamlessly in docker

It is possible to make an alias to seamlessly run curl inside a container. Define an alias like the following:

```
1 # Bash alias example
2 alias curl='docker run -it --rm curlimages/curl'
```

And simply invoke `curl www.example.com` to make a request

Running curl in kubernetes

Sometimes it can be useful to troubleshoot k8s networking with curl, just like :

```
1 kubectl run -i --tty curl --image=curlimages/curl --restart=Never -- "-m 5" www.example.com
```

Source code

The source code is, of course, the actual engine parts of this project. After all, it is a software project. curl and libcurl are written in C.

Hosting and download

You can always find the source code for the latest curl and libcurl release on the official curl website curl.se. There are also checksums and digital signatures provided to help you verify that what ends up on your local system when you download the files, are the same bytes in the same order as were originally uploaded there by the curl team.

If you would rather work directly with the curl source code off the source code repository, you find all details in [the curl GitHub repository](#).

Clone the code

```
1 git clone https://github.com/curl/curl.git
```

This will get the latest curl code downloaded and unpacked in a directory on your local system.

Open Source

What is Open Source

Generally, Open Source software is software that can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone. Open Source software is typically made by many people, and distributed under licenses that comply with the definition.

Free Software is an older and related term that mostly says the same thing for all our intents and purposes, but we stick to the term Open Source in this document for simplicity.

License

curl and libcurl are distributed under an Open Source license known as a MIT license derivative. It is short, simple and easy to grasp. It follows here in full:

```
1 COPYRIGHT AND PERMISSION NOTICE
2
3 Copyright © 1996 – 2022, Daniel Stenberg, <daniel@haxx.se>, and many
4 contributors, see the THANKS file.
5
6 All rights reserved.
7
8 Permission to use, copy, modify, and distribute this software for any purpose
9 with or without fee is hereby granted, provided that the above copyright
10 notice and this permission notice appear in all copies.
11
12 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
13 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
14 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN
15 NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
16 DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
17 OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
18 OR OTHER DEALINGS IN THE SOFTWARE.
19
20 Except as contained in this notice, the name of a copyright holder shall not
21 be used in advertising or otherwise to promote the sale, use or other dealings
22 in this Software without prior written authorization of the copyright holder.
```

This is legalese that says you are allowed to change the code, redistribute the code, redistribute binaries built from the code and build proprietary code with it, without anyone requiring you to give any changes back to the project—but you may not claim that you wrote it.

Early on in the project we iterated over a few different other licenses before we settled on this. We started out GPL, then tried MPL and landed on this MIT derivative. We will never change the license again.

Copyright and Legal

Copyright

Copyright is a legal right granted by the law of a country that gives the creator of an original work exclusive rights for its use and distribution.

The copyright owner(s) can agree to allow others to use their work by licensing it. That is what we do in the curl project. The copyright is the foundation on which the licensing works.

Daniel Stenberg is the owner of most copyrights in the curl project.

Independent

A lot of Open Source projects are run within umbrella organizations. Such organizations include the GNU project, the Apache Software Foundation, a larger company that funds the project or similar. The curl project is not part of any such larger organization but is completely independent and free.

No company controls curl's destiny and the curl project does not need to follow any umbrella organization's guidelines.

curl is not a formal company, organization or a legal entity of any kind. curl is just an informal collection of humans, distributed across the globe, who work together on a software project.

Legal

The curl project obeys national laws of the countries in which it works. However, it is a highly visible international project, downloadable and usable in effectively every country on earth, so some local laws could be broken when using curl. That is just the nature of it and if uncertain, you should check your own local situation.

There have been law suits involving technology that curl provides. One such case known to the author of this was a patent case in the US that insisted they had the rights to resumed file transfers.

As a generic software component that is usable everywhere to everyone, there are times when libcurl—in particular—is used in nefarious or downright malicious ways. Examples include being used in virus and malware software. That is unfortunate but nothing we can prevent.

Code layout

The curl source code tree is neither large nor complicated. A key thing to remember is, that libcurl is the library and that this library is the biggest component of the curl command-line tool.

root

We try to keep the number of files in the source tree root to a minimum. You will see a slight difference in files if you check a release archive compared to what is stored in the git repository as several files are generated by the release scripts.

Some of the more notable ones include:

- `buildconf` : used to build configure and more when building curl from source out of the git repository.
- `buildconf.bat` : the Windows version of buildconf. Run this after having checked out the full source code from git.
- `CHANGES` : generated at release and put into the release archive. It contains the 1000 latest changes to the source repository.
- `configure` : a generated script that is used on Unix-like systems to generate a setup when building curl.
- `COPYING` : the license detailing the rules for your using the code.

- `GIT-INFO` : only present in git and contains information about how to build curl after having checked out the code from git.
 - `maketgz` : the script used to produce release archives and daily snapshots
 - `README` : a short summary of what curl and libcurl are.
 - `RELEASE-NOTES` : contains the changes done for the latest release; when found in git it contains the changes done since the previous release that are destined to end up in the coming release.
-

lib

This directory contains the full source code for libcurl. It is the same source code for all platforms—over one hundred C source files and a few more private header files. The header files used when building applications against libcurl are not stored in this directory; see `include/curl` for those.

Depending on what features are enabled in your own build and what functions your platform provides, some of the source files or portions of the source files may contain code that is not used in your particular build.

lib/vtls

The VTLS sub section within libcurl is the home of all the TLS backends libcurl can be built to support. The "virtual" TLS internal API is a backend agnostic API used internally to access TLS and crypto functions without the main code knowing which specific TLS library is used. This allows the person who builds libcurl to select from a wide variety TLS libraries to build with.

We also maintain a [SSL comparison table](#) on the website to aid users.

- AmiSSL: an OpenSSL fork made for AmigaOS (uses `openssl.c`)
- BearSSL
- BoringSSL: an OpenSSL fork maintained by Google. (uses `openssl.c`)
- GnuTLS
- GSKit: the native TLS library on OS/400.
- LibreSSL: an OpenSSL fork maintained by the OpenBSD team. (uses `openssl.c`)
- mbedTLS
- NSS: TLS library most known for being used by the Firefox web browser.
- OpenSSL
- rustls: a TLS library written in rust
- Schannel: the native TLS library on Windows.
- Secure Transport: the native TLS library on macOS
- wolfSSL

src

This directory holds the source code for the curl command-line tool. It is the same source code for all platforms that run the tool.

Most of what the command-line tool does is to convert given command line options into the corresponding libcurl options or set of options and then makes sure to issue them correctly to drive the network transfer according to the user's wishes.

This code uses libcurl just as any other application would.

include/curl

Here are the public header files that are provided for libcurl-using applications. Some of them are generated at configure or release time so they do not look identical in the git repository as they do in a release archive.

With modern libcurl, all an application is expected to include in its C source code is `#include <curl/curl.h>`

docs

The main documentation location. Text files in this directory are typically plain text files. We have slowly started to move towards Markdown format so a few (but growing number of) files use the `.md` extension to signify that.

Most of these documents are also shown on the curl website automatically converted from text to a web friendly format/look.

- `BINDINGS` : lists all known libcurl language bindings and where to find them
- `BUGS` : how to report bugs and where
- `CODE_OF_CONDUCT.md` : how we expect people to behave in this project
- `CONTRIBUTE` : what to think about when contributing to the project
- `curl.1` : the curl command-line tool man page, in nroff format
- `curl-config.1` : the curl-config man page, in nroff format
- `FAQ` : frequently asked questions about various curl-related subjects
- `FEATURES` : an incomplete list of curl features
- `HISTORY` : describes how the project started and has evolved over the years
- `HTTP2.md` : how to use HTTP/2 with curl and libcurl
- `HTTP-COOKIES` : how curl supports and works with HTTP cookies

- `index.html` : a basic HTML page as a documentation index page
 - `INSTALL` : how to build and install curl and libcurl from source
 - `INSTALL.cmake` : how to build curl and libcurl with CMake
 - `INSTALL.devcpp` : how to build curl and libcurl with devcpp
 - `INTERNAL` : details curl and libcurl internal structures
 - `KNOWN_BUGS` : list of known bugs and problems
 - `LICENSE-MIXING` : describes how to combine different third party modules and their individual licenses
 - `MAIL-ETIQUETTE` : this is how to communicate on our mailing lists
 - `MANUAL` : a tutorial-like guide on how to use curl
 - `mk-ca-bundle.1` : the mk-ca-bundle tool man page, in nroff format
 - `README.cmake` : CMake details
 - `README.netware` : Netware details
 - `README.win32` : win32 details
 - `RELEASE-PROCEDURE` : how to do a curl and libcurl release
 - `RESOURCES` : further resources for further reading on what, why and how curl does things
 - `ROADMAP.md` : what we want to work on in the future
 - `SECURITY` : how we work on security vulnerabilities
 - `SSLCERTS` : TLS certificate handling documented
 - `SSL-PROBLEMS` : common SSL problems and their causes
 - `THANKS` : thanks to this extensive list of friendly people, curl exists today!
 - `TheArtOfHttpScripting` : a tutorial into HTTP scripting with curl
 - `TODO` : things we or you can work on implementing
 - `VERSIONS` : how the version numbering of libcurl works
-

docs/libcurl

All libcurl functions have their own man pages in individual files with .3 extensions, using nroff format, in this directory. There are also a few other files that are described below.

- `ABI`
- `index.html`
- `libcurl.3`
- `libcurl-easy.3`
- `libcurl-errors.3`

- `libcurl.m4`
 - `libcurl-multi.3`
 - `libcurl-share.3`
 - `libcurl-thread.3`
 - `libcurl-tutorial.3`
 - `symbols-in-versions`
-

docs/libcurl/opts

This directory contains the man pages for the individual options for three different libcurl functions.

`curl_easy_setopt()` options start with `CURLOPT_`, `curl_multi_setopt()` options start with `CURLOPT_` and `curl_easy_getinfo()` options start with `CURLINFO_`.

docs/examples

Contains around 100 stand-alone examples that are meant to help readers understand how libcurl can be used.

See also the [libcurl examples](#) section of this book.

scripts

Handy scripts.

- `contributors.sh`: extracts all contributors from the git repository since a given hash/tag. The purpose is to generate a list for the RELEASE-NOTES file and to allow manually added names to remain in there even on updates. The script uses the `THANKS-filter` file to rewrite some names.
- `contrithanks.sh`: extracts contributors from the git repository since a given hash/tag, filters out all the names that are already mentioned in `THANKS`, and then outputs `THANKS` to stdout with the list of new contributors appended at the end; it's meant to allow easier updates of the THANKS document. The script uses the `THANKS-filter` file to rewrite some names.
- `log2changes.pl`: generates the `CHANGES` file for releases, as used by the release script. It simply converts git log output.
- `zsh.pl`: helper script to provide curl command-line completions to users of the zsh shell.

Handling build options

The curl and libcurl source code has been carefully written to build and run on virtually every computer platform in existence. This can only be done through hard work and by adhering to a few guidelines (and, of course, a fair amount of testing).

A golden rule is to always add #ifdefs that checks for specific features, and then have the setup scripts (configure or CMake or hard-coded) check for the presence of said features in a user's computer setup before the program is compiled there. Additionally and as a bonus, thanks to this way of writing the code, some features can be explicitly turned off even if they are present in the system and *could* be used. Examples of that would be when users want to, for example, build a version of the library with a smaller footprint or with support for certain protocols disabled, etc.

The project sometimes uses #ifdef protection around entire source files when, for example, a single file is provided for a specific operating system or perhaps for a specific feature that is not always present. This is to make it possible for all platforms to always build all files—it simplifies the build scripts and makefiles a lot. A file entirely #ifndefed out hardly adds anything to the build time, anyway.

Rather than sprinkling the code with #ifdefs, to the extent where it is possible, we provide functions and macros that make the code look and work the same, independent of present features. Some of those are then empty macros for the builds that lack the features.

Both TLS handling and name resolving are handled with an internal API that hides the specific implementation and choice of 3rd party software library. That way, most of the internals work the same independent of which TLS library or name resolving system libcurl is told to use.

Code style

Source code that has a common style is easier to read than code that uses different styles in different places. It helps making the code feel like one continuous code base. Easy-to-read is an important property of code and helps make it easier to review when new things are added and it helps debugging code when developers are trying to figure out why things go wrong. A unified style is more important than individual contributors having their own personal tastes satisfied.

Our C code has a few style rules. Most of them are verified and upheld by the `checksrc.pl` script. Invoked with `make checksrc` or even by default by the build system when built after `./configure --enable-debug` has been used.

It is normally not a problem for anyone to follow the guidelines as you just need to copy the style already used in the source code, and there are no particularly unusual rules in our set of rules.

We also work hard on writing code that is warning-free on all the major platforms and in general on as many platforms as possible. Code that obviously will cause warnings will not be accepted as-is.

Naming

Try using a non-confusing naming scheme for your new functions and variable names. It does not

necessarily have to mean that you should use the same as in other places of the code, just that the names should be logical, understandable and be named according to what they are used for. File-local functions should be made static. We like lower case names.

All symbols meant for public use must start with `curl`. Global internal symbols start with `Curl`.

Indentation

We use only spaces for indentation, never TABs. We use two spaces for each new open brace.

```
1 if(something_is_true) {  
2     while(second_statement == fine) {  
3         moo();  
4     }  
5 }
```

Comments

Since we write C89 code, `//` comments are not allowed. They were not introduced in the C standard until C99. We use only `/* comments */`.

```
1 /* this is a comment */
```

Long lines

Source code in curl may never be wider than 79 columns. There are two reasons for maintaining this even in the modern era of large and high resolution screens:

1. Narrower columns are easier to read than wide ones. There is a reason newspapers have used columns for decades or centuries.
 2. Narrower columns allow developers to more easily view multiple pieces of code next to each other in different windows. I often have two or three source code windows next to each other on the same screen, as well as multiple terminal and debugging windows.
-

Braces

In if/while/do/for expressions, we write the open brace on the same line as the keyword and we then set the closing brace on the same indentation level as the initial keyword. Like this:

```
1 if(age < 40) {  
2     /* clearly a youngster */  
3 }
```

You may omit the braces if they would contain only a one-line statement:

```
1 if(!x)  
2     continue;
```

For functions the opening brace should be on a separate line:

```
1 int main(int argc, char **argv)  
2 {  
3     return 1;  
4 }
```

else on the following line

When adding an `else` clause to a conditional expression using braces, we add it on a new line after the closing brace. Like this:

```
1 if(age < 40) {  
2     /* clearly a youngster */  
3 }  
4 else {  
5     /* probably intelligent */  
6 }
```

No space before parentheses

When writing expressions using if/while/do/for, there shall be no space between the keyword and the open parenthesis. Like this:

```
1 while(1) {  
2     /* loop forever */  
3 }
```

~~Use boolean conditions~~

Rather than test a conditional value such as a bool against TRUE or FALSE, a pointer against NULL or != NULL and an int against zero or not zero in if/while conditions we prefer:

```
1 result = do_something();
2 if(!result) {
3     /* something went wrong */
4     return result;
5 }
```

No assignments in conditions

To increase readability and reduce complexity of conditionals, we avoid assigning variables within if/while conditions. We frown upon this style:

```
1 if((ptr = malloc(100)) == NULL)
2     return NULL;
```

and instead we encourage the above version to be spelled out more clearly:

```
1 ptr = malloc(100);
2 if(!ptr)
3     return NULL;
```

New block on a new line

We never write multiple statements on the same source line, even for short if() conditions.

```
1 if(a)
2     return TRUE;
3 else if(b)
4     return FALSE;
```

and NEVER:

```
1 if(a) return TRUE;
2 else if(b) return FALSE;
```

~~General coding conventions~~

Please use spaces on both sides of operators in C expressions. Postfix `0`, `[], ->, .., ++, --` and Unary `+, -, !, ~, &` operators excluded they should have no space.

Examples:

```
1 bla = func();
2 who = name[0];
3 age += 1;
4 true = !false;
5 size += -2 + 3 * (a + b);
6 ptr->member = a++;
7 struct.field = b--;
8 ptr = &address;
9 contents = *pointer;
10 complement = ~bits;
11 empty = (!*string) ? TRUE : FALSE;
```

No parentheses for return values

We use the 'return' statement without extra parentheses around the value:

```
1 int works(void)
2 {
3     return TRUE;
4 }
```

Parentheses for sizeof arguments

When using the sizeof operator in code, we prefer it to be written with parentheses around its argument:

```
1 int size = sizeof(int);
```

Column alignment

Some statements cannot be completed on a single line because the line would be too long, the statement too hard to read, or due to other style guidelines above. In such a case the statement will span multiple lines.

If a continuation line is part of an expression or sub-expression then you should align on the appropriate column so that it's easy to tell what part of the statement it is. Operators should not start continuation lines. In other cases follow the 2-space indent guideline. Here are some examples from libcurl:

```

1 if(Curl_pipeline_wanted(handle->multi, CURLPIPE_HTTP1) &&
2     (handle->set.httpversion != CURL_HTTP_VERSION_1_0) &&
3     (handle->set.httpreq == HTTPREQ_GET || 
4      handle->set.httpreq == HTTPREQ_HEAD))
5 /* did not ask for HTTP/1.0 and a GET or HEAD */
6 return TRUE;

```

If no parenthesis, use the default indent:

```

1 data->set.http_disable_hostname_check_before_authentication =
2 (0 != va_arg(param, long)) ? TRUE : FALSE;

```

Function invoke with an open parenthesis:

```

1 if(option) {
2     result = parse_login_details(option, strlen(option),
3                                     (userp ? &user : NULL),
4                                     (passwdp ? &passwd : NULL),
5                                     NULL);
6 }

```

Align with the "current open" parenthesis:

```

1 DEBUGF(infof(data, "Curl_pp_readresp_ %d bytes of trailing "
2                 "server response left\n",
3                 (int)clipamount));

```

Platform dependent code

Use `#ifdef HAVE_FEATURE` to do conditional code. We avoid checking for particular operating systems or hardware in the `#ifdef` lines. The `HAVE_FEATURE` shall be generated by the configure script for unix-like systems and they are hard-coded in the `config-[system].h` files for the others.

We also encourage use of macros/functions that possibly are empty or defined to constants when libcurl is built without that feature, to make the code seamless. Like this example where the `magic()` function works differently depending on a build-time conditional:

```

1 #ifdef HAVE_MAGIC
2 void magic(int a)
3 {
4     return a + 2;
5 }
6 #else
7 #define magic(x) 1

```

```
8 #endif  
9  
10 int content = magic(3);
```

No typedefed structs

Use structs by all means, but do not typedef them. Use the `struct` name way of identifying them:

```
1 struct something {  
2     void *valid;  
3     size_t way_to_write;  
4 };  
5 struct something instance;
```

Not okay:

```
1 typedef struct {  
2     void *wrong;  
3     size_t way_to_write;  
4 } something;  
5 something instance;
```

Contributing

Contributing

Contributing means helping out.

When you contribute anything to the project—code, documentation, bug fixes, suggestions or just good advice—we assume you do this with permission and you are not breaking any contracts or laws by providing that to us. If you do not have permission, do not contribute it to us.

Contributing to a project like curl could be many different things. While source code is the stuff that is needed to build the products, we are also depending on good documentation, testing (both test code and test infrastructure), web content, user support and more.

Send your changes or suggestions to the team and by working together we can fix problems, improve functionality, clarify documentation, add features or make anything else you help out with land in the proper place. We will make sure improved code and docs get merged into the source tree properly and other sorts of contributions are suitable received.

Send your contributions on a [mailing list](#), file an issue or submit a pull request.

Suggestions

Ideas are easy, implementations are hard. Yes, we do appreciate good ideas and suggestions of what to do and how to do it, but the chances that the ideas actually turn into real features grow substantially if you also volunteer to participate in converting the idea into reality.

We already gather ideas in the `TODO` document and we are generally aware of the current trends in the popular networking protocols so there is usually no need to remind us about those.

What to add

The best approach to add anything to curl or libcurl is, of course, to first bring the idea and suggestion to the curl project team members and then discuss with them if the idea is feasible for inclusion and then how an implementation is best done—and done in the best possible way to get merged into the source code repository, assuming that is what you want.

The project generally approves of functions that improves the support for the current protocols, especially features that popular clients or browsers have but that curl still lacks.

Of course, you can also add contents to the project that is not code, like documentation, graphics or website contents, but the general rules apply equally to that.

If you are fixing a problem you have or a problem that others are reporting, we will be thrilled to receive your fix and merge it as soon as possible,

What not to add

There are no good rules that say what features you can or cannot add or that we will never accept, but let me instead try to mention a few things you should avoid to get less friction and to be successful, faster:

- Do not write up a huge patch first and then send it to the list for discussion. Always start out by discussing on the list, and send your initial review requests early to get feedback on your design and approach. It saves you from wasting time going down a route that might need rewriting in the end anyway.
- When introducing things in the code, you need to follow the style and architecture that already exists. When you add code to the ordinary transfer code path, it must, for example, work asynchronously in a non-blocking manner. We will not accept new code that introduces blocking behaviors—we already have too many of those that we have not managed to remove yet.
- Quick hacks or dirty solutions that have a high risk of not working on platforms you do not run or on architectures you do not know. We do not care if you are in a hurry or that it works for you. We do not accept high risk code or code that is hard to read or understand.
- Code that breaks the build. Sure, we accept that we sometimes have to add code to certain areas that makes the new functionality perhaps depend on a specific 3rd party library or a specific operating system and similar, but we can **never** do that at the expense of all other systems. We do not break the build, and we make sure all tests keep running successfully.

git

Our preferred source control tool is `git`.

While git is sometimes not the easiest tool to learn and master, all the basic steps a casual developer and contributor needs to know are straight-forward and do not take much time or effort to learn.

This book will not help you learn git. All software developers in this day and age should learn git anyway.

The curl git tree can be browsed with a web browser on our GitHub page at <https://github.com/curl/curl>.

To check out the curl source code from git, you can clone it like this:

```
1 git clone https://github.com/curl/curl.git
```

Pull request

A popular and convenient way to make your own changes and contribute them back to the project is by doing a so-called pull request on GitHub.

First, you create your own version of the source tree, called a fork, on the Github website. That way you get your own version of the curl git tree that you can clone to a local copy.

You edit your own local copy, commit the changes, push them to the git repository on Github and then on the Github website you can select to create a pull request based on your changes done to your local repository clone of the original curl repository.

We recommend doing your work meant for a pull request in a dedicated separate branch and not in master, just to make it easier for you to update a pull request, like after review, for example, or if you realize it was a dead end and you decide to just throw it away.

Make a patch for the mailing list

Even if you opt to not make a pull request but prefer the old fashioned and trusted method of sending a patch to the curl-library mailing list, it is still a good to work in a local git branch and commit your changes there.

A branch makes it easy to edit and rebase when you need to change things and it makes it easy to keep syncing to the master branch when things are updated upstream.

Once your commits are fine enough to get sent to the mailing list, you just create patches with `git format-patch` and send them away. Even more fancy users go directly to `git send-email` and have git send the email itself.

git commit style

When you commit a patch to git, you give it a commit message that describes the change you are committing. We have a certain style in the project that we ask you to use:

```
1 [area]: [short line describing the main effect]
2
3 [separate the above single line from the rest with an empty line]
4
5 [full description, no wider than 72 columns that describes as much as
6 possible as to why this change is made, and possibly what things
```

```
8 [it fixes and everything else that is related]
9 [Bug: link to source of the report or more related discussion]
10 [Reported-by: John Doe—credit the reporter]
11 [whatever-else-by: credit all helpers, finders, doers]
```

Do not forget to use `git commit --author="Jane Doe <jane@example.com>"` if you commit someone else's work, and make sure that you have your own Github username and email setup correctly in git before you commit via commands below:

```
1 git config --global user.name "johndoe"
2 git config --global user.email "johndoe@example.com"
```

The author and the *-by: lines are, of course, there to make sure we give the proper credit in the project. We do not want to take someone else's work without clearly attributing where it comes from. Giving correct credit is of utmost importance.

Who decides what goes in?

First, it might not be obvious to everyone but there is, of course, only a limited set of people that can actually merge commits into the actual official git repository. Let's call them the core team.

Everyone else can fork off their own curl repository to which they can commit and push changes and host them online and build their own curl versions from and so on, but in order to get changes into the *official* repository they need to be pushed by a trusted person.

The core team is a small set of curl developers who have been around for several years and have shown they are skilled developers and that they fully comprehend the values and the style of development we do in this project. They are some of the people listed in the [The development team](#) section.

You can always bring a discussion to the mailing list and argue why you think your changes should get accepted, or perhaps even object to other changes that are getting in and so forth. You can even suggest yourself or someone else to be given "push rights" and become one of the selected few in that team.

Daniel remains the project leader and while it is rarely needed, he has the final say in debates that do not seem to sway in either direction or fail to reach consensus.

Reporting vulnerabilities

All known and public curl or libcurl related vulnerabilities are listed on [the curl website security page](#).

Security vulnerabilities should not be entered in the project's public bug tracker unless the necessary configuration is in place to limit access to the issue to only the reporter and the project's security team.

Vulnerability handling

The typical process for handling a new security vulnerability is as follows.

No information should be made public about a vulnerability until it is formally announced at the end of this process. That means, for example, that a bug tracker entry must NOT be created to track the issue since that will make the issue public and it should not be discussed on any of the project's public mailing lists. Also messages associated with any commits should not make any reference to the security nature of the commit if done prior to the public announcement.

- The person discovering the issue, the reporter, reports the vulnerability on <https://hackerone.com/curl>. Issues filed there reach a handful of selected and trusted people.
- Messages that do not relate to the reporting or managing of an undisclosed security vulnerability in curl or libcurl are ignored and no further action is required.
- A person in the security team sends an email to the original reporter to acknowledge the report.
- The security team investigates the report and either rejects it or accepts it.
- If the report is rejected, the team writes to the reporter to explain why.
- If the report is accepted, the team writes to the reporter to let him/her know it is accepted and that they are working on a fix.
- The security team discusses the problem, works out a fix, considers the impact of the problem and suggests a release schedule. This discussion should involve the reporter as much as possible.
- The release of the information should be "as soon as possible" and is most often synced with an upcoming release that contains the fix. If the reporter, or anyone else, thinks the next planned release is too far away then a separate earlier release for security reasons should be considered.
- Write a security advisory draft about the problem that explains what the problem is, its impact, which versions it affects, any solutions or workarounds and when the fix was released, making sure to credit all contributors properly.
- Request a CVE number ([Common Vulnerabilities and Exposures](#)) using HackerOne's form for this purpose.
- Update the "security advisory" with the CVE number.
- Consider informing distros@openwall to prepare them about the upcoming public security vulnerability announcement - attach the advisory draft for information. Note that 'distros' do not accept an embargo longer than 14 days and they do not care for Windows-specific flaws.
- The security team commits the fix in a private branch. The commit message should ideally contain the CVE number. This fix is usually also distributed to the 'distros' mailing list to allow them to use the fix prior to the public announcement.
- At the day of the next release, the private branch is merged into the master branch and pushed. Once pushed, the information is accessible to the public and the actual release should follow suit immediately afterwards.
- The project team creates a release that includes the fix.
- The project team announces the release and the vulnerability to the world in the same manner we always announce releases—it gets sent to the curl-announce, curl-library and curl-users mailing lists.
- The security web page on the website should get the new vulnerability mentioned.

curl-security@haxx.se

Who is on this list? There are a couple of criteria you must meet, and then we might ask you to join the list or you can ask to join it. It really is not formal. We only require that you have a long-term presence in the curl project and you have shown an understanding for the project and its way of working. You must have been around for a good while and you should have no plans on vanishing in the near future.

We do not make the list of participants public mostly because it tends to vary somewhat over time and a list somewhere will only risk getting outdated.

Website

Most of the curl website is also available in a public git repository, although separate from the source code repository since it generally is not interesting to the same people and we can maintain a different list of people that have push rights, etc.

The website git repository is available on GitHub at this URL: <https://github.com/curl/curl-www> and you can clone a copy of the web code like this:

```
1 git clone https://github.com/curl/curl-www.git
```

Building the web

The website is a custom-made setup that mostly builds static HTML files from a set of source files. The source files are preprocessed with what is a souped-up C preprocessor called [fcpp](#) and a set of perl scripts. The man pages get converted to HTML with [roffit](#). Make sure fcpp, perl, roffit, make and curl are all in your \$PATH.

Once you have cloned the git repository the first time, invoke `sh bootstrap.sh` once to get a symlink and some initial local files setup, and then you can build the website locally by invoking `make` in the source root tree.

Note that this does not make you a complete website mirror, as some scripts and files are only available on the real actual site, but should give you enough to let you view most HTML pages locally.

Run a local clone

The website is built in a way that makes it easy and convenient to host a local copy for browsing and testing changes before we push them to the official site in production. We then recommend you call the site `curl.local` and add that as an entry in your local `/etc/hosts` file. Then point the document root of your HTTP server to the curl-www source code root.

Website infrastructure

- The public curl website is hosted at [curl.se](#).
- The domain name is owned by **Daniel Stenberg**
- The main origin machine is sponsored by **Haxx**
- The curl.se domain is served by anycast distributed DNS servers sponsored by **Kirei**
- The site is delivered to the world via a CDN run by **Fastly**
- The website updates itself from GitHub every N minutes. The CDN front-ends cache content for Y minutes (different types cache content different times)

Build curl

The source code for this project is written in a way that allows it to get compiled and built on just about any operating system and platform, with as few restraints and requirements as possible.

If you have a 32bit (or larger) CPU architecture, if you have a C89 compliant compiler and if you have roughly a POSIX supporting sockets API, then you can most likely build curl and libcurl for your target system.

For the most popular platforms, the curl project comes with build systems already done and prepared to allow you to easily build it yourself.

There are also friendly people and organizations who put together binary packages of curl and libcurl and make them available for download. The different options will be explored below.

The latest version?

Looking at the [curl website](#), you can see the latest curl and libcurl version released from the project. That is the latest source code release package you can get.

When you opt for a prebuilt and prepackaged version for your operating system or distribution of choice, you may not always find the latest version but you might have to either be satisfied with the latest version someone has packaged for your environment, or you need to build it yourself from source.

The curl project also provides info about the latest version in a somewhat more machine-readable format on this URL: <https://curl.se/info>.

Releases source code

The curl project creates source code that can be built to produce the two products curl and libcurl. The conversion from source code to binaries is often referred to as "building". You build curl and libcurl from source.

The curl project does not provide any built binaries at all — it only ships the source code. The binaries which can be found on the download page of the curl web and installed from other places on the Internet are all built and provided to the world by other friendly people and organizations.

The source code consists of a large number of files containing C code. Generally speaking, the same set of files are used to build binaries for all platforms and computer architectures that curl supports. curl can be built and run on a vast number of platforms. If you use a rare operating system yourself, chances are that building curl from source is the easiest or perhaps the only way to get curl.

Making it easy to build curl is a priority to the curl project, although we do not always necessarily succeed!

git vs release tarballs

When release tarballs are created, a few files are generated and included in the final release bundle. Those generated files are not present in the git repository, because they are generated and there is no need to store them in git.

Of course, can also opt to build the latest version that exist in the [git repository](#). It could however be a bit more fragile and probably requires slightly more attention to detail.

If you build curl from a git checkout, you need to generate some files yourself before you can build. On Linux and Unix-like systems, do this by running `./buildconf` and on Windows, run `buildconf.bat`.

On Linux and Unix-like systems

There are two distinctly different ways to build curl on Linux and other Unix-like systems; there is the one using [the configure script](#) and there is [the CMake approach](#).

There are two different build environments to cater to people's different opinions and tastes. The configure-based build is arguably the more mature and more encompassing build system and should probably be considered the default one.

On Windows

On Windows there are at least four different ways to build. The above mentioned ways, [the CMake approach](#) and using [configure](#) with msys work, but the more popular and common methods are probably building with Microsoft's Visual Studio compiler using either `nmake` or project files. See the build on [windows](#) section.

Learn more

- [Autotools](#) - build with configure
- [CMake](#)
- [On Windows](#) - Windows-specific ways to build
- [Dependencies](#)
- [TLS libraries](#)

Autotools

The "Autotools" are a collection of different tools that used together generate the `configure` script. The `configure` script is run by the user who wants to build curl and it does a whole bunch of things:

- It checks for features and functions present in your system.
- It offers command-line options so that you as a builder can decide what to enable and disable in the build. Features and protocols, etc., can be toggled on/off, even compiler warning levels and more.
- It offers command-line options to let the builder point to specific installation paths for various third-party dependencies that curl can be built to use.
- It specifies on which file path the generated installation should be placed when ultimately the build is made and "make install" is invoked.

In the most basic usage, just running `./configure` in the source directory is enough. When the script completes, it outputs a summary of what options it has detected/enabled and what features that are still disabled, some of which possibly because it failed to detect the presence of necessary third-party dependencies that are needed for those functions to work. If the summary is not what you expected it to be, invoke `configure` again with new options or with the previously used options adjusted.

After `configure` has completed, you invoke `make` to build the entire thing and then finally `make install` to install curl, libcurl and associated things. `make install` requires that you have the correct rights in your system to create and write files in the installation directory or you will get some errors.

Cross-compiling

Cross-compiling means that you build the source on one architecture but the output is created to be run on a different one. For example, you could build the source on a Linux machine but have the output work on a Windows machine.

For cross-compiling to work, you need a dedicated compiler and build system setup for the particular target system for which you want to build. How to get and install that system is not covered in this book.

Once you have a cross compiler, you can instruct `configure` to use that compiler instead of the "native" compiler when it builds curl so that the end result then can be moved over and used on the other machine.

Static linking

By default, configure will setup the build files so that the following 'make' command will create both shared and static versions of libcurl. You can change that with the `--disable-static` or `--disable-shared` options to configure.

If you instead want to build with static versions of third party libraries instead of shared libraries, you need to prepare yourself for an uphill battle. curl's configure script is focused on setting up and building with shared libraries.

One of the differences between linking with a static library compared to linking with a shared one is in how shared libraries handle their own dependencies while static ones do not. In order to link with library `xyz` as a shared library, it is as basically a matter of adding `-lxyz` to the linker command line no matter which other libraries `xyz` itself was built to use. But, if that `xyz` is instead a static library we also need to specify each dependency of `xyz` on the linker command line. curl's configure cannot keep up with or know all possible dependencies for all the libraries it can be made to build with, so users wanting to build with static libs mostly need to provide that list of libraries to link with.

Select TLS backend

The configure-based build offers the user to select from a wide variety of different TLS libraries when building. You select them by using the correct command line options. Before curl 7.77.0, the configure script would automatically check for OpenSSL, but modern versions do not.

- AmiSSL: `--with-amissl`
- BearSSL: `--with-bearssl`
- BoringSSL: `--withOpenssl`
- GnuTLS: `--with-gnutls`
- LibreSSL: `--with-openssl`
- mbedTLS: `--with-mbedtls`
- NSS: `--with-nss`
- OpenSSL: `--with-openssl`
- Rustls: `--with-rustls` (point to the rustls-ffi install path)
- Schannel: `--with-schannel`
- Secure Transport: `--with-secure-transport`
- wolfSSL: `--with-wolfssl`

If you do not specify which TLS library to use, the configure script will fail. If you want to build *without* TLS support, you must explicitly ask for that with `--without-ssl`.

These `--with-*` options also allow you to provide the install prefix so that configure will search for the specific library where you tell it to. Like this:

```
1 ./configure --with-gnutls=/home/user/custom-gnutls
```

You can opt to build with support for **multiple** TLS libraries by specifying multiple `--with-*` options on the configure command line. Pick which one to make the default TLS backend with `--with-default-ssl-backend=[NAME]`. For example, build with support for both GnuTLS and OpenSSL and default to OpenSSL:

```
1 ./configure --with-openssl --with-gnutls --with-default-ssl-backend=openssl
```

Select SSH backend

The configure-based build offers the user to select from a variety of different SSH libraries when building. You select them by using the correct command-line options.

- libssh2: `--with-libssh2`
- libssh: `--with-libssh`
- wolfSSH: `--with-wolfssh`

These `--with-*` options also allow you to provide the install prefix so that configure will search for the specific library where you tell it to. Like this:

```
1 ./configure --with-libssh2=/home/user/custom-libssh2
```

Select HTTP/3 backend

The configure-based build offers the user to select different HTTP/3 libraries when building. You select them by using the correct command-line options.

- quiche: `--with-quiche`
- ngtcp2: `--with-ngtcp2 --with-nghttp3`

CMake

CMake is an alternative build method that works on most modern platforms, including Windows. Using this method you first need to have cmake installed on your build machine, invoke cmake to generate the build files and then build. With cmake's `-G` flag, you select which build system to generate files for. See `cmake --help` for the list of "generators" your cmake installation supports.

On the cmake command line, the first argument specifies where to find the cmake source files, which is `.` (a single dot) if in the same directory.

To build on Linux using plain make with CMakeLists.txt in the same directory, you can do:

```
1 cmake -G "Unix Makefiles" .
2 make
```

Or rely on the fact that unix makefiles are the default there:

```
1 cmake .
2 make
```

To create a subdirectory for the build and run make in there:

```
1 mkdir build
2 cd build
3 cmake ..
4 make
```

Windows

You can build curl on Windows in several different ways. We recommend using the MSVC compiler from Microsoft or the free and open source mingw compiler. The build process is, however, not limited to these.

If you use mingw, you might want to use the [autotools](#) build system.

winbuild

This is how to build curl and libcurl using the command line.

Build with MSVC using the `nmake` utility like this:

```
1 cd winbuild
```

Decide what options to enable/disable in your build. The `README.md` file in that directory details them all, but an example command line could look like this (split into several lines for readability):

```
1 nmake WITH_SSL=dll WITH_NGHTTP2=dll ENABLE_IPV6=yes \
2 WITH_ZLIB=dll MACHINE=x64
```

Visual C++ project files

curl tarballs ship with pre-generated project files that you can load and build curl with.

Project files are provided for several different Visual C++ versions.

To build with VC++, you will of course have to first install VC++ which is part of Visual Studio.

Once you have VC++ installed you should launch the application and open one of the solution or workspace files. The VC directory names are based on the version of Visual C++ that you will be using. Each version of Visual Studio has a default version of Visual C++. We offer these versions:

- VC10 (Visual Studio 2010 Version 10.0)
- VC11 (Visual Studio 2012 Version 11.0)
- VC12 (Visual Studio 2013 Version 12.0)
- VC14 (Visual Studio 2015 Version 14.0)
- VC14.10 (Visual Studio 2017 Version 15.0)
- VC14.30 (Visual Studio 2022 Version 17.0)

Separate solutions are provided for both libcurl and the curl command line tool as well as a solution that includes both projects. libcurl.sln, curl.sln and curl-all.sln, respectively. We recommend using curl-all.sln to build both projects.

For example, if you are using Visual Studio 2022 then you should be able to use `VC14.30\curl-all.sln` to build curl and libcurl.

Running DLL based configurations

If you are a developer and plan to run the curl tool from Visual Studio (eg you are debugging) with any third-party libraries (such as OpenSSL, wolfSSL or libSSH2) then you will need to add the search path of these DLLs to the configuration's PATH environment. To do that:

1. Open the 'curl-all.sln' or 'curl.sln' solutions
2. Right-click on the 'curl' project and select Properties
3. Navigate to 'Configuration Properties > Debugging > Environment'
4. Add `PATH='Path to DLL';C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem`

```
DLL';C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem
```

... where 'Path to DLL` is the configuration specific path. For example the following configurations in Visual Studio 2010 might be:

DLL Debug - DLL OpenSSL (Win32):

```
1 PATH=..\..\..\..\..\openssl\build\Win32\VC10\DLL_Debug;C:\Windows\system32;
2 C:\Windows;C:\Windows\System32\Wbem
```

DLL Debug - DLL OpenSSL (x64):

```
1 PATH=..\..\..\..\..\..\openssl\build\Win64\VC10\DLL_Debug;C:\Windows\system32;
2 C:\Windows;C:\Windows\System32\Wbem
```

DLL Debug - DLL wolfSSL (Win32):

DLL Debug - DLL wolfSSL (x64):

If you are using a configuration that uses multiple third-party library DLLs (such as `DLL Debug - DLL OpenSSL - DLL LibSSH2`) then `Path to DLL` will need to contain the path to both of these.

Notes

The following keywords have been used in the directory hierarchy:

- `<platform>` - The platform (For example: Windows)
 - `<ide>` - The IDE (For example: VC10)
 - `<architecture>` - The platform architecture (For example: Win32, Win64)
 - `<configuration>` - The target configuration (For example: DLL Debug, LIB Release - LIB OpenSSL)

If you are using the source code from the git repository, rather than a release archive or nightly build, you will need to generate the project files. Please run "generate -help" for usage details.

Should you wish to help out with some of the items on the TODO list, or find bugs in the project files that need correcting, and would like to submit updated files back then please note that, whilst the solution files can be edited directly, the templates for the project files (which are stored in the git repository) will need to be modified rather than the generated project files that Visual Studio uses.

Dependencies

A key to making good software is to build on top of other great software. By using libraries that many others use, we reinvent the same things fewer times and we get more reliable software as there are more people using the same code.

A whole slew of features that curl provides require that it is built to use one or more external libraries. They are then dependencies of curl. None of them are *required*, but most users will want to use at least some of them.

HTTP Compression

curl can do automatic decompression of data transferred over HTTP if built with the proper 3rd party libraries. You can build curl to use one or more of these libraries:

- gzip compression with [zlib](#)
- brotli compression with [brotli](#)
- zstd compression with [libzstd](#)

Getting compressed data over the wire will use less bandwidth, which might also result in shorter transfer times.

c-ares

<https://c-ares.org/>

curl can be built with c-ares to be able to do asynchronous name resolution. Another option to enable asynchronous name resolution is to build curl with the threaded name resolver backend, which will then instead create a separate helper thread for each name resolve. c-ares does it all within the same thread.

nghttp2

<https://nghttp2.org/>

This is a library for handling HTTP/2 framing and is a prerequisite for curl to support HTTP version 2.

openldap

<https://www.openldap.org/>

This library is one option to allow curl to get support for the LDAP and LDAPS URL schemes. On Windows, you can also opt to build curl to use the winldap library.

librtmp

<https://rtmpdump.mplayerhq.hu/>

To enable curl's support for the RTMP URL scheme, you must build curl with the librtmp library that comes from the RTMPDump project.

libpsl

<https://rockdaboot.github.io/libpsl/>

When you build curl with support for libpsl, the cookie parser will know about the Public Suffix List and thus handle such cookies appropriately.

libidn2

<https://www.gnu.org/software/libidn/libidn2/manual/libidn2.html>

curl handles International Domain Names (IDN) with the help of the libidn2 library.

SSH libraries

If you want curl to have SCP and SFTP support, build with one of these SSH libraries:

- [libssh2](#)
 - [libssh](#)
 - [wolfSSH](#)
-

TLS libraries

There are many different TLS libraries to choose from, so they are covered in a [separate section](#).

TLS libraries

To make curl support TLS based protocols, such as HTTPS, FTPS, SMTPS, POP3S, IMAPS and more, you need to build with a third-party TLS library since curl does not implement the TLS protocol itself.

curl is written to work with a large number of TLS libraries:

- AmiSSL
- BearSSL
- BoringSSL
- GnuTLS
- GSKit (OS/400 specific)

- libressl
- mbedTLS
- NSS
- OpenSSL
- rustls
- Schannel (native Windows)
- Secure Transport (native macOS)
- WolfSSL

When you build curl and libcurl to use one of these libraries, it is important that you have the library and its include headers installed on your build machine.

configure

Below, you will learn how to tell configure to use the different libraries. The configure script will not select any TLS library by default. You must select one, or instruct configure that you want to build without TLS support using `--without-ssl`.

OpenSSL, BoringSSL, libressl

```
1 ./configure --with-openssl
```

configure will detect OpenSSL in its default path by default. You can optionally point configure to a custom install path prefix where it can find OpenSSL:

```
1 ./configure --with-openssl=/home/user/installed/openssl
```

The alternatives [BoringSSL](#) and libressl look similar enough that configure will detect them the same way as OpenSSL. It then uses additional measures to figure out which of the particular flavors it is using.

GnuTLS

```
1 ./configure --with-gnutls
```

configure will detect GnuTLS in its default path by default. You can optionally point configure to a custom install path prefix where it can find gnutls:

```
1 ./configure --with-gnutls=/home/user/installed/gnutls
```

```
1 ./configure --with-nss
```

configure will detect NSS in its default path by default. You can optionally point configure to a custom install path prefix where it can find NSS:

```
1 ./configure --with-nss=/home/user/installed/nss
```

WolfSSL

```
1 ./configure --with-wolfssl
```

configure will detect WolfSSL in its default path by default. You can optionally point configure to a custom install path prefix where it can find WolfSSL:

```
1 ./configure --with-wolfssl=/home/user/installed/wolfssl
```

mbedTLS

```
1 ./configure --with-mbedtls
```

configure will detect mbedTLS in its default path by default. You can optionally point configure to a custom install path prefix where it can find mbedTLS:

```
1 ./configure --with-mbedtls=/home/user/installed/mbedtls
```

Secure Transport

```
1 ./configure --with-secure-transport
```

configure will detect Secure Transport in its default path by default. You can optionally point configure to a custom install path prefix where it can find Secure Transport:

```
1 ./configure --with-secure-transport=/home/user/installed/darwinssl
```

Schannel

```
1 ./configure --with-schannel
```

configure will detect Schannel in its default path by default.

(WinSSL was previously an alternative name for Schannel, and earlier curl versions instead needed --with-winssl)

BearSSL

```
1 ./configure --with-bearssl
```

configure will detect BearSSL in its default path by default. You can optionally point configure to a custom install path prefix where it can find BearSSL:

```
1 ./configure --with-bearssl=/home/user/installed/bearssl
```

Rustls

```
1 ./configure --with-rustls
```

When told to use "rustls", curl is actually trying to find and use the rustls-ffi library - the C API for the rustls library. configure will detect rustls-ffi in its default path by default. You can optionally point configure to a custom install path prefix where it can find rustls-ffi:

```
1 ./configure --with-rustls=/home/user/installed/rustls-ffi
```

BoringSSL

build boringssl

\$HOME/src is where I put the code in this example. You can pick wherever you like.

```
1 $ cd $HOME/src
2 $ git clone https://boringssl.googlesource.com/boringssl
3 $ cd boringssl
4 $ mkdir build
5 $ cd build
6 $ cmake -DCMAKE_POSITION_INDEPENDENT_CODE=on ..
7 $ make
```

set up the build tree to get detected by curl's configure

In the boringssl source tree root, make sure there is a `lib` and an `include` dir. The `lib` dir should

contain the two libs (I made them symlinks into the build dir). The `include` dir is already present by default. Make and populate `lib` like this (commands issued in the source tree root, not in the `build/` subdirectory).

```
1 $ mkdir lib
2 $ cd lib
3 $ ln -s ../build/ssl/libssl.a
4 $ ln -s ../build/crypto/libcrypto.a
```

configure curl

`LIBS=-lpthread ./configure --with-ssl=$HOME/src/boringssl` (where I point out the root of the boringssl tree)

Verify that at the end of the configuration, it says it detected BoringSSL to be used.

build curl

Run `make` in the curl source tree.

Now you can install curl normally with `make install` etc.

Command line basics

curl started out as a command-line tool and it has been invoked from shell prompts and from within scripts by thousands of users over the years. curl has established itself as one of those trusty tools that is there for you to help you get your work done.

Binaries and different platforms

The command-line tool "curl" is a binary executable file. The curl project does not by itself distribute or provide binaries. Binary files are highly system specific and oftentimes also bound to specific system versions.

To get a curl for your platform and your system, you need to get a curl executable from somewhere. Many people build their own from the source code provided by the curl project, lots of people install it using a package tool for their operating system and yet another portion of users download binary install packages from sources they trust.

No matter how you do it, make sure you are getting your version from a trusted source and that you verify

digital signatures or the authenticity of the packages in other ways.

Also, remember that curl is often built to use third-party libraries to perform and unless curl is built to use them statically you must also have those third-party libraries installed; the exact set of libraries will vary depending on the particular build you get.

Command lines, quotes and aliases

There are many different command lines, shells and prompts in which curl can be used. They all come with their own sets of limitations, rules and guidelines to follow. The curl tool is designed to work with any of them without causing troubles but there may be times when your specific command line system does not match what others use or what is otherwise documented.

One way that command-line systems differ, for example, is how you can put quotes around arguments such as to embed spaces or special symbols. In most Unix-like shells you use double quotes ("") and single quotes ('') depending if you want to allow variable expansions or not within the quoted string, but on Windows there is no support for the single quote version.

In some environments, like PowerShell on Windows, the authors of the command line system decided they know better and "help" the user to use another tool instead of curl when `curl` is typed, by providing an alias that takes precedence when a command line is executed. In order to use curl properly with PowerShell, you need to type in its full name including the extension: "curl.exe".

Different command-line environments will also have different maximum command line lengths and force the users to limit how large amount of data that can be put into a single line. curl adapts to this by offering a way to provide command-line options through a file—or from stdin—using the [-K option](#).

Garbage in gives garbage out

curl has little will of its own. It tries to please you and your wishes to a large extent. It also means that it will try to play with what you give it. If you misspell an option, it might do something unintended. If you pass in a slightly illegal URL, chances are curl will still deal with it and proceed. It means that you can pass in crazy data in some options and you can have curl pass on that crazy data in its transfer operation.

This is a design choice, as it allows you to really tweak how curl does its protocol communications and you can have curl massage your server implementations in the most creative ways.

Command line options

When telling curl to do something, you invoke curl with zero, one or several command-line options to accompany the URL or set of URLs you want the transfer to be about. curl supports over two hundred different options.

Short options

Command line options pass on information to curl about how you want it to behave. Like you can ask curl to switch on verbose mode with the `-v` option:

```
1 curl -v http://example.com
```

`-v` is here used as a "short option". You write those with the minus symbol and a single letter immediately following it. Many options are just switches that switch something on or change something between two known states. They can be used with just that option name. You can then also combine several single-letter options after the minus. To ask for both verbose mode and that curl follows HTTP redirects:

```
1 curl -vL http://example.com
```

The command-line parser in curl always parses the entire line and you can put the options anywhere you like; they can also appear after the URL:

```
1 curl http://example.com -Lv
```

and the two separate short options can of course also be specified separately, like:

```
1 curl -v -L http://example.com
```

Long options

Single-letter options are convenient since they are quick to write and use, but as there are only a limited number of letters in the alphabet and there are many things to control, not all options are available like that. Long option names are therefore provided for those. Also, as a convenience and to allow scripts to become more readable, most short options have longer name aliases.

Long options are always written with *two* minuses (or *dashes*, whichever you prefer to call them) and then the name and you can only write one option name per double-minus. Asking for verbose mode using the long option format looks like:

```
1 curl --verbose http://example.com
```

and asking for HTTP redirects as well using the long format looks like:

```
1 curl --verbose --location http://example.com
```

Arguments to options

Not all options are just simple boolean flags that enable or disable features. For some of them you need to pass on data, like perhaps a user name or a path to a file. You do this by writing first the option and then the argument, separated with a space. Like, for example, if you want to send an arbitrary string of data in an HTTP POST to a server:

```
1 curl -d arbitrary http://example.com
```

and it works the same way even if you use the long form of the option:

```
1 curl --data arbitrary http://example.com
```

When you use the short options with arguments, you can, in fact, also write the data without the space separator:

```
1 curl -darbitrary http://example.com
```

Arguments with spaces

At times you want to pass on an argument to an option, and that argument contains one or more spaces. For example you want to set the user-agent field curl uses to be exactly `I am your father`, including those three spaces. Then you need to put quotes around the string when you pass it to curl on the command line. The exact quotes to use varies depending on your shell/command prompt, but generally it will work with double quotes in most places:

```
1 curl -A "I am your father" http://example.com
```

Failing to use quotes, like if you would write the command line like this:

```
1 curl -A I am your father http://example.com
```

... will make curl only use 'I' as a user-agent string, and the following strings, 'am', your, etc will instead all be treated as separate URLs since they do not start with `-` to indicate that they are options and curl only ever handles options and URLs.

To make the string itself contain double quotes, which is common when you for example want to send a string of JSON to the server, you may need to use single quotes (except on Windows, where single quotes does not work the same way). Send the JSON string `{ "name": "Darth" }`:

```
1 curl -d '{ "name": "Darth" }' http://example.com
```

Or if you want to avoid the single quote thing, you may prefer to send the data to curl via a file, which then does not need the extra quoting. Assuming we call the file 'json' that contains the above mentioned data:

```
1 curl -d @json http://example.com
```

Negative options

For options that switch on something, there is also a way to switch it off. You then use the long form of the option with an initial `no-` prefix before the name. As an example, to switch off verbose mode:

```
1 curl --no-verbose http://example.com
```

Options depend on version

`curl` was first typed on a command line back in the glorious year of 1998. It already then worked on the specified URL and none, one or more command-line options given to it.

Since then we have added more options. We add options as we go along and almost every new release of curl has one or a few new options that allow users to modify certain aspects of its operation.

With the curl project's rather speedy release chain with a new release shipping every eight weeks, it is almost inevitable that you are at least not always using the latest released version of curl. Sometimes you may even use a curl version that is a few years old.

All command-line options described in this book were, of course, added to curl at some point and only a small portion of them were available that fine spring day in 1998 when curl first shipped. You may have reason to check your version of curl and crosscheck with the curl man page for when certain options were added. This is especially important if you want to take a curl command line using a modern curl version back to an older system that might be running an older installation.

The developers of curl are working hard to not change existing behavior. Command lines written to use curl in 1998, 2003 or 2010 should all be possible to run unmodified even today.

URLs

curl is called curl because a substring in its name is URL (Uniform Resource Locator). It operates on URLs. URL is the name we casually use for the web address strings, like the ones we usually see prefixed with `HTTP://` or starting with `www.`

URL is, strictly speaking, the former name for these. URI (Uniform Resource Identifier) is the more modern and correct name for them. The syntax is defined in [RFC 3986](#).

Where curl accepts a "URL" as input, it is then really a "URI". Most of the protocols curl understands also have a corresponding URI syntax document that describes how that particular URI format works. curl assumes that you give it a valid URL and it only does limited checks of the format in order to extract the information it deems necessary to perform its operation. You can, for example, most probably pass in illegal characters in the URL without curl noticing or caring and it will just pass them on.

- [Scheme](#)
- [Name and password](#)
- [Host](#)
- [Port number](#)
- [Path](#)
- [FTP type](#)
- [Fragment](#)
- [Browsers' address bar](#)
- [Many options and URLs](#)
- [Connection reuse](#)
- [Parallel transfers](#)

Scheme

URLs start with the "scheme", which is the official name for the `http://` part. That tells which protocol the URL uses. The scheme must be a known one that this version of curl supports or it will show an error message and stop. Additionally, the scheme must neither start with nor contain any whitespace.

The scheme separator

The scheme identifier is separated from the rest of the URL by the `://` sequence. That is a colon and two forward slashes. There exists URL formats with only one slash, but curl does not support any of them. There are two additional notes to be aware of, about the number of slashes:

curl allows some illegal syntax and tries to correct it internally; so it will also understand and accept URLs with one or three slashes, even though they are in fact not properly formed URLs. curl does this because the browsers started this practice so it has lead to such URLs being used in the wild every now and then.

`file://` URLs are written as `file://<hostname>/<path>` but the only hostnames that are okay to use are `localhost`, `127.0.0.1` or a blank (nothing at all):

```
1 file://localhost/path/to/file
2 file://127.0.0.1/path/to/file
3 file:///path/to/file
```

Inserting any other host name in there will make recent versions of curl to return an error.

Pay special attention to the third example above (`file:///path/to/file`). That is *three* slashes before the path. That is again an area with common mistakes and where browsers allow users to use the wrong syntax so as a special exception, curl on Windows also allows this incorrect format:

```
1 file:///X:/path/to/file
```

... where X is a windows-style drive letter.

Without scheme

As a convenience, curl also allows users to leave out the scheme part from URLs. Then it guesses which protocol to use based on the first part of the host name. That guessing is basic, as it just checks if the first part of the host name matches one of a set of protocols, and assumes you meant to use that protocol. This heuristic is based on the fact that servers traditionally used to be named like that. The protocols that are detected this way are FTP, DICT, LDAP, IMAP, SMTP and POP3. Any other host name in a scheme-less URL will make curl default to HTTP.

For example, this gets a file from an FTP site:

```
1 curl ftp.funet.fi/README
```

While this gets data from an HTTP server:

```
1 curl example.com
```

You can modify the default protocol to something other than HTTP with the `--proto-default` option.

Supported schemes

curl supports or can be made to support (if built so) the following transfer schemes and protocols:

DICT, FILE, FTP, FTPS, GOPHER, GOPHERS, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, MQTT, POP3, POP3S, RTMP, RTMPS, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP

Name and password

Following the scheme in a URL, there can be a possible user name and password field embedded. The use of this syntax is usually frowned upon these days since you easily leak this information in scripts or otherwise. For example, listing the directory of an FTP server using a given name and password:

```
1 curl ftp://user:password@example.com/
```

The presence of user name and password in the URL is completely optional. curl also allows that information to be provided with normal command-line options, outside of the URL.

If you want a non-ASCII letter or maybe a `:` or `@` as part of the user name and/or password, remember to "URL-encode" that letter: write it as `%HH` where `HH` is the hexadecimal byte value. `:` is `%3a` and `@` is `%40`.

Host

The host name part of the URL is, of course, simply a name that can be resolved to an numerical IP address, or the numerical address itself.

```
1 curl http://example.com
```

When specifying a numerical address, use the dotted version for IPv4 addresses:

```
1 curl http://127.0.0.1/
```

...and for IPv6 addresses the numerical version needs to be within square brackets:

```
1 curl http://[2a04:4e42::561]/
```

When a host name is used, the converting of the name to an IP address is typically done using the system's resolver functions. That normally lets a sysadmin provide local name lookups in the `/etc/hosts` file (or equivalent).

International Domain Names (IDN)

curl knows how to deal with IDN names and you just pass them on like you would a normal name:

```
1 curl https://räksmörgås.se
```

Port number

Each protocol has a "default port" that curl will use for it, unless a specified port number is given. The optional port number can be provided within the URL after the host name part, as a colon and the port number written in decimal. For example, asking for an HTTP document on port 8080:

```
1 curl http://example.com:8080/
```

With the name specified as an IPv4 address:

```
1 curl http://127.0.0.1:8080/
```

With the name given as an IPv6 address:

```
1 curl http://[fdea::1]:8080/
```

The port number is an unsigned 16 bit number, so it has to be within the range 0 to 65535.

TCP vs UDP

The given port number is used when setting up the connection to the server specified in the URL. The port is either a TCP port number or a UDP port number depending on which actual underlying transport protocol that is used. TCP is the most common one, but TFTP and HTTP/3 use UDP.

URLs using the `file://` scheme cannot have a port number.

Path

Every URL contains a path. If there is none given, `/` is implied. For example when you use just the host name like in:

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

The path is sent to the specified server to identify exactly which resource that is requested or that will be provided.

The exact use of the path is protocol dependent. For example, getting the file `README` from the default anonymous user from an FTP server:

```
1 curl ftp://ftp.example.com/README
```

For the protocols that have a directory concept, ending the URL with a trailing slash means that it is a directory and not a file. Thus asking for a directory list from an FTP server is implied with such a slash:

```
1 curl ftp://ftp.example.com/tmp/
```

If you want a non-ASCII letter or maybe even space () as part of the path field, remember to "URL-encode" that letter: write it as %HH where HH is the hexadecimal byte value. is %20 .

FTP type

This is not a feature that is widely used.

URLs that identify files on FTP servers have a special feature that allows you to also tell the client (curl in this case) which file type the resource is. This is because FTP is a little special and can change mode for a transfer and thus handle the file differently than if it would use another mode.

You tell curl that the FTP resource is an ASCII type by appending ;type=A to the URL. Getting the foo file from the root directory of example.com using ASCII could then be made with:

```
1 curl "ftp://example.com/foo;type=A"
```

And while curl defaults to binary transfers for FTP, the URL format allows you to also specify the binary type with type=l:

```
1 curl "ftp://example.com/foo;type=l"
```

Finally, you can tell curl that the identified resource is a directory if the type you pass is D:

```
1 curl "ftp://example.com/foo;type=D"
```

...this can then work as an alternative format, instead of ending the path with a trailing slash as mentioned above.

Fragment

URLs offer a fragment part. That is usually seen as a hash symbol (#) and a name for a specific name within a web page in browsers. An example of such a URL might look like:

```
1 https://www.example.com/info.html#the-plot
```

curl supports fragments fine when a URL is passed to it, but the fragment part is never actually sent over the wire so it does not make a difference to curl's operations whether it is present or not.

If you want to make the # character as part of the path and not separating the fragment, make sure to pass it URL-encoded, as %23 :

```
1 curl https://www.example.com/info.html%23the-plot
```

The "fragment trick"

The fact that the fragment part is not actually used over the network can be taken advantage of when you craft command lines.

For example, if you want to request the same URL from a server 10 times, you can make a loop and put the loop instruction in the fragment part. Like this:

```
1 curl https://example.com/#[1-10]
```

Browsers' address bar

It is important to realize that when you use a modern web browser, the address bar they tend to feature at the top of their main windows are not using URLs or even URIs. They are in fact mostly using IRIs, which is a superset of URIs to allow internationalization like non-Latin symbols and more, but it usually goes beyond that, too, as they tend to, for example, handle spaces and do magic things on percent encoding in ways none of these mentioned specifications say a client should do.

The address bar is quite simply an interface for humans to enter and see URI-like strings.

Sometimes the differences between what you see in a browser's address bar and what you can pass in to curl is significant.

Many options and URLs

As mentioned above, curl supports hundreds of command-line options and it also supports an unlimited number of URLs. If your shell or command-line system supports it, there is really no limit to how long a command line you can pass to curl.

curl will parse the entire command line first, apply the wishes from the command-line options used, and then go over the URLs one by one (in a left to right order) to perform the operations.

For some options (for example `-o` or `-O` that tell curl where to store the transfer), you may want to specify one option for each URL on the command line.

curl will return an exit code for its operation on the last URL used. If you instead rather want curl to exit with an error on the first URL in the set that fails, use the `--fail-early` option.

One output for each given URL

If you use a command-line with two URLs, you must tell curl how to handle both of them. The `-o` and `-O` options instruct curl how to save the output for *one* URL of the URLs, so you might want to have as many of those options as you have URLs on the command line.

If you have more URLs than output options on the command line, the URL content without corresponding output instructions will then instead be sent to stdout.

Using the `--remote-name-all` flag will automatically make curl act as if `-O` was used for all given URLs that do not have any output option.

Separate options per URL

In previous sections we described how curl always parses all options in the whole command line and applies those to all the URLs that it transfers.

That was a simplification: curl also offers an option (`-:`, `--next`) that inserts a boundary between a set of options and URLs for which it will apply the options. When the command-line parser finds a `--next` option, it applies the following options to the next set of URLs. The `--next` option thus works as a *divider* between a set of options and URLs. You can use as many `--next` options as you please.

As an example, we do an HTTP GET to a URL and follow redirects, we then make a second HTTP POST to a different URL and we round it up with a HEAD request to a third URL. All in a single command line:

```
1 curl --location http://example.com/1 --next
2   --data sendthis http://example.com/2 --next
3   --head http://example.com/3
```

Trying something like that *without* the `--next` options on the command line would generate an illegal command line since curl would attempt to combine both a POST and a HEAD:

```
1 Warning: You can only select one HTTP request method! You asked for both
2 Warning: POST (-d, --data) and HEAD (-I, --head).
```

Connection reuse

Setting up a TCP connection and especially a TLS connection can be a slow process, even on high bandwidth networks.

It can be useful to remember that curl has a connection pool internally which keeps previously used connections alive and around for a while after they were used so that subsequent requests to the same hosts can reuse an already established connection.

Of course, they can only be kept alive for as long as the curl tool is running. It is a good reason for trying to get several transfers done within the same command line instead of running several independent curl

command line invocations.

Parallel transfers

The default behavior of getting the specified URLs one by one in a serial fashion makes it easy to understand exactly when each URL is fetched but it can be slow.

curl offers the `-Z` (or `--parallel`) option that instead instructs curl to attempt to do the specified transfers in a parallel fashion. When this is enabled, curl will do a lot of transfers simultaneously instead of serially. It will do up to 50 transfers at the same time by default and as soon as one of them has completed, the next one will be kicked off.

For cases where you want to download many files from different sources and a few of them might be slow, a few fast, this can speed things up tremendously.

If 50 parallel transfer is wrong for you, the `--parallel-max` option is there to allow you to change that amount.

Parallel transfer progress meter

Naturally, the ordinary progress meter display that shows file transfer progress for a single transfer is not that useful for parallel transfers so when curl performs parallel transfers, it will show a different progress meter that displays information about all the current ongoing transfers in a single line.

Connection before multiplex

When curl is asked to do parallel transfers, it prioritizes to have the additional transfer reuse and multiplex over other already existing connections. This can potentially lower the total amount of connections (and thereby resources) necessary but it might be slightly slower at start-up.

With `--parallel-immediate`, curl is instructed to reverse the prioritization and instead prefer creating a new connection immediately rather than risk waiting a little to see if the transfer can be multiplexed of another connection.

URL globbing

At times you want to get a range of URLs that are mostly the same, with only a small portion of it changing between the requests. Maybe it is a numeric range or maybe a set of names. curl offers "globbing" as a way to specify many URLs like that easily.

The globbing uses the reserved symbols `[]` and `{}` for this, symbols that normally cannot be part of a legal

URL (except for numerical IPv6 addresses but curl handles them fine anyway). If the globbing gets in your way disable it with `--nobeoff`

When using [] or {} sequences when invoked from a command line prompt, you probably have to put the full URL within double quotes to avoid the shell from interfering with it. This also goes for other characters treated special, like for example '&', '?' and '*'.

While most transfer related functionality in curl is provided by the libcurl library, the URL globbing feature is not!

Numerical ranges

You can ask for a numerical range with [N-M] syntax, where N is the start index and it goes up to and including M. For example, you can ask for 100 images one by one that are named numerically:

```
1 curl -O "http://example.com/[1-100].png"
```

and it can even do the ranges with zero prefixes, like if the number is three digits all the time:

```
1 curl -O "http://example.com/[001-100].png"
```

Or maybe you only want even-numbered images so you tell curl a step counter too. This example range goes from 0 to 100 with an increment of 2:

```
1 curl -O "http://example.com/[0-100:2].png"
```

Alphabetical ranges

curl can also do alphabetical ranges, like when a site has sections named a to z:

```
1 curl -O "http://example.com/section[a-z].html"
```

A list

Sometimes the parts do not follow such an easy pattern, and then you can instead give the full list yourself but then within the curly braces instead of the brackets used for the ranges:

```
1 curl -O "http://example.com/{one,two,three,alpha,beta}.html"
```

Combinations

You can use several globs in the same URL which then will make curl iterate over those, too. To download the images of Ben, Alice and Frank, in both the resolutions 100 x 100 and 1000 x 1000, a command line could look like:

```
1 curl -O "http://example.com/{Ben,Alice,Frank}-{100x100,1000x1000}.jpg"
```

Or download all the images of a chess board, indexed by two coordinates ranged 0 to 7:

```
1 curl -O "http://example.com/chess-[0-7]x[0-7].jpg"
```

And you can, of course, mix ranges and series. Get a week's worth of logs for both the web server and the mail server:

```
1 curl -O "http://example.com/{web,mail}-log[0-6].txt"
```

Output variables for globbing

In all the globbing examples previously in this chapter we have selected to use the `-O` / `--remote-name` option, which makes curl save the target file using the file name part of the used URL.

Sometimes that is not enough. You are downloading multiple files and maybe you want to save them in a different subdirectory or create the saved file names differently. curl, of course, has a solution for these situations as well: output file name variables.

Each "glob" used in a URL gets a separate variable. They are referenced as `# [num]` - that means the single character `#` followed by the glob number which starts with 1 for the first glob and ends with the last glob.

Save the main pages of two different sites:

```
1 curl "http://{one,two}.example.com" -o "file_#{1..2}.txt"
```

Save the outputs from a command line with two globs in a subdirectory:

```
1 curl "http://{site,host}.host[1-5].example.com" -o "subdir/#1_#2"
```

List options

curl has more than two hundred command-line options and the number of options keep increasing over time. Chances are the number of options will reach 250 within a few years.

To find out which options you need to perform as certain action, you can get curl to list them. First, `curl --help` or simply `curl -h` will get you a list of the most important and frequently used options. You can then provide an additional "category" to `-h` to get more options listed for that specific area. Use `curl -h category` to list all existing categories or `curl -h all` to list all available options.

Before curl 7.73.0, the help category system did not exist and then `curl --help` or simply `curl -h` would simply list all existing options in alphabetical order with a brief explanation next to each.

The `curl --manual` option outputs the entire man page for curl. That is a thorough and complete document on how each option works amassing several thousand lines of documentation. To wade through that is also a tedious work and we encourage use of a search function through those text masses. Some people will appreciate the man page in its [web version](#).

Config file

You can easily end up with curl command lines that use a large number of command-line options, making them rather hard to work with. Sometimes the length of the command line you want to enter even hits the maximum length your command-line system allows. The Microsoft Windows command prompt being an example of something that has a fairly small maximum line length.

To aid such situations, curl offers a feature we call "config file". It allows you to write command-line options in a text file instead and then tell curl to read options from that file in addition to the command line.

You tell curl to read more command-line options from a specific file with the `-K/--config` option, like this:

```
1 curl -K cmdline.txt http://example.com
```

...and in the `cmdline.txt` file (which, of course, can use any file name you please) you enter each command line per line:

```
1 # this is a comment, we ask to follow redirects
2 --location
3 # ask to do a HEAD request
4 --head
```

The config file accepts both short and long options, exactly as you would write them on a command line. As a special extra feature, it also allows you to write the long format of the options without the leading two dashes to make it easier to read. Using that style, the config file shown above can alternatively be written as:

```
1 # this is a comment, we ask to follow redirects
2 location
3 # ask to do a HEAD request
4 head
```

Command line options that take an argument must have its argument provided on the same line as the option. For example changing the User-Agent HTTP header can be done with

```
1 user-agent "Everything-is-an-agent"
```

To allow the config files to look even more like a true config file, it also allows you to use '=' or ':' between the option and its argument. As you see above it is not necessary, but some like the clarity it offers. Setting the user-agent option again:

```
1 user-agent = "Everything-is-an-agent"
```

If the parameter contains whitespace (or starts with : or =), the parameter must be enclosed within quotes. Within double quotes, the following escape sequences are available: \\, \" , \\t, \\n, \\r and \\v. A backslash preceding any other letter is ignored.

The argument to an option can be specified without double quotes and then curl will treat the next space or newline as the end of the argument.

The user agent string example we have used above has no white spaces and therefore it can also be provided without the quotes like:

```
1 user-agent = Everything-is-an-agent
```

Finally, if you want to provide a URL in a config file, you must do that the `--url` way, or just with `url`, and not like on the command line where everything that is not an option is assumed to be a URL. So you provide a URL for curl like this:

```
1 url = "http://example.com"
```

Default config file

When curl is invoked, it always (unless `-q` is used) checks for a default config file and uses it if found.

The default config file is checked for in the following places in this order:

1. `$CURL_HOME/.curlrc`
2. `$XDG_CONFIG_HOME/.curlrc` (Added in 7.73.0)
3. `$HOME/.curlrc`
4. Windows: `%USERPROFILE%\curlrc`
5. Windows: `%APPDATA%\curlrc`
6. Windows: `%USERPROFILE%\Application Data\curlrc`

- Non-Windows: use getpwuid to find the home directory
- 8. On Windows, if it finds no `.curlrc` file in the sequence described above, it checks for one in the same dir the curl executable is placed.

On Windows two filenames are checked per location: `.curlrc` and `_curlrc`, preferring the former. Ancient curl versions on Windows checked for `_curlrc` only.

Passwords

Passwords are tricky and sensitive. Leaking a password can make someone other than you access the resources and the data otherwise protected.

curl offers several ways to receive passwords from the user and then subsequently pass them on or use them to something else.

The most basic curl authentication option is `-u` / `--user`. It accepts an argument that is the user name and password, colon separated. Like when `alice` wants to request a page requiring HTTP authentication and her password is `12345` :

```
1 $ curl -u alice:12345 http://example.com/
```

Command line leakage

Several potentially bad things are going on here. First, we are entering a password on the command line and the command line might be readable for other users on the same system (assuming you have a multi-user system). curl will help minimize that risk by trying to blank out passwords from process listings.

One way to avoid passing the user name and password on the command line is to instead use a [.netrc file](#) or a [config file](#). You can also use the `-u` option without specifying the password, and then curl will instead prompt the user for it when it runs.

Network leakage

Secondly, this command line sends the user credentials to an HTTP server, which is a clear-text protocol that is open for man-in-the-middle or other snoopers to spy on the connection and see what is sent. In this command line example, it makes curl use HTTP Basic authentication and that is completely insecure.

There are several ways to avoid this, and the key is, of course, then to avoid protocols or authentication schemes that sends credentials in plain text over the network. Easiest is perhaps to make sure you use encrypted versions of protocols. Use HTTPS instead of HTTP, use FTPS instead of FTP and so on.

If you need to stick to a plain text and insecure protocol, then see if you can switch to using an authentication

method that avoids sending the credentials in the clear. If you want HTTP, such methods would include Digest (`--digest`), Negotiate (`--negotiate.`) and NTLM (`--ntlm`).

Progress meter

curl has a built-in progress meter. When curl is invoked to transfer data (either uploading or downloading) it can show that meter in the terminal screen to show how the transfer is progressing, namely the current transfer speed, how long it has been going on and how long it thinks it might be left until completion.

The progress meter is inhibited if curl deems that there is output going to the terminal, as the progress meter would interfere with that output and just mess up what gets displayed. A user can also forcibly switch off the progress meter with the `-s` / `--silent` option, which tells curl to hush.

If you invoke curl and do not get the progress meter, make sure your output is directed somewhere other than the terminal.

curl also features an alternative and simpler progress meter that you enable with `-#` / `--progress-bar`. As the long name implies, it instead shows the transfer as progress bar.

At times when curl is asked to transfer data, it cannot figure out the total size of the requested operation and that then subsequently makes the progress meter contain fewer details and it cannot, for example, make forecasts for transfer times, etc.

Units

The progress meter displays bytes and bytes per second.

It will also use suffixes for larger amounts of bytes, using the 1024 base system so 1024 is one kilobyte (1K), 2048 is 2K, etc. curl supports these:

Suffix	Amount	Name
K	2^{10}	kilobyte
M	2^{20}	megabyte
G	2^{30}	gigabyte
T	2^{40}	terabyte
P	2^{50}	petabyte

The times are displayed using H:MM:SS for hours, minutes and seconds.

The progress meter exists to show a user that something actually is happening. The different fields in the output have the following meaning:

1 % Total	% Received	% Xferd	Average Speed	Dload	Upload	Total	Time	Current	Left	Curr. Speed
2										
3 0 151M	0 38608	0 0	9406 0	4:41:43	0:00:04	4:41:39	9287			

From left to right:

Title	Meaning
%	Percentage completed of the whole transfer
Total	Total size of the whole expected transfer (if known)
%	Percentage completed of the download
Received	Currently downloaded number of bytes
%	Percentage completed of the upload
Xferd	Currently uploaded number of bytes
Average Speed Dload	Average transfer speed of the entire download so far, in number of bytes per second
Average Speed Upload	Average transfer speed of the entire upload so far in number of bytes per second
Time Total	Expected time to complete the operation, in HH:MM:SS notation for hours, minutes and seconds
Time Current	Time passed since the start of the transfer, in HH:MM:SS notation for hours, minutes and seconds
Time Left	Expected time left to completion, in HH:MM:SS notation for hours, minutes and seconds
Curr. Speed	Average transfer speed over the last 5 seconds (the first 5 seconds of a transfer is based on less time, course) in number of bytes per second

Using curl

Previous chapters have described some basic details on what curl is and something about the basic

command lines. You use command line options and you pass an URL or to work with

In this chapter, we are going to dive deeper into a variety of different concepts of what the curl tool can do and how to tell curl to use these features. You should consider all these features as different tools that are here to help you do your file transfer tasks as conveniently as possible.

Verbose

If your curl command does not execute or return what you expected it to, your first gut reaction should always be to run the command with the `-v` / `--verbose` option to get more information.

When verbose mode is enabled, curl gets more talkative and will explain and show a lot more of its doings. It will add informational tests and prefix them with `*`. For example, let's see what curl might say when trying a simple HTTP example (saving the downloaded data in the file called 'saved'):

```
1 $ curl -v http://example.com -o saved
2 * Rebuilt URL to: http://example.com/
```

Ok so we invoked curl with a URL that it considers incomplete so it helps us and it adds a trailing slash before it moves on.

```
1 * Trying 93.184.216.34...
```

This tells us curl now tries to connect to this IP address. It means the name 'example.com' has been resolved to one or more addresses and this is the first (and possibly only) address curl will try to connect to.

```
1 * Connected to example.com (93.184.216.34) port 80 (#0)
```

It worked! curl connected to the site and here it explains how the name maps to the IP address and on which port it has connected to. The '(#0)' part is which internal number curl has given this connection. If you try multiple URLs in the same command line you can see it use more connections or reuse connections, so the connection counter may increase or not increase depending on what curl decides it needs to do.

If we use an HTTPS:// URL instead of an HTTP one, there will also be a whole bunch of lines explaining how curl uses CA certs to verify the server's certificate and some details from the server's certificate, etc. Including which ciphers were selected and more TLS details.

In addition to the added information given from curl internals, the `-v` verbose mode will also make curl show all headers it sends and receives. For protocols without headers (like FTP, SMTP, POP3 and so on), we can consider commands and responses as headers and they will thus also be shown with `-v`.

If we then continue the output seen from the command above (but ignore the actual HTML response), curl will show:

```
1 > GET / HTTP/1.1
2 > Host: example.com
3 > User-Agent: curl/7.45.0
```

```
5    > Accept: */*
```

This is the full HTTP request to the site. This request is how it looks in a default curl 7.45.0 installation and it may, of course, differ slightly between different releases and in particular it will change if you add command line options.

The last line of the HTTP request headers looks empty, and it is. It signals the separation between the headers and the body, and in this request there is no "body" to send.

Moving on and assuming everything goes according to plan, the sent request will get a corresponding response from the server and that HTTP response will start with a set of headers before the response body:

```
1 < HTTP/1.1 200 OK
2 < Accept-Ranges: bytes
3 < Cache-Control: max-age=604800
4 < Content-Type: text/html
5 < Date: Sat, 19 Dec 2015 22:01:03 GMT
6 < Etag: "359670651"
7 < Expires: Sat, 26 Dec 2015 22:01:03 GMT
8 < Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
9 < Server: ECS (ewr/15BD)
10 < Vary: Accept-Encoding
11 < X-Cache: HIT
12 < x-ec-custom-error: 1
13 < Content-Length: 1270
14 <
```

This may look mostly like mumbo jumbo to you, but this is normal set of HTTP headers—metadata—about the response. The first line's "200" might be the most important piece of information in there and means "everything is fine".

The last line of the received headers is, as you can see, empty, and that is the marker used for the HTTP protocol to signal the end of the headers.

After the headers comes the actual response body, the data payload. The regular -v verbose mode does not show that data but only displays

```
1 { [1270 bytes data]
```

That 1270 bytes should then be in the 'saved' file. You can also see that there was a header named Content-Length: in the response that contained the exact file length (it will not always be present in responses).

HTTP/2 and HTTP/3

When doing file transfers using version two or three of the HTTP protocol, curl sends and receives **compressed** headers. So to display outgoing and incoming HTTP/2 and HTTP/3 headers in a readable

and understandable way, curl will actually show the uncompressed versions in a style similar to how they

Silence

The opposite of verbose is, of course, to make curl more silent. With the `-s` (or `--silent`) option you make curl switch off the progress meter and not output any error messages for when errors occur. It gets mute. It will still output the downloaded data you ask it to.

With silence activated, you can ask for it to still output the error message on failures by adding `-S` or `--show-error`.

Trace options

There are times when `-v` is not enough. In particular, when you want to store the complete stream including the actual transferred data.

For situations when curl does encrypted file transfers with protocols such as HTTPS, FTPS or SFTP, other network monitoring tools (like Wireshark or tcpdump) will not be able to do this job as easily for you.

For this, curl offers two other options that you use instead of `-v`.

`--trace [filename]` will save a full trace in the given file name. You can also use '`-`' (a single minus) instead of a file name to get it passed to stdout. You would use it like this:

```
1 $ curl --trace dump http://example.com
```

When completed, there is a 'dump' file that can turn out pretty sizable. In this case, the 15 first lines of the dump file looks like:

```
1 == Info: Rebuilt URL to: http://example.com/
2 == Info: Trying 93.184.216.34...
3 == Info: Connected to example.com (93.184.216.34) port 80 (#0)
4 => Send header, 75 bytes (0x4b)
5 0000: 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a GET / HTTP/1.1..
6 0010: 48 6f 73 74 3a 20 65 78 61 6d 70 6c 65 2e 63 6f Host: example.co
7 0020: 6d 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 m..User-Agent: c
8 0030: 75 72 6c 2f 37 2e 34 35 2e 30 0d 0a 41 63 63 65 url/7.45.0..Acce
9 0040: 70 74 3a 20 2a 2f 2a 0d 0a 0d 0a pt: */*....
10 <= Recv header, 17 bytes (0x11)
11 0000: 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d HTTP/1.1 200 OK.
12 0010: 0a .
13 <= Recv header, 22 bytes (0x16)
14 0000: 41 63 63 65 70 74 2d 52 61 6e 67 65 73 3a 20 62 Accept-Ranges: b
15 0010: 79 74 65 73 0d 0a ytes..
```

Every single sent and received byte get displayed individually in hexadecimal numbers. Received headers will be output line by line.

If you think the hexadecimals are not helping, you can try `--trace-ascii [filename]` instead, also this accepting '-' for stdout and that makes the 15 first lines of tracing look like:

```
1 == Info: Rebuilt URL to: http://example.com/
2 == Info: Trying 93.184.216.34...
3 == Info: Connected to example.com (93.184.216.34) port 80 (#0)
4 => Send header, 75 bytes (0x4b)
5 0000: GET / HTTP/1.1
6 0010: Host: example.com
7 0023: User-Agent: curl/7.45.0
8 003c: Accept: */*
9 0049:
10 <= Recv header, 17 bytes (0x11)
11 0000: HTTP/1.1 200 OK
12 <= Recv header, 22 bytes (0x16)
13 0000: Accept-Ranges: bytes
14 <= Recv header, 31 bytes (0x1f)
15 0000: Cache-Control: max-age=604800
```

-trace-time

This option prefixes all verbose/trace outputs with a high resolution timer for when the line is printed. It works with the regular `-v` / `--verbose` option as well as with `--trace` and `--trace-ascii`.

An example could look like this:

```
1 $ curl -v --trace-time http://example.com
2 23:38:56.837164 * Rebuilt URL to: http://example.com/
3 23:38:56.841456 * Trying 93.184.216.34...
4 23:38:56.935155 * Connected to example.com (93.184.216.34) port 80 (#0)
5 23:38:56.935296 > GET / HTTP/1.1
6 23:38:56.935296 > Host: example.com
7 23:38:56.935296 > User-Agent: curl/7.45.0
8 23:38:56.935296 > Accept: */*
9 23:38:56.935296 >
10 23:38:57.029570 < HTTP/1.1 200 OK
11 23:38:57.029699 < Accept-Ranges: bytes
12 23:38:57.029803 < Cache-Control: max-age=604800
13 23:38:57.029903 < Content-Type: text/html
14 ---- snip ----
```

The lines are all the local time as hours:minutes:seconds and then number of microseconds in that second.

Write out

--write-out or just -w for short, outputs text and information after a transfer is completed. It offers a large range of variables that you can include in the output, variables that have been set with values and information from the transfer.

Instruct curl to output a string by passing plain text to this option:

```
1 curl -w "formatted string" http://example.com/
```

...and you can also have curl read that string from a given file instead if you prefix the string with '@':

```
1 curl -w @filename http://example.com/
```

...or even have curl read the string from stdin if you use '-' as filename:

```
1 curl -w @- http://example.com/
```

The variables that are available are accessed by writing `%{variable_name}` in the string and that variable will then be substituted by the correct value. To output a normal '%' you just write it as '%}'. You can also output a newline by using '\n', a carriage return with '\r' and a tab space with '\t'.

(The %-symbol is special on the Windows command line, where all occurrences of % must be doubled when using this option.)

As an example, we can output the Content-Type and the response code from an HTTP transfer, separated with newlines and some extra text like this:

```
1 curl -w "Type: %{content_type}\nCode: %{response_code}\n" http://example.com
```

This feature writes the output to stdout so you probably want to make sure that you do not also send the downloaded content to stdout as then you might have a hard time to separate out the data.

Available --write-out variables

Some of these variables are not available in really old curl versions.

Variable	Description
<code>content_type</code>	Content-Type of the requested document, if there was any.
<code>errormsg</code>	Error message from the transfer. Empty if no error occurred. (Introduced in 7.75.0)

<code>exitcode</code>	Numerical exit code from the transfer. 0 if no error occurred. (Introduced in 7.75.0)
<code>filename_effective</code>	The ultimate filename that curl writes out to. Practical if curl is told to write to a file with the <code>--remote-name</code> or <code>--output</code> option. It's most useful in combination with the <code>--remote-header-name</code> option.
<code>ftp_entry_path</code>	The initial path curl ended up in when logging on the remote FTP server.
<code>http_code</code>	The former variable name for what is now known as <code>response_code</code> .
<code>http_connect</code>	the numerical code that was found in the last response (from a proxy) to a curl CONNECT request.
<code>http_version</code>	The HTTP version that was used.
<code>json</code>	all write-out variables as a single JSON object.
<code>local_ip</code>	IP address of the local end of the most recently used connection - can be either IPv4 or IPv6
<code>local_port</code>	Local port number of the most recently used connection
<code>method</code>	HTTP method the most recent request used
<code>num_connects</code>	Number of new connects made in the recent transfer.
<code>num_headers</code>	Number of response headers in the last response
<code>num_redirects</code>	Number of redirects that were followed in the request.
<code>onerror</code>	If the transfer ended with an error, show the rest of the string, otherwise stop here. (Introduced in 7.75.0)
<code>proxy_ssl_verify_result</code>	The result of the SSL peer certificate verification that was requested when communicating with a proxy. 0 means the verification was successful.

`redirect_url`

The actual URL a redirect *would* take you to were an HTTP request was made without `-L` to follow redirects.

`remote_ip`

The remote IP address of the most recently used connection — can be either IPv4 or IPv6.

`remote_port`

The remote port number of the most recently made connection.

Version

Version

To get to know what version of curl you have installed, run

```
1 curl --version
```

or use the shorthand version:

```
1 curl -V
```

The output from that command line is typically four lines, out of which some will be rather long and might wrap in your terminal window.

An example output from a Debian Linux in June 2020:

```
1 curl 7.68.0 (x86_64-pc-linux-gnu) libcurl/7.68.0 OpenSSL/1.1.1g zlib/1.2.11 brotli/1.0.7 l...
2 Release-Date: 2020-01-08
3 Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtmp rtsp
4 Features: AsynchDNS brotli GSS-API HTTP2 HTTPS-proxy IDN IPv6 Kerberos Largefile libz NTLM
```

while the same command line invoked on a Windows 10 machine on the same date looks like:

```
1 curl 7.55.1 (Windows) libcurl/7.55.1 WinSSL
2 Release-Date: [unreleased]

3 Protocols: dict file ftp ftps http https imap imaps pop3 pop3s smtp smtps telnet tftp
4 Features: AsynchDNS IPv6 Largefile SSPI Kerberos SPNEGO NTLM SSL
```

The meaning of the four lines?

Line 1: curl

The first line starts with `curl` and first shows the main version number of the tool. Then follows the "platform" the tool was built for within parentheses and the libcurl version. Those three fields are common for all curl builds.

If the curl version number has `-DEV` appended to it, it means the version is built straight from a in-development source code and it is not an officially released and "blessed" version.

The rest of this line contains names of third party components this build of curl uses, often with their individual version number next to it with a slash separator. Like `OpenSSL/1.1.1g` and `nnghttp2/1.41.0`. This can for example tell you which TLS backends this curl uses.

Line 1: TLS versions

Line 1 may contain one or more TLS libraries. curl can be built to support more than one TLS library which

then makes curl - at start-up - select which particular backend to use for this invoke.

If curl supports more than one TLS library like this, the ones that are *not* selected by default will be listed within parentheses. Thus, if you do not specify which backend to use (with the `CURL_SSL_BACKEND` environment variable) the one listed without parentheses will be used.

Line 2: Release-Date

This line shows the date this curl version was released by the curl project, and it can also show a secondary "Patch date" if it has been updated somehow after it was originally released.

This says `[unreleased]` if curl was built another way than from a release tarball, and as you can see above that is how Microsoft did it for Windows 10 and the curl project does not recommend it.

Line 3: Protocols

This is a list of all transfer protocols (URL schemes really) in alphabetical order that this curl build supports. All names are shown in lowercase letters.

This list can contain these protocols:

dict, file, ftp, ftps, gopher, http, https, imap, imaps, ldap, ldaps, mqtt, pop3, pop3s, rtmp, rtsp, scp, sftp, smb, smbs, smtp, smtps, telnet and tftp

Line 4: Features

The list of features this build of curl supports. If the name is present in the list, that feature is enabled. If the name is not present, that feature is not enabled.

Features that can be present there:

- **alt-svc** - Support for the alt-svc: header
- **AsynchDNS** - This curl uses asynchronous name resolves. Asynchronous name resolves can be done using either the c-ares or the threaded resolver backends.
- **brotli** - support for automatic brotli compression over HTTP(S)
- **CharConv** - curl was built with support for character set conversions (like EBCDIC)
- **Debug** - This curl uses a libcurl built with Debug. This enables more error-tracking and memory debugging etc. For curl-developers only!
- **GSS-API** - GSS-API authentication is enabled
- **HTTP2** - HTTP/2 support has been built-in.
- **HTTP3** - HTTP/3 support has been built-in.
- **HTTPS-proxy** - This curl is built to support HTTPS proxy.
- **IDN** - This curl supports IDN - international domain names.
- **IPv6** - You can use IPv6 with this.
- **krb4** - Krb4 for FTP is supported
- **Largefile** - This curl supports transfers of large files, files larger than 2GB.

- **libz** - Automatic gzip decompression of compressed files over HTTP is supported.
- **Metalink** - This curl supports Metalink. In modern curl versions this option is never available.
- **MultiSSL** - This curl supports multiple TLS backends. The first line will detail exactly which TLS libraries.
- **NTLM** - NTLM authentication is supported.
- **NTLM_WB** - NTLM authentication is supported.
- **PSL** - Public Suffix List (PSL) is available and means that this curl has been built with knowledge about "public suffixes", used for cookies.
- **SPNEGO** - SPNEGO authentication is supported.
- **SSL** - SSL versions of various protocols are supported, such as HTTPS, FTPS, POP3S and so on.
- **SSPI** - SSPI is supported
- **TLS-SRP** - SRP (Secure Remote Password) authentication is supported for TLS.
- **UnixSockets** - Unix sockets support is provided.

Persistent connections

When setting up TCP connections to sites, curl will keep the old connection around for a while so that if the next transfer is to the same host it can reuse the same connection again and thus save a lot of time. We call this persistent connections. curl will always try to keep connections alive and reuse existing connections as far as it can.

The curl command-line tool can, however, only keep connections alive for as long as it runs, so as soon as it exits back to your command line it has to close down all currently open connections (and also free and clean up all the other caches it uses to decrease time of subsequent operations). We call the pool of alive connections the "connection cache".

If you want to perform N transfers or operations against the same host or same base URL, you could gain a lot of speed by trying to do them in as few curl command lines as possible instead of repeatedly invoking curl with one URL at a time.

Downloads

"Download" means getting data from a server on a network, and the server is then clearly considered to be "above" you. This is loading data down from the server onto your machine where you are running curl.

Downloading is probably the most common use case for curl — retrieving the specific data pointed to by a URL onto your machine.

- [What exactly is downloading?](#)
- [Storing downloads](#)
- [Download to a file named by the URL](#)

- [Use the target file name from the server](#)
- [HTML and charsets](#)
- [Compression](#)
- [Shell redirects](#)
- [Multiple downloads](#)
- [My browser shows something else](#)
- [Rate limiting](#)
- [Maximum file size](#)
- [Storing metadata in file system](#)
- [Raw](#)
- [Retry](#)
- [Resuming and ranges](#)

What exactly is downloading?

You specify the resource to download by giving curl a URL. curl defaults to downloading a URL unless told otherwise, and the URL identifies what to download. In this example the URL to download is `http://example.com`:

```
1 curl http://example.com
```

The URL is broken down into its individual components ([as explained elsewhere](#)), the correct server is contacted and is then asked to deliver the specific resource—often a file. The server then delivers the data, or it refuses or perhaps the client asked for the wrong data and then that data is delivered.

A request for a resource is protocol-specific so an `FTP://` URL works differently than an `HTTP://` URL or an `SFTP://` URL.

A URL without a path part, that is a URL that has a host name part only (like the `http://example.com` example above) will get a slash ('/') appended to it internally and then that is the resource curl will ask for from the server.

If you specify multiple URLs on the command line, curl will download each URL one by one. It will not start the second transfer until the first one is complete, etc.

Storing downloads

If you try the example download as in the previous section, you will notice that curl will output the downloaded data to stdout unless told to do something else. Outputting data to stdout is really useful when

You want to pipe it into another program or similar, but it is not obvious the optimal way to deal with your

Give curl a specific file name to save the download in with `-o [filename]` (with `--output` as the long version of the option), where filename is either just a file name, a relative path to a file name or a full path to the file.

Also note that you can put the `-o` before or after the URL; it makes no difference:

```
1 curl -o output.html http://example.com/
2 curl -o /tmp/index.html http://example.com/
3 curl http://example.com -o ../../folder/savethis.html
```

This is, of course, not limited to `http://` URLs but works the same way no matter which type of URL you download:

```
1 curl -o file.txt ftp://example.com/path/to/file-name.ext
```

If you ask curl to send the output to the terminal, it attempts to detect and prevent binary data from being sent there since that can seriously mess up your terminal (sometimes to the point where it stops working). You can override curl's binary-output-prevention and force the output to get sent to stdout by using `-o -`.

curl has several other ways to store and name the downloaded data. Details follow.

Download to a file named by the URL

Many URLs, however, already contain the file name part in the rightmost end. curl lets you use that as a shortcut so you do not have to repeat it with `-o`. So instead of:

```
1 curl -o file.html http://example.com/file.html
```

You can save the remote URL resource into the local file 'file.html' with this:

```
1 curl -O http://example.com/file.html
```

This is the `-O` (uppercase letter o) option, or `--remote-name` for the long name version. The `-O` option selects the local file name to use by picking the file name part of the URL that you provide. This is important. You specify the URL and curl picks the name from this data. If the site redirects curl further (and if you tell curl to follow redirects), it does not change the file name curl will use for storing this.

Use the URL's file name part for all URLs

As a reaction to adding a hundred `-O` options when using a hundred URLs, we introduced an option called `--remote-name-all`. This makes `-O` the default operation for all given URLs. You can still provide

individual "storage instructions" for URLs but if you leave one out for a URL that gets downloaded, the default action is then switched from `stdout` to `-O` style.

Use the target file name from the server

HTTP servers have the option to provide a header named `Content-Disposition:` in responses. That header may contain a suggested file name for the contents delivered, and curl can be told to use that hint to name its local file. The `-J` / `--remote-header-name` enables this. If you also use the `-O` option, it makes curl use the file name from the URL by default and only *if* there is actually a valid `Content-Disposition` header available, it switches to saving using that name.

`-J` has some problems and risks associated with it that users need to be aware of:

1. It will only use the rightmost part of the suggested file name, so any path or directories the server suggests will be stripped out.
2. Since the file name is entirely selected by the server, curl will, of course, overwrite any preexisting local file in your current directory if the server happens to provide such a file name.
3. File name encoding and character sets issues. curl does not decode the name in any way, so you may end up with a URL-encoded file name where a browser would otherwise decode it to something more readable using a sensible character set.

HTML and charsets

curl will download the exact binary data that the server sends. This might be of importance to you in case, for example, you download an HTML page or other text data that uses a certain character encoding that your browser then displays as expected. curl will then not translate the arriving data.

A common example where this causes some surprising results is when a user downloads a web page with something like:

```
1 curl https://example.com/ -o storage.html
```

...and when inspecting the `storage.html` file after the fact, the user realizes that one or more characters look funny or downright wrong. This might occur because the server sent the characters using charset X, while your editor and environment use charset Y. In an ideal world, we would all use UTF-8 everywhere but unfortunately, that is still not the case.

A common work-around for this issue that works decently is to use the common `iconv` utility to translate a text file to and from different charsets.

Compression

curl allows you to ask HTTP and HTTPS servers to provide compressed versions of the data and then

perform automatic decompression of it on arrival. In situations where bandwidth is more limited than CPU this will help you receive more data in a shorter amount of time.

HTTP compression can be done using two different mechanisms, one which might be considered "The Right Way" and the other that is the way that everyone actually uses and is the widespread and popular way to do it. The common way to compress HTTP content is using the **Content-Encoding** header. You ask curl to use this with the `--compressed` option:

```
1 curl --compressed http://example.com/
```

With this option enabled (and if the server supports it) it delivers the data in a compressed way and curl will decompress it before saving it or sending it to stdout. This usually means that as a user you do not really see or experience the compression other than possibly noticing a faster transfer.

The `--compressed` option asks for Content-Encoding compression using one of the supported compression algorithms. There is also the rare **Transfer-Encoding** method, which is the request header that was created for this automated method but was never really widely adopted. You can tell curl to ask for Transfer-Encoded compression with `--tr-encoding`:

```
1 curl --tr-encoding http://example.com/
```

In theory, there is nothing that prevents you from using both in the same command line, although in practice, you may then experience that some servers get a little confused when asked to compress in two different ways. It is generally safer to just pick one.

Shell redirects

When you invoke curl from a shell or some other command-line prompt system, that environment generally provides you with a set of output redirection abilities. In most Linux and Unix shells and with Windows' command prompts, you direct stdout to a file with `> filename`. Using this, of course, makes the use of `-o` or `-O` superfluous.

```
1 curl http://example.com/ > example.html
```

Redirecting output to a file redirects all output from curl to that file, so even if you ask to transfer more than one URL to stdout, redirecting the output will get all the URLs' output stored in that single file.

```
1 curl http://example.com/1 http://example.com/2 > files
```

Unix shells usually allow you to redirect the `stderr` stream separately. The `stderr` stream is usually a stream that also gets shown in the terminal, but you can redirect it separately from the `stdout` stream. The `stdout` stream is for the data while `stderr` is metadata and errors, etc., that are not data. You can redirect `stderr` with `2>file` like this:

```
1 curl http://example.com > files.html 2>errors
```

Multiple downloads

As curl can be told to download many URLs in a single command line, there are, of course, times when you want to store these downloads in nicely named local files.

The key to understanding this is that each download URL needs its own "storage instruction". Without said "storage instruction", curl will default to sending the data to stdout. If you ask for two URLs and only tell curl where to save the first URL, the second one is sent to stdout. Like this:

```
1 curl -o one.html http://example.com/1 http://example.com/2
```

The "storage instructions" are read and handled in the same order as the download URLs so they do not have to be next to the URL in any way. You can round up all the output options first, last or interleaved with the URLs. You choose.

These examples all work the same way:

```
1 curl -o 1.txt -o 2.txt http://example.com/1 http://example.com/2
2 curl http://example.com/1 http://example.com/2 -o 1.txt -o 2.txt
3 curl -o 1.txt http://example.com/1 http://example.com/2 -o 2.txt
4 curl -o 1.txt http://example.com/1 -o 2.txt http://example.com/2
```

The `-O` is similarly just an instruction for a single download so if you download multiple URLs, use more of them:

```
1 curl -O -O http://example.com/1 http://example.com/2
```

My browser shows something else

A common use case is using curl to get a URL that you can get in your browser when you paste the URL in the browser's address bar.

A browser getting a URL as input does so much more and in so many different ways than curl that what curl shows in your terminal output is probably not at all what you see in your browser window.

Client differences

Curl only gets exactly what you ask it to get and it never parses the actual content—the data—that the server delivers. A browser gets data and it activates different parsers depending on what kind of content it thinks it

gets. For example, if the data is HTML it will parse it to display a web page and possibly download other sub resources such as images, JavaScript and CSS files. When curl downloads HTML it will just get that single HTML resource, even if it, when parsed by a browser, would trigger a whole busload of more downloads. If you want curl to download any sub-resources as well, you need to pass those URLs to curl and ask it to get those just like any other URLs.

Clients also differ in how they send their requests, and some aspects of a request for a resource include, for example, format preferences, asking for compressed data, or just telling the server from which previous page we are "coming from". curl's requests will differ a little or a lot from how your browser sends its requests.

Server differences

The server that receives the request and delivers data is often setup to act in certain ways depending on what kind of client it thinks communicates with it. Sometimes it is as innocent as trying to deliver the best content for the client, sometimes it is to hide some content for some clients or even to try to work around known problems in specific browsers. Then there is also, of course, various kind of login systems that might rely on HTTP authentication or cookies or the client being from the pre-validated IP address range.

Sometimes getting the same response from a server using curl as the response you get with a browser ends up really hard work. Users then typically record their browser sessions with the browser's networking tools and then compare that recording with recorded data from curl's `--trace-ascii` option and proceed to modify curl's requests (often with `-H` / `--header`) until the server starts to respond the same to both.

This type of work can be both time consuming and tedious. You should always do this with permission from the server owners or admins.

Intermediaries' fiddlings

Intermediaries are proxies, explicit or implicit ones. Some environments will force you to use one or you may choose to use one for various reasons, but there are also the transparent ones that will intercept your network traffic silently and proxy it for you no matter what you want.

Proxies are "middle men" that terminate the traffic and then act on your behalf to the remote server. This can introduce all sorts of explicit filtering and "saving" you from certain content or even "protecting" the remote server from what data you try to send to it, but even more so it introduces another software's view on how the protocol works and what the right things to do are.

Interfering intermediaries are often the cause of lots of head aches and mysteries down to downright malicious modifications of content.

We strongly encourage you to use HTTPS or other means to verify that the contents you are downloading or uploading are really the data that the remote server has sent to you and that your precious bytes end up verbatim at the intended destination.

Rate limiting

When curl transfers data, it will attempt to do that as fast as possible. It goes for both uploads and downloads. Exactly how fast that will be depends on several factors, including your computer's ability, your own network connection's bandwidth, the load on the remote server you are transferring to/from and the latency to that server. And your curl transfers are also likely to compete with other transfers on the networks the data travels over, from other users or just other apps by the same user.

In many setups, however, you will find that you can more or less saturate your own network connection with a single curl command line. If you have a 10 megabit per second connection to the Internet, chances are curl can use all of those 10 megabits to transfer data.

For most use cases, using as much bandwidth as possible is a good thing. It makes the transfer faster, it makes the curl command complete sooner and it will make the transfer use resources from the server for a shorter period of time.

Sometimes you will, however, find that having curl starve out other network functions on your local network connection is inconvenient. In these situations you may want to tell curl to slow down so that other network users get a better chance to get their data through as well. With `--limit-rate [speed]` you can tell curl to not go faster than the given number of bytes per second. The rate limit value can be given with a letter suffix using one of K, M and G for kilobytes, megabytes and gigabytes.

To make curl not download data any faster than 200 kilobytes per second:

```
1 curl https://example.com/ --limit-rate 200K
```

The given limit is the maximum *average speed* allowed, counted during the entire transfer. It means that curl might use higher transfer speeds in short bursts, but over time it uses no more than the given rate.

Also note that curl never knows what the maximum possible speed is—it will simply go as fast as it can and is allowed. You may know your connection's maximum speed, but curl does not.

Maximum file size

When you want to make sure your curl command line will not try to download a too-large file, you can instruct curl to stop before doing that, if it knows the size before the transfer starts! Maybe that would use too much bandwidth, take too long time or you do not have enough space on your hard drive:

```
1 curl --max-filesize 100000 https://example.com/
```

Give curl the largest download you will accept in number of bytes and if curl can figure out the size before the transfer starts it will abort before trying to download something larger.

There are many situations in which curl cannot figure out the size at the time the transfer starts and this option will not affect those transfers, even if they may end up larger than the specified amount.

Storing metadata in file system

When saving a download to a file with curl, the `--xattr` option tells curl to also store certain file metadata in "extended file attributes". These extended attributes are standardized name/value pairs stored in the file system, assuming one of the supported file systems and operating systems are used.

Currently, the URL is stored in the `xdg.origin.url` attribute and, for HTTP, the content type is stored in the `mime_type` attribute. If the file system does not support extended attributes when this option is set, a warning is issued.

Raw

When `--raw` is used, it disables all internal HTTP decoding of content or transfer encodings and instead makes curl pass on unaltered, raw, data.

This is typically used if you are writing a middle software and you want to pass on the content to another HTTP client and allow that to do the decoding instead.

Retry

Normally curl will only make a single attempt to perform a transfer and return an error if not successful. Using the `--retry` option you can tell curl to retry certain failed transfers.

If a transient error is returned when curl tries to perform a transfer, it will retry this number of times before giving up. Setting the number to 0 makes curl do no retries (which is the default). Transient error means either: a timeout, an FTP 4xx response code or an HTTP 5xx response code.

Tweak your retries

When curl is about to retry a transfer, it will first wait one second and then for all forthcoming retries it will double the waiting time until it reaches 10 minutes which then will be the delay between the rest of the retries. Using `--retry-delay` you can disable this exponential backoff algorithm and set your own delay between the attempts. With `--retry-max-time` you cap the total time allowed for retries. The `--max-time` option will still specify the longest time a single of these transfers is allowed to spend.

Make curl retry up to 5 times, but no less than two minutes:

```
1 curl --retry 5 --retry-max-time 120 https://example.com
```

Connection refused

The default retry mechanism only retries transfers for what are considered transient errors. Those are errors that the server itself hints and qualifies as being there right now but that might be gone at a later time.

Sometimes you as a user know more about the situation and you can then help out curl to do better retries. For starters, you can tell curl to consider "connection refused" to be a transient error. Maybe you know that the server you communicate with is a flaky one or maybe you know that you sometimes try to download from it when it reboots or similar. You use `--retry-connrefused` for this.

For example: retry up to 5 times and consider `ECONNREFUSED` a reason for retry:

```
1 curl --retry 5 --retry-connrefused https://example.com
```

Retry on any and all errors

The most aggressive form of retry is for the cases where you **know** that the URL is supposed to work and you will not tolerate any failures. Using `--retry-all-errors` makes curl treat all transfers failures as reason for retry.

For example: retry up to 12 times for all errors:

```
1 curl --retry 12 --retry-all-errors https://example.com
```

Resuming and ranges

Resuming a download means first checking the size of what is already present locally and then asking the server to send the rest of it so it can be appended. curl also allows resuming the transfer at a custom point without actually having anything already locally present.

curl supports resumed downloads on several protocols. Tell it where to start the transfer with the `-C`, `--continue-at` option that takes either a plain numerical byte counter offset where to start or the string `-` that asks curl to figure it out itself based on what it knows. When using `-`, curl will use the destination file name to figure out how much data that is already present locally and ask use that as an offset when asking for more data from the server.

To start downloading an FTP file from byte offset 100:

```
1 curl --continue-at 100 ftp://example.com/bigfile
```

Continue downloading a previously interrupted download:

```
1 curl --continue-at - http://example.com/bigfile -O
```

If you instead just want a specific byte range from the remote resource transferred, you can ask for only that. For example, when you only want 1000 bytes from offset 100 to avoid having to download the entire huge remote file:

```
1 curl --range 100-1099 http://example.com/bigfile
```

Uploads

Uploading is a term for sending data to a remote server. Uploading is done differently for each protocol, and several protocols may even allow different ways of uploading data.

Protocols allowing upload

You can upload data using one of these protocols: FILE, FTP, FTPS, HTTP, HTTPS, IMAP, IMAPS, SCP, SFTP, SMB, SMBS, SMTP, SMTPS and TFTP.

HTTP offers several "uploads"

HTTP, and its bigger brother HTTPS, offer several different ways to upload data to a server and curl provides easy command-line options to do it the three most common ways, described below.

An interesting detail with HTTP is also that an upload can also be a download, in the same operation and in fact many downloads are initiated with an HTTP POST.

POST

POST is the HTTP method that was invented to send data to a receiving web application, and it is, for example, how most common HTML forms on the web work. It usually sends a chunk of relatively small amounts of data to the receiver.

The upload kind is usually done with the `-d` or `--data` options, but there are a few additional alterations.

Read the detailed description on how to do this with curl in the [HTTP POST with curl](#) chapter.

multipart formpost

Multipart formposts are also used in HTML forms on websites; typically when there is a file upload involved. This type of upload is also an HTTP POST but it sends the data formatted according to some special rules, which is what the "multipart" name means.

Since it sends the data formatted completely differently, you cannot select which type of POST to use at your own whim but it entirely depends on what the receiving server end expects and can handle.

HTTP multipart formposts are done with `-F`. See the detailed description in the [HTTP multipart formposts](#) chapter.

PUT

HTTP PUT is the upload method that was designed to send a complete resource meant to be put as-is on the remote site or even replace an existing resource there. That said, this is also the least used upload method for HTTP on the web today and lots, if not most, web servers do not even have PUT enabled.

You send off an HTTP upload using the `-T` option with the file to upload:

```
1 curl -T uploadthis http://example.com/
```

FTP uploads

Working with FTP, you get to see the remote file system you will be accessing. You tell the server exactly in which directory you want the upload to be placed and which file name to use. If you specify the upload URL with a trailing slash, curl will append the locally used file name to the URL and then that will be the file name used when stored remotely:

```
1 curl -T uploadthis ftp://example.com/this/directory/
```

So if you prefer to select a different file name on the remote side than what you have used locally, you specify it in the URL:

```
1 curl -T uploadthis ftp://example.com/this/directory/remotename
```

Learn much more about FTPing in the [FTP with curl](#) section.

SMTP uploads

You may not consider sending an email to be "uploading", but to curl it is. You upload the mail body to the SMTP server. With SMTP, you also need to include all the mail headers you need (To:, From:, Date:, etc.) in the mail body as curl will not add any at all.

```
1 curl -T mail smtp://mail.example.com/ --mail-from user@example.com
```

Learn more about using SMTP with curl in the [Using curl/SMTP](#) section.

Progress meter for uploads

The general progress meter curl provides (see the [Progress meter](#) section) works fine for uploads as well. What needs to be remembered is that the progress meter is automatically disabled when you are sending output to stdout, and most protocols curl support can output something even for an upload.

Therefore, you may need to explicitly redirect the downloaded data to a file (using shell redirect '>', `-o` or similar) to get the progress meter displayed for upload.

Rate limiting

Rate limiting works exactly the same for uploads as for downloads and curl, in fact, only has a single limit that will limit the speed in both directions.

See further details in the [Download Rate limiting section](#).

Connections

Most of the protocols you use with curl speak TCP. With TCP, a client such as curl must first figure out the IP address(es) of the host you want to communicate with, then connect to it. "Connecting to it" means performing a TCP protocol handshake.

For ordinary command line usage, operating on a URL, these are details which are taken care of under the hood, and which you can mostly ignore. But at times you might find yourself wanting to tweak the specifics...

- [Name resolve tricks](#)
- [Connection timeout](#)
- [Network interface](#)
- [Local port number](#)
- [Keep alive](#)

Name resolve tricks

curl offers many ways to make use another host than the one it normally would connect to.

Edit the hosts file

Maybe you want the command `curl http://example.com` to connect to your local server instead of the actual server.

You can normally and easily do that by editing your `hosts` file (`/etc/hosts` on Linux and Unix-like systems) and adding, for example, `127.0.0.1 example.com` to redirect the host to your localhost. However this edit requires admin access and it has the downside that it affects all other applications at the same time.

Change the Host: header

The `Host:` header is the normal way an HTTP client tells the HTTP server which server it speaks to, as typically an HTTP server serves many different names using the same software instance.

So, by passing in a custom modified `Host:` header you can have the server respond with the contents of the site even when you did not actually connect to that host name.

For example, you run a test instance of your main site `www.example.com` on your local machine and you want to have curl ask for the index html:

```
1 curl -H "Host: www.example.com" http://localhost/
```

When setting a custom `Host:` header and using cookies, curl will extract the custom name and use that as host when matching cookies to send off.

The `Host:` header is not enough when communicating with an HTTPS server. With HTTPS there is a separate extension field in the TLS protocol called SNI (Server Name Indication) that lets the client tell the server the name of the server it wants to talk to. curl will only extract the SNI name to send from the given URL.

Provide a custom IP address for a name

Do you know better than the name resolver where curl should go? Then you can give an IP address to curl yourself! If you want to redirect port 80 access for `example.com` to instead reach your localhost:

```
1 curl --resolve example.com:80:127.0.0.1 http://example.com/
```

You can even specify multiple `--resolve` switches to provide multiple redirects of this sort, which can be handy if the URL you work with uses HTTP redirects or if you just want to have your command line work with multiple URLs.

`--resolve` inserts the address into curl's DNS cache, so it will effectively make curl believe that is the address it got when it resolved the name.

When talking HTTPS, this will send SNI for the name in the URL and curl will verify the server's response to

make sure it serves for the name in the URL.

The pattern you specify in the option needs be a host name and its corresponding port number and only if that exact pair is used in the URL will the address be substituted. For example, if you want to replace a host name in an HTTPS URL on its default port number, you need to tell curl it is for port 443, like:

```
1 curl --resolve example.com:443:192.168.0.1 https://example.com/
```

Provide a replacement name

As a close relative to the `--resolve` option, the `--connect-to` option provides a minor variation. It allows you to specify a replacement name and port number for curl to use under the hood when a specific name and port number is used to connect.

For example, suppose you have a single site called `www.example.com` that in turn is actually served by three different individual HTTP servers: `load1`, `load2` and `load3`, for load balancing purposes. In a typical normal procedure, curl resolves the main site and gets to speak to one of the load balanced servers (as it gets a list back and just picks one of them) and all is well. If you want to send a test request to one specific server out of the load balanced set (`load1.example.com` for example) you can instruct curl to do that.

You can still use `--resolve` to accomplish this if you know the specific IP address of `load1`. But without having to first resolve and fix the IP address separately, you can tell curl:

```
1 curl --connect-to www.example.com:80:load1.example.com:80 http://www.example.com
```

It redirects from a SOURCE NAME + SOURCE PORT to a DESTINATION NAME + DESTINATION PORT. curl will then resolve the `load1.example.com` name and connect, but in all other ways still assume it is talking to `www.example.com`.

Name resolve tricks with c-ares

As should be detailed elsewhere in this book, curl may be built with several different name resolving backends. One of those backends is powered by the c-ares library and when curl is built to use c-ares, it gets a few extra superpowers that curl built to use other name resolve backends do not get. Namely, it gains the ability to more specifically instruct what DNS servers to use and how that DNS traffic is using the network.

With `--dns-servers`, you can specify exactly which DNS server curl should use instead of the default one. This lets you run your own experimental server that answers differently, or use a backup one if your regular one is unreliable or dead.

With `--dns-ipv4-addr` and `--dns-ipv6-addr` you ask curl to "bind" its local end of the DNS communication to a specific IP address and with `--dns-interface` you can instruct curl to use a specific network interface to use for its DNS requests.

These `--dns-*` options are advanced and are only meant for people who know what they are doing and understand what these options do. But they offer customizable DNS name resolution operations.

Connection timeout

curl will typically make a TCP connection to the host as an initial part of its network transfer. This TCP connection can fail or be slow, if there are shaky network conditions or faulty remote servers.

To reduce the impact on your scripts or other use, you can set the maximum time in seconds which curl will allow for the connection attempt. With `--connect-timeout` you tell curl the maximum time to allow for connecting, and if curl has not connected in that time it returns a failure.

The connection timeout only limits the time curl is allowed to spend up until the moment it connects, so once the TCP connection has been established it can take longer time. See the [Timeouts](#) section for more on generic curl timeouts.

If you specify a low timeout, you effectively disable curl's ability to connect to remote servers, slow servers or servers you access over unreliable networks.

The connection timeout can be specified as a decimal value for sub-second precision. For example, to allow 2781 milliseconds to connect:

```
1 curl --connect-timeout 2.781 https://example.com/
```

Network interface

On machines with multiple network interfaces that are connected to multiple networks, there are situations where you can decide which network interface you would prefer the outgoing network traffic to use. Or which originating IP address (out of the multiple ones you have) to use in the communication.

Tell curl which network interface, which IP address or even host name that you would like to "bind" your local end of the communication to, with the `--interface` option:

```
1 curl --interface eth1 https://www.example.com/
2
3 curl --interface 192.168.0.2 https://www.example.com/
4
5 curl --interface machine2 https://www.example.com/
```

Local port number

A TCP connection is created between an IP address and a port number in the local end and an IP address

and a port number in the remote end. The remote port number can be specified in the URL and usually helps identify which service you are targeting.

The local port number is usually randomly assigned to your TCP connection by the network stack and you normally do not have to think about it much further. However, in some circumstances you find yourself behind network equipment, firewalls or similar setups that put restrictions on what source port numbers that can be allowed to set up the outgoing connections.

For situations like this, you can specify which local ports curl should bind the connection to. You can specify a single port number to use, or a range of ports. We recommend using a range because ports are scarce resources and the exact one you want may already be in use. If you ask for a local port number (or range) that curl cannot obtain for you, it will exit with a failure.

Also, on most operating systems you cannot bind to port numbers below 1024 without having a higher privilege level (root) and we generally advise against running curl as root if you can avoid it.

Ask curl to use a local port number between 4000 and 4200 when getting this HTTPS page:

```
1 curl --local-port 4000-4200 https://example.com/
```

Keep alive

TCP connections can be totally without traffic in either direction when they are not used. A totally idle connection can therefore not be clearly separated from a connection that has gone completely stale because of network or server issues.

At the same time, lots of network equipment such as firewalls or NATs are keeping track of TCP connections these days, so that they can translate addresses, block "wrong" incoming packets, etc. These devices often count completely idle connections as dead after N minutes, where N varies between device to device but at times is as short as 10 minutes or even less.

One way to help avoid a really slow connection (or an idle one) getting treated as dead and wrongly killed, is to make sure TCP keep alive is used. TCP keepalive is a feature in the TCP protocol that makes it send "ping frames" back and forth when it would otherwise be totally idle. It helps idle connections to detect breakage even when no traffic is moving over it, and helps intermediate systems not consider the connection dead.

curl uses TCP keepalive by default for the reasons mentioned here. But there might be times when you want to *disable* keepalive or you may want to change the interval between the TCP "pings" (curl defaults to 60 seconds). You can switch off keepalive with:

```
1 curl --no-keepalive https://example.com/
```

or change the interval to 5 minutes (300 seconds) with:

```
1 curl --keepalive-time 300 https://example.com/
```

Timeouts

Network operations are by their nature rather unreliable or perhaps fragile operations as they depend on a set of services and networks to be up and working for things to work. The availability of these services can come and go and the performance of them may also vary greatly from time to time.

The design of TCP even allows the network to get completely disconnected for an extended period of time without it necessarily getting noticed by the participants in the transfer.

The result of this is that sometimes Internet transfers take a long time. Further, most operations in curl have no time-out by default!

Maximum time allowed to spend

Tell curl with `-m` / `--max-time` the maximum time, in seconds, that you allow the command line to spend before curl exits with a timeout error code (28). When the set time has elapsed, curl will exit no matter what is going on at that moment—including if it is transferring data. It really is the maximum time allowed.

The given maximum time can be specified with a decimal precision; `0.5` means 500 milliseconds and `2.37` equals 2370 milliseconds.

Example:

```
1 curl --max-time 5.5 https://example.com/
```

(Your locale may use another symbol than a dot for expressing numerical fractions.)

Never spend more than this to connect

`--connect-timeout` limits the time curl will spend trying to connect to the host. All the necessary steps done before the connection is considered complete have to be completed within the given time frame. Failing to connect within the given time will cause curl to exit with a timeout exit code (28).

The given maximum connect time can be specified with a decimal precision; `0.5` means 500 milliseconds and `2.37` equals 2370 milliseconds:

```
1 curl --connect-timeout 2.37 https://example.com/
```

Transfer speeds slower than this means exit

Having a fixed maximum time for a curl operation can be cumbersome, especially if you, for example, do scripted transfers and the file sizes and transfer times vary a lot. A fixed timeout value then needs to be set unnecessarily high to cover for worst cases.

As an alternative to a fixed time-out, you can tell curl to abandon the transfer if it gets below a certain speed and stays below that threshold for a specific period of time.

For example, if a transfer speed goes below 1000 bytes per second during 15 seconds, stop it:

```
1 curl --speed-time 15 --speed-limit 1000 https://example.com/
```

Keep connections alive

curl enables TCP keep-alive by default. TCP keep-alive is a feature that makes the TCP stack send a probe to the other side when there is no traffic, to make sure that it is still there and "alive". By using keep-alive, curl is much more likely to discover that the TCP connection is dead.

Use `--keepalive-time` to specify how often in full seconds you would like the probe to get sent to the peer. The default value is 60 seconds.

Sometimes this probing disturbs what you are doing and then you can easily disable it with `--no-keepalive`.

.netrc

Unix systems have for a long time offered a way for users to store their user name and password for remote FTP servers. ftp clients have supported this for decades and this way allowed users to quickly login to known servers without manually having to reenter the credentials each time. The `.netrc` file is typically stored in a user's home directory. (On Windows, curl will look for it with the name `_netrc`).

This being a widespread and well used concept, curl also supports it—if you ask it to. curl does not, however, limit this feature to FTP, but can get credentials for machines for any protocol with this. See further below for how.

The .netrc file format

The `.netrc` file format is simple: you specify lines with a machine name and follow that with lines for the login and password that are associated with that machine.

machine name

Identifies a remote machine name. curl searches the `.netrc` file for a machine token that matches the remote machine specified in the URL. Once a match is made, the subsequent `.netrc` tokens are processed, stopping

when the end of file is reached or another machine is encountered.

default

This is the same as machine name except that `default` matches any name. There can be only one default token, and it must be after all machine tokens. To provide a default anonymous login for hosts that are not otherwise matched, add a line similar to this in the end:

```
1 default login anonymous password user@domain
```

login name

The user name string for the remote machine. You cannot use a space in the name.

password string

Supply a password. If this token is present, curl will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the `.netrc` file you really **should** make sure the file is not readable by anyone besides the user. You cannot use a space when you enter the password.

macdef name

Define a macro. This is **not supported by curl**. In order for the rest of the `.netrc` to still work fine, curl will properly skip every definition done with `macdef` that it finds.

Example

An example `.netrc` for the host `example.com` with a user named '`daniel`', using the password '`qwerty`' would look like:

```
1 machine example.com
2 login daniel
3 password qwerty
```

It can also be written on a single line with the same functionality:

```
1 machine example.com login daniel password qwerty
```

User name matching

When a URL is provided with a user name and `.netrc` is used, then curl will try to find the matching password for that machine and login combination.

Enable netrc

`-n, --netrc` tells curl to look for and use the `.netrc` file.

`--netrc-file [file]` is similar to `--netrc`, except that you also provide the path to the actual file to use. This is useful when you want to provide the information in another directory or with another file name.

`--netrc-optional` is similar to `--netrc`, but this option makes the `.netrc` usage optional and not mandatory as the `--netrc` option.

Proxies

A proxy is a machine or software that does something on behalf of you, the client.

You can also see it as a middle man that sits between you and the server you want to work with, a middle man that you connect to instead of the actual remote server. You ask the proxy to perform your desired operation for you and then it will run off and do that and then return the data to you.

There are several different types of proxies and we shall list and discuss them in subsections below.

- [Discover your proxy](#)
- [PAC](#)
- [Captive portals](#)
- [Proxy type](#)
- [HTTP proxy](#)
- [SOCKS proxy](#)
- [MITM proxy](#)
- [Authentication](#)
- [HTTPS proxy](#)
- [Proxy environment variables](#)
- [Proxy headers](#)

Discover your proxy

Some networks are setup to require a proxy in order for you to reach the Internet or perhaps that special network you are interested in. The use of proxies are introduced on your network by the people and management that run your network for policy or technical reasons.

In the networking space there are a few methods for the automatic detection of proxies and how to connect to them, but none of those methods are truly universal and curl supports none of them. Furthermore, when

you communicate to the outside world through a proxy that often means that you have to put a lot of trust on the proxy as it will be able to see and modify all the non-secure network traffic you send or get through it. That trust is not easy to assume automatically.

If you check your browser's network settings, sometimes under an advanced settings tab, you can learn what proxy or proxies your browser is configured to use. Chances are big that you should use the same one or ones when you use curl.

As an example, you can find [proxy settings for Firefox browser](#) in Preferences => General => Network Settings as shown below:

proxy
settin
gs for
Firefo
x

PAC

Some network environments provides several different proxies that should be used in different situations, and a customizable way to handle that is supported by the browsers. This is called "[proxy auto-config](#)", or PAC.

A PAC file contains a JavaScript function that decides which proxy a given network connection (URL) should use, and even if it should not use a proxy at all. Browsers most typically read the PAC file off a URL on the local network.

Since curl has no JavaScript capabilities, curl does not support PAC files. If your browser and network use PAC files, the easiest route forward is usually to read the PAC file manually and figure out the proxy you need to specify to run curl successfully.

Captive portals

These are not proxies but they are blocking the way between you and the server you want to access.

A "captive portal" is one of these systems that are popular to use in hotels, airports and for other sorts of network access to a larger audience. The portal will "capture" all network traffic and redirect you to a login web page until you have either clicked OK and verified that you have read their conditions or perhaps even made sure that you have paid plenty of money for the right to use the network.

curl's traffic will of course also captured by such portals and often the best way is to use a browser to accept the conditions and "get rid of" the portal since from then on they often allow all other traffic originating from that same machine (MAC address) for a period of time.

Most often you can use curl too to submit that affirmation, if you just figure out how to submit the form and what fields to include in it. If this is something you end up doing many times, it may be worth exploring.

Proxy type

curl supports several different types of proxies.

The default proxy type is HTTP so if you specify a proxy host name (or IP address) without a scheme part (the part that is often written as `http://`) curl goes with assuming it's an HTTP proxy.

curl provides a number of options to set the proxy type instead of using the scheme prefix. See the [SOCKS](#) section.

HTTP proxy

An HTTP proxy is a proxy that the client speaks HTTP with to get the transfer done. curl will, by default, assume that a host you point out with `-x` or `--proxy` is an HTTP proxy, and unless you also specify a port number it will default to port 1080 (and the reason for that particular port number is purely historical).

If you want to request the example.com web page using a proxy on 192.168.0.1 port 8080, a command line could look like:

```
1 curl -x 192.168.0.1:8080 http://example.com/
```

Recall that the proxy receives your request, forwards it to the real server, then reads the response from the server and then hands that back to the client.

If you enable verbose mode with `-v` when talking to a proxy, you will see that curl connects to the proxy instead of the remote server, and you will see that it uses a slightly different request line.

HTTPS with HTTP proxy

HTTPS was designed to allow and provide secure and safe end-to-end privacy from the client to the server (and back). In order to provide that when speaking to an HTTP proxy, the HTTP protocol has a special request that curl uses to setup a tunnel through the proxy that it then can encrypt and verify. This HTTP method is known as `CONNECT`.

When the proxy tunnels encrypted data through to the remote server after a CONNECT method sets it up, the proxy cannot see nor modify the traffic without breaking the encryption:

```
1 curl -x proxy.example.com:80 https://example.com/
```

Non-HTTP protocols over an HTTP proxy

An "HTTP proxy" means the proxy itself speaks HTTP. HTTP proxies are primarily used to proxy HTTP but it is also fairly common that they support other protocols as well. In particular, FTP is fairly commonly supported.

When talking FTP "over" an HTTP proxy, it is usually done by more or less pretending the other protocol works like HTTP and asking the proxy to "get this URL" even if the URL is not using HTTP. This distinction is important because it means that when sent over an HTTP proxy like this, curl does not really speak FTP even though given an FTP URL; thus FTP-specific features will not work:

```
1 curl -x http://proxy.example.com:80 ftp://ftp.example.com/file.txt
```

What you can do instead then, is to "tunnel through" the HTTP proxy!

HTTP proxy tunneling

Most HTTP proxies allow clients to "tunnel through" it to a server on the other side. That is exactly what's done every time you use HTTPS through the HTTP proxy.

You tunnel through an HTTP proxy with curl using `-p` or `--proxytunnel`.

When you do HTTPS through a proxy you normally connect through to the default HTTPS remote TCP port number 443, so therefore you will find that most HTTP proxies white list and allow connections only to hosts on that port number and perhaps a few others. Most proxies will deny clients from connecting to just any random port (for reasons only the proxy administrators know).

Still, assuming that the HTTP proxy allows it, you can ask it to tunnel through to a remote server on any port number so you can do other protocols "normally" even when tunneling. You can do FTP tunneling like this:

```
1 curl -p -x http://proxy.example.com:80 ftp://ftp.example.com/file.txt
```

You can tell curl to use HTTP/1.0 in its CONNECT request issued to the HTTP proxy by using `--proxy1.0 [proxy]` instead of `-x`.

SOCKS proxy

SOCKS is a protocol used for proxies and curl supports it. curl supports both SOCKS version 4 as well as version 5, and both versions come in two flavors.

You can select the specific SOCKS version to use by using the correct scheme part for the given proxy host with `-x`, or you can specify it with a separate option instead of `-x`.

SOCKS4 is for the version 4 and SOCKS4a is for the version 4 without resolving the host name locally:

```
1 curl -x socks4://proxy.example.com http://www.example.com/
2
3 curl --socks4 proxy.example.com http://www.example.com/
```

The SOCKS4a versions:

```
1 curl -x socks4a://proxy.example.com http://www.example.com/
2
3 curl --socks4a proxy.example.com http://www.example.com/
```

SOCKS5 is for the version 5 and SOCKS5-hostname is for the version 5 without resolving the host name locally:

```
1 curl -x socks5://proxy.example.com http://www.example.com/
2
3 curl --socks5 proxy.example.com http://www.example.com/
```

The SOCKS5-hostname versions. This sends the host name to the server so there is no name resolving done locally:

```
1 curl -x socks5h://proxy.example.com http://www.example.com/
2
3 curl --socks5-hostname proxy.example.com http://www.example.com/
```

MITM proxy

MITM means Man-In-The-Middle. MITM proxies are usually deployed by companies in "enterprise environments" and elsewhere, where the owners of the network have a desire to investigate even TLS encrypted traffic.

To do this, they require users to install a custom "trust root" (Certificate Authority (CA) certificate) in the client, and then the proxy terminates all TLS traffic from the client, impersonates the remote server and acts like a proxy. The proxy then sends back a generated certificate signed by the custom CA. Such proxy setups usually transparently capture all traffic from clients to TCP port 443 on a remote machine. Running curl in such a network would also get its HTTPS traffic captured.

This practice, of course, allows the middle man to decrypt and snoop on all TLS traffic.

Authentication

HTTP and SOCKS proxies can require authentication, so curl then needs to provide the proper credentials to the proxy to be allowed to use it, and failing to do will only make the proxy return HTTP responses using code 407.

Authentication for proxies is similar to "normal" HTTP authentication. It is separate from the server authentication to allow clients to independently use both normal host authentication as well as proxy authentication.

With curl, you set the user name and password for the proxy authentication with the `-U user:password` or `--proxy-user user:password` option:

```
1 curl -U daniel:secr3t -x myproxy:80 http://example.com
```

This example will default to using the Basic authentication scheme. Some proxies will require another authentication scheme (and the headers that are returned when you get a 407 response will tell you which) and then you can ask for a specific method with `--proxy-digest`, `--proxy-negotiate`, `--proxy-ntlm`. The above example command again, but asking for NTLM auth with the proxy:

```
1 curl -U daniel:secr3t -x myproxy:80 http://example.com --proxy-ntlm
```

There is also the option that asks curl to figure out which method the proxy wants and supports and then go with that (with the possible expense of extra round-trips) using `--proxy-anyauth`. Asking curl to use any method the proxy wants is then like this:

```
1 curl -U daniel:secr3t -x myproxy:80 http://example.com --proxy-anyauth
```

HTTPS proxy

All the other mentioned protocols to speak with the proxy are clear text protocols, HTTP and the SOCKS versions. Using those methods could allow someone to eavesdrop on your traffic the local network where you or the proxy reside. Because over the connection between curl and the proxy, the data is sent in the clear.

One solution for that is to use an HTTPS proxy, speaking HTTPS to the proxy, which then establishes a secure and encrypted connection that is safe from easy surveillance.

When an HTTPS proxy is specified, the default port used on that host will be 443.

In all other ways, HTTPS proxies work like [HTTP proxies](#).

Proxy environment variables

curl checks for the existence of specially named environment variables before it runs to see if a proxy is requested to get used.

You specify the proxy by setting a variable named `[scheme]_proxy` to hold the proxy host name (the same way you would specify the host with `-x`). So if you want to tell curl to use a proxy when access an

HTTP server, you set the 'http_proxy' environment variable. Like this:

```
1 http_proxy=http://proxy.example.com:80
2 curl -v www.example.com
```

While the above example shows HTTP, you can, of course, also set ftp_proxy, https_proxy, and so on. All these proxy environment variable names except http_proxy can also be specified in uppercase, like HTTPS_PROXY.

To set a single variable that controls *all* protocols, the ALL_PROXY exists. If a specific protocol variable one exists, such a one will take precedence.

When using environment variables to set a proxy, you could easily end up in a situation where one or a few host names should be excluded from going through the proxy. This is then done with the NO_PROXY variable. Set that to a comma-separated list of host names that should not use a proxy when being accessed. You can set NO_PROXY to be a single asterisk (*) to match all hosts.

As an alternative to the NO_PROXY variable, there is also a --noproxy command line option that serves the same purpose and works the same way.

http_proxy in lower case only

The HTTP version of the proxy environment variables is treated differently than the others. It is only accepted in its lower case version because of the CGI protocol, which lets users run scripts in a server when invoked by an HTTP server. When a CGI script is invoked by a server, it automatically creates environment variables for the script based on the incoming headers in the request. Those environment variables are prefixed with uppercase HTTP_!

An incoming request to an HTTP server using a request header like `Proxy: yada` will therefore create the environment variable `HTTP_PROXY` set to contain `yada` before the CGI script is started. If that CGI script runs curl...

Accepting the upper case version of this environment variable has been the source for many security problems in lots of software through times.

Proxy headers

When you want to add HTTP headers meant specifically for an HTTP or HTTPS proxy, and not for the remote server, the --header option falls short.

For example, if you issue an HTTPS request through an HTTP proxy, it will be done by first issuing a `CONNECT` to the proxy that establishes a tunnel to the remote server and then it sends the request to that server. That first `CONNECT` is only issued to the proxy and you may want to make sure only that receives your special header, and send another set of custom headers to the remote server.

Set a specific different `User-Agent`: only to the proxy:

```
1 curl --proxy-header "User-Agent: magic/3000" -x proxy https://example.com/
```

Exit status

A lot of effort has gone into the project to make curl return a usable exit code when something goes wrong and it will always return 0 (zero) when the operation went as planned.

If you write a shell script or batch file that invokes curl, you can always check the return code to detect problems in the invoked command. Below, you will find a list of return codes as of the time of this writing. Over time we tend to slowly add new ones so if you get a code back not listed here, please refer to more updated curl documentation for aid.

A basic Unix shell script could look like something like this:

```
1#!/bin/sh
2curl http://example.com
3res=$?
4if test "$res" != "0"; then
5  echo "the curl command failed with: $res"
6fi
```

Available exit codes

1. Unsupported protocol. This build of curl has no support for this protocol. Usually this happens because the URL was misspelled to use a scheme part that either has a space in front of it or spells `http` like `htpt` or similar. Another common mistake is that you use a libcurl installation that was built with one or more protocols disabled and you now ask libcurl to use one of those protocols that were disabled in the build.
2. Failed to initialize. This is mostly an internal error or a problem with the libcurl installation or system libcurl runs in.
3. URL malformed. The syntax was not correct. This happens when you mistype a URL so that it ends up wrong, or in rare situations you are using a URL that is accepted by another tool that curl does not support only because there is no universal URL standard that everyone adheres to.
4. A feature or option that was needed to perform the desired request was not enabled or was explicitly disabled at build-time. To make curl able to do this, you probably need another build of libcurl.
5. Couldn't resolve proxy. The address of the given proxy host could not be resolved. Either the given proxy name is just wrong, or the DNS server is misbehaving and does not know about this name when it

- should or perhaps even the system you run curl on is misconfigured so that it does not find/use the
- 6. **Couldn't DNS resolve host.** The given remote host's address was not resolved. The address of the given server could not be resolved. Either the given host name is just wrong, or the DNS server is misbehaving and does not know about this name when it should or perhaps even the system you run curl on is misconfigured so that it does not find/use the correct DNS server.
 - 7. Failed to connect to host. curl managed to get an IP address to the machine and it tried to setup a TCP connection to the host but failed. This can be because you have specified the wrong port number, entered the wrong host name, the wrong protocol or perhaps because there is a firewall or another network equipment in between that blocks the traffic from getting through.
 - 8. Unknown FTP server response. The server sent data curl could not parse. This is either because of a bug in curl, a bug in the server or because the server is using an FTP protocol extension that curl does not support. The only real work-around for this is to tweak curl options to try it to use other FTP commands that perhaps will not get this unknown server response back.
 - 9. FTP access denied. The server denied login or denied access to the particular resource or directory you wanted to reach. Most often you tried to change to a directory that does not exist on the server. The directory of course is what you specify in the URL.
 - 10. FTP accept failed. While waiting for the server to connect back when an active FTP session is used, an error code was sent over the control connection or similar.
 - 11. FTP weird PASS reply. Curl could not parse the reply sent to the PASS request. PASS in the command curl sends the password to the server with, and even anonymous connections to FTP server actually sends a password - a fixed anonymous string. Getting a response back from this command that curl does not understand is a strong indication that this is not an FTP server at all or that the server is badly broken.
 - 12. During an active FTP session (PORT is used) while waiting for the server to connect, the timeout expired. It took too long for the server to get back. This is usually a sign that something is preventing the server from reaching curl successfully. Like a firewall or other network arrangements. .
 - 13. Unknown response to FTP PASV command, Curl could not parse the reply sent to the PASV request. This is a strange server. PASV is used to setup the second data transfer connection in passive mode, see the [FTP uses two connections](#) section for more on that. You might be able to work-around this problem by using PORT instead, with the `--ftp-port` option.
 - 14. Unknown FTP 227 format. Curl could not parse the 227-line the server sent. This is most certainly a broken server. A 227 is the FTP server's response when sending back information on how curl should connect back to it in passive mode. You might be able to work-around this problem by using PORT instead, with the `--ftp-port` option.
 - 15. FTP cannot get host. Couldn't use the host IP address we got in the 227-line. This is most likely an internal error.
 - 16. HTTP/2 error. A problem was detected in the HTTP2 framing layer. This is somewhat generic and can be one out of several problems, see the error message for details.
 - 17. FTP could not set binary. Couldn't change transfer method to binary. This server is broken. curl needs to set the transfer to the correct mode before it is started as otherwise the transfer cannot work.
 - 18. Partial file. Only a part of the file was transferred. When the transfer is considered complete, curl will verify that it actually received the same amount of data that it was told before-hand that it was going to get. If the two numbers do not match, this is the error code. It could mean that curl got fewer bytes than

advertised or that it got more. curl itself cannot know which number that is wrong or which is correct. If any [FTP](#) could not download/access the given file. The RETR (or similar) command failed. curl got an error from the server when trying to download the file.

20. Not used

21. Quote error. A quote command returned an error from the server. curl allows several different ways to send custom commands to a IMAP, POP3, SMTP or FTP server and features a generic check that the commands work. When any of the individually issued commands fails, this exit status is returned. The advice is generally to watch the headers in the FTP communication to better understand exactly what failed and how.

22. HTTP page not retrieved. The requested URL was not found or returned another error with the HTTP error code being 400 or above. This return code only appears if `-f`, `--fail` is used.

23. Write error. Curl could not write data to a local filesystem or similar. curl receives data chunk by chunk from the network and it stores it like at (or writes it to stdout), one piece at a time. If that write action gets an error, this is the exit status.

24. Not used

25. Upload failed. The server refused to accept or store the file that curl tried to send to it. This is usually due to wrong access rights on the server but can also happen due to out of disk space or other resource constraints. This error can happen for many protocols.

26. Read error. Various reading problems. The inverse to exit status 23. When curl sends data to a server, it reads data chunk by chunk from a local file or stdin or similar, and if that reading fails in some way this is the exit status curl will return.

27. Out of memory. A memory allocation request failed. curl needed to allocate more memory than what the system was willing to give it and curl had to exit. Try using smaller files or make sure that curl gets more memory to work with.

28. Operation timeout. The specified time-out period was reached according to the conditions. curl offers several [timeouts](#), and this exit code tells one of those timeout limits were reached. Extend the timeout or try changing something else that allows curl to finish its operation faster. Often, this happens due to network and remote server situations that you cannot affect locally.

29. Not used

30. FTP PORT failed. The PORT command failed. Not all FTP servers support the PORT command; try doing a transfer using PASV instead. The PORT command is used to ask the server to create the data connection by *connecting back* to curl. See also the [FTP uses two connections](#) section.

31. FTP could not use REST. The REST command failed. This command is used for resumed FTP transfers. curl needs to issue the REST command to do range or resumed transfers. The server is broken, try the same operation without range/resume as a crude work-around.

32. Not used

33. HTTP range error. The range request did not work. Resumed HTTP requests are not necessarily acknowledged or supported, so this exit code signals that for this resource on this server, there can be no range or resumed transfers.

34. HTTP post error. Internal post-request generation error. If you get this error, please report the exact circumstances to the curl project.

35. A TLS/SSL connect error. The SSL handshake failed. The SSL handshake can fail due to numerous

different reasons so the error message may offer some additional clues. Maybe the parties could not agree to a SSL/TLS version, an agreeable cipher suite or similar.

36. Bad download resume. Could not continue an earlier aborted download. When asking to resume a transfer that then ends up not possible to do, this error can get returned. For FILE, FTP or SFTP.
37. Couldn't read the given file when using the FILE:// scheme. Failed to open the file. The file could be non-existing or is it a permission problem perhaps?
38. LDAP cannot bind. LDAP "bind" operation failed, which is a necessary step in the LDAP operation and thus this means the LDAP query could not be performed. This might happen because of wrong username or password, or for other reasons.
39. LDAP search failed. The given search terms caused the LDAP search to return an error.
40. **Not used**
41. **Not used**
42. Aborted by callback. An application told libcurl to abort the operation. This error code is not generally made visible to users and not to users of the curl tool.
43. Bad function argument. A function was called with a bad parameter - this return code is present to help application authors to understand why libcurl cannot perform certain actions and should never be return by the curl tool. Please file a bug report to the curl project if this happens to you.
44. **Not used**
45. Interface error. A specified outgoing network interface could not be used. curl will typically decide outgoing network and IP addresses by itself but when explicitly asked to use a specific one that curl cannot use, this error can occur.
46. **Not used**
47. Too many redirects. When following HTTP redirects, libcurl hit the maximum number set by the application. The maximum number of redirects is unlimited by libcurl but is set to 50 by default by the curl tool. The limit is present to stop endless redirect loops. Change the limit with `--max-redirects`.
48. Unknown option specified to libcurl. This could happen if you use a curl version that is out of sync with the underlying libcurl version. Perhaps your newer curl tries to use an option in the older libcurl that was not introduced until after the libcurl version you are using but is known to your curl tool code as that is newer. To decrease the risk of this and make sure it does not happen: use curl and libcurl of the same version number.
49. Malformed telnet option. The telnet options you provide to curl was not using the correct syntax.
50. **Not used**
51. The server's SSL/TLS certificate or SSH fingerprint failed verification. curl can then not be sure of the server being who it claims to be. See the [using TLS with curl](#) section for more TLS details and [using SCP and SFTP with curl](#) for more SSH specific details.
52. The server did not reply anything, which in this context is considered an error. When an HTTP(S) server responds to an HTTP(S) request, it will always return *something* as long as it is alive and sound. All valid HTTP responses have a status line and responses header. Not getting anything at all back is an indication the server is faulty or perhaps that something prevented curl from reaching the right server or that you are trying to connect to the wrong port number etc.
53. SSL crypto engine not found.
- 54.

- Cannot set SSL crypto engine as default.
- 55. Failed sending network data. Sending data over the network is a crucial part of most curl operations and when curl gets an error from the lowest networking layers that the sending failed, this exit status gets returned. To pinpoint why this happens, some serious digging is usually required. Start with enabling verbose mode, do tracing and if possible check the network traffic with a tool like Wireshark or similar.
 - 56. Failure in receiving network data. Receiving data over the network is a crucial part of most curl operations and when curl gets an error from the lowest networking layers that the receiving of data failed, this exit status gets returned. To pinpoint why this happens, some serious digging is usually required. Start with enabling verbose mode, do tracing and if possible check the network traffic with a tool like Wireshark or similar.
57. **Not used**
- 58. Problem with the local certificate. The client certificate had a problem so it could not be used.
Permissions? The wrong pass phrase?
 - 59. Couldn't use the specified SSL cipher. The cipher names need to be specified exact and they are also unfortunately specific to the particular TLS backend curl has been built to use. For the current list of support ciphers and how to write them, see the online docs at <https://curl.se/docs/ssl-ciphers.html>.
 - 60. Peer certificate cannot be authenticated with known CA certificates. This usually means that the certificate is either self-signed or signed by a CA (Certificate Authority) that is not present in the CA store curl uses.
 - 61. Unrecognized transfer encoding. Content received from the server could not be parsed by curl.
 - 62. Invalid LDAP URL.
 - 63. Maximum file size exceeded.
 - 64. Requested FTP SSL level failed.
 - 65. Sending the data requires a rewind that failed.
 - 66. Failed to initialize SSL Engine.
 - 67. The user name, password, or similar was not accepted and curl failed to log in.
 - 68. File not found on TFTP server.
 - 69. Permission problem on TFTP server.
 - 70. Out of disk space on TFTP server.
 - 71. Illegal TFTP operation.
 - 72. Unknown TFTP transfer ID.
 - 73. File already exists (TFTP).
 - 74. No such user (TFTP).
 - 75. Character conversion failed.
 - 76. Character conversion functions required.
 - 77. Problem with reading the SSL CA cert
 - 78. The resource referenced in the URL does not exist.
 - 79. An unspecified error occurred during the SSH session.
80. Failed to shut down the SSL connection.

- 81. Not used
 - 82. Could not load CRL file, missing or wrong format
 - 83. TLS certificate issuer check failed
 - 84. The FTP PRET command failed
 - 85. RTSP: mismatch of CSeq numbers
 - 86. RTSP: mismatch of Session Identifiers
 - 87. Unable to parse FTP file list
 - 88. FTP chunk callback reported error
 - 89. No connection available, the session will be queued
 - 90. SSL public key does not matched pinned public key
 - 91. Invalid SSL certificate status
 - 92. Stream error in HTTP/2 framing layer
 - 93. An API function was called from inside a callback. If the curl tool returns this, something has gone wrong internally
 - 94. Authentication error.
 - 95. HTTP/3 layer error. This is somewhat generic and can be one out of several problems, see the error message for details.
 - 96. QUIC connection error. This error may be caused by an TLS library error. QUIC is the transport protocol used for HTTP/3.
-

Error message

When curl exits with a non-zero code, it will also output an error message (unless `--silent` is used). That error message may add some additional information or circumstances to the exit status number itself so the same error number can get different error messages.

"Not used"

The list of exit codes above contains a number of values marked as 'not used'. Those are exit status codes that are not used in modern versions of curl but that have been used or were intended to be used in the past. They may be used in a future version of curl.

Additionally, the highest used error status in this list is 92, but there is no guarantee that a future curl version will not add more exit codes after that number.

SCP and SFTP

curl supports the SCP and SFTP protocols if built with a prerequisite 3rd party library: [libssh2](#), [libssh](#) or [wolfSSH](#).

SCP and SFTP are both protocols that are built on top of SSH, a secure and encrypted data protocol that is similar to TLS but differs in a few important ways. For example, SSH does not use certificates of any sort but instead it uses public and private keys. Both SSH and TLS provide strong crypto and secure transfers when used correctly.

The SCP protocol is generally considered to be the black sheep of the two since it is not portable and usually only works between Unix systems.

URLs

SFTP and SCP URLs are similar to other URLs and you download files using these protocols the same as with others:

```
1 curl sftp://example.com/file.zip -u user
```

and:

```
1 curl scp://example.com/file.zip -u user
```

SFTP (but not SCP) supports getting a file listing back when the URL ends with a trailing slash:

```
1 curl sftp://example.com/ -u user
```

Note that both these protocols work with "users" and you do not ask for a file anonymously or with a standard generic name. Most systems will require that users authenticate, as outlined below.

When requesting a file from an SFTP or SCP URL, the file path given is considered to be the absolute path on the remote server unless you specifically ask for the path relative to the user's home directory. You do that by making sure the path starts with `/~/`. This is quite the opposite to how FTP URLs work and is a common cause for confusion among users.

For user `daniel` to transfer `todo.txt` from his home directory, it would look similar to this:

```
1 curl sftp://example.com/~/todo.txt -u daniel
```

Authentication

Authentication with curl against an SSH server (when you specify an SCP or SFTP URL) is done like this:

1. curl connects to the server and learns which authentication methods that this server offers

2. curl then tries the offered methods one by one until one works or they all failed

curl will attempt to use your public key as found in the `.ssh` subdirectory in your home directory if the server offers public key authentication. When doing so, you still need to tell curl which user name to use on the server. For example, the user 'john' lists the entries in his home directory on the remote SFTP server called 'sftp.example.com':

```
1 curl -u john: sftp://sftp.example.com/
```

If curl cannot authenticate with the public key for any reason, it will instead attempt to use the user name + password if the server allows it and the credentials are passed on the command line.

For example, the same user from above has the password `RHvx6wUA` on a remote system and can download a file via SCP like this:

```
1 curl -u john:RHvx6wUA -O scp://ssh.example.com/file.tar.gz
```

Known hosts

A secure network client needs to make sure that the remote host is exactly the host that it thinks it is communicating with. With TLS based protocols, it is done by the client verifying the server's certificate.

With SSH protocols there are no server certificates, but instead each server can provide its unique key. And unlike TLS, SSH has no certificate authorities or anything so the client simply has to make sure that the host's key matches what it already knows (via other means) it should look like.

The matching of keys is typically done using hashes of the key and the file that the client stores the hashes for known servers is often called `known_hosts` and is put in a dedicated SSH directory. On Linux systems that is usually called `~/.ssh`.

When curl connects to a SFTP and SCP host, it will make sure that the host's key hash is already present in the known hosts file or it will deny continued operation because it cannot trust that the server is the right one. Once the correct hash exists in `known_hosts` curl can perform transfers.

To force curl to skip checking and obeying to the `known_hosts` file, you can use the `-k` / `--insecure` command-line option. You must use this option with extreme care since it makes it possible for man-in-the-middle attacks not to be detected.

Reading email

There are two dominant protocols on the Internet for reading/downloading email from servers (at least if we do not count web based reading), and they are IMAP and POP3. The former being the slightly more modern alternative. curl supports both.

POP3

To list message numbers and sizes:

```
1 curl pop3://mail.example.com/
```

To download message 1:

```
1 curl pop3://mail.example.com/1
```

To delete message 1:

```
1 curl --request DELE pop3://mail.example.com/1
```

IMAP

Get the mail using the UID 57 from mailbox 'stuff':

```
1 curl imap://server.example.com/stuff;UID=57
```

Instead, get the mail with index 57 from the mailbox 'fun':

```
1 curl imap://server.example.com/fun;MAILINDEX=57
```

List the mails in the mailbox 'boring':

```
1 curl imap://server.example.com/boring
```

List the mails in the mailbox 'boring' and provide user and password:

```
1 curl imap://server.example.com/boring -u user:password
```

TLS for emails

POP3 and IMAP can both be done over a secure connection and both can be done using either explicit or implicit TLS. The "explicit" method is probably the most common approach and it means that the client connects to the server using an insecure connection and *upgrades* it to TLS as it goes, using the

`STARTTLS` command. You tell curl to attempt this with `--ssl` or if you want to *insist* on a secure connection you use `--ssl-reqd`. Like this:

```
1 curl pop3://mail.example.com/ --ssl-reqd
```

or

```
1 curl --ssl imap://mail.example.com/inbox
```

"Implicit" SSL means that the connection gets secured already at first connect, which you make curl attempt by specifying a scheme in the URL that uses SSL. In this case either `pop3s://` or `imaps://`. For such connections, curl will insist on connecting and negotiating a TLS connection already from the start, or it will fail its operation.

The previous explicit examples done with implicit SSL:

```
1 curl pop3s://mail.example.com/
```

or

```
1 curl imaps://mail.example.com/inbox
```

Sending email

Sending email with curl is done with the SMTP protocol. SMTP stands for [Simple Mail Transfer Protocol](#).

curl supports sending data to an SMTP server, which combined with the right set of command line options makes an email get sent to a set of receivers of your choice.

When sending SMTP with curl, there are two necessary command line options that **must** be used.

- You need to tell the server at least one recipient with `--mail-rcpt`. You can use this option several times and then curl will tell the server that all those email addresses should receive the email.
- You need to tell the server which email address that is the sender of the email with `--mail-from`. It is important to realize that this email address is not necessarily the same as is shown in the `From:` line of the email text.

Then, you need to provide the actual email data. This is a (text) file formatted according to [RFC 5322](#). It is a set of headers and a body. Both the headers and the body need to be correctly encoded. The headers typically include `To:`, `From:`, `Subject:`, `Date:` etc.

A basic command to send an email:

```
1 curl smtp://mail.example.com --mail-from myself@example.com --mail-rcpt \
```

```
2 receiver@example.com --upload-file email.txt
```

Example email.txt

```
1 From: John Smith <john@example.com>
2 To: Joe Smith <smith@example.com>
3 Subject: an example.com example email
4 Date: Mon, 7 Nov 2016 08:45:16
5
6 Dear Joe,
7 Welcome to this example email. What a lovely day.
```

Secure mail transfer

Some mail providers allow or require using SSL for SMTP. They may use a dedicated port for SSL or allow SSL upgrading over a clear-text connection.

If your mail provider has a dedicated SSL port you can use `smtps://` instead of `smtp://`, which uses the SMTP SSL port of 465 by default and requires the entire connection to be SSL. For example `smtps://smtp.gmail.com/`.

However, if your provider allows upgrading from clear-text to secure transfers you can use one of these options:

```
1 --ssl           Try SSL/TLS (FTP, IMAP, POP3, SMTP)
2 --ssl-reqd     Require SSL/TLS (FTP, IMAP, POP3, SMTP)
```

You can tell curl to *try* but not require upgrading to secure transfers by adding `--ssl` to the command:

```
1 curl --ssl smtp://mail.example.com --mail-from myself@example.com \
2     --mail-rcpt receiver@example.com --upload-file email.txt \
3     --user 'user@your-account.com:your-account-password'
```

You can tell curl to *require* upgrading to using secure transfers by adding `--ssl-reqd` to the command:

```
1 curl --ssl-reqd smtp://mail.example.com --mail-from myself@example.com \
2     --mail-rcpt receiver@example.com --upload-file email.txt \
3     --user 'user@your-account.com:your-account-password'
```

The SMTP URI

The path part of a SMTP request specifies the host name to present during communication with the mail server. If the path is omitted then curl will attempt to figure out the local computer's host name and use that. However, this may not return the fully qualified domain name that is required by some mail servers and specifying this path allows you to set an alternative name, such as your machine's fully qualified domain name, which you might have obtained from an external function such as `gethostname` or `getaddrinfo`.

To connect to the mail server at `mail.example.com` and send your local computer's host name in the HELO / EHLO command:

```
1 curl smtp://mail.example.com
```

You can of course as always use the `-v` option to get to see the client-server communication.

To instead have curl send `client.example.com` in the HELO / EHLO command to the mail server at `mail.example.com`, use:

```
1 curl smtp://mail.example.com/client.example.com
```

No MX lookup!

When you send email with an ordinary mail client, it will first check for an MX record for the particular domain you want to send email to. If you send an email to `joe@example.com`, the client will get the MX records for `example.com` to learn which mail server(s) to use when sending email to example.com users.

curl does no MX lookups by itself. If you want to figure out which server to send an email to for a particular domain, we recommend you figure that out first and then call curl to use those servers. Useful command line tools to get MX records include 'dig' and 'nslookup'.

MQTT

A plain "GET" subscribes to the topic and prints all published messages. Doing a "POST" publishes the post data to the topic and exits.

Subscribe to the temperature in the "home/bedroom" subject published by example.com:

```
1 curl mqtt://example.com/home/bedroom/temp
```

Send the value '75' to the "home/bedroom/dimmer" subject hosted by the example.com server:

```
1 curl -d 75 mqtt://example.com/home/bedroom/dimmer
```

What does curl deliver as a response to a subscribe

It outputs two bytes topic length (MSB | LSB), the topic followed by the payload.

Caveats

Remaining limitations in curl's MQTT support as of September 2020:

- No username support
- Only QoS level 0 is implemented for publish
- No way to set retain flag for publish
- No username/password support
- No TLS (mqtts) support

TFTP

Trivial File Transfer Protocol (TFTP) is a simple clear-text protocol that allows a client to get a file from or put a file onto a remote host.

Primary use cases for this protocol have been to get the boot image over a local network. TFTP also stands out a little next to many other protocols by the fact that it is done over UDP as opposed to TCP which most other protocols use.

There is no secure version or flavor of TFTP.

Download

Download a file from the TFTP server of choice:

```
1 curl -O tftp://localserver/file.boot
```

Upload

Upload a file to the TFTP server of choice:

```
1 curl -T file boot tftp://localserver/
```

TFTP options

The TFTP protocols transmits data to the other end of the communication using "blocks". When a TFTP transfer is setup, both parties either agree on using the default block size of 512 bytes or negotiate a different one. curl supports block sizes between 8 to 65464 bytes.

You ask curl to use a different size than the default with `--tftp-blksize`. Ask for 8192 bytes blocks like this:

```
1 curl --tftp-blksize 8192 tftp://localserver/file
```

It has been shown that there are server implementations that do not handle option negotiation at all, so curl also has the ability to completely switch off all attempts of setting options. If you are in the unfortunate of working with such a server, use the flag like this:

```
1 curl --tftp-no-options tftp://localserver/file
```

TELNET

Telnet is an ancient application protocol for bidirectional **clear-text** communication. It was designed for interactive text-oriented communications and *there is no encrypted or secure version* of Telnet.

TELNET is not a perfect match for curl. The protocol is not done to handle plain uploads or downloads so the usual curl paradigms have had to be stretched somewhat to make curl deal with it appropriately.

curl sends received data to stdout and it reads input to send on stdin. The transfer is complete when the connection is dropped or when the user presses control-c.

Historic TELNET

Once upon the time, systems provided telnet access for login. Then you could connect to a server and login to it, much like how you would do it with SSH today. That practice have fortunately now mostly been moved into the museum cabinets due to the insecure nature of the protocol.

The default port number for telnet is 23.

Debugging with TELNET

The fact that TELNET is basically just a simple clear-text TCP connection to the target host and port makes it somewhat useful to debug other protocols and services at times.

Example, connect to your local HTTP server on port 80 and send a (broken) request to it by manually entering `GET /` and press return twice:

```
1 curl telnet://localhost:80
```

Your web server will most probably return something like this back:

```
1 HTTP/1.1 400 Bad Request
2 Date: Tue, 07 Dec 2021 07:41:16 GMT
3 Server: softare/7.8.9
4 Content-Length: 31
5 Connection: close
6 Content-Type: text/html
7
8 [message]
```

Options

When curl sets up a TELNET connection to a server, you can ask it to pass on options. You do this with `--telnet-option` (or `-t`), and there are three options available to use:

- `TTYPE=<term>` sets the "terminal type" for the session to be `<term>`.
- `XDISPLOC=<X display>` sets the X display location
- `NEW_ENV=<var, val>` sets the environment variable `var` to the value `val` in the remote session

Login to your local machine's telnet server and tell it you use a vt100 terminal:

```
1 curl --telnet-option TTYPE=vt100 telnet://localhost
```

You need to manually enter your name and password when prompted

DICT

DICT is a protocol for dictionary lookups.

Usage

For fun try

```
1 curl dict://dict.org/m:curl  
2 curl dict://dict.org/d:heisenbug:jargon  
3 curl dict://dict.org/d:daniel:gcide
```

Aliases for 'm' are 'match' and 'find', and aliases for 'd' are 'define' and 'lookup'. For example,

```
1 curl dict://dict.org/find:curl
```

Commands that break the URL description of the RFC (but not the DICT protocol) are

```
1 curl dict://dict.org/show:db  
2 curl dict://dict.org/show:strat
```

TLS

TLS stands for Transport Layer Security and is the name for the technology that was formerly called SSL. The term SSL has not really died though so these days both the terms TLS and SSL are often used interchangeably to describe the same thing.

TLS is a cryptographic security layer "on top" of TCP that makes the data tamper proof and guarantees server authenticity, based on strong public key cryptography and digital signatures.

- [Ciphers](#)
- [Enable TLS](#)
- [TLS versions](#)
- [Verifying server certificates](#)
- [Certificate pinning](#)
- [OCSP stapling](#)
- [Client certificates](#)
- [TLS auth](#)
- [TLS backends](#)
- [SSLKEYLOGFILE](#)

Ciphers

When curl connects to a TLS server, it negotiates how to speak the protocol and that negotiation involves several parameters and variables that both parties need to agree to. One of the parameters is which

cryptography algorithms to use, the so called cipher. Over time, security researchers figure out flaws and Using the verbose option, `-v`, you can get information about which cipher and TLS version are negotiated. By using the `--ciphers` option, you can change what cipher to prefer in the negotiation, but mind you, this is a power feature that takes knowledge to know how to use in ways that do not just make things worse.

Enable TLS

curl supports the TLS version of many protocols. HTTP has HTTPS, FTP has FTPS, LDAP has LDAPS, POP3 has POP3S, IMAP has IMAPS and SMTP has SMTPS.

If the server side supports it, you can use the TLS version of these protocols with curl.

There are two general approaches to do TLS with protocols. One of them is to speak TLS already from the first connection handshake while the other is to "upgrade" the connection from plain-text to TLS using protocol specific instructions.

With curl, if you explicitly specify the TLS version of the protocol (the one that has a name that ends with an 'S' character) in the URL, curl will try to connect with TLS from start, while if you specify the non-TLS version in the URL you can *usually* upgrade the connection to TLS-based with the `--ssl` option.

The support table looks like this:

Clear	TLS version	--ssl
HTTP	HTTPS	no
LDAP	LDAPS	no
FTP	FTPS	yes
POP3	POP3S	yes
IMAP	IMAPS	yes
SMTP	SMTPS	yes

The protocols that *can* do `--ssl` all favor that method. Using `--ssl` means that curl will *attempt* to upgrade the connection to TLS but if that fails, it will still continue with the transfer using the plain-text version of the protocol. To make the `--ssl` option **require** TLS to continue, there is instead the `--ssl-reqd` option which will make the transfer fail if curl cannot successfully negotiate TLS.

Require TLS security for your FTP transfer:

```
1 curl --ssl-reqd ftp://ftp.example.com/file.txt
```

Suggest TLS to be used for your FTP transfer:

```
1 curl --ssl ftp://ftp.example.com/file.txt
```

Connecting directly with TLS (to HTTPS://, LDAPS://, FTPS:// etc) means that TLS is mandatory and curl will return an error if TLS is not negotiated.

Get a file over HTTPS:

```
1 curl https://www.example.com/
```

TLS versions

SSL was invented in the mid 90s and has developed ever since. SSL version 2 was the first widespread version used on the Internet but that was deemed insecure already a long time ago. SSL version 3 took over from there, and it too has been deemed not safe enough for use.

TLS version 1.0 was the first "standard". RFC 2246 was published 1999. TLS 1.1 came out in 2006, further improving security, followed by TLS 1.2 in 2008. TLS 1.2 came to be the gold standard for TLS for a decade.

TLS 1.3 (RFC 8446) was finalized and published as a standard by the IETF in August 2018. This is the most secure and fastest TLS version as of date. It is however so new that a lot of software, tools and libraries do not yet support it.

curl is designed to use a "safe version" of SSL/TLS by default. It means that it will not negotiate SSLv2 or SSLv3 unless specifically told to, and in fact several TLS libraries no longer provide support for those protocols so in many cases curl is not even able to speak those protocol versions unless you make a serious effort.

Option	Use
--sslv2	SSL version 2
--sslv3	SSL version 3
--tlsv1	TLS >= version 1.0
--tlsv1.0	TLS >= version 1.0
--tlsv1.1	TLS >= version 1.1
--tlsv1.2	TLS >= version 1.2
--tlsv1.3	TLS >= version 1.3

Verifying server certificates

Having a secure connection to a server is not worth a lot if you cannot also be certain that you are

communicating with the **correct** host. If we do not know that, we could just as well be talking with an

To check that it communicates with the right TLS server, curl uses a set of locally stored CA certificates to verify the signature of the server's certificate. All servers provide a certificate to the client as part of the TLS handshake and all public TLS-using servers have acquired that certificate from an established Certificate Authority.

After some applied crypto magic, curl knows that the server is in fact the correct one that acquired that certificate for the host name that curl used to connect to it. Failing to verify the server's certificate is a TLS handshake failure and curl exits with an error.

In rare circumstances, you may decide that you still want to communicate with a TLS server even if the certificate verification fails. You then accept the fact that your communication may be subject to Man-In-The-Middle attacks. You lower your guards with the `-k` or `--insecure` option.

CA store

curl needs a "CA store", a collection of CA certificates, to verify the TLS server it talks to.

If curl is built to use a TLS library that is "native" to your platform, chances are that library will use the native CA store as well. If not, curl has to either have been built to know where the local CA store is, or users need to provide a path to the CA store when curl is invoked.

You can point out a specific CA bundle to use in the TLS handshake with the `--cacert` command line option. That bundle needs to be in PEM format. You can also set the environment variable `CURL_CA_BUNDLE` to the full path.

CA store on windows

curl built on windows that is not using the native TLS library (Schannel), have an extra sequence for how the CA store can be found and used.

curl will search for a CA cert file named "curl-ca-bundle.crt" in these directories and in this order:

1. application's directory
2. current working directory
3. Windows System directory (e.g. `C:\windows\system32`)
4. Windows Directory (e.g. `C:\windows`)
5. all directories along `%PATH%`

Certificate pinning

TLS certificate pinning is a way to verify that the public key used to sign the servers certificate has not changed. It is "pinned".

When negotiating a TLS or SSL connection, the server sends a certificate indicating its identity. A public key is extracted from this certificate and if it does not exactly match the public key provided to this option, curl will abort the connection before sending or receiving any data.

You tell curl a file name to read the sha256 value from, or you specify the base64 encoded hash directly in the command line with a `sha256//` prefix. You can specify one or more hashes like that, separated with semicolons (;).

```
1 curl --pinnedpubkey "sha256//83d34tasd3rt..." https://example.com/
```

This feature is not supported by all TLS backends.

OCSP stapling

This uses the TLS extension called Certificate Status Request to ask the server to provide a fresh "proof" from the CA in the handshake, that the certificate that it returns is still valid. This is a way to make really sure the server's certificate has not been revoked.

If the server does not support this extension, the test will fail and curl returns an error. And it is still far too common that servers do not support this.

Ask for the handshake to use the status request like this:

```
1 curl --cert-status https://example.com/
```

This feature is only supported by the OpenSSL, GnuTLS and NSS backends.

Client certificates

TLS client certificates are a way for clients to cryptographically prove to servers that they are truly the right peer (also sometimes known as Mutual TLS or mTLS). A command line that uses a client certificate specifies the certificate and the corresponding key, and they are then passed on the TLS handshake with the server.

You need to have your client certificate already stored in a file when doing this and you should supposedly have gotten it from the right instance via a different channel previously.

The key is typically protected by a password that you need to provide or get prompted for interactively.

curl offers options to let you specify a single file that is both the client certificate and the private key concatenated using `--cert`, or you can specify the key file independently with `--key`:

```
1 curl --cert mycert:mypassword https://example.com
2 curl --cert mycert:mypassword --key mykey https://example.com
```

For some TLS backends you can also pass in the key and certificate using different types:

```
1 curl --cert mycert:mypassword --cert-type PEM \
2     --key mykey --key-type PEM https://example.com
```

TLS auth

TLS connections offer a (rarely used) feature called Secure Remote Passwords. Using this, you authenticate the connection for the server using a name and password and the command line flags for this are `--tlsuser <name>` and `--tlspassword <secret>`. Like this:

```
1 curl --tlsuser daniel --tlspassword secret https://example.com
```

TLS backends

When curl is built, it gets told to use a specific TLS library. That TLS library is the engine that provides curl with the powers to speak TLS over the wire. We often refer to them as different "backends" as they can be seen as different pluggable pieces into the curl machine. curl can be built to be able to use one or more of these backends.

Sometimes features and behaviors differ slightly when curl is built with different TLS backends, but the developers work hard on making those differences as small and unnoticeable as possible.

Showing the curl version information with `curl --version` will always include the TLS library and version in the first line of output.

Multiple TLS backends

When curl is built with *multiple* TLS backends, it can be told which one to use each time it is started. It is always built to use a specific one by default unless one is asked for.

If you invoke `curl --version` for a curl with multiple backends it will mention `MultiSSL` as a feature in the last line. The first line will then include all the supported TLS backends with the non-default ones within parentheses.

To set a specific one to get used, set the environment variable `CURL_SSL_BACKEND` to its name.

SSLKEYLOGFILE

Wireshark interface showing network traffic analysis. The packet list displays various TCP and HTTP2 frames. The details pane shows Stream settings for a specific connection, and the bytes pane shows raw hex and ASCII data.

Wireshark interface showing decrypted SSL traffic. The tabs at the top indicate decrypted SSL records and data. The bottom status bar shows the number of packets and the current display range.

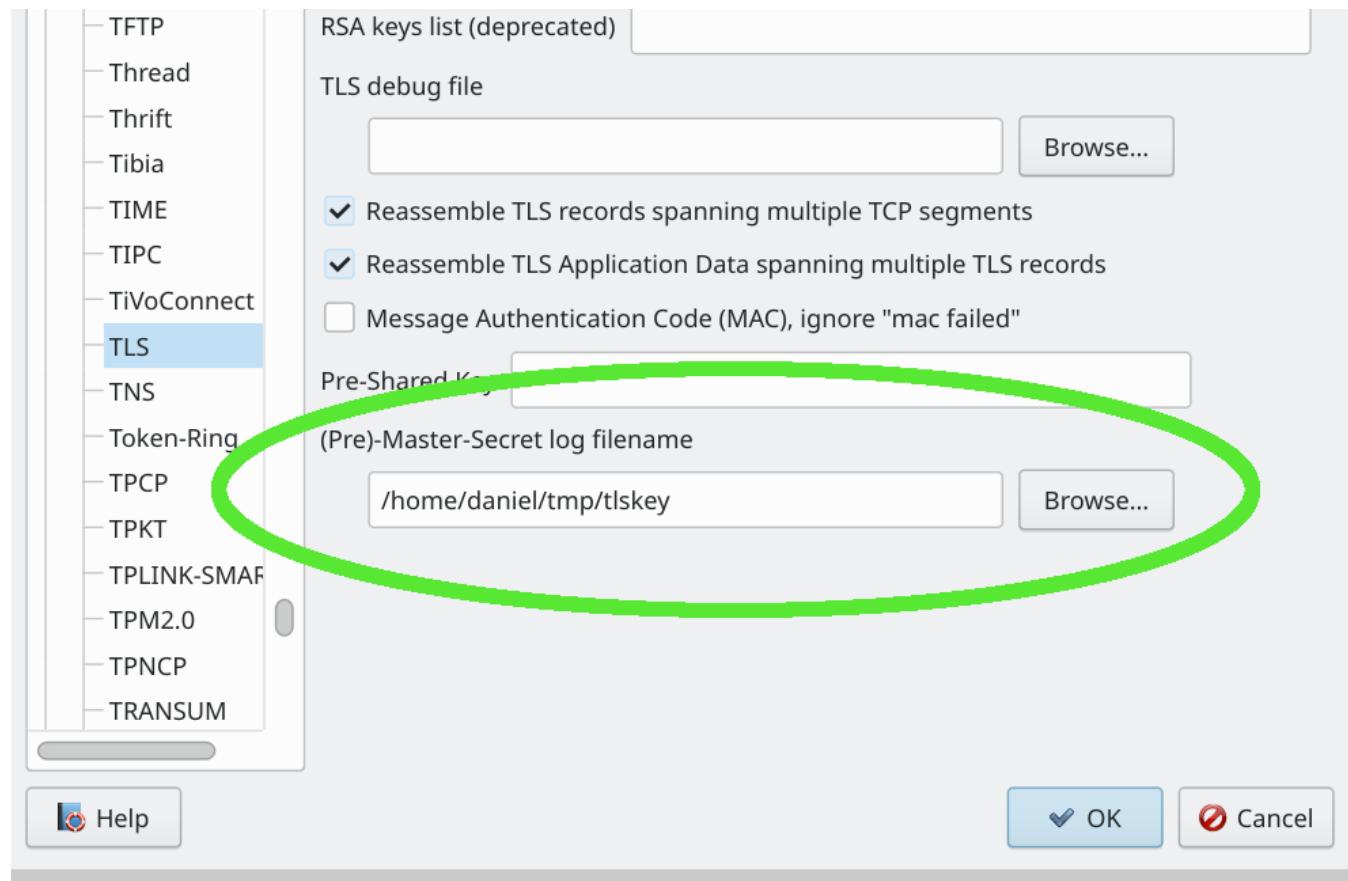
Since a long time back, the venerable network analyzer tool Wireshark (screenshot above) has provided a way to decrypt and inspect TLS traffic when sent and received by Firefox and Chrome.

This is similarly possible to do with curl.

You do this by making the browser or curl tell Wireshark the encryption secrets so that it can decrypt them:

1. set the environment variable named `SSLKEYLOGFILE` to a file name of your choice before you start the browser or curl
2. Setting the same file name path in the Master-secret field in Wireshark. Go to Preferences->Protocols->TLS and edit the path as shown in the screenshot below.

Wireshark Preferences window showing the Transport Layer Security (TLS) settings. The RSA keys list button is highlighted.



set the ssl key file name

Having done this simple operation, you can now inspect curl's or your browser's HTTPS traffic in Wireshark. Just super handy and awesome.

Just remember that if you record TLS traffic and want to save it for analyzing later, you need to also save the file with the secrets so that you can decrypt that traffic capture at a later time as well.

libcurl-using applications too!

Support for `SSLKEYLOGFILE` is provided by libcurl itself - making it possible for you to trace and inspect the TLS network data for any application built to use libcurl - not just the curl command line tool.

Restrictions

The support for `SSLKEYLOGFILE` requires that curl was built with a TLS backend that supports this feature. The backends that support `SSLKEYLOGFILE` are: OpenSSL, libressl, BoringSSL, GnuTLS, NSS and wolfSSL.

If curl was built to use another backend, you cannot record your curl TLS traffic this way.

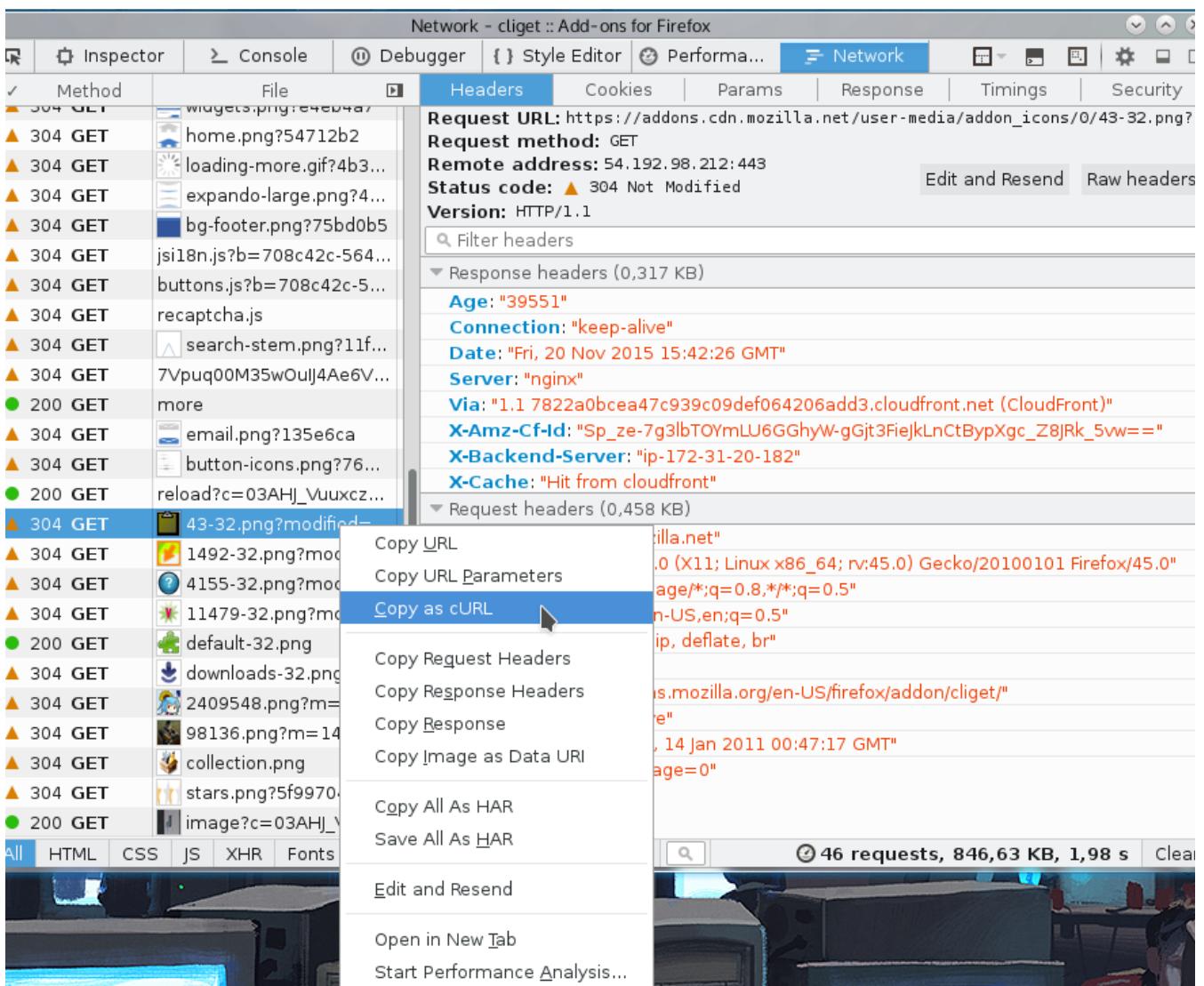
Copy as curl

Using curl to reproduce an operation a user just managed to do with his or her browser is a common request and area people ask for help about.

How do you get a curl command line to get a resource, just like the browser would get it, nice and easy? Chrome, Firefox, Edge and Safari all have this feature.

From Firefox

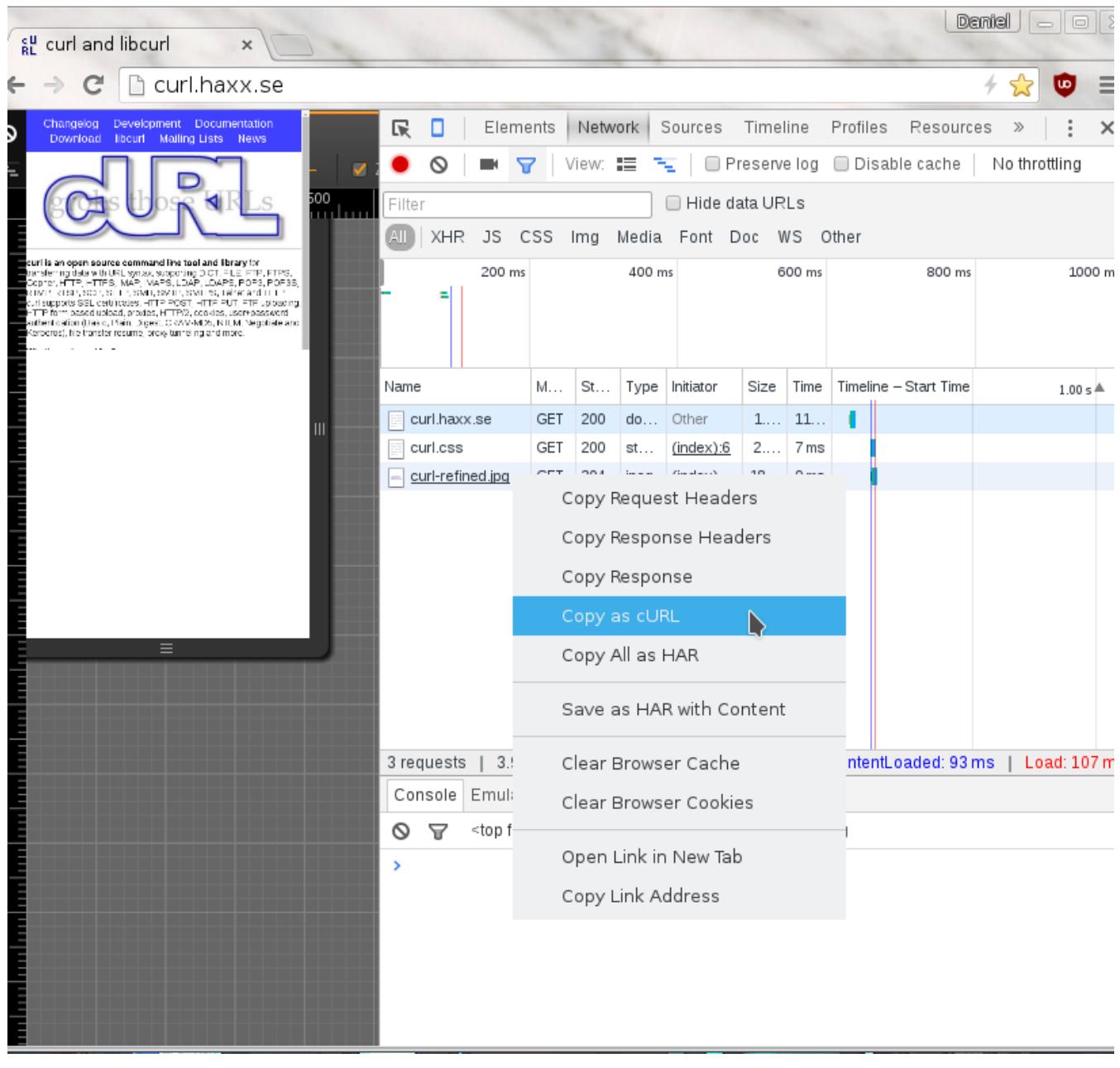
You get the site shown with Firefox's network tools. You then right-click on the specific request you want to repeat in the "Web Developer->Network" tool when you see the HTTP traffic, and in the menu that appears you select "Copy as cURL". Like this screenshot below shows. The operation then generates a curl command line to your clipboard and you can then paste that into your favorite shell window. This feature is available by default in all Firefox installations.



copy as curl with Firefox

From Chrome and Edge

When you pop up the More tools->Developer mode in Chrome or Edge, and you select the Network tab you see the HTTP traffic used to get the resources of the site. On the line of the specific resource you are interested in, you right-click with the mouse and you select "Copy as cURL" and it will generate a command line for you in your clipboard. Paste that in a shell to get a curl command line that makes the transfer. This feature is available by default in all Chrome and Chromium installations.

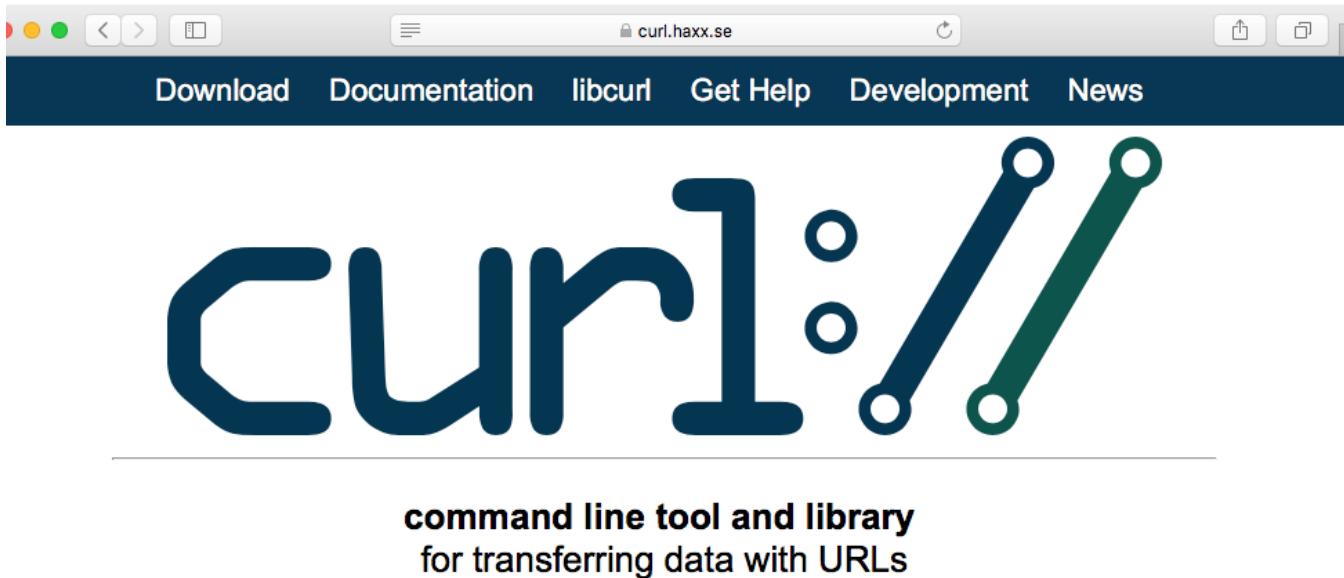


From Safari

In Safari, the "development" menu is not visible until you go into **preferences->Advanced** and enable it. But once you have done that, you can select **Show web inspector** in that development menu and get to see a new console pop up that is similar to the development tools of Firefox and Chrome.

Select the network tab, reload the web page and then you can right click the particular resources that you

want to fetch with curl, as if you did it with Safari..



The screenshot shows the curl.haxx.se homepage. The page features a large, stylized 'curl' logo with a green and blue color scheme. Below the logo, the text 'command line tool and library' and 'for transferring data with URLs' is displayed. A 'Supports...' section lists various protocols and features. At the bottom, there's a screenshot of the Safari developer tools Network tab, showing a list of resources for curl.haxx.se. One resource, 'curl.haxx.se', has a context menu open with options like 'Open in New Tab', 'Reveal in Resources Tab', 'Copy Link Address', 'Copy as curl', and 'Save File'. The 'Copy as curl' option is highlighted.

curl.haxx.se

command line tool and library
for transferring data with URLs

Supports...

DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, Telnet and TFTP. curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form-based upload, proxies, HTTP/2, cookies, user+password authentication /Basic

Name	Domain	Type	M...	S...	St...	Cached	Size	Transf...	Start...	Latency	Duration	100.00ms
curl.haxx.se	curl.haxx.se	Do...	G...	H...	200	No	8.0...	2.82 ...	0.055ms	51.06ms	0.011ms	
curl.css		Sty...	G...	H...	200	Yes (...	7.8...	0 B	119.1ms	0.112ms	0.087ms	
curl-logo.svg		Ima...	G...	H...	200	Yes (...	8.4...	0 B	120.0ms	0.044ms	0.045ms	
everything-curl.jpg		Ima...	G...	H...	200	Yes (...	15....	0 B	120.4ms	0.050ms	0.046ms	

copy as curl with Safari

On Firefox, without using the devtools

If this is something you would like to get done more often, you probably find using the developer tools a bit inconvenient and cumbersome to pop up just to get the command line copied. Then [cliget](#) is the perfect add-on for you as it gives you a new option in the right-click menu, so you can get a quick command line generated really quickly, like this example when I right-click an image in Firefox:



The screenshot shows a Firefox browser window with the title bar 'Daniel Stenberg - daniel.haxx.se - Nightly'. The address bar shows 'daniel.haxx.se'. Below the address bar, there are navigation buttons, a search bar, and other browser controls. The main content area displays the text 'front | blog | photos | contact | projects' and the heading 'Daniel Stenberg - daniel.haxx.se'. On the right side of the screen, there is a dark image thumbnail. A right-click context menu is open over the image thumbnail, with options 'View Image', 'Copy Image', and 'Copy Image Location' visible.

Daniel Stenberg - daniel.haxx.se - Nightly

daniel.haxx.se

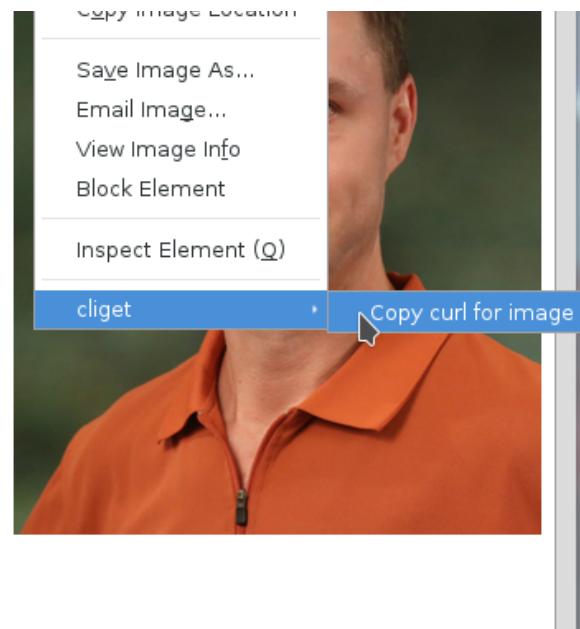
front | blog | photos | contact | projects

Daniel Stenberg -
daniel.haxx.se

View Image
Copy Image
Copy Image Location

*I write only open source software.
I work on open standards. I write
open documents. I share
knowledge.*

I am the author and maintainer of [cURL](#) and [libcurl](#). An internet protocol geek, an open source person and a hacker. I've been programming for fun and profit since 1985. You'll find lots of info about my various [projects](#) on these web pages.



cliget with Firefox

Not perfect

These methods all give you a command line to reproduce their HTTP transfers. You will also learn they are still often not the perfect solution to your problems. Why? Well mostly because these tools are written to rerun the exact same request that you copied, while you often want to rerun the same logic but not sending an exact copy of the same cookies and file contents etc.

These tools will give you command lines with static and fixed cookie contents to send in the request, because that is the contents of the cookies that were sent in the browser's requests. You will most likely want to rewrite the command line to dynamically adapt to whatever the content is in the cookie that the server told you in a previous response. And so on.

The copy as curl functionality is also often notoriously bad at using `-F` and instead they provide handcrafted `--data-binary` solutions including the mime separator strings etc.

HTTP with curl

In all user surveys and during all curl's lifetime, HTTP has been the most important and most frequently used protocol that curl supports. This chapter will explain how to do effective HTTP transfers and general fiddling with curl.

This will mostly work the same way for HTTPS, as they are really the same thing under the hood, as HTTPS is HTTP with an extra security TLS layer. See also the specific [HTTPS](#) section below.

HTTP methods

In every HTTP request, there is a method. Sometimes called a verb. The most commonly used ones are GET, POST, HEAD and PUT.

Normally however you do not specify the method in the command line, but instead the exact method used depends on the specific options you use. GET is default, using `-d` or `-F` makes it a POST, `-I` generates a HEAD and `-T` sends a PUT.

More about this in the [Modify the HTTP request](#) section.

Protocol basics

HTTP protocol basics

(This assumes you have read the [Network and protocols](#) section or are otherwise already familiar with protocols.)

HTTP is a protocol that is easy to learn the basics of. A client connects to a server—and it is always the client that takes the initiative—sends a request and receives a response. Both the request and the response consist of headers and a body. There can be little or a lot of information going in both directions.

An HTTP request sent by a client starts with a request line, followed by headers and then optionally a body. The most common HTTP request is probably the GET request which asks the server to return a specific resource, and this request does not contain a body.

When a client connects to 'example.com' and asks for the '/' resource, it sends a GET without a request body:

```
1 GET / HTTP/1.1
2 User-agent: curl/2000
3 Host: example.com
```

...the server could respond with something like below, with response headers and a response body ('hello'). The first line in the response also contains the response code and the specific version the server supports:

```
1 HTTP/1.1 200 OK
2 Server: example-server/1.1
3 Content-Length: 5
4 Content-Type: plain/text
5
6 hello
```

If the client would instead send a request with a small request body ('hello'), it could look like this:

```
1 POST / HTTP/1.1
2 Host: example.com
3 User-agent: curl/2000
4 Content-Length: 5
```

A server always responds to an HTTP request unless something is wrong.

The URL converted to a request

So when an HTTP client is given a URL to operate on, that URL is then used, picked apart and those parts are used in various places in the outgoing request to the server. Let's take the an example URL:

```
1 https://www.example.com/path/to/file
```

- **https** means that curl will use TLS to the remote port 443 (which is the default port number when no specified is used in the URL).
- **www.example.com** is the host name that curl will resolve to one or more IP address to connect to. This host name will also be used in the HTTP request in the `Host:` header.
- **/path/to/file** is used in the HTTP request to tell the server which exact document/resources curl wants to fetch

Responses

HTTP responses

When an HTTP client talks HTTP to a server, the server *will* respond with an HTTP response message or curl will consider it an error and returns 52 with the error message "Empty reply from server".

Size of an HTTP response

An HTTP response has a certain size and curl needs to figure it out. There are several different ways to signal the end of an HTTP response but the most basic way is to use the `Content-Length:` header in the response and with that specify the exact number of bytes in the response body.

Some early HTTP server implementations had problems with file sizes greater than 2GB and wrongly managed to send Content-Length: headers with negative sizes or otherwise just plain wrong data. curl can be told to ignore the Content-Length: header completely with `--ignore-content-length`. Doing so may have some other negative side-effects but should at least let you get the data.

HTTP response codes

An HTTP transfer gets a 3 digit response code back in the first response line. The response code is the server's way of giving the client a hint about how the request was handled.

It is important to note that curl does not consider it an error even if the response code would indicate that the requested document could not be delivered (or similar). curl considers a successful sending and receiving of HTTP to be good.

The first digit of the HTTP response code is a kind of "error class":

- 1xx: transient response, more is coming
- 2xx: success
- 3xx: a redirect
- 4xx: the client asked for something the server could not or would not deliver
- 5xx: there is a problem in the server

Remember that you can use curl's `--write-out` option to extract the response code. See the [--write-out](#) section.

To make curl return an error for response codes ≥ 400 , you need to use `--fail` or `--fail-with-body`. Then curl will exit with error code 22 for such occurrences.

CONNECT response codes

Since there can be an HTTP request and a separate CONNECT request in the same curl transfer, we often separate the CONNECT response (from the proxy) from the remote server's HTTP response.

The CONNECT is also an HTTP request so it gets response codes in the same numeric range and you can use `--write-out` to extract that code as well.

Chunked transfer encoding

An HTTP 1.1 server can decide to respond with a "chunked" encoded response, a feature that was not present in HTTP 1.0.

When receiving a chunked response, there is no Content-Length: for the response to indicate its size. Instead, there is a `Transfer-Encoding: chunked` header that tells curl there is chunked data coming and then in the response body, the data comes in a series of "chunks". Every individual chunk starts with the size of that particular chunk (in hexadecimal), then a newline and then the contents of the chunk. This is repeated over and over until the end of the response, which is signaled with a zero sized chunk. The point of this response encoding is for the client to be able to figure out when the response has ended even though the server did not know the full size before it started to send it. This is usually the case when the response is dynamic and generated at the point when the request comes.

Clients like curl will, of course, decode the chunks and not show the chunk sizes to users.

Gzipped transfers

Responses over HTTP can be sent in compressed format. This is most commonly done by the server when it includes a `Content-Encoding: gzip` in the response as a hint to the client. Compressed responses make a lot of sense when either static resources are sent (that were compressed previously) or even in runtime when there is more CPU power available than bandwidth. Sending a much smaller amount of data is often preferred.

You can ask curl to both ask for compressed content *and* automatically and transparently uncompress gzipped data when receiving content encoded gzip (or in fact any other compression algorithm that curl understands) by using `--compressed`:

```
1 curl --compressed http://example.com/
```

Transfer encoding

A less common feature used with transfer encoding is compression.

Compression in itself is common. Over time the dominant and web compatible way to do compression for HTTP has become to use `Content-Encoding` as described in the section above. But HTTP was originally intended and specified to allow transparent compression as a transfer encoding, and curl supports this feature.

The client then simply asks the server to do compression transfer encoding and if acceptable, it will respond with a header indicating that it will and curl will then transparently uncompress that data on arrival. A user enables asking for compressed transfer encoding with `--tr-encoding`:

```
1 curl --tr-encoding http://example.com/
```

It should be noted that not many HTTP servers in the wild support this.

Pass on transfer encoding

In some situations you may want to use curl as a proxy or other in-between software. In those cases, curl's way to deal with transfer-encoding headers and then decoding the actual data transparently may not be desired, if the end receiver *also* expects to do the same.

You can then ask curl to pass on the received data, without decoding it. That means passing on the sizes in the chunked encoding format or the compressed format when compressed transfer encoding is used etc.

```
1 curl --raw http://example.com/
```

Authentication

Each HTTP request can be made authenticated. If a server or a proxy want the user to provide proof that they have the correct credentials to access a URL or perform an action, it can send an HTTP response code that informs the client that it needs to provide a correct HTTP authentication header in the request to be allowed.

A server that requires authentication sends back a 401 response code and an associated `WWW-Authenticate`: header that lists all the authentication methods that the server supports.

An HTTP proxy that requires authentication sends back a 407 response code and an associated `Proxy-Authenticate`: header that lists all the authentication methods that the proxy supports.

It might be worth to note that most websites of today do not require HTTP authentication for login etc, but

they will instead ask users to login on web pages and then the browser will issue a POST with the user and password etc. and then subsequently maintain cookies for the session.

To tell curl to do an authenticated HTTP request, you use the `-u`, `--user` option to provide user name and password (separated with a colon). Like this:

```
1 curl --user daniel:secret http://example.com/
```

This will make curl use the default "Basic" HTTP authentication method. Yes, it is actually called Basic and it is truly basic. To explicitly ask for the basic method, use `--basic`.

The Basic authentication method sends the user name and password in clear text over the network (base64 encoded) and should be avoided for HTTP transport.

When asking to do an HTTP transfer using a single (specified or implied), authentication method, curl will insert the authentication header already in the first request on the wire.

If you would rather have curl first test if the authentication is really required, you can ask curl to figure that out and then automatically use the most safe method it knows about with `--anyauth`. This makes curl try the request unauthenticated, and then switch over to authentication if necessary:

```
1 curl --anyauth --user daniel:secret http://example.com/
```

and the same concept works for HTTP operations that may require authentication:

```
1 curl --proxy-anyauth --proxy-user daniel:secret http://example.com/ \
2      --proxy http://proxy.example.com:80/
```

curl typically (a little depending on how it was built) speaks several other authentication methods as well, including Digest, Negotiate and NTLM. You can ask for those methods too specifically:

```
1 curl --digest --user daniel:secret http://example.com/
2 curl --negotiate --user daniel:secret http://example.com/
3 curl --ntlm --user daniel:secret http://example.com/
```

Ranges

What if the client only wants the first 200 bytes out of a remote resource or perhaps 300 bytes somewhere in the middle? The HTTP protocol allows a client to ask for only a specific data range. The client asks the server for the specific range with a start offset and an end offset. It can even combine things and ask for several ranges in the same request by just listing a bunch of pieces next to each other. When a server sends back multiple independent pieces to answer such a request, you will get them separated with mime boundary strings and it will be up to the user application to handle that accordingly. curl will not further separate such a response.

However, a byte range is only a request to the server. It does not have to respect the request and in many cases, like when the server automatically generates the contents on the fly when it is being asked, it will simply refuse to do it and it then instead responds with the full contents anyway.

You can make curl ask for a range with `-r` or `--range`. If you want the first 200 bytes out of something:

```
1 curl -r 0-199 http://example.com
```

Or everything in the file starting from index 200:

```
1 curl -r 200- http://example.com
```

Get 200 bytes from index 0 *and* 200 bytes from index 1000:

```
1 curl -r 0-199,1000-1199 http://example.com/
```

HTTP versions

As any other Internet protocol, the HTTP protocol has kept evolving over the years and now there are clients and servers distributed over the world and over time that speak different versions with varying levels of success. So in order to get libcurl to work with the URLs you pass in libcurl offers ways for you to specify which HTTP version that request and transfer should use. libcurl is designed in a way so that it tries to use the most common, the most sensible if you want, default values first but sometimes that is not enough and then you may need to instruct libcurl on what to do.

curl defaults to HTTP/1.1 for HTTP servers but if you connect to HTTPS and you have a curl that has HTTP/2 abilities built-in, it attempts to negotiate HTTP/2 automatically or falls back to 1.1 in case the negotiation failed. Non-HTTP/2 capable curs get 1.1 over HTTPS by default.

Option	Description
<code>--http1.0</code>	HTTP/1.0
<code>--http1.1</code>	HTTP/1.1
<code>--http2</code>	HTTP/2
<code>--http2-prior-knowledge</code>	HTTP/2
<code>--http3</code>	HTTP/3

Accepting HTTP/0.9

The HTTP version used before HTTP/1.0 was made available is often referred to as HTTP/0.9. Back in

those days, HTTP responses had no headers as they would only return a response body and then immediately close the connection.

curl can be told to support such responses but will by default not recognize them, for security reasons. Almost anything bad will look like an HTTP/0.9 response to curl so the option needs to be used with caution.

The HTTP/0.9 option to curl is different than the other HTTP command line options for HTTP versions mentioned above as this controls what response to accept, while the others are about what HTTP protocol version to try to use.

Tell curl to accept an HTTP/0.9 response like this:

```
1 curl --http0.9 https://example.com/
```

Conditionals

Sometimes users want to avoid downloading a file again if the same file maybe already has been downloaded the day before. This can be done by making the HTTP transfer "conditioned" on something. curl supports two different conditions: the file timestamp and etag.

Check by modification date

Download the file only if it is newer than a specific date with the use of the `-z` or `--time-cond` option:

```
1 curl -z "Jan 10, 2017" https://example.com/file -O
```

Or the reverse, get the file only if it is older than the specific time by prefixing the date with a dash:

```
1 curl --time-cond "-Jan 10, 2017" https://example.com/file -O
```

The date parser is liberal and accepts most formats you can write the date with, and you can also specify it complete with a time:

```
1 curl --time-cond "Sun, 12 Sep 2004 15:05:58 -0700" https://www.example.org/file.html
```

The `-z` option can also extract and use the timestamp from a local file, which is handy to only download a file if it has been updated remotely:

```
1 curl -z file.html https://example.com/file.html -O
```

It is often useful to combine the use of `-z` with the `--remote-time` flag, which sets the time of the locally created file to the same timestamp as the remote file had:

```
1 curl -z file.html -o file.html --remote-time https://example.com/file.html
```

Check by modification of content

HTTP servers can return a specific "ETag" for a given resource version. If the resource at a given URL changes, a new Etag value must be generated, so a client knows that as long as the ETag remains the same, the content has not changed.

Using ETags, curl can check for remote updates without having to rely on times or file dates. It also then makes the check able to detect sub-second changes, which the timestamp based checks cannot.

Using curl you can download a remote file and save its ETag (if it provides any) in a separate "cache" by using the `--etag-save` command line option. Like this:

```
1 curl --etag-save etags.txt https://example.com/file -o output
```

A subsequent command line can then use that previously saved etag and make sure to only download the file again if it has changed, like this:

```
1 curl --etag-compare etag.txt https://example.com/file -o output
```

The two etag options can also be combined in the same command line, so that if the file actually was updated, curl would save the update ETag again in the file:

```
1 curl --etag-compare etag.txt --etag-save etag.txt \
2   https://example.com/file -o output
```

HTTPS

HTTPS is in effect Secure HTTP. The "secure" part means that the TCP transport layer is enhanced to provide authentication, privacy (encryption) and data integrity by the use of TLS.

See the [Using TLS](#) section for in-depth details on how to modify and tweak the TLS details in an HTTPS transfer.

HTTP POST

POST is the HTTP method that was invented to send data to a receiving web application, and it is how most common HTML forms on the web works. It usually sends a chunk of relatively small amounts of data to the

receiver.

This section describes the simple posts, for multipart formposts done with `-F`, check out [Multipart formposts](#).

- [Simple POST](#)
- [Content-Type](#)
- [Posting binary](#)
- [JSON](#)
- [URL encoding](#)
- [Convert to GET](#)
- [Expect 100-continue](#)
- [Chunked encoded POSTs](#)
- [Hidden form fields](#)
- [Figure out what a browser sends](#)
- [JavaScript and forms](#)

Simple POST

When the data is sent by a browser after data have been filled in a form, it will send it URL encoded, as a serialized name=value pairs separated with ampersand symbols (`&`). You send such data with curl's `-d` or `--data` option like this:

```
1 curl -d 'name=admin&shoesize=12' http://example.com/
```

When specifying multiple `-d` options on the command line, curl will concatenate them and insert ampersands in between, so the above example could also be made like this:

```
1 curl -d name=admin -d shoesize=12 http://example.com/
```

If the amount of data to send is not really fit to put in a mere string on the command line, you can also read it off a file name in standard curl style:

```
1 curl -d @filename http://example.com
```

While the server might assume that the data is encoded in some special way, curl does not encode or change the data you tell it to send. **curl sends exactly the bytes you give it.**

To send a POST body that starts with a `@` symbol, to avoid that curl tries to load that as a file name, use `--data-raw` instead. This option has no file loading capability:

```
1 curl -d "now" | curl https://example.com
```

Content-Type

POSTing with curl's `-d` option will make it include a default header that looks like `Content-Type: application/x-www-form-urlencoded`. That is what your typical browser will use for a plain POST.

Many receivers of POST data do not care about or check the Content-Type header.

If that header is not good enough for you, you should, of course, replace that and instead provide the correct one. Such as if you POST JSON to a server and want to more accurately tell the server about what the content is:

```
1 curl -d '{json}' -H 'Content-Type: application/json' https://example.com
```

Posting binary

When reading data to post from a file, `-d` will strip out carriage return and newlines. Use `--data-binary` if you want curl to read and use the given file in binary exactly as given:

```
1 curl --data-binary @filename http://example.com/
```

JSON

curl 7.82.0 introduced the `--json` option as a new way to send JSON formatted data to HTTP servers using POST. This option works as a shortcut and provides a single option that replaces these three:

```
1 --data [arg]
2 --header "Content-Type: application/json"
3 --header "Accept: application/json"
```

This option does not make curl actually understand or know about the JSON data it sends, but it makes it easier to send it. curl will not touch or parse the data that it sends, so you need to make sure it is valid JSON yourself.

Send a basic JSON object to a server:

```
1 curl --json '{"tool": "curl"}' https://example.com/
```

Send JSON from a local file:

```
1 curl --json @json.txt https://example.com/
```

Send JSON passed to curl on stdin:

```
1 echo '{"a":"b"}' | curl --json @- https://example.com/
```

You can use multiple `--json` options on the same command line. This makes curl concatenate the contents from the options and send all data in one go to the server. Note that the concatenation is plain text based and will not in any way merge the JSON object as per JSON.

Send JSON from a file and concatenate a string to the end:

```
1 curl --json @json.txt --json ", \"end\": \"true\"}" https://example.com/
```

Crafting JSON to send

The quotes used in JSON data sometimes makes it a bit difficult and cumbersome to write and use in shells and scripts.

Using a separate tool for this purpose might make things easier for you, and one tool in particular that might help you accomplish this is [jo](#).

Send a basic JSON object to a server with jo and `--json`

```
1 jo -p name=jo n=17 parser=false | curl --json @- https://example.com/
```

Receiving JSON

curl itself does not know or understand the contents it sends or receives, including when the server returns JSON in its response.

Using a separate tool for the purpose of parsing or pretty-printing JSON responses might make things easier for you, and one tool in particular that might help you accomplish this is [jq](#).

Send a basic JSON object to a server, and pretty-print the JSON response:

```
1 curl --json '{"tool": "curl"}' https://example.com/ | jq
```

Send the JSON with `jo`, print the response with `jq`:

```
1 jo -p name=jo n=17 | curl --json @- https://example.com/ | jq
```

jq is a powerful and capable tool for extracting, filtering and managing JSON content that goes way beyond just pretty-printing.

URL encoding

Percent-encoding, also known as URL encoding, is technically a mechanism for encoding data so that it can appear in URLs. This encoding is typically used when sending POSTs with the `application/x-www-form-urlencoded` content type, such as the ones curl sends with `--data` and `--data-binary` etc.

The command-line options mentioned above all require that you provide properly encoded data, data you need to make sure already exists in the right format. While that gives you a lot of freedom, it is also a bit inconvenient at times.

To help you send data you have not already encoded, curl offers the `--data-urlencode` option. This option offers several different ways to URL encode the data you give it.

You use it like `--data-urlencode data` in the same style as the other `--data` options. To be CGI-compliant, the **data** part should begin with a name followed by a separator and a content specification. The **data** part can be passed to curl using one of the following syntaxes:

- `content` : This will make curl URL encode the content and pass that on. Just be careful so that the content does not contain any = or @ symbols, as that will then make the syntax match one of the other cases below!
- `=content` : This will make curl URL encode the content and pass that on. The initial '=' symbol is not included in the data.
- `name=content` : This will make curl URL encode the content part and pass that on. Note that the name part is expected to be URL encoded already.
- `@filename` : This will make curl load data from the given file (including any newlines), URL encode that data and pass it on in the POST.
- `name@filename` : This will make curl load data from the given file (including any newlines), URL encode that data and pass it on in the POST. The name part gets an equal sign appended, resulting in `name=urlencoded-file-content`. Note that the name is expected to be URL encoded already.

As an example, you could POST a name to have it encoded by curl:

```
1 curl --data-urlencode "name=John Doe (Junior)" http://example.com
```

...which would send the following data in the actual request body:

```
1 name=John%20Doe%20%28Junior%29
```

If you store the string `John Doe (Junior)` in a file named `contents.txt`, you can tell curl to send that contents URL encoded using the field name 'user' like this:

```
1 curl --data-urlencode user@contents.txt http://example.com
```

In both these examples above, the field name is not URL encoded but is passed on as-is. If you want to URL encode the field name as well, like if you want to pass on a field name called `user name`, you can ask curl to encode the entire string by prefixing it with an equals sign (that will not get sent):

```
1 curl --data-urlencode "=user name=John Doe (Junior)" http://example.com
```

Convert to GET

A little convenience feature that could be suitable to mention in this context is the `-G` or `--get` option, which takes all data you have specified with the different `-d` variants and appends that data to the inputted URL e.g. `http://example.com` separated with a '?' and then makes curl send a GET instead.

This option makes it easy to switch between POSTing and GETting a form, for example.

An example that adds an encoded piece of data as a query in the URL:

```
1 curl -G --data-urlencode "name=daniel stenberg" https://example.com/
```

Expect 100-continue

HTTP/1 has no proper way to stop an ongoing transfer (in any direction) and still maintain the connection. So, if we figure out that the transfer had better stop after the transfer has started, there are only two ways to proceed: cut the connection and pay the price of reestablishing the connection again for the next request, or keep the transfer going and waste bandwidth but be able to reuse the connection next time.

One example of when this can happen is when you send a large file over HTTP, only to discover that the server requires authentication and immediately sends back a 401 response code.

The mitigation that exists to make this scenario less frequent is to have curl pass on an extra header, `Expect: 100-continue`, which gives the server a chance to deny the request before a lot of data is sent off. curl sends this `Expect:` header by default if the POST it will do is known or suspected to be larger than just minuscule. curl also does this for PUT requests.

When a server gets a request with an 100-continue and deems the request fine, it will respond with a 100 response that makes the client continue. If the server does not like the request, it sends back response code for the error it thinks it is.

Unfortunately, lots of servers in the world do not properly support the `Expect:` header or do not handle it correctly, so curl will only wait 1000 milliseconds for that first response before it will continue anyway.

Those are 1000 wasted milliseconds. You can then remove the use of `Expect:` from the request and avoid the waiting with `-H`:

```
1 curl -H Expect: -d "payload to send" http://example.com
```

In some situations, curl will inhibit the use of the `Expect` header if the content it is about to send is small (like below one kilobyte), as having to waste such a small chunk of data is not considered much of a problem.

HTTP/2 and later

HTTP/2 and later versions of HTTP can stop an ongoing transfer without shutting down the connection, which makes `Expect:` pointless.

Chunked encoded POSTs

When talking to an HTTP 1.1 server, you can tell curl to send the request body without a `Content-Length:` header upfront that specifies exactly how big the POST is. By insisting on curl using chunked Transfer-Encoding, curl will send the POST chunked piece by piece in a special style that also sends the size for each such chunk as it goes along.

You send a chunked POST with curl like this:

```
1 curl -H "Transfer-Encoding: chunked" -d @file http://example.com
```

Caveats

This assumes that you know you do this against an HTTP/1.1 server. Before 1.1, there was no chunked encoding, and after version 1.1 chunked encoding has been deprecated.

Hidden form fields

Sending a post with `-d` is the equivalent of what a browser does when an HTML form is filled in and submitted.

Submitting such forms is a common operation with curl; effectively, to have curl fill in a web form in an automated fashion.

If you want to submit a form with curl and make it look as if it has been done with a browser, it is important to provide all the input fields from the form. A common method for web pages is to set a few hidden input fields to the form and have them assigned values directly in the HTML. A successful form submission, of course, also includes those fields and in order to do that automatically you may be forced to first download the

HTML page that holds the form, parse it, and extract the hidden field values so that you can send them off.

Figure out what a browser sends

A common shortcut is to simply fill in the form with your browser and submit it and check in the browser's network development tools exactly what it sent.

A slightly different way is to save the HTML page containing the form, and then edit that HTML page to redirect the 'action=' part of the form to your own server or a test server that just outputs exactly what it gets. Completing that form submission will then show you exactly how a browser sends it.

A third option is, of course, to use a network capture tool such as Wireshark to check exactly what is sent over the wire. If you are working with HTTPS, you cannot see form submissions in clear text on the wire but instead you need to make sure you can have Wireshark extract your TLS private key from your browser. See the Wireshark documentation for details on doing that.

JavaScript and forms

A common mitigation against automated agents or scripts using curl is to have the page with the HTML form use JavaScript to set values of some input fields, usually one of the hidden ones. Often, there is some JavaScript code that executes on page load or when the submit button is pressed which sets a magic value that the server then can verify before it considers the submission to be valid.

You can usually work around that by just reading the JavaScript code and redoing that logic in your script. Using the tricks in [Figure out what a browser sends](#) to check exactly what a browser sends is then also a good help.

Multipart formposts

A multipart formpost is what an HTTP client sends when an HTML form is submitted with `enctype` set to "multipart/form-data". It is an HTTP POST request sent with the request body specially formatted as a series of "parts", separated with MIME boundaries.

An example piece of HTML would look like this:

```
1 <form action="submit.cgi" method="post" enctype="multipart/form-data">
2   Name: <input type="text" name="person"><br>
3   File: <input type="file" name="secret"><br>
4   <input type="submit" value="Submit">
5 </form>
```

Which could look something like this in a web browser:

Name:

File: Browse... No file selected.

a multipart form

A user can fill in text in the 'Name' field and by pressing the 'Browse' button a local file can be selected that will be uploaded when 'Submit' is pressed.

Sending such a form with curl

With curl, you add each separate multipart with one `-F` (or `--form`) flag and you then continue and add one `-F` for every input field in the form that you want to send.

The above small example form has two parts, one named 'person' that is a plain text field and one named 'secret' that is a file.

Send your data to that form like this:

```
1 curl -F person=anonymous -F secret=@file.txt http://example.com/submit.cgi
```

The HTTP this generates

The **action** specifies where the POST is sent. **method** says it is a POST and **enctype** tells us it is a multipart formpost.

With the fields filled in as shown above, curl generates and sends these HTTP request headers to the host example.com:

```
1 POST /submit.cgi HTTP/1.1
2 Host: example.com
3 User-Agent: curl/7.46.0
4 Accept: */*
5 Content-Length: 313
6 Expect: 100-continue
7 Content-Type: multipart/form-data; boundary=-----d74496d66958873e
```

Content-Length, of course, tells the server how much data to expect. This example's 313 bytes is really small.

The **Expect** header is explained in the [Expect 100 continue](#) chapter.

The **Content-Type** header is a bit special. It tells that this is a multipart formpost and then it sets the "boundary" string. The boundary string is a line of characters with a bunch of random digits somewhere in it, that serves as a separator between the different parts of the form that will be submitted. The particular boundary you see in this example has the random part `d74496d66958873e` but you will, of course, get something different when you run curl (or when you submit such a form with a browser).

So after that initial set of headers follows the request body

```
1 -----d74496d66958873e
2 Content-Disposition: form-data; name="person"
3
4 anonymous
5 -----d74496d66958873e
6 Content-Disposition: form-data; name="secret"; filename="file.txt"
7 Content-Type: text/plain
8
9 contents of the file
10 -----d74496d66958873e--
```

Here you clearly see the two parts sent, separated with the boundary strings. Each part starts with one or more headers describing the individual part with its name and possibly some more details. Then after the part's headers come the actual data of the part, without any sort of encoding.

The last boundary string has two extra dashes `--` appended to signal the end.

Content-Type

POSTing with curl's `-F` option will make it include a default **Content-Type** header in its request, as shown in the above example. This says `multipart/form-data` and then specifies the MIME boundary string. That `content-type` is the default for multipart formposts but you can, of course, still modify that for your own commands and if you do, curl is clever enough to still append the boundary magic to the replaced header. You cannot really alter the boundary string, since curl needs that for producing the POST stream.

To replace the header, use `-H` like this:

```
1 curl -F 'name=Dan' -H 'Content-Type: multipart/magic' https://example.com
```

Converting a web form

There are a few different ways to figure out how to write a curl command line to submit a multipart form as seen in HTML.

- Save the HTML locally, run `nc` locally to listen on a chosen port number, change the `action` URL to submit the POST to your local `nc` instance. Submit the form and watch how `nc` shows it. Then
2. Use the "development tools" in your favorite browser and inspect the POST request in the network tab after you have submitted it. Then convert that HTTP data to a curl command line. Unfortunately, the [copy as curl](#) feature in the browsers usually do not actually do multipart formposts particularly well.
 3. Inspect the source HTML and convert into a curl command line directly from that.
-

From `<form>` to `-F`

In a `<form>` that uses `enctype="multipart/form-data"`, the first step is to find the `action=` property as that tells the target for the POST. You need to convert that into a full URL for your curl command line.

An example action looks like this:

```
1 <form action="submit.cgi" method="post" enctype="multipart/form-data">
```

If the form is found in a web page hosted on a URL like for example

`https://example.com/user/login` the `action=submit.cgi` is a relative path within the same "directory" as the form itself. The full URL to submit this form thus becomes

`https://example.com/user/submit.cgi`. That is the URL to use in the curl command line.

Next, you must identify every `<input>` tag used within the form, including the ones that are marked as "hidden". Hidden just means that they are not shown in the web page, but they should still be sent in the POST.

For every `<input>` in the form there should be a corresponding `-F` in the command line.

text input

A regular tag using type "text" in the style like

```
1 <input type="text" name="person">
```

should then set the field name with content like this:

```
1 curl -F "person=Mr Smith" https://example.com/
```

file input

When the input type is set to a "file", like in:

```
1 curl -F image=@funnycat.gif https://example.com/
```

You provide a file for this part by specifying the file name and use `@` and the path to the file to include:

```
1 curl -F image=@funnycat.gif https://example.com/
```

hidden input

A common technique to pass on state from a form is to set a number of `<input>` tags as `type="hidden"`. This is basically the same thing as an already filled in form field, so you convert this to a command line by using the name and value. For example:

```
1 <input type="hidden" name="username" value="bob123">
```

This is converted like for the normal text field, and here you know what the content should be:

```
1 curl -F "username=bob123" https://example.com/
```

All fields at once

If we toy with the idea that all the three different `<input>` tags showed in the examples above were used in the same `<form>`, then a complete curl command line to send, including the correct URL as extracted above, would look like:

```
1 curl -F "person=Mr Smith" -F image=@funnycat.gif -F "username=bob123" \
2 https://example.com/user/submit.cgi
```

-d vs -F

Previous chapters talked about [regular POST](#) and [multipart formpost](#), and in your typical command lines you do them with `-d` or `-F`.

When do you use which of them?

As described in the chapters mentioned above, both these options send the specified data to the server. The difference is in how the data is formatted over the wire. Most of the time, the receiving end is written to expect a specific format and it expects that the sender formats and sends the data correctly. A client cannot just pick a format of its own choice.

HTML web forms

When we are talking browsers and HTML, the standard way is to offer a form to the user that sends off data

when the form has been filled in. The `<form>` tag is what makes one of those appear on the web page. The tag instructs the browser how to format its POST. If the form tag includes `enctype=multipart/form-data`, it tells the browser to send the data as a [multipart formpost](#) which you make with curl's `-F` option. This method is typically used when the form includes a `<input type=file>` tag, for file uploads.

The default `enctype` used by forms, which is rarely spelled out in HTML since it is default, is `application/x-www-form-urlencoded`. It makes the browser "URL encode" the input as name=value pairs with the data encoded to avoid unsafe characters. We often refer to that as a [regular POST](#), and you perform one with curl's `-d` and friends.

POST outside of HTML

POST is a regular HTTP method and there is no requirement that it be triggered by HTML or involve a browser. Lots of services, APIs and other systems allow you to pass in data these days in order to get things done.

If these services expect plain "raw" data or perhaps data formatted as JSON or similar, you want the [regular POST](#) approach. curl's `-d` option does not alter or encode the data at all but will just send exactly what you tell it to. Just pay attention that `-d` sets a default `Content-Type:` that might not be what you want.

Redirects

HTTP redirects

The "redirect" is a fundamental part of the HTTP protocol. The concept was present and is documented already in the first spec (RFC 1945), published in 1996, and it has remained well-used ever since.

A redirect is exactly what it sounds like. It is the server sending back an instruction to the client instead of giving back the contents the client wanted. The server says "go look over *here* instead for that thing you asked for".

Redirects are not all alike. How permanent is the redirect? What request method should the client use in the next request?

All redirects also need to send back a `Location:` header with the new URI to ask for, which can be absolute or relative.

Permanent and temporary

Is the redirect meant to last or just remain valid for now? If you want a GET to permanently redirect users to resource B with another GET, send back a 301. It also means that the user-agent (browser) is meant to cache this and keep going to the new URI from now on when the original URI is requested.

The temporary alternative is 302. Right now the server wants the client to send a GET request to B, but it

~~should not cache this but keep trying the original URI when directed to it next time~~

Note that both 301 and 302 will make browsers do a GET in the next request, which possibly means changing the method if it started with a POST (and only if POST). This changing of the HTTP method to GET for 301 and 302 responses is said to be “for historical reasons”, but that’s still what browsers do so most of the public web will behave this way.

In practice, the 303 code is similar to 302. It will not be cached and it will make the client issue a GET in the next request. The differences between a 302 and 303 are subtle, but 303 seems to be more designed for an “indirect response” to the original request rather than just a redirect.

These three codes were the only redirect codes in the HTTP/1.0 spec.

curl however, does not remember or cache any redirects at all so to it, there is really no difference between permanent and temporary redirects.

Tell curl to follow redirects

In curl’s tradition of only doing the basics unless you tell it differently, it does not follow HTTP redirects by default. Use the `-L`, `--location` option to tell it to do that.

When following redirects is enabled, curl will follow up to 50 redirects by default. There is a maximum limit mostly to avoid the risk of getting caught in endless loops. If 50 is not sufficient for you, you can change the maximum number of redirects to follow with the `--max-redirects` option.

GET or POST?

All three of these response codes, 301 and 302/303, will assume that the client sends a GET to get the new URI, even if the client might have sent a POST in the first request. This is important, at least if you do something that does not use GET.

If the server instead wants to redirect the client to a new URI and wants it to send the same method in the second request as it did in the first, like if it first sent POST it’d like it to send POST again in the next request, the server would use different response codes.

To tell the client “the URI you sent a POST to, is permanently redirected to B where you should instead send your POST now and in the future”, the server responds with a 308. And to complicate matters, the 308 code is only recently defined (the [spec](#) was published in June 2014) so older clients may not treat it correctly! If so, then the only response code left for you is...

The (older) response code to tell a client to send a POST also in the next request but temporarily is 307. This redirect will not be cached by the client though, so it’ll again post to A if requested to again. The 307 code was introduced in HTTP/1.1.

Oh, and redirects work the same way in HTTP/2 as they do in HTTP/1.1.

Text	Permanent	Temporary
Switch to GET	301	302 and 303
Keep original method	308	307

Decide what method to use in redirects

It turns out that there are web services out there in the world that want a POST sent to the original URL, but are responding with HTTP redirects that use a 301, 302 or 303 response codes and *still* want the HTTP client to send the next request as a POST. As explained above, browsers won't do that and neither will curl—by default.

Since these setups exist, and they're actually not terribly rare, curl offers options to alter its behavior.

You can tell curl to not change the non-GET request method to GET after a 30x response by using the dedicated options for that: `--post301`, `--post302` and `--post303`. If you are instead writing a libcurl based application, you control that behavior with the `CURLOPT_POSTREDIR` option.

Redirecting to other host names

When you use curl you may provide credentials like user name and password for a particular site, but since an HTTP redirect might move away to a different host curl limits what it sends away to other hosts than the original within the same transfer.

So if you want the credentials to also get sent to the following host names even though they are not the same as the original—presumably because you trust them and know that there is no harm in doing that—you can tell curl that it is fine to do so by using the `--location-trusted` option.

Non-HTTP redirects

Browsers support more ways to do redirects that sometimes make life complicated to a curl user as these methods are not supported or recognized by curl.

HTML redirects

If the above was not enough, the web world also provides a method to redirect browsers by plain HTML. See the example `<meta>` tag below. This is somewhat complicated with curl since curl never parses HTML and thus has no knowledge of these kinds of redirects.

```
1 <meta http-equiv="refresh" content="0; url=http://example.com/">
```

JavaScript redirects

The modern web is full of JavaScript and as you know, JavaScript is a language and a full runtime that allows code to execute in the browser when visiting websites.

JavaScript also provides means for it to instruct the browser to move on to another site—a redirect, if you will.

Modify the HTTP request

As described earlier, each HTTP transfer starts with curl sending an HTTP request. That request consists of a request line and a number of request headers, and this chapter details how you can modify all of those.

- [Request method](#)
- [Request target](#)
- [Fragment](#)
- [Customize headers](#)
- [Referer](#)
- [User-agent](#)

Of course, changing the [HTTP version](#) is another way to alter the request.

Request method

The first line of an HTTP request includes the *method* - sometimes also referred to as the verb. When doing a simple GET request as this command line would do:

```
1 curl http://example.com/file
```

...the initial request line looks like this:

```
1 GET /file HTTP/1.1
```

You can tell curl to change the method into something else by using the `-X` or `--request` command-line options followed by the actual method name. You can, for example, send a `DELETE` instead of `GET` like this:

```
1 curl http://example.com/file -X DELETE
```

This command-line option only changes the text in the outgoing request, it does not change any behavior. This is particularly important if you, for example, ask curl to send a HEAD with `-X`, as HEAD is specified to send all the headers a GET response would get but *never* send a response body, even if the headers otherwise imply that one would come. So, adding `-X HEAD` to a command line that would otherwise do a GET will cause curl to hang, waiting for a response body that will not come.

When asking curl to perform HTTP transfers, it will pick the correct method based on the option so you should only rarely have to explicitly ask for it with `-X`. It should also be noted that when curl follows redirects like asked to with `-L`, the request method set with `-X` will be sent even on the subsequent redirects.

Request target

When given an input URL such as `http://example.com/file`, the path section of the URL gets extracted and is turned into `/file` in the HTTP request line. That item in the protocol is called the *request target* in HTTP. That is the resource this request will interact with. Normally this request target is extracted from the URL and then used in the request and as a user you do not need to think about it.

In some rare circumstances, user may want to go creative and change this request target in ways that the URL does not really allow. For example, the HTTP OPTIONS method has a specially define request target for magic that concerns *the server* and not a specific path, and it uses `*` for that. Yes, a single asterisk. There is no way to specify a URL for this, so if you want to pass a single asterisk in the request target to a server, like for OPTIONS, you have to do it like this:

```
1 curl -X OPTIONS --request-target "*" http://example.com/
```

That example command line makes the first line of the outgoing HTTP request to look like this:

```
1 OPTIONS * HTTP/1.1
```

--path-as-is

The path part of the URL is the part that starts with the first slash after the host name and ends either at the end of the URL or at a '?' or '#' (roughly speaking).

If you include substrings including `/.../` or `./` in the path, curl will automatically squash them before the path is sent to the server, as is dictated by standards and how such strings tend to work in local file systems. The `/.../` sequence will remove the previous section so that `/hello/sir/.../` ends up just `/hello/` and `./` is simply removed so that `/hello./sir/` becomes `/hello/sir/`.

To prevent curl from squashing those magic sequences before they are sent to the server and thus allow them through, the `--path-as-is` option exists.

Lame attempt to trick the server to deliver its `/etc/passwd` file:

```
1 curl --path-as-is https://example.com/.../etc/passwd
```

Fragment

A URL may contain an "anchor", also known as a fragment, which is written with a pound sign and string at the end of the URL. Like for example `http://example.com/foo.html#here-it-is`. That fragment

part, everything from the pound/hash sign to the end of the URL, is only intended for local use and will not

Customize headers

In an HTTP request, after the initial request-line, there will typically follow a number of request headers. That is a set of `name: value` pairs that ends with a blank line that separates the headers from the following request body (that sometimes is empty).

curl will by default and on its own account pass a few headers in requests, like for example `Host:` , `Accept:` , `User-Agent:` and a few others that may depend on what the user asks curl to do.

All headers set by curl itself can be overridden, replaced if you will, by the user. You just then tell curl's `-H` or `--header` the new header to use and it will then replace the internal one if the header field matches one of those headers, or it will add the specified header to the list of headers to send in the request.

To change the `Host:` header, do this:

```
1 curl -H "Host: test.example" http://example.com/
```

To add a `Elevator: floor-9` header, do this:

```
1 curl -H "Elevator: floor-9" http://example.com/
```

If you just want to delete an internally generated header, just give it to curl without a value, just nothing on the right side of the colon.

To switch off the `User-Agent:` header, do this:

```
1 curl -H "User-Agent:" http://example.com/
```

Finally, if you then truly want to add a header with no contents on the right side of the colon (which is a rare thing), the magic marker for that is to instead end the header field name with a *semicolon*. Like this:

```
1 curl -H "Empty;" http://example.com
```

Referer

When a user clicks on a link on a web page and the browser takes the user away to the next URL, it will send the new URL a `Referer:` header in the new request telling it where it came from. That is the referer header. The `Referer:` is misspelled but that is how it is supposed to be!

With curl you set the referer header with `-e` or `--referer` , like this:

```
1 curl --referer http://comes-from.example.com https://www.example.com/
```

User-agent

The User-Agent is a header that each client can set in the request to inform the server which user-agent it is. Sometimes servers will look at this header and determine how to act based on its contents.

The default header value is 'curl/[version]', as in `User-Agent: curl/7.54.1` for curl version 7.54.1.

You can set any value you like, using the option `-A` or `--user-agent` plus the string to use or, as it's just a header, `-H "User-Agent: foobar/2000"`.

As comparison, a test version of Firefox on a Linux machine once sent this User-Agent header:

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101
Firefox/58.0
```

HTTP PUT

The difference between a PUT and a POST is subtle. They are virtually identical transmissions except for the different method strings. Where POST is meant to pass on data to a remote resource, PUT is supposed to be the new version of that resource.

In that aspect, PUT is similar to good old standard file upload found in other protocols. You upload a new version of the resource with PUT. The URL identifies the resource and you point out the local file to put there:

```
1 curl -T localfile http://example.com/new/resource/file
```

...so `-T` will imply a PUT and tell curl which file to send off. But the similarities between POST and PUT also allows you to send a PUT with a string by using the regular curl POST mechanism using `-d` but asking for it to use a PUT instead:

```
1 curl -d "data to PUT" -X PUT http://example.com/new/resource/file
```

Cookies

HTTP cookies are key/value pairs that a client stores on the behalf of a server. They are sent back in subsequent requests to allow clients to keep state between requests. Remember that the HTTP protocol itself has no state but instead the client has to resend all data in subsequent requests that it wants the server to be aware of.

Cookies are set by the server with the `Set-Cookie:` header and with each cookie the server sends a bunch of extra properties that need to match for the client to send the cookie back. Like domain name and path and perhaps most important for how long the cookie should live on.

The expiry of a cookie is either set to a fixed time in the future (or to live a number of seconds) or it gets no expiry at all. A cookie without an expire time is called a "session cookie" and is meant to live during the "session" but not longer. A session in this aspect is typically thought to be the life time of the browser used to view a site. When you close the browser, you end your session. Doing HTTP operations with a command-line client that supports cookies begs the question of when a session really ends...

Cookie engine

The general concept of curl only doing the bare minimum unless you tell it differently makes it not acknowledge cookies by default. You need to switch on "the cookie engine" to make curl keep track of cookies it receives and then subsequently send them out on requests that have matching cookies.

You enable the cookie engine by asking curl to read or write cookies. If you tell curl to read cookies from a non-existing file, you will only switch on the engine but start off with an empty internal cookie store:

```
1 curl -b non-existing http://example.com
```

Just switching on the cookie engine, getting a single resource and then quitting would be pointless as curl would have no chance to actually send any cookies it received. Assuming the site in this example would set cookies and then do a redirect we would do:

```
1 curl -L -b non-existing http://example.com
```

Reading cookies from file

Starting off with a blank cookie store may not be desirable. Why not start off with cookies you stored in a previous fetch or that you otherwise acquired? The file format curl uses for cookies is called the Netscape cookie format because it was once the file format used by browsers and then you could easily tell curl to use the browser's cookies!

As a convenience, curl also supports a cookie file being a set of HTTP headers that set cookies. It's an inferior format but may be the only thing you have.

Tell curl which file to read the initial cookies from:

```
1 curl -L -b cookies.txt http://example.com
```

Remember that this only *reads* from the file. If the server would update the cookies in its response, curl would update that cookie in its in-memory store but then eventually throw them all away when it exits and a

subsequent invocation of the same input file would use the original cookie contents again.

Writing cookies to file

The place where cookies are stored is sometimes referred to as the "cookie jar". When you enable the cookie engine in curl and it has received cookies, you can instruct curl to write down all its known cookies to a file, the cookie jar, before it exits. It is important to remember that curl only updates the output cookie jar on exit and not during its lifetime, no matter how long the handling of the given input takes.

You point out the cookie jar output with `-c`:

```
1 curl -c cookie-jar.txt http://example.com
```

`-c` is the instruction to *write* cookies to a file, `-b` is the instruction to *read* cookies from a file. Oftentimes you want both.

When curl writes cookies to this file, it will save all known cookies including those that are session cookies (without a given lifetime). curl itself has no notion of a session and it does not know when a session ends so it will not flush session cookies unless you tell it to.

New cookie session

Instead of telling curl when a session ends, curl features an option that lets the user decide when a new session *begins*.

A new cookie session means that all the old session cookies will be thrown away. It is the equivalent of closing a browser and starting it up again.

Tell curl a new cookie session starts by using `-j`, `--junk-session-cookies`:

```
1 curl -j -b cookies.txt http://example.com/
```

Cookie file format

Netscape once created a file format for storing cookies on disk so that they would survive browser restarts. curl adopted that file format to allow sharing the cookies with browsers, only to soon watch the browsers move away from that format. Modern browsers no longer use it, while curl still does.

The Netscape cookie file format stores one cookie per physical line in the file with a bunch of associated meta data, each field separated with TAB. That file is called the cookiejar in curl terminology.

When libcurl saves a cookiejar, it creates a file header of its own in which there is a URL mention that will link to the web version of this document.

File format

The cookie file format is text based and stores one cookie per line. Lines that start with `#` are treated as comments.

Each line that specifies a single cookie consists of seven text fields separated with TAB characters (ASCII octet 9). A valid line must end with a newline character.

Fields in the file

Field number, what type and example data and the meaning of it:

1. string `example.com` - the domain name
2. boolean `FALSE` - include subdomains
3. string `/foobar/` - path
4. boolean `TRUE` - send/receive over HTTPS only
5. number `1462299217` - expires at - seconds since Jan 1st 1970, or 0
6. string `person` - name of the cookie
7. string `daniel` - value of the cookie

HTTP/2

curl supports HTTP/2 for both HTTP:// and HTTPS:// URLs assuming that curl was built with the proper prerequisites. It will even default to using HTTP/2 when given an HTTPS URL since doing so implies no penalty and when curl is used with sites that do not support HTTP/2 the request will instead negotiate HTTP/1.1.

With HTTP:// URLs however, the upgrade to HTTP/2 is done with an `Upgrade:` header that may cause an extra round-trip and perhaps even more troublesome, a sizable share of old servers will return a 400 response when seeing such a header.

It should also be noted that some (most?) servers that support HTTP/2 for HTTP:// (which in itself is not all servers) will not acknowledge the `Upgrade:` header on POST, for example.

To ask a server to use HTTP/2, just:

```
1 curl --http2 http://example.com/
```

If your curl does not support HTTP/2, that command line will return an error saying so. Running `curl -V`

will show if your version of curl supports it.

If you by some chance already know that your server speaks HTTP/2 (for example, within your own controlled environment where you know exactly what runs in your machines) you can shortcut the HTTP/2 "negotiation" with `--http2-prior-knowledge`.

Multiplexing

One of the primary features in the HTTP/2 protocol is the ability to multiplex several logical stream over the same physical connection. When using the curl command-line tool, you cannot take advantage of that cool feature since curl is doing all its network requests in a strictly serial manner, one after the next, with the second only ever starting once the previous one has ended.

Preferably, a future curl version will be enhanced to allow the use of this feature.

Alternative Services

[RFC 7838](#) defines an HTTP header which lets a server tell a client that there is one or more *alternatives* for that server at "another place" with the use of the `Alt-Svc:` response header.

The *alternatives* the server suggests can include a server running on another port on the same host, on another completely different host name and it can even perhaps offer the service over another protocol.

Enable

To make curl consider offered alternatives, tell curl to use a specific alt-svc cache file like this:

```
1 curl --alt-svc altcache.txt https://example.com/
```

then curl will load existing alternative service entries from the file at start-up and consider those when doing HTTP requests, and if the servers sends new or updated `Alt-Svc:` headers, curl will store those in the cache at exit.

The alt-svc cache

The alt-svc cache is similar to a cookie jar. It is a text based file that stores one alternative per line and each entry also has an expiry time for which duration that particular alternative is valid.

HTTPS only

`Alt-Svc:` is only trusted and parsed from servers when connected to over HTTPS.

HTTP/3

The use of `Alt-Svc:` headers is as of August 2019 the only defined way to bootstrap a client and server into using HTTP/3. The server then hints to the client over HTTP/1 or HTTP/2 that it also is available over HTTP/3 and then curl can connect to it using HTTP/3 in the subsequent request if the alt-svc cache says so.

HTTP/3

This feature is marked **experimental** as of this time and needs to be explicitly enabled in the build to function.

Drafts, not standard

As of January 2022, **the HTTP/3 protocol has not yet been finalized**. Everything and everyone that speaks HTTP/3 at this point does it with the knowledge and awareness that it might change going forward.

QUIC

HTTP/3 is the HTTP version that is designed to communicate over QUIC. QUIC can for all particular purposes to be considered as a TCP+TLS replacement.

All requests that do HTTP/3 will therefore not use TCP. They will use QUIC. QUIC is a reliable transport protocol built over UDP. HTTP/3 implies use of QUIC.

HTTPS only

HTTP/3 is performed over QUIC which is always using TLS, so HTTP/3 is by definition always encrypted and secure. Therefore, curl only uses HTTP/3 for `HTTPS://` URLs.

Enable

As a shortcut straight to HTTP/3, to make curl attempt a QUIC connect directly to the given host name and port number, use `--http3`. Like this:

```
1 curl --http3 https://example.com/
```

Normally, without the `--http3` option, an `HTTPS://` URL implies that a client needs to connect to it using TCP (and TLS).

Alt-svc:

The [alt-svc](#) method of changing to HTTP/3 is the official way to bootstrap into HTTP/3 for a server.

Note that you need that feature built-in and that it does not switch to HTTP/3 for the *current* request unless the alt-svc cache is already populated, but it will rather store the info for use in the *next* request to the host.

When QUIC is denied

A certain amount of QUIC connection attempts will fail, partly because many networks and hosts block or throttle the traffic.

Currently, curl features no fall-back logic but if an HTTP/3 (or QUIC rather) connection fails it will be reported as, yes, a failure.

Web browsers will upgrade to HTTP/3 in the background and only switch over once they know it works, which is a smoother way that does not break things for users as much.

Future curl versions will likely offer better fall-back and error handling for this.

HSTS

HTTP Strict Transport Security, HSTS, is a protocol mechanism that helps to protect HTTPS servers against man-in-the-middle attacks such as protocol downgrade attacks and cookie hijacking. It allows an HTTPS server to declare that clients should automatically interact with this host name using only HTTPS connections going forward - and explicitly not use clear text protocols with it.

HSTS cache

The HSTS status for a certain server name is set in a response header and has an expire time. The status for every HSTS host name needs to be saved in a file for curl to pick it up and to update the status and expire time.

Invoke curl and tell it which file to use as a hsts cache:

```
1 curl --hsts hsts.txt https://example.com
```

curl will only update the hsts info if the header is read over a secure transfer, so not when done over a clear text protocol.

Use HSTS to update insecure protocols

If the cache file now contains an entry for the given host name, it will automatically switch over to a secure protocol even if you try to connect to it with an insecure one:

```
1 curl --hsts hsts.txt http://example.com
```

HTTP cheat sheet

[online here](#)

Verbose	Hide progress	extra info	Write output	Timeout
-v	-s	-w "format"	-O	-m
--trace-ascii			-o	
POST	multipart	PUT	HEAD	custom
-d "string"	-F name=value	-T	-I	-X "METHOD"
-d @file	-F name=@file			
Basic auth	read cookies	write cookies	send cookies	user-agent
-u user:password	-b	-c	-b "c=1; d=2"	-A "string"
Use proxy	Headers, add/remove	follow redirs	gzip	insecure
-x	-H "name: value"	-L	--compressed	-k
	-H "name:"			

Scripting browser-like tasks

curl can do almost every HTTP operation and transfer your favorite browser can. It can actually do a lot more than so as well, but in this chapter we will focus on the fact that you can use curl to reproduce, or script, what you would otherwise have to do manually with a browser.

Here are some tricks and advice on how to proceed when doing this.

Figure out what the browser does

This is really a necessary first step. Second-guessing what it does risks having you chase down the wrong problem rat-hole. The scientific approach to this problem pretty much requires that you first understand what the browser does.

To learn what the browser does to perform a certain task, you can either read the HTML pages that you operate on and with a deep enough knowledge you can see what a browser would do to accomplish it and then start trying to do the same with curl.

The slightly more effective way, that also works even for the cases when the page is shock-full of obfuscated JavaScript, is to run the browser and monitor what HTTP operations it performs.

The [Copy as curl](#) section describes how you can record a browser's request and easily convert that to a curl command line.

Those copied curl command lines are often not good enough though since they tend to copy *exactly* that request, while you probably want to be a bad bit more dynamic so that you can reproduce the same operation and not just resend the verbatim request.

Cookies

A lot of the web today works with a user name and password login prompt somewhere. In many cases you even logged in a while ago with your browser but it has kept the state and keeps you logged in.

The logged-in state is almost always done by using [cookies](#). A common operation would be to first login and save the returned cookies in a file, and then let the site update the cookies in the subsequent command lines when you traverse the site with curl.

Web logins and sessions

The site at <https://example.com/> features a login prompt. The login on the web site is an HTML form to which you send a [HTTP POST](#) to. Save the response cookies and the response (HTML) output.

Although the login page is visible (if you would use a browser) on <https://example.com/>, the HTML form tag on that page informs you about which exact URL to send the POST to, using the `action` parameter.

In our imaginary case, the form tag looks like this:

```
1 <form action="login.cgi" method="POST">
2   <input type="text" name="user">
3   <input type="password" name="secret">
4   <input type="hidden" name="id" value="bc76">
5 </form>
```

There are three fields of importance. **text**, **secret** and **id**. The last one, the id, is marked `hidden` which means that it will not show up in the browser and it is not a field that a user fills in. It is generated by the site itself, and for your curl login to succeed, you need extract that value and use that in your POST submission together with the rest of the data.

Send correct contents to the fields to the correct destination URL:

```
1 curl -d user=daniel -d secret=qwerty -d id=bc76 https://example.com/login.cgi -o out
```

Many login pages even send you a session cookie already when presenting the login, and since you often need to extract the hidden fields from the `<form>` tag anyway, you could do something like this first:

```
1 curl -c cookies https://example.com/ -o loginform
```

You would often need an HTML parser or some scripting language to extract the id field from there and then you can proceed and login as mentioned above, but with the added cookie loading (I am splitting the line into two lines to make it more readable):

```
1 curl -d user=daniel -d secret=qwerty -d id=bc76 https://example.com/login.cgi \
2 -b cookies -c cookies -o out
```

You can see that it uses both `-b` for reading cookies from the file and `-c` to store cookies again, for when the server sends back updated cookies.

Always, *always*, add `-v` to the command lines when working out the details. See also the [verbose](#) section for more details on that.

Redirects

It is common for servers to use [redirects](#) when responding to a login POST. It is so common I would probably say it is rare that it is not solved with a redirect.

You then just need to remember that curl does not follow redirects automatically. You need to instruct it to do this by adding the `-L` command line option. Adding that to the previous command line then makes the full one look like:

```
1 curl -d user=daniel -d secret=qwerty -d id=bc76 https://example.com/login.cgi \
2 -b cookies -c cookies -L -o out
```

Post-login

In the above example command lines, we save the login response output in a file named 'out' and in your script you should probably verify that it contains some text or something that confirms that the login is successful.

Once successfully logged in, get the files or perform the HTTP operations you need and remember to keep using both `-b` and `-c` on the command lines to use and update the cookies.

Referer

Some sites will check that the `Referer:` is actually identifying the legitimate parent URL when you request something or when you login or similar. You can then inform the server from which URL you arrived by using `-e https://example.com/` etc. Appending that to the previous login attempt then makes it:

```
1 curl -d user=daniel -d secret=qwerty -d id=bc76 https://example.com/login.cgi \
2 -b cookies -c cookies -L -e "https://example.com/" -o out
```

TLS fingerprinting

Anti-bot detections nowadays use TLS fingerprinting to figure out whether a request is coming from a browser. Curl's fingerprint can vary depending on your environment and most likely is different from those of browsers. Curl's CLI does not have options to change all the various parts of the fingerprint, however an advanced user can customize the fingerprint through the use of libcurl and by compiling curl from source themselves.

FTP with curl

FTP, the File Transfer Protocol, is probably the oldest network protocol that curl supports—it was created in the early 1970s. The official spec that still is the go-to documentation is [RFC 959](#), from 1985, published well over a decade before the first curl release.

FTP was created in a different era of the Internet and computers and as such it works a little bit differently than most other protocols. These differences can often be ignored and things will just work, but they are also important to know at times when things do not run as planned.

Ping-pong

The FTP protocol is a command and response protocol; the client sends a command and the server responds. If you use curl's `-v` option you will get to see all the commands and responses during a transfer.

For an ordinary transfer, there are something like 5 to 8 commands necessary to send and as many responses to wait for and read. Perhaps needlessly to say, if the server is in a remote location there will be a lot of time waiting for the ping pong to go through before the actual file transfer can be set up and get started. For small files, the initial commands can take longer time than the actual data transfer.

Transfer mode

When an FTP client is about to transfer data, it specifies to the server which "transfer mode" it would like the upcoming transfer to use. The two transfer modes curl supports are 'ASCII' and 'BINARY'. Ascii is for text and usually means that the server will send the files with converted newlines while binary means sending the data unaltered and assuming the file is not text.

curl will default to binary transfer mode for FTP, and you ask for ascii mode instead with `-B`, `--use-ascii` or by making sure the URL ends with `;type=A`.

Authentication

FTP is one of the protocols you normally do not access without a user name and password. It just happens that for systems that allow "anonymous" FTP access you can login with pretty much any name and password you like. When curl is used on an FTP URL to do transfer without any given user name or password, it uses the name `anonymous` with the password `ftp@example.com`.

If you want to provide another user name and password, you can pass them on to curl either with the `-u`, `--user` option or embed the info in the URL:

```
1 curl --user daniel:secret ftp://example.com/download
2
3 curl ftp://daniel:secret@example.com/download
```

FTP Directory listing

You can list a remote FTP directory with curl by making sure the URL ends with a trailing slash. If the URL ends with a slash, curl will presume that it is a directory you want to list. If it is not actually a directory, you will most likely instead get an error.

```
1 curl ftp://ftp.example.com/directory/
```

With FTP there is no standard syntax for the directory output that is returned for this sort of command that uses the standard FTP command `LIST`. The listing is usually humanly readable and perfectly understandable but you will see that different servers will return the listing in slightly different ways.

One way to get just a listing of all the names in a directory and thus to avoid the special formatting of the regular directory listings is to tell curl to `--list-only` (or just `-l`). curl then issues the `NLST` FTP command instead:

```
1 curl --list-only ftp://ftp.example.com/directory/
```

NLST has its own quirks though, as some FTP servers list only actual *files* in their response to NLST; they do not include directories and symbolic links!

Uploading with FTP

To upload to an FTP server, you specify the entire target file path and name in the URL, and you specify the local file name to upload with `-T`, `--upload-file`. Optionally, you end the target URL with a slash and then the file component from the local path will be appended by curl and used as the remote file name.

Like:

```
1 curl -T localfile ftp://ftp.example.com/dir/path/remote-file
```

or to use the local file name as the remote name:

```
1 curl -T localfile ftp://ftp.example.com/dir/path/
```

curl also supports [globbing](#) in the `-T` argument so you can opt to easily upload a range or a series of files:

```
1 curl -T 'image[1-99].jpg' ftp://ftp.example.com/upload/
```

or

```
1 curl -T '{Huey,Dewey,Louie}.jpg' ftp://ftp.example.com/nephews/
```

Custom FTP commands

The FTP protocol offers a wide variety of different commands that allow the client to perform actions, other than the plain file transfers that curl is focused on.

A curl user can pass on such extra (custom) commands to the server as a step in the file transfer sequence. curl even offers to have those commands run at different points in the process.

Quote

In the old days the standard old ftp client had a command called *quote*. It was used to send commands verbatim to the server. curl uses the same name for virtually the same functionality: send the specified command verbatim to the server. Actually one or more commands. `-Q` or `--quote`.

To know what commands that are available and possible to send to a server, you need to know a little about the FTP protocol, and possibly read up a bit on RFC 959 on the details.

To send a simple `NOOP` to the server (which does nothing!) **before** the transfer starts, provide it to curl like this:

```
1 curl -Q NOOP ftp://example.com/file
```

To instead send the same command immediately **after** the transfer, prefix the FTP command with a dash:

```
1 curl -Q -NOOP ftp://example.com/file
```

The third "position in time" that curl offers to send the commands, is after curl has changed the working directory, just **before the commands** that kick off the transfer are sent. To send command then, prefix the command with a '+' (plus).

A series of commands

You can in fact send commands in all three different times by using multiple `-Q` on the command line. You can also send multiple commands in the same position by using more `-Q` options.

By default, if any of the given commands returns an error from the server, curl will stop its operations, abort the transfer (if it happens before transfer has started) and not send any more of the custom commands.

Example, rename a file then do a transfer:

```
1 curl -Q "RNFR original" -Q "RNTO newname" ftp://example.com/newname
```

Fallible commands

You can opt to send individual quote commands that are allowed to fail, to get an error returned from the server without causing everything to stop.

You make the command "fallible" by prefixing it with an asterisk (`*`). For example, send a delete (`DELE`)

after a transfer and allow it to fail:

```
1 curl -Q "-*DELE file" ftp://example.com/moo
```

Two connections

FTP uses two TCP connections! The first connection is setup by the client when it connects to an FTP server, and is called the *control connection*. As the initial connection, it gets to handle authentication and changing to the correct directory on the remote server, etc. When the client then is ready to transfer a file, a second TCP connection is established and the data is transferred over that.

This setting up of a second connection causes nuisances and headaches for several reasons.

Active connections

The client can opt to ask the server to connect to the client to set it up, a so-called "active" connection. This is done with the PORT or EPRT commands. Allowing a remote host to connect back to a client on a port that the client opens up requires that there is no firewall or other network appliance in between that refuses that to go through and that is far from always the case. You ask for an active transfer using `curl -P [arg]` (also known as `--ftp-port` in long form) and while the option allows you to specify exactly which address to use, just setting the same as you come from is almost always the correct choice and you do that with `-P -`, like this way to ask for a file:

```
1 curl -P - ftp://example.com/foobar.txt
```

You can also explicitly ask curl to not use EPRT (which is a slightly newer command than PORT) with the `-no-eprt` command-line option.

Passive connections

Curl defaults to asking for a "passive" connection, which means it sends a PASV or EPSV command to the server and then the server opens up a new port for the second connection that then curl connects to. Outgoing connections to a new port are generally easier and less restricted for end users and clients, but it then requires that the network in the server's end allows it.

Passive connections are enabled by default, but if you have switched on active before, you can switch back to passive with `--ftp-pasv`.

You can also explicitly ask curl not to use EPSV (which is a slightly newer command than PASV) with the `-no-epsv` command-line option.

Sometimes the server is running a funky setup so that when curl issues the PASV command and the server responds with an IP address for curl to connect to, that address is wrong and then curl fails to setup the data connection. For this (rare) situation, you can ask curl to ignore the IP address mentioned in the PASV response (`--ftp-skip-pasv-ip`) and instead use the same IP address it has for the control connection even for the second connection.

Firewall issues

Using either active or passive transfers, any existing firewalls in the network path pretty much have to have stateful inspection of the FTP traffic to figure out the new port to open that up and accept it for the second connection.

Directory traversing

When doing FTP commands to traverse the remote file system, there are a few different ways curl can proceed to reach the target file, the file the user wants to transfer.

multicwd

curl can do one change directory (CWD) command for every individual directory down the file tree hierarchy. If the full path is `one/two/three/file.txt`, that method means doing three CWD commands before asking for the `file.txt` file to get transferred. This method thus creates quite a large number of commands if the path is many levels deep. This method is mandated by an early spec (RFC 1738) and is how curl acts by default:

```
1 curl --ftp-method multicwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

CWD one < 250 OK. Current directory is /one
CWD two < 250 OK. Current directory is /one/two
CWD three < 250 OK. Current directory is /one/two/three
RETR file.txt

nocwd

The opposite to doing one CWD for each directory part is to not change the directory at all. This method asks the server using the entire path at once and is thus fast. Occasionally servers have a problem with this and it is not purely standards-compliant:

```
1 curl --ftp-method nocwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

```
RETR one/two/three/file.txt
```

single cwd

This is the in-between the other two FTP methods. This makes a single `CWD` command to the target directory and then it asks for the given file:

```
1 curl --ftp-method single cwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

```
CWD one/two/three < 250 OK. Current directory is /one/two/three RETR file.txt
```

FTPS

FTPS is FTP secure by TLS. It negotiates fully secured TLS connections where plain FTP uses clear text unsafe connections.

There are two ways to do FTPS with curl. The *implicit* way and the *explicit* way. (These terms originate from the FTPS RFC). Usually the server you work with dictates which of these methods you can and shall use against it.

Implicit FTPS

The *implicit* way is when you use `ftps://` in your URL. This makes curl connect to the host and do a TLS handshake immediately, without doing anything in the clear. If no port number is specified in the URL, curl will use port 990 for this. This is usually not how FTPS is done.

Explicit FTPS

The *explicit* way of doing FTPS is to keep using an `ftp://` URL, but instruct curl to upgrade the connection into a secure one using the `STARTTLS` FTP command.

You can tell curl to either *attempt* an upgrade and continue as usual if the upgrade fails with `--ssl`, or you

can force curl to either upgrade or fail the whole thing hard if the upgrade fails by using `--ssl-reqd`. We strongly recommend using the latter so that you can be sure that a secure transfer is done - if any

Common FTPS problems

The single most common problem with FTPS comes from the fact that the FTP protocol (that FTP still mostly uses) uses a separate connection setup for the data transfer. This connection is done to another port and when FTP is done over clear text (non-FTPS), firewalls and network inspectors etc can figure out that this is FTP in progress and they can adapt things and rules for the new connection.

When everything is encrypted, with FTPS, firewalls cannot see what is going on and no outside can dynamically adapt network rules or permission based on this.

Using libcurl

The engine in the curl command-line tool is libcurl. libcurl is also the engine in thousands of tools, services and applications out there today, performing their Internet data transfers.

C API

libcurl is a library of functions that are provided with a C API, for applications written in C. You can easily use it from C++ too, with only a few considerations (see [libcurl for C++ programmers](#)). For other languages, there exist "bindings" that work as intermediate layers between libcurl the library and corresponding functions for the particular language you like.

Transfer oriented

We have designed libcurl to be transfer oriented usually without forcing users to be protocol experts or in fact know much at all about networking or the protocols involved. You setup a transfer with as many details and specific information as you can and want, and then you tell libcurl to perform that transfer.

That said, networking and protocols are areas with lots of pitfalls and special cases so the more you know about these things, the more you will be able to understand about libcurl's options and ways of working. Not to mention, such knowledge is invaluable when you are debugging and need to understand what to do next when things do not go as you intended.

The most basic libcurl using application can be as small as just a couple of lines of code, but most applications will, of course, need more code than that.

Simple by default more on demand

libcurl generally does the simple and basic transfer by default, and if you want to add more advanced features, you add that by setting the correct options. For example, libcurl does not support HTTP cookies by default but it does once you tell it.

This makes libcurl's behaviors easier to guess and depend on, and also it makes it easier to maintain old behavior and add new features. Only applications that actually ask for and use the new features will get that behavior.

Easy handle

The fundamentals you need to learn with libcurl:

First you create an "easy handle", which is your handle to a transfer, really:

```
1 CURL *easy_handle = curl_easy_init();
```

You then set options in that handle to control the upcoming transfer. This example sets the URL:

```
1 /* set URL to operate on */
2 res = curl_easy_setopt(easy_handle, CURLOPT_URL, "http://example.com/");
```

If `curl_easy_setopt()` returns `CURLE_OK`, we know it stored the option fine.

Creating the easy handle and setting options on it does not make any transfer happen, and usually do not even make much more happen other than libcurl storing your wish to be used later when the transfer actually occurs. Lots of syntax checking and validation of the input may also be postponed, so just because `curl_easy_setopt` did not complain, it does not mean that the input was correct and valid; you may get an error returned later.

Read more on [easy options](#) in its separate section.

When you are done setting options to your easy handle, you can fire off the actual transfer.

The actual performing of the transfer can be done using different methods and function calls, depending on what kind of behavior you want in your application and how libcurl is best integrated into your architecture. Those are further described later in this chapter.

While the transfer is ongoing, libcurl calls your specified functions—known as [callbacks](#) — to deliver data, to read data and to do a variety of things.

After the transfer has completed, you can figure out if it succeeded or not and you can extract statistics and other information that libcurl gathered during the transfer from the easy handle. See [Post transfer information](#).

Reuse

Easy handles are meant and designed to be reused. When you have done a single transfer with the easy handle, you can immediately use it again for your next transfer. There are lots of gains to be had by this.

All options are "sticky". If you make a second transfer with the same handle, the same options are used. They remain set in the handle until you change them again, or call `curl_easy_reset()` on the handle.

Reset

By calling `curl_easy_reset()`, all options for the given easy handle will be reset and restored to their default values. The same values the options had when the handle was initially created. The caches remain intact.

Duplicate

An easy handle, with all its currently set options, can be duplicated using `curl_easy_duphandle()`. It returns a copy of the handle passed in to it.

The caches and other state information will not be carried over.

Drive transfers

libcurl provides three different ways to perform the transfer. Which way to use in your case is entirely up to you and what you need.

1. The 'easy' interface lets you do a single transfer in a synchronous fashion. libcurl will do the entire transfer and return control back to your application when it is completed—successful or failed.
2. The 'multi' interface is for when you want to do more than one transfer at the same time, or you just want a non-blocking transfer.
3. The 'multi_socket' interface is a slight variation of the regular multi one, but is event-based and is really the suggested API to use if you intend to scale up the number of simultaneous transfers to hundreds or thousands or so.

Let's look at each one a little closer...

Drive with easy

The name 'easy' was picked simply because this is really the easy way to use libcurl, and with easy, of course, comes a few limitations. Like, for example, that it can only do one transfer at a time and that it does

the entire transfer in a single function call and returns once it is completed:

```
1 res = curl_easy_perform( easy_handle );
```

If the server is slow, if the transfer is large or if you have some unpleasant timeouts in the network or similar, this function call can end up taking a long time. You can, of course, set timeouts to not allow it to spend more than N seconds, but it could still mean a substantial amount of time depending on the particular conditions.

If you want your application to do something else while libcurl is transferring with the easy interface, you need to use multiple threads. If you want to do multiple simultaneous transfers when using the easy interface, you need to perform each of the transfers in its own thread.

Drive with multi

The name 'multi' is for multiple, as in multiple parallel transfers, all done in the same single thread. The multi API is non-blocking so it can also make sense to use it for single transfers.

The transfer is still set in an "easy" `CURL *` handle as described [above](#), but with the multi interface you also need a multi `CURLM *` handle created and use that to drive all the individual transfers. The multi handle can "hold" one or many easy handles:

```
1 CURLM *multi_handle = curl_multi_init();
```

A multi handle can also get certain options set, which you do with `curl_multi_setopt()`, but in the simplest case you might not have anything to set there.

To drive a multi interface transfer, you first need to add all the individual easy handles that should be transferred to the multi handle. You can add them to the multi handle at any point and you can remove them again whenever you like. Removing an easy handle from a multi handle will, of course, remove the association and that particular transfer would stop immediately.

Adding an easy handle to the multi handle is easy:

```
1 curl_multi_add_handle( multi_handle, easy_handle );
```

Removing one is just as easily done:

```
1 curl_multi_remove_handle( multi_handle, easy_handle );
```

Having added the easy handles representing the transfers you want to perform, you write the transfer loop. With the multi interface, you do the looping so you can ask libcurl for a set of file descriptors and a timeout value and do the `select()` call yourself, or you can use the slightly simplified version which does that for us, with `curl_multi_wait`. The simplest loop could look like this: (*note that a real application would check return codes*)

```

1 int transfers_running;
2 do {
3     curl_multi_wait ( multi_handle, NULL, 0, 1000, NULL);
4     curl_multi_perform ( multi_handle, &transfers_running );
5 } while (transfers_running);

```

The fourth argument to `curl_multi_wait`, set to 1000 in the example above, is a timeout in milliseconds. It is the longest time the function will wait for any activity before it returns anyway. You do not want to lock up for too long before calling `curl_multi_perform` again as there are timeouts, progress callbacks and more that may lose precision if you do so.

To instead do `select()` on our own, we extract the file descriptors and timeout value from libcurl like this (*note that a real application would check return codes*):

```

1 int transfers_running;
2 do {
3     fd_set fdread;
4     fd_set fdwrite;
5     fd_set fdexcep;
6     int maxfd = -1;
7     long timeout;
8
9     /* extract timeout value */
10    curl_multi_timeout(multi_handle, &timeout);
11    if (timeout < 0)
12        timeout = 1000;
13
14    /* convert to struct usable by select */
15    timeout.tv_sec = timeout / 1000;
16    timeout.tv_usec = (timeout % 1000) * 1000;
17
18    FD_ZERO(&fdread);
19    FD_ZERO(&fdwrite);
20    FD_ZERO(&fdexcep);
21
22    /* get file descriptors from the transfers */
23    mc = curl_multi_fdset(multi_handle, &fdread, &fdwrite, &fdexcep, &maxfd);
24
25    if (maxfd == -1) {
26        SHORT_SLEEP;
27    }
28    else
29        select(maxfd+1, &fdread, &fdwrite, &fdexcep, &timeout);
30
31    /* timeout or readable/writable sockets */
32    curl_multi_perform(multi_handle, &transfers_running);
33 } while ( transfers_running );

```

Both these loops let you use one or more file descriptors of your own on which to wait, like if you read from your own sockets or a pipe or similar.

And again, you can add and remove easy handles to the multi handle at any point during the looping. Removing a handle mid-transfer will, of course, abort that transfer.

When is a single transfer done?

As the examples above show, a program can detect when an individual transfer completes by seeing that the `transfers_running` variable decreases.

It can also call `curl_multi_info_read()`, which will return a pointer to a struct (a "message") if a transfer has ended and you can then find out the result of that transfer using that struct.

When you do multiple parallel transfers, more than one transfer can of course complete in the same `curl_multi_perform` invocation and then you might need more than one call to `curl_multi_info_read` to get info about each completed transfer.

Drive with multi_socket

Driving with the "multi_socket" interface

`multi_socket` is the extra spicy version of the regular multi interface and is designed for event-driven applications. Make sure you read the [Drive with multi interface](#) section first.

`multi_socket` supports multiple parallel transfers—all done in the same single thread—and have been used to run several tens of thousands of transfers in a single application. It is usually the API that makes the most sense if you do a large number (>100 or so) of parallel transfers.

Event-driven in this case means that your application uses a system level library or setup that "subscribes" to a number of sockets and it lets your application know when one of those sockets are readable or writable and it tells you exactly which one.

This setup allows clients to scale up the number of simultaneous transfers much higher than with other systems, and still maintain good performance. The "regular" APIs otherwise waste far too much time scanning through lists of all the sockets.

Pick one

There are numerous event based systems to select from out there, and libcurl is completely agnostic to which one you use. libevent, libev and libuv are three popular ones but you can also go directly to your operating system's native solutions such as epoll, kqueue, /dev/poll, pollset or Event Completion.

Many easy handles

Just like with the regular multi interface, you add easy handles to a multi handle with `curl_multi_add_handle()`. One easy handle for each transfer you want to perform.

You can add them at any time while the transfers are running and you can also similarly remove easy

handles at any time using the `curl_multi_remove_handle` call. Typically though, you remove a handle only after its transfer is completed.

multi_socket callbacks

As explained above, this event-based mechanism relies on the application to know which sockets are used by libcurl and what libcurl waits for on those sockets: if it waits for the socket to become readable, writable or both!

It also needs to tell libcurl when its timeout time has expired, as it is control of driving everything libcurl cannot do it itself. So libcurl must tell the application an updated timeout value, too.

socket_callback

libcurl informs the application about socket activity to wait for with a callback called [CURLMOPT_SOCKETFUNCTION](#). Your application needs to implement such a function:

```
1 int socket_callback(CURL *easy,          /* easy handle */
2                      curl_socket_t s, /* socket */
3                      int what,        /* what to wait for */
4                      void *userp,     /* private callback pointer */
5                      void *socketp)   /* private socket pointer */
6 {
7     /* told about the socket 's' */
8 }
9
10 /* set the callback in the multi handle */
11 curl_multi_setopt(multi_handle, CURLMOPT_SOCKETFUNCTION, socket_callback);
```

Using this, libcurl will set and remove sockets your application should monitor. Your application tells the underlying event-based system to wait for the sockets. This callback will be called multiple times if there are multiple sockets to wait for, and it will be called again when the status changes and perhaps you should switch from waiting for a writable socket to instead wait for it to become readable.

When one of the sockets that the application is monitoring on libcurl's behalf registers that it becomes readable or writable, as requested, you tell libcurl about it by calling `curl_multi_socket_action()` and passing in the affected socket and an associated bitmask specifying which socket activity that was registered:

```
1 int running_handles;
2 ret = curl_multi_socket_action(multi_handle,
3                                sockfd, /* the socket with activity */
4                                ev_bitmask, /* the specific activity */
5                                &running_handles);
```

timer_callback

The application is in control and will wait for socket activity. But even without socket activity there will be

things libcurl needs to do. Timeout things, calling the progress callback, starting over a retry or failing a transfer that takes too long, etc. To make that work, the application must also make sure to handle a single-shot timeout that libcurl sets.

libcurl sets the timeout with the timer_callback [CURLMOPT_TIMERFUNCTION](#):

```
1 int timer_callback(multi_handle, /* multi handle */
2                     timeout_ms,    /* milliseconds to wait */
3                     userp)        /* private callback pointer */
4 {
5     /* the new time-out value to wait for is in 'timeout_ms' */
6 }
7
8 /* set the callback in the multi handle */
9 curl_multi_setopt(multi_handle, CURLMOPT_TIMERFUNCTION, timer_callback);
```

There is only one timeout for the application to handle for the entire multi handle, no matter how many individual easy handles that have been added or transfers that are in progress. The timer callback will be updated with the current nearest-in-time period to wait. If libcurl gets called before the timeout expiry time because of socket activity, it may update the timeout value again before it expires.

When the event system of your choice eventually tells you that the timer has expired, you need to tell libcurl about it:

```
1 curl_multi_socket_action(multi, CURL_SOCKET_TIMEOUT, 0, &running);
```

...in many cases, this will make libcurl call the timer_callback again and set a new timeout for the next expiry period.

How to start everything

When you have added one or more easy handles to the multi handle and set the socket and timer callbacks in the multi handle, you are ready to start the transfer.

To kick it all off, you tell libcurl it timed out (because all easy handles start out with a short timeout) which will make libcurl call the callbacks to set things up and from then on you can just let your event system drive:

```
1 /* all easy handles and callbacks are setup */
2
3 curl_multi_socket_action(multi, CURL_SOCKET_TIMEOUT, 0, &running);
4
5 /* now the callbacks should have been called and we have sockets to wait for
6   and possibly a timeout, too. Make the event system do its magic */
7
8 event_base_dispatch(event_base); /* libevent2 has this API */
9
10 /* at this point we have exited the event loop */
```

When is it done?

The 'running_handles' counter returned by `curl_multi_socket_action` holds the number of current transfers not completed. When that number reaches zero, we know there are no transfers going on.

Each time the 'running_handles' counter changes, `curl_multi_info_read()` will return info about the specific transfers that completed.

Connection reuse

libcurl keeps a pool of old connections alive. When one transfer has completed it will keep N connections alive in a "connection pool" (sometimes also called connection cache) so that a subsequent transfer that happens to be able to reuse one of the existing connections can use it instead of creating a new one. Reusing a connection instead of creating a new one offers significant benefits in speed and required resources.

When libcurl is about to make a new connection for the purposes of doing a transfer, it will first check to see if there is an existing connection in the pool that it can reuse instead. The connection re-use check is done before any DNS or other name resolving mechanism is used, so it is purely host name based. If there is an existing live connection to the right host name, a lot of other properties (port number, protocol, etc) are also checked to see that it can be used.

Easy API pool

When you are using the easy API, or, more specifically, `curl_easy_perform()`, libcurl will keep the pool associated with the specific easy handle. Then reusing the same easy handle will ensure it can reuse its connection.

Multi API pool

When you are using the multi API, the connection pool is instead kept associated with the multi handle. This allows you to cleanup and re-create easy handles freely without risking losing the connection pool, and it allows the connection used by one easy handle to get reused by a separate one in a later transfer. Just reuse the multi handle!

Sharing the "connection cache"

Since libcurl 7.57.0, applications can use the [share interface](#) to have otherwise independent transfers share the same connection pool.

Callbacks

Lots of operations within libcurl are controlled with the use of *callbacks*. A callback is a function pointer provided to libcurl that libcurl then calls at some point to get a particular job done.

Each callback has its specific documented purpose and it requires that you write it with the exact function prototype to accept the correct arguments and return the documented return code and return value so that libcurl will perform the way you want it to.

Each callback option also has a companion option that sets the associated "user pointer". This user pointer is a pointer that libcurl does not touch or care about, but just passes on as an argument to the callback. This allows you to, for example, pass in pointers to local data all the way through to your callback function.

Unless explicitly stated in a libcurl function documentation, it is not legal to invoke libcurl functions from within a libcurl callback.

- [Write data](#)
- [Read data](#)
- [Progress information](#)
- [Header data](#)
- [Debug](#)
- [sockopt](#)
- [SSL context](#)
- [Seek and ioctl](#)
- [Network data conversion](#)
- [Opensocket and closesocket](#)
- [SSH key](#)
- [RTSP interleaved data](#)
- [FTP wildcard matching](#)
- [Resolver start](#)
- [Sending trailers](#)
- [HSTS](#)
- [Prereq](#)

Write data

The write callback is set with `CURLOPT_WRITEFUNCTION`:

```
1 curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, write_callback);
```

The `write_callback` function must match this prototype:

```
1 size_t write_callback(char *ptr, size_t size, size_t nmemb, void *userdata);
```

This callback function gets called by libcurl as soon as there is data received that needs to be saved. *ptr* points to the delivered data, and the size of that data is *size* multiplied with *nmemb*.

If this callback is not set, libcurl instead uses 'fwrite' by default.

The write callback will be passed as much data as possible in all invokes, but it must not make any assumptions. It may be one byte, it may be thousands. The maximum amount of body data that will be passed to the write callback is defined in the curl.h header file: CURL_MAX_WRITE_SIZE (the usual default is 16KB). If CURLOPT_HEADER is enabled for this transfer, which makes header data get passed to the write callback, you can get up to CURL_MAX_HTTP_HEADER bytes of header data passed into it. This usually means 100KB.

This function may be called with zero bytes data if the transferred file is empty.

The data passed to this function will not be zero terminated. You cannot, for example, use printf's %s operator to display the contents nor strcpy to copy it.

This callback should return the number of bytes actually taken care of. If that number differs from the number passed to your callback function, it will signal an error condition to the library. This will cause the transfer to get aborted and the libcurl function used will return CURLE_WRITE_ERROR .

The user pointer passed in to the callback in the *userdata* argument is set with CURLOPT_WRITEDATA :

```
1 curl_easy_setopt(handle, CURLOPT_WRITEDATA, custom_pointer);
```

Store in memory

A popular demand is to store the retrieved response in memory, and the callback explained above supports that. When doing this, just be careful as the response can potentially be enormous.

You implement the callback in a manner similar to:

```
1 struct response {
2     char *memory;
3     size_t size;
4 };
5
6 static size_t
7 mem_cb(void *contents, size_t size, size_t nmemb, void *userp)
8 {
9     size_t realsize = size * nmemb;
10    struct response *mem = (struct response *)userp;
11
12    char *ptr = realloc(mem->memory, mem->size + realsize + 1);
13    if(!ptr) {
```

```

15     printf("from memory\n");
16     return 0;
17 }
18
19 mem->memory = ptr;
20 memcpy(&(mem->memory[mem->size]), contents, realsize);
21 mem->size += realsize;
22 mem->memory[mem->size] = 0;
23
24 return realsize;
25 }
26
27 int main()
28 {
29     struct response chunk = {.memory = malloc(0),
30                             .size = 0};
31
32 /* send all data to this function */
33 curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, mem_cb);
34
35 /* we pass our 'chunk' to the callback function */
36 curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
37
38 free(chunk.memory);
39 }
```

Read data

The read callback is set with `CURLOPT_READFUNCTION` :

```
1 curl_easy_setopt(handle, CURLOPT_READFUNCTION, read_callback);
```

The `read_callback` function must match this prototype:

```
1 size_t read_callback(char *buffer, size_t size, size_t nitems, void *stream);
```

This callback function gets called by libcurl when it wants to send data to the server. This is a transfer that you have set up to upload data or otherwise send it off to the server. This callback will be called over and over until all data has been delivered or the transfer failed.

The `stream` pointer points to the private data set with `CURLOPT_READDATA` :

```
1 curl_easy_setopt(handle, CURLOPT_READDATA, custom_pointer);
```

If this callback is not set, libcurl instead uses 'fread' by default.

The data area pointed at by the pointer `buffer` should be filled up with at most `size` multiplied with `nitems`

number of bytes by your function. The callback should then return the number of bytes that it stored in that memory area, or 0 if we have reached the end of the data. The callback can also return a few "magic" return codes to cause libcurl to return failure immediately or to pause the particular transfer. See the [CURLOPT_READFUNCTION man page](#) for details.

Progress information

The progress callback is what gets called regularly and repeatedly for each transfer during the entire lifetime of the transfer. The old callback was set with `CURLOPT_PROGRESSFUNCTION` but the modern and preferred callback is set with `CURLOPT_XFERINFOFUNCTION`:

```
1 curl_easy_setopt(handle, CURLOPT_XFERINFOFUNCTION, xfer_callback);
```

The `xfer_callback` function must match this prototype:

```
1 int xfer_callback(void *clientp, curl_off_t dltotal, curl_off_t dlnow,
2                      curl_off_t ultotal, curl_off_t ulnow);
```

If this option is set and `CURLOPT_NOPROGRESS` is set to 0 (zero), this callback function gets called by libcurl with a frequent interval. While data is being transferred it will be called frequently, and during slow periods like when nothing is being transferred it can slow down to about one call per second.

The `clientp` pointer points to the private data set with `CURLOPT_XFERINFODATA`:

```
1 curl_easy_setopt(handle, CURLOPT_XFERINFODATA, custom_pointer);
```

The callback gets told how much data libcurl will transfer and has transferred, in number of bytes:

- `dltotal` is the total number of bytes libcurl expects to download in this transfer.
- `dlnow` is the number of bytes downloaded so far.
- `ultotal` is the total number of bytes libcurl expects to upload in this transfer.
- `ulnow` is the number of bytes uploaded so far.

Unknown/unused argument values passed to the callback will be set to zero (like if you only download data, the upload size will remain 0). Many times the callback will be called one or more times first, before it knows the data sizes, so a program must be made to handle that.

Returning a non-zero value from this callback will cause libcurl to abort the transfer and return `CURLE_ABORTED_BY_CALLBACK`.

If you transfer data with the multi interface, this function will not be called during periods of idleness unless you call the appropriate libcurl function that performs transfers.

(The deprecated callback `CURLOPT_PROGRESSFUNCTION` worked identically but instead of taking

arguments of type `CURL off + it used double \`

Header data

The header callback is set with `CURLOPT_HEADERFUNCTION`:

```
1 curl_easy_setopt(handle, CURLOPT_HEADERFUNCTION, header_callback);
```

The `header_callback` function must match this prototype:

```
1 size_t header_callback(char *ptr, size_t size, size_t nmemb, void *userdata);
```

This callback function gets called by libcurl as soon as a header has been received. `ptr` points to the delivered data, and the size of that data is `size` multiplied with `nmemb`. libcurl buffers headers and delivers only "full" headers, one by one, to this callback.

The data passed to this function will not be zero terminated! You cannot, for example, use `printf`'s `%s` operator to display the contents nor `strcpy` to copy it.

This callback should return the number of bytes actually taken care of. If that number differs from the number passed to your callback function, it signals an error condition to the library. This will cause the transfer to abort and the libcurl function used will return `CURLE_WRITE_ERROR`.

The user pointer passed in to the callback in the `userdata` argument is set with `CURLOPT_HEADERDATA`:

```
1 curl_easy_setopt(handle, CURLOPT_HEADERDATA, custom_pointer);
```

Debug

The debug callback is set with `CURLOPT_DEBUGFUNCTION`:

```
1 curl_easy_setopt(handle, CURLOPT_DEBUGFUNCTION, debug_callback);
```

The `debug_callback` function must match this prototype:

```
1 int debug_callback(CURL *handle,
2                     curl_infotype type,
3                     char *data,
4                     size_t size,
5                     void *userdata);
```

This callback function replaces the default verbose output function in the library and will get called for all debug and trace messages to aid applications to understand what's going on. The `type` argument explains

what sort of data that is provided: header, data or SSL data and in which direction it flows.

A common use for this callback is to get a full trace of all data that libcurl sends and receives. The data sent to this callback is always the unencrypted version, even when, for example, HTTPS or other encrypted protocols are used.

This callback must return zero or cause the transfer to stop with an error code.

The user pointer passed in to the callback in the `userdata` argument is set with `CURLOPT_DEBUGDATA`:

```
1 curl_easy_setopt(handle, CURLOPT_DEBUGDATA, custom_pointer);
```

sockopt

The sockopt callback is set with `CURLOPT_SOCKOPTFUNCTION`:

```
1 curl_easy_setopt(handle, CURLOPT_SOCKOPTFUNCTION, sockopt_callback);
```

The `sockopt_callback` function must match this prototype:

```
1 int sockopt_callback(void *clientp,
2                      curl_socket_t curlfd,
3                      curlsocktype purpose);
```

This callback function gets called by libcurl when a new socket has been created but before the connect call, to allow applications to change specific socket options.

The `clientp` pointer points to the private data set with `CURLOPT_SOCKOPTDATA`:

```
1 curl_easy_setopt(handle, CURLOPT_SOCKOPTDATA, custom_pointer);
```

This callback should return:

- `CURL_SOCKOPT_OK` on success
- `CURL_SOCKOPT_ERROR` to signal an unrecoverable error to libcurl
- `CURL_SOCKOPT_ALREADY_CONNECTED` to signal success but also that the socket is in fact already connected to the destination

SSL context

libcurl offers a special TLS related callback called `CURLOPT_SSL_CTX_FUNCTION`. This option only works for libcurl powered by OpenSSL, wolfSSL or mbedTLS and it does nothing if libcurl is built with another TLS backend.

This callback gets called by libcurl just before the initialization of a TLS connection after having processed all other TLS related options to give a last chance to an application to modify the behavior of the TLS initialization. The `ssl_ctx` parameter passed to the callback in the second argument is actually a pointer to the SSL library's `SSL_CTX` for OpenSSL or wolfSSL, and a pointer to `mbedtls_ssl_config` for mbedTLS. If an error is returned from the callback no attempt to establish a connection is made and the operation will return the callback's error code. Set the `userptr` argument with the `CURLOPT_SSL_CTX_DATA` option.

This function gets called on all new connections made to a server, during the TLS negotiation. The TLS context will point to a newly initialized object each time.

Seek and ioctl

This callback is set with `CURLOPT_SEEKFUNCTION`.

The callback gets called by libcurl to seek to a certain position in the input stream and can be used to fast forward a file in a resumed upload (instead of reading all uploaded bytes with the normal read function/callback). It is also called to rewind a stream when data has already been sent to the server and needs to be sent again. This may happen when doing an HTTP PUT or POST with a multi-pass authentication method, or when an existing HTTP connection is reused too late and the server closes the connection. The function shall work like `fseek(3)` or `lseek(3)` and it gets `SEEK_SET`, `SEEK_CUR` or `SEEK_END` as argument for origin, although libcurl currently only passes `SEEK_SET`.

The custom `userp` sent to the callback is the pointer you set with `CURLOPT_SEEKDATA`.

The callback function must return `CURL_SEEKFUNC_OK` on success, `CURL_SEEKFUNC_FAIL` to cause the upload operation to fail or `CURL_SEEKFUNC_CANTSEEK` to indicate that while the seek failed, libcurl is free to work around the problem if possible. The latter can sometimes be done by instead reading from the input or similar.

If you forward the input arguments directly to `fseek(3)` or `lseek(3)`, note that the data type for offset is not the same as defined for `curl_off_t` on many systems!

Network data conversion

Convert to and from network callbacks

For non-ASCII platforms, `CURLOPT_CONV_FROM_NETWORK_FUNCTION` is provided. This function should convert **to** host encoding **from** the network encoding.

`CURLOPT_CONV_TO_NETWORK_FUNCTION` should convert from **host** encoding **to** the network encoding. It is used when commands or ASCII data are sent over the network.

Convert from UTF-8 callback

`CURLOPT_CONV_FROM_UTF8_FUNCTION` should convert to host encoding from UTF-8 encoding. It is required only for SSL processing.

Opensocket and closesocket

Occasionally you end up in a situation where you want your application to control with more precision exactly what socket libcurl will use for its operations. libcurl offers this pair of callbacks that replaces libcurl's own call to `socket()` and the subsequent `close()` of the same file descriptor.

Provide a file descriptor

By setting the `CURLOPT_OPENSOCKETFUNCTION` callback, you can provide a custom function to return a file descriptor for libcurl to use:

```
1 curl_easy_setopt(handle, CURLOPT_OPENSOCKETFUNCTION, opensocket_callback);
```

The `opensocket_callback` function must match this prototype:

```
1 curl_socket_t opensocket_callback(void *clientp,
2                                     curlsocktype purpose,
3                                     struct curl_sockaddr *address);
```

The callback gets the `clientp` as first argument, which is simply an opaque pointer you set with `CURLOPT_OPENSOCKETDATA`.

The other two arguments pass in data that identifies for what `purpose` and `address` the socket is to be used. The `purpose` is a typedef with a value of `CURLSOCKTYPE_IPCXN` or `CURLSOCKTYPE_ACCEPT`, identifying in which circumstance the socket is created. The "accept" case being when libcurl is used to accept an incoming FTP connection for when FTP active mode is used, and all other cases when libcurl creates a socket for its own outgoing connections the `IPCXN` value is passed in.

The `address` pointer points to a `struct curl_sockaddr` that describes the IP address of the network destination for which this socket is created. Your callback can for example use this information to whitelist or blacklist specific addresses or address ranges.

The `socketopen` callback is also explicitly allowed to modify the target address in that struct, if you would like to offer some sort of network filter or translation layer.

The callback should return a file descriptor or `CURL_SOCKET_BAD`, which then will cause an unrecoverable error within libcurl and it will eventually return `CURLE_COULDNT_CONNECT` from its `perform` function.

If you want to return a file descriptor that is *already connected* to a server, then you must also set the [sockopt callback](#) and make sure that returns the correct return value.

The `curl_sockaddr` struct looks like this:

```
1 struct curl_sockaddr {  
2     int family;  
3     int socktype;  
4     int protocol;  
5     unsigned int addrlen;  
6     struct sockaddr addr;  
7 };
```

Socket close callback

The corresponding callback to the open socket is of course the close socket. Usually when you provide a custom way to provide a file descriptor you want to provide your own cleanup version as well:

```
1 curl_easy_setopt(handle, CURLOPT_CLOSESOCKETFUNCTION, closesocket_callback);
```

The `closesocket_callback` function must match this prototype:

```
1 int closesocket_callback(void *clientp, curl_socket_t item);
```

SSH key

This callback is set with `CURLOPT_SSH_KEYFUNCTION`.

It gets called when the `known_host` matching has been done, to allow the application to act and decide for libcurl how to proceed. The callback will only be called if `CURLOPT_SSH_KNOWNHOSTS` is also set.

In the arguments to the callback are the old key and the new key and the callback is expected to return a return code that tells libcurl how to act:

`CURLKHSTAT_FINE_REPLACE` - The new host+key is accepted and libcurl will replace the old host+key into the `known_hosts` file before continuing with the connection. This will also add the new host+key combo to the `known_host` pool kept in memory if it was not already present there. The adding of data to the file is done by completely replacing the file with a new copy, so the permissions of the file must allow this.

`CURLKHSTAT_FINE_ADD_TO_FILE` - The host+key is accepted and libcurl will append it to the `known_hosts` file before continuing with the connection. This will also add the host+key combo to the `known_host` pool kept in memory if it was not already present there. The adding of data to the file is done by completely replacing the file with a new copy, so the permissions of the file must allow this.

`CURLKHSTAT_FINE` - The host+key is accepted libcurl will continue with the connection. This will also add the host+key combo to the `known_host` pool kept in memory if it was not already present there.

`CURLKHSTAT_REJECT` - The host+key is rejected. libcurl will deny the connection to continue and it will be closed.

`CURLKHSTAT_DEFER` - The host+key is rejected, but the SSH connection is asked to be kept alive. This feature could be used when the app wants to somehow return and act on the host+key situation and then retry without needing the overhead of setting it up from scratch again.

RTSP interleaved data

The callback with the `CURLOPT_INTERLEAVEFUNCTION` option.

This callback gets called by libcurl as soon as it has received interleaved RTP data when doing an RTSP transfer. It gets called for each `$` block and therefore contains exactly one upper-layer protocol unit (e.g. one RTP packet). libcurl writes the interleaved header as well as the included data for each call. The first byte is always an ASCII dollar sign. The dollar sign is followed by a one byte channel identifier and then a 2 byte integer length in network byte order. See RFC2326 Section 10.12 for more information on how RTP interleaving behaves. If unset or set to NULL, curl will use the default write function.

The `CURLOPT_INTERLEAVEDATA` pointer is passed in the userdata argument in the callback.

FTP wildcard matching

libcurl supports FTP wildcard matching. You use this feature by setting `CURLOPT_WILDCARDMATCH` to `1L` and then use a "wildcard pattern" in the in the file name part of the URL.

Wildcard patterns

The default libcurl wildcard matching function supports:

`*` - ASTERISK

```
1 ftp://example.com/some/path/*.txt
```

To match all txt files in the directory `some/path`. Only two asterisks are allowed within the same pattern string.

`?` - QUESTION MARK"

A question mark matches any (exactly one) character. Like if you have files called `photo1.jpeg` and `photo7.jpeg` this pattern could match them:

```
1 ftp://example.com/some/path/photo?.jpeg
```

[- BRACKET EXPRESSION

The left bracket opens a bracket expression. The question mark and asterisk have no special meaning in a bracket expression. Each bracket expression ends by the right bracket (]) and matches exactly one character. Some examples follow:

[a-zA-Z0-9] or [f-gF-G] - character intervals

[abc] - character enumeration

[^abc] or [!abc] - negation

[[:name:]] class expression. Supported classes are alnum, lower, space, alpha, digit, print, upper, blank, graph, xdigit.

[] [-!] - special case, matches only \-, [, ! or ^.

[\\\\[\]\\]] - escape syntax. Matches [,] or \\.

Using the rules above, a file name pattern can be constructed:

```
1 ftp://example.com/some/path/[a-z[:upper:]]\\.jpeg
```

FTP chunk callbacks

When FTP wildcard matching is used, the `CURLOPT_CHUNK_BGN_FUNCTION` callback is called before a transfer is initiated for a file that matches.

The callback can then opt to return one of these return codes to tell libcurl what to do with the file:

- `CURL_CHUNK_BGN_FUNC_OK` transfer the file
- `CURL_CHUNK_BGN_FUNC_SKIP`
- `CURL_CHUNK_BGN_FUNC_FAIL` stop because of error

After the matched file has been transferred or skipped, the `CURLOPT_CHUNK_END_FUNCTION` callback is called.

The end chunk callback can only return success or error.

FTP matching callback

If the default pattern matching function is not to your liking, you can provide your own replacement function by setting the `CURLOPT_FNMATCH_FUNCTION` option to your alternative.

Resolver start

This callback function, set with `CURLOPT_RESOLVER_START_FUNCTION` gets called by libcurl every time before a new resolve request is started, and it specifies for which `CURL *` handle the resolve is intended.

Sending trailers

"Trailers" is an HTTP/1 feature where headers can be passed on *at the end of a transfer*. This callback is used for when you want to send trailers with curl after an upload has been performed. An upload in the form of a chunked encoded POST.

The callback set with `CURLOPT_TRAILERFUNCTION` will be called and the function can then append headers to a list. One or many. When done, libcurl sends off those as trailers to the server.

HSTS

For HSTS, HTTP Strict Transport Security, libcurl provides two callbacks to allow an allocation to implement the storage for rules. The callbacks are then set to read and/or write the HSTS policies from a persistent storage.

With `CURLOPT_HSTSREADFUNCTION`, the application provides a function using which HSTS data into libcurl is read. `CURLOPT_HSTSWRITEFUNCTION` is the corresponding function that libcurl calls to write data.

Prereq

"Prereq" here means immediately before the request is issued. That's the moment where this callback is called.

Set the function with `CURLOPT_PREREQFUNCTION` and it will be called and pass on IP address and port numbers in the arguments. This allows the application to know about the transfer just before it starts and also allows it to cancel this particular transfer should it want to.

Cleanup

In previous sections we have discussed how to setup handles and how to drive the transfers. All transfers will, of course, end up at some point, either successfully or with a failure.

Multi API

When you have finished a single transfer with the multi API, you use `curl_multi_info_read()` to identify exactly which easy handle was completed and you remove that easy handle from the multi handle with `curl_multi_remove_handle()`.

If you remove the last easy handle from the multi handle so there are no more transfers going on, you can close the multi handle like this:

```
1 curl_multi_cleanup( multi_handle );
```

easy handle

When the easy handle is done serving its purpose, you can close it. If you intend to do another transfer, you are however advised to rather reuse the handle rather than to close it and create a new one.

If you do not intend to do another transfer with the easy handle, you simply ask libcurl to cleanup:

```
1 curl_easy_cleanup( easy_handle );
```

Name resolving

Most transfers libcurl can do involves a name that first needs to be translated to an Internet address. That is "name resolving". Using a numerical IP address directly in the URL usually avoids the name resolve phase, but in many cases it is not easy to manually replace the name with the IP address.

libcurl tries hard to [re-use an existing connection](#) rather than to create a new one. The function that checks for an existing connection to use is based purely on the name and is performed before any name resolving is attempted. That is one of the reasons the re-use is so much faster. A transfer using a reused connection will not resolve the host name again.

If no connection can be reused, libcurl resolves the host name to the set of addresses it resolves to. Typically this means asking for both IPv4 and IPv6 addresses and there may be a whole set of those returned to libcurl. That set of addresses is then tried until one works, or it returns failure.

An application can force libcurl to use only an IPv4 or IPv6 resolved address by setting `CURLOPT_IPRESOLVE` to the preferred value. For example, ask to only use IPv6 addresses:

```
1 curl_easy_setopt(easy, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_V6);
```

Name resolver backends

libcurl can be built to do name resolves in one out of these three different ways and depending on which backend way that is used, it gets a slightly different feature set and sometimes modified behavior.

1. The default backend is invoking the "normal" libc resolver functions in a new helper-thread, so that it can still do fine-grained timeouts if wanted and there will be no blocking calls involved.
2. On older systems, libcurl uses the standard synchronous name resolver functions. They unfortunately make all transfers within a multi handle block during its operation and it is much harder to time out nicely.
3. There is also support for resolving with the c-ares third party library, which supports asynchronous name resolving without the use of threads. This scales better to huge number of parallel transfers but it is not always 100% compatible with the native name resolver functionality.

DNS over HTTPS

Independently of what resolver backend that libcurl is built to use, since 7.62.0 it also provides a way for the user to ask a specific DoH (DNS over HTTPS) server for the address of a name. This will avoid using the normal, native resolver method and server and instead asks a dedicated separate one.

A DoH server is specified as a full URL with the `CURLOPT_DOH_URL` option like this:

```
1 curl_easy_setopt(easy, CURLOPT_DOH_URL, "https://example.com/doh");
```

The URL passed to this option *must* be using https:// and it is generally recommended that you have HTTP/2 support enabled so that libcurl can perform multiple DoH requests multiplexed over the connection to the DoH server.

Caching

When a name has been resolved, the result will be put in libcurl's in-memory cache so that subsequent resolves of the same name will be near instant for as long the name is kept in the DNS cache. By default, each entry is kept in the cache for 60 seconds, but that value can be changed with `CURLOPT_DNS_CACHE_TIMEOUT`.

The DNS cache is kept within the easy handle when `curl_easy_perform` is used, or within the multi handle when the multi interface is used. It can also be made shared between multiple easy handles using the [share interface](#).

Custom addresses for hosts

Sometimes it is handy to provide "fake" addresses to real host names so that libcurl will connect to a

different address instead of one an actual name resolve would suggest

With the help of the `CURLOPT_RESOLVE` option, an application can pre-populate libcurl's DNS cache with a custom address for a given host name and port number.

To make libcurl connect to 127.0.0.1 when example.com on port 443 is requested, an application can do:

```
1 struct curl_slist *dns;
2 dns = curl_slist_append(NULL, "example.com:443:127.0.0.1");
3 curl_easy_setopt(curl, CURLOPT_RESOLVE, dns);
```

Since this puts the "fake" address into the DNS cache, it will work even when following redirects etc.

Name server options

For libcurl built to use c-ares, there is a few options available that offer fine-grained control of what DNS servers to use and how. This is limited to c-ares build purely because these are powers that are not available when the standard system calls for name resolving are used.

- With `CURLOPT_DNS_SERVERS`, the application can select to use a set of dedicated DNS servers.
 - With `CURLOPT_DNS_INTERFACE` it can tell libcurl which network interface to speak DNS over instead of the default one.
 - With `CURLOPT_DNS_LOCAL_IP4` and `CURLOPT_DNS_LOCAL_IP6`, the application can specify which specific network addresses to bind DNS resolves to.
-

No global DNS cache

The option called `CURLOPT_DNS_USE_GLOBAL_CACHE` once told curl to use a global DNS cache. This functionality has been removed since 7.65.0, so while this option still exists it does nothing.

Proxies

A proxy in a network context is a middle man, a server in between you as a client and the remote server you want to communicate with. The client contacts the middle man which then goes on to contact the remote server for you.

This style of proxy use is sometimes used by companies and organizations, in which case you are usually required to use them to reach the target server.

There are several different kinds of proxies and different protocols to use when communicating with a proxy, and libcurl supports a few of the most common proxy protocols. It is important to realize that the protocol used to the proxy is not necessarily the same protocol used to the remote server.

When setting up a transfer with libcurl you need to point out the server name and port number of the proxy. You may find that your favorite browsers can do this in slightly more advanced ways than libcurl can, and we will get into such details in later sections.

Proxy types

libcurl supports the two major proxy types: SOCKS and HTTP proxies. More specifically, it supports both SOCKS4 and SOCKS5 with or without remote name lookup, as well as both HTTP and HTTPS to the local proxy.

The easiest way to specify which kind of proxy you are talking to is to set the scheme part of the proxy host name string (`CURLOPT_PROXY`) to match it:

```
1 socks4://proxy.example.com:12345/
2 socks4a://proxy.example.com:12345/
3 socks5://proxy.example.com:12345/
4 socks5h://proxy.example.com:12345/
5 http://proxy.example.com:12345/
6 https://proxy.example.com:12345/
```

`socks4` - means SOCKS4 with local name resolving

`socks4a` - means SOCKS4 with proxy's name resolving

`socks5` - means SOCKS5 with local name resolving

`socks5h` - means SOCKS5 with proxy's name resolving

`http` - means HTTP, which always lets the proxy resolve names

`https` - means HTTPS **to the proxy**, which always lets the proxy resolve names (Note that HTTPS proxy support was added recently, in curl 7.52.0, and it still only works with a subset of the TLS libraries: OpenSSL, GnuTLS and NSS.)

You can also opt to set the type of the proxy with a separate option if you prefer to only set the host name, using `CURLOPT_PROXYTYPE`. Similarly, you can set the proxy port number to use with `CURLOPT_PROXYPORT`.

Local or proxy name lookup

In a section above you can see that different proxy setups allow the name resolving to be done by different parties involved in the transfer. You can in several cases either have the client resolve the server host name and pass on the IP address to the proxy to connect to - which of course assumes that the name lookup works accurately on the client system - or you can hand over the name to the proxy to have the proxy resolve the name; converting it to an IP address to connect to.

When you are using an HTTP or HTTPS proxy, you always give the name to the proxy to resolve.

Which proxy?

If your network connection requires the use of a proxy to reach the destination, you must figure this out and tell libcurl to use the correct proxy. There is no support in libcurl to make it automatically figure out or detect a proxy.

When using a browser, it is popular to provide the proxy with a PAC script or other means but none of those are recognized by libcurl.

Proxy environment variables

If no proxy option has been set, libcurl will check for the existence of specially named environment variables before it performs its transfer to see if a proxy is requested to get used.

You can specify the proxy by setting a variable named `[scheme]_proxy` to hold the proxy host name (the same way you would specify the host with `-x`). So if you want to tell curl to use a proxy when accessing an HTTP server, you set the `http_proxy` environment variable. Like this:

```
1 http_proxy=http://proxy.example.com:80
```

The proxy example above is for HTTP, but can of course also set `ftp_proxy`, `https_proxy`, and so on for the specific protocols you want to proxy. All these proxy environment variable names except `http_proxy` can also be specified in uppercase, like `HTTPS_PROXY`.

To set a single variable that controls *all* protocols, the `ALL_PROXY` exists. If a specific protocol variable one exists, such a one will take precedence.

When using environment variables to set a proxy, you could easily end up in a situation where one or a few host names should be excluded from going through the proxy. This can be done with the `NO_PROXY` variable - or the corresponding `CURLOPT_NOPROXY` libcurl option. Set that to a comma-separated list of host names that should not use a proxy when being accessed. You can set `NO_PROXY` to be a single asterisk (*) to match all hosts.

HTTP proxy

The HTTP protocol details exactly how an HTTP proxy should be used. Instead of sending the request to the actual remote server, the client (libcurl) instead asks the proxy for the specific resource. The connection to the HTTP proxy is made using plain unencrypted HTTP.

If an HTTPS resource is requested, libcurl will instead issue a `CONNECT` request to the proxy. Such a request opens a tunnel through the proxy, where it passes data through without understanding it. This way, libcurl can establish a secure end-to-end TLS connection even when an HTTP proxy is present.

You can proxy non-HTTP protocols over an HTTP proxy, but since this is mostly done by the CONNECT method to tunnel data through it requires that the proxy is configured to allow the client to connect to those other particular remote port numbers. Many HTTP proxies are setup to inhibit connections to other port numbers than 80 and 443.

HTTPS proxy

An HTTPS proxy is similar to an HTTP proxy but allows the client to connect to it using a secure HTTPS connection. Since the proxy connection is separate from the connection to the remote site even in this situation, as HTTPS to the remote site will be tunneled through the HTTPS connection to the proxy, libcurl provides a whole set of TLS options for the proxy connection that are separate from the connection to the remote host.

For example, `CURLOPT_PROXY_CAINFO` is the same functionality for the HTTPS proxy as `CURLOPT_CAINFO` is for the remote host. `CURLOPT_PROXY_SSL_VERIFYPEER` is the proxy version of `CURLOPT_SSL_VERIFYPEER` and so on.

HTTPS proxies are still today fairly unusual in organizations and companies.

Proxy authentication

Authentication with a proxy means that you need to provide valid credentials in the handshake negotiation with the proxy itself. The proxy authentication is then in addition to and separate of the possible authentication or lack of authentication with the remote host.

libcurl supports authentication with HTTP, HTTPS and SOCKS5 proxies. The key option is then `CURLOPT_PROXYUSERPWD` which sets the user name and password to use - unless you set it within the `CURLOPT_PROXY` string.

HTTP Proxy headers

With an HTTP or HTTPS proxy, libcurl will issue a request to the proxy that includes a set of headers. An application can of course modify the headers, just like for requests sent to servers.

libcurl offers the `CURLOPT_PROXYHEADER` for controlling the headers that are sent to a proxy **when there is a separate request sent to the server**. This typically means the initial `CONNECT` request sent to a proxy for setting up a tunnel through the proxy.

Post transfer info

Remember how libcurl transfers are associated with an "easy handle"! Each transfer has such a handle and

when a transfer is completed, before the handle is cleaned or reused for another transfer, it can be used to extract information from the previous operation.

Your friend for doing this is called `curl_easy_getinfo()` and you tell it which specific information you are interested in and it will return that to you if it can.

When you use this function, you pass in the easy handle, which information you want and a pointer to a variable to hold the answer. You must pass in a pointer to a variable of the correct type or you risk that things will go side-ways. These information values are designed to be provided *after* the transfer is completed.

The data you receive can be a long, a 'char *', a 'struct curl_slist *', a double or a socket.

This is how you extract the `Content-Type:` value from the previous HTTP transfer:

```
1 CURLcode res;
2 char *content_type;
3 res = curl_easy_getinfo(curl, CURLINFO_CONTENT_TYPE, &content_type);
```

If you want to extract the local port number that was used in that connection:

```
1 CURLcode res;
2 long port_number;
3 res = curl_easy_getinfo(curl, CURLINFO_LOCAL_PORT, &port_number);
```

Available information

Getinfo option	Type	Description
CURLINFO_ACTIVE SOCKET	curl_socket_t	The session's active socket
CURLINFO_APPCONNECT_ TIME	double	Time from start until SSL/SSH handshake completed
CURLINFO_APPCONNECT_ TIME_T	curl_off_t	Time from start until SSL/SSH handshake completed (in microseconds)
CURLINFO_CERTINFO	struct curl_slist *	Certificate chain
CURLINFO_CONDITION_ UNMET	long	Whether or not a time conditional was met
CURLINFO_CONNECT_ TIME	double	Time from start until remote host or proxy completed
CURLINFO_CONNECT_ TIME_ T	curl_off_t	Time from start until remote host or proxy completed (in

		microseconds)
CURLINFO_CONTENT_LENGTH_DOWNLOAD	double	Content length from the Content Length header
CURLINFO_CONTENT_LENGTH_UPLOAD	double	Upload size
CURLINFO_CONTENT_TYPE	char *	Content type from the Content-Type header
CURLINFO_COOKIELIST	struct curl_slist *	List of all known cookies
CURLINFO_EFFECTIVE_METHOD	char *	Last used HTTP request method
CURLINFO_EFFECTIVE_URL	char *	Last used URL
CURLINFO_FILETIME	long	Remote time of the retrieved document
CURLINFO_FTP_ENTRY_PATH	char *	The entry path after logging in to an FTP server
CURLINFO_HEADER_SIZE	long	Number of bytes of all headers received
CURLINFO_HTTP_CONNECT_CODE	long	Last proxy CONNECT response code
CURLINFO_HTTP_VERSION	long	The HTTP version used in the connection
CURLINFO_HTTPAUTH_AVAIL	long	Available HTTP authentication methods (bitmask)
CURLINFO_LASTSOCKET	long	Last socket used

Share data between handles

Sometimes applications need to share data between transfers. All easy handles added to the same multi handle automatically get a lot of sharing done between the handles in that same multi handle, but sometimes that is not exactly what you want.

Multi handle

All easy handles added to the same multi handle automatically share [connection cache](#) and [dns cache](#).

Sharing between easy handles

libcurl has a generic "sharing interface", where the application creates a "share object" that then holds data that can be shared by any number of easy handles. The data is then stored and read from the shared object instead of kept within the handles that are sharing it.

```
1 CURLSH *share = curl_share_init();
```

The shared object can be set to share all or any of cookies, connection cache, dns cache and SSL session id cache.

For example, setting up the share to hold cookies and dns cache:

```
1 curl_share_setopt(share, CURLOPT_SHARE, CURL_LOCK_DATA_COOKIE);
2 curl_share_setopt(share, CURLOPT_SHARE, CURL_LOCK_DATA_DNS);
```

You then set up the corresponding transfer to use this share object:

```
1 curl_easy_setopt(curl, CURLOPT_SHARE, share);
```

Transfers done with this `curl` handle will thus use and store its cookie and dns information in the `share` handle. You can set several easy handles to share the same share object.

What to share

`CURL_LOCK_DATA_COOKIE` - set this bit to share cookie jar. Note that each easy handle still needs to get its cookie "engine" started properly to start using cookies.

`CURL_LOCK_DATA_DNS` - the DNS cache is where libcurl stores addresses for resolved host names for a while to make subsequent lookups faster.

`CURL_LOCK_DATA_SSL_SESSION` - the SSL session ID cache is where libcurl store resume information for SSL connections to be able to resume a previous connection faster.

`CURL_LOCK_DATA_CONNECT` - when set, this handle will use a shared connection cache and thus will probably be more likely to find existing connections to re-use etc, which may result in faster performance when doing multiple transfers to the same host in a serial manner.

Locking

If you want have the share object shared by transfers in a multi-threaded environment. Perhaps you have a CPU with many cores and you want each core to run its own thread and transfer data, but you still want the different transfers to share data. Then you need to set the mutex callbacks.

If you do not use threading and you *know* you access the shared object in a serial one-at-a-time manner you do not need to set any locks. But if there is ever more than one transfer that access share object at a time, it needs to get mutex callbacks setup to prevent data destruction and possibly even crashes.

Since libcurl itself does not know how to lock things or even what threading model you are using, you must make sure to do mutex locks that only allows one access at a time. A lock callback for a pthreads-using application could look similar to:

```
1 static void lock_cb(CURL *handle, curl_lock_data data,
2                      curl_lock_access access, void *userptr)
3 {
4     pthread_mutex_lock(&lock[data]); /* uses a global lock array */
5 }
6 curl_share_setopt(share, CURLSHOPT_LOCKFUNC, lock_cb);
```

With the corresponding unlock callback could look like:

```
1 static void unlock_cb(CURL *handle, curl_lock_data data,
2                       void *userptr)
3 {
4     pthread_mutex_unlock(&lock[data]); /* uses a global lock array */
5 }
6 curl_share_setopt(share, CURLSHOPT_UNLOCKFUNC, unlock_cb);
```

Unshare

A transfer will use the share object during its transfer and share what that object has been specified to share with other handles sharing the same object.

In a subsequent transfer, `CURLOPT_SHARE` can be set to `NULL` to prevent a transfer from continuing to share. In that case, the handle may start the next transfer with empty caches for the data that was previously shared.

Between two transfers, a share object can also get updated to share a different set of properties so that the handles that share that object will share a different set of data next time. You remove an item to share from a shared object with the `curl_share_setopt()`'s `CURLSHOPT_UNSHARE` option like this when unsharing DNS data:

```
1 curl_share_setopt(share, CURLSHOPT_UNSHARE, CURL_LOCK_DATA_DNS);
```

URL API

libcurl offers an API for parsing, updating and generating URLs. Using this, applications can take advantage of using libcurl's URL parser for its own purposes. By using the same parser, security problems due to different interpretations can be avoided.

- [Include files](#)
- [Create, cleanup, duplicate](#)
- [Parse a URL](#)
- [Redirect to a relative URL](#)
- [Get a URL](#)
- [Get individual URL parts](#)
- [Set individual URL parts](#)
- [Append to the query](#)
- [CURLOPT_CURLU](#)

Include files

You include `<curl/curl.h>` in your code when you want to use the URL API.

```
1 #include <curl/curl.h>
2
3 CURLU *h = curl_url();
4 rc = curl_url_set(h, CURLUPART_URL, "ftp://example.com/no/where", 0);
```

Create, cleanup, duplicate

The first step when using this API is to create a `CURLU *` handle that holds URL info and resources. The handle is a reference to an associated data object that holds information about a single URL and all its different components.

The API allows you to set or get each URL component separately or as a full URL.

Create a URL handle like this:

```
1 CURLU *h = curl_url();
```

When you are done with it, clean it up:

```
1 curl_url_cleanup(h);
```

When you need a copy of a handle, just duplicate it:

```
1 CURLU *nh = curl_url_dup(h);
```

Parse a URL

You parse a full URL by *setting* the `CURLUPART_URL` part in the handle:

```
1 CURLU *h = curl_url();
2 rc = curl_url_set(h, CURLUPART_URL,
3                     "https://example.com:449/foo/bar?name=moo", 0);
```

If successful, `rc` contains `CURLUE_OK` and the different URL components are held in the handle. It means that the URL was valid as far as libcurl concerns.

The function call's forth argument is a bitmask for changing specific features. You can set none, one more bits in that to alter the parser's behavior:

CURLU_NON_SUPPORT_SCHEME

Makes `curl_url_set()` accept a non-supported scheme. If not set, the only acceptable schemes are for the protocols libcurl knows and have built-in support for.

CURLU_URLENCODE

Makes the function URL encode the path part if any bytes in it would benefit from that: like spaces or "control characters".

CURLU_DEFAULT_SCHEME

If the passed in string does not use a scheme, assume that the default one was intended. The default scheme is HTTPS. If this is not set, a URL without a scheme part will not be accepted as valid. Overrides the `CURLU_GUESS_SCHEME` option if both are set.

CURLU_GUESS_SCHEME

Makes libcurl allow the URL to be set without a scheme and it instead "guesses" which scheme that was intended based on the host name. If the outermost sub-domain name matches DICT, FTP, IMAP, LDAP, POP3 or SMTP then that scheme will be used, otherwise it picks HTTP. Conflicts with the `CURLU_DEFAULT_SCHEME` option which takes precedence if both are set.

CURLU_NO_AUTHORITY

Skips authority checks. The RFC allows individual schemes to omit the host part (normally the only mandatory part of the authority), but libcurl cannot know whether this is permitted for custom schemes. Specifying the flag permits empty authority sections, similar to how file scheme is handled. Really only usable in combination with `CURLU_NON_SUPPORT_SCHEME`.

CURLU_PATH_AS_IS

Makes libcurl skip the normalization of the path. That is the procedure where curl otherwise removes sequences of dot-slash and dot-dot etc. The same option used for transfers is called `CURLOPT_PATH_AS_IS`.

CURLU_ALLOW_SPACE

Makes the URL parser allow space (ASCII 32) where possible. The URL syntax does normally not allow spaces anywhere, but they should be encoded as `%20` or `+`. When spaces are allowed, they are still not allowed in the scheme. When space is used and allowed in a URL, it will be stored as-is unless `CURLU_URLENCODE` is also set, which then makes libcurl URL-encode the space before stored. This affects how the URL will be constructed when `curl_url_get()` is subsequently used to extract the full URL or individual parts.

Redirect to a relative URL

When the handle already has parsed a URL, setting a second relative URL will make it "redirect" to adapt to it.

Example, first set the original URL then set the one we "redirect" to:

```
1 CURLU *h = curl_url();
2 rc = curl_url_set(h, CURLUPART_URL,
3                     "https://example.com/foo/bar?name=moo", 0);
4
5 rc = curl_url_set(h, CURLUPART_URL, "../test?another", 0);
```

Get a URL

The `CURLU *` handle represents a URL, or at parts of a URL, and you can easily extract that URL at any point:

```
1 char *url;
2 rc = curl_url_get(h, CURLUPART_URL, &url, CURLU_NO_DEFAULT_PORT);
3 curl_free(url);
```

If the handle does not have enough information to extra a full URL, it will return error.

The returned string must be freed with `curl_free()` after you are done with it.

The zero in the function call's forth argument is a flag bitmask for changing specific features.

CURLU_DEFAULT_PORT

If the URL handle has no port number stored, this option will make `curl_url_get()` return the default port for the used scheme.

CURLU_DEFAULT_SCHEME

If the handle has no scheme stored, this option will make `curl_url_get()` return the default scheme instead of error.

CURLU_NO_DEFAULT_PORT

Instructs `curl_url_get()` to *not* use a port number in the generated URL if that port number matches the default port used for the scheme. For example, if port number 443 is set and the scheme is `https`, the extracted URL will not include the port number.

CURLU_URLENCODE

If set, will make `curl_url_get()` URL encode the host name part when a full URL is retrieved. If not set (default), libcurl returns the URL with the host name "raw" to support IDN names to appear as-is. IDN host names are typically using non-ASCII bytes that otherwise will be percent-encoded.

Note that even when not asking for URL encoding, the `%` (byte 37) will be URL encoded in host names to make sure the host name remains valid.

Get individual URL parts

When a URL has been parsed or individual parts have been set in the `CURLU` handle, you can extract those pieces again from the handle at any time.

The second argument to `curl_url_get()` specifies which part you want extracted. They are all extracted as null-terminated `char *` data, so you pass a pointer to such a variable.

```
1 char *host;
2 rc = curl_url_get(h, CURLUPART_HOST, &host, 0);
3
4 char *scheme;
5 rc = curl_url_get(h, CURLUPART_SCHEME, &scheme, 0);
6
7 char *user;
8 rc = curl_url_get(h, CURLUPART_USER, &user, 0);
9
10 char *password;
11 rc = curl_url_get(h, CURLUPART_PASSWORD, &password, 0);
12
13 char *port;
14 rc = curl_url_get(h, CURLUPART_PORT, &port, 0);
15
16 char *path;
17 rc = curl_url_get(h, CURLUPART_PATH, &path, 0);
18
19 char *query;
20 rc = curl_url_get(h, CURLUPART_QUERY, &query, 0);
21
22 char *fragment;
23 rc = curl_url_get(h, CURLUPART_FRAGMENT, &fragment, 0);
```

Remember to free the returned string with `curl_free` when you are done with it!

Extracted parts are not URL decoded unless the user asks for it with the `CURLU_URLDECODE` flag.

Set individual URL parts

The API allows the application to set individual parts of a URL held in the `CURLU` handle, either after having parsed a full URL or instead of parsing such.

```
1 rc = curl_url_set(urlp, CURLUPART_HOST, "www.example.com", 0);
2 rc = curl_url_set(urlp, CURLUPART_SCHEME, "https", 0);
3 rc = curl_url_set(urlp, CURLUPART_USER, "john", 0);
4 rc = curl_url_set(urlp, CURLUPART_PASSWORD, "doe", 0);
5 rc = curl_url_set(urlp, CURLUPART_PORT, "443", 0);
6 rc = curl_url_set(urlp, CURLUPART_PATH, "/index.html", 0);
7 rc = curl_url_set(urlp, CURLUPART_QUERY, "name=john", 0);
8 rc = curl_url_set(urlp, CURLUPART_FRAGMENT, "anchor", 0);
```

The API always expects a null-terminated `char *` string in the third argument, or NULL to clear the field. Note that the port number is also provided as a string this way.

Set parts are not URL encoded unless the user asks for it with the `CURLU_URLENCODE` flag in the forth argument.

Append to the query

An application can append a string to the right end of the existing query part with the `CURLU_APPENDQUERY` flag.

Consider a handle that holds the URL `https://example.com/?shoes=2`. An application can then add the string `hat=1` to the query part like this:

```
1 rc = curl_url_set(urlp, CURLUPART_QUERY, "hat=1", CURLU_APPENDQUERY);
```

It will even notice the lack of an ampersand (`&`) separator so it will inject one too, and the handle's full URL would then equal `https://example.com/?shoes=2&hat=1`.

The appended string can of course also get URL encoded on add, and if asked, the encoding will skip the '=' character. For example, append `candy=M&M` to what we already have, and URL encode it to deal with the ampersand in the data:

```
1 rc = curl_url_set(urlp, CURLUPART_QUERY, "candy=M&M",
2                      CURLU_APPENDQUERY | CURLU_URLENCODE);
```

Now the URL looks like `https://example.com/?shoes=2&hat=1&candy=M%26M`.

CURLOPT_CURLU

As a convenience to applications, they can pass in an already parsed URL to libcurl to work with, as an alternative to `CURLOPT_URL`.

You pass in a `CURLU` handle instead of a URL string with the `CURLOPT_CURLU` option.

Example:

```
1 CURLU *h = curl_url();
2 rc = curl_url_set(h, CURLUPART_URL, "https://example.com/", 0);
3
4 CURL *easy = curl_easy_init();
5 curl_easy_setopt(easy, CURLOPT_CURLU, h);
```

API compatibility

libcurl promises API stability and guarantees that your program written today will remain working in the future. We do not break compatibility.

Over time, we add features, new options and new functions to the APIs but we do not change behavior in a non-compatible way or remove functions.

The last time we changed the API in a non-compatible way was for 7.16.0 in 2006 and we plan to never do it again.

Version numbers

Curl and libcurl are individually versioned, but they mostly follow each other rather closely.

The version numbering is always built up using the same system:

```
1 X.Y.Z
```

- X is main version number
- Y is release number
- Z is patch number

Bumping numbers

One of these X.Y.Z numbers will get bumped in every new release. The numbers to the right of a bumped number will be reset to zero.

The main version number X is bumped when *really* big, world colliding changes are made. The release number Y is bumped when changes are performed or things/features are added. The patch number Z is bumped when the changes are mere bugfixes.

It means that after a release 1.2.3, we can release 2.0.0 if something really big has been made, 1.3.0 if not that big changes were made or 1.2.4 if mostly bugs were fixed.

Bumping, as in increasing the number with 1, is unconditionally only affecting one of the numbers (and the ones to the right of it are set to zero). 1 becomes 2, 3 becomes 4, 9 becomes 10, 88 becomes 89 and 99 becomes 100. So, after 1.2.9 comes 1.2.10. After 3.99.3, 3.100.0 might come.

All original curl source release archives are named according to the libcurl version (not according to the curl client version that, as said before, might differ).

Which libcurl version

As a service to any application that might want to support new libcurl features while still being able to build with older versions, all releases have the libcurl version stored in the `curl/curlver.h` file using a static numbering scheme that can be used for comparison. The version number is defined as:

```
1 #define LIBCURL_VERSION_NUM 0xXXYYZZ
```

Where `XX`, `YY` and `ZZ` are the main version, release and patch numbers in hexadecimal. All three number fields are always represented using two digits (eight bits each). 1.2.0 would appear as "0x010200" while version 9.11.7 appears as "0x090b07".

This 6-digit hexadecimal number is always a greater number in a more recent release. It makes comparisons with greater than and less than work.

This number is also available as three separate defines: `LIBCURL_VERSION_MAJOR`, `LIBCURL_VERSION_MINOR` and `LIBCURL_VERSION_PATCH`.

These defines are, of course, only suitable to figure out the version number built *just now* and they will not help you figuring out which libcurl version that is used at runtime three years from now.

Which libcurl version runs

To figure out which libcurl version that your application is using *right now*, `curl_version_info()` is there for you.

Applications should use this function to judge if things are possible to do or not, instead of using compile-

time checks, as dynamic/DLL libraries can be changed independent of applications.

`curl_version_info()` returns a pointer to a struct with information about version numbers and various features and in the running version of libcurl. You call it by giving it a special age counter so that libcurl knows the "age" of the libcurl that calls it. The age is a define called `CURLVERSION_NOW` and is a counter that is increased at irregular intervals throughout the curl development. The age number tells libcurl what struct set it can return.

You call the function like this:

```
1 curl_version_info_data *version = curl_version_info( CURLVERSION_NOW );
```

The data will then be pointing at struct that has or at least can have the following layout:

```
1 struct {
2     CURLversion age;           /* see description below */
3
4     /* when 'age' is 0 or higher, the members below also exist: */
5     const char *version;       /* human readable string */
6     unsigned int version_num; /* numeric representation */
7     const char *host;         /* human readable string */
8     int features;            /* bitmask, see below */
9     char *ssl_version;        /* human readable string */
10    long ssl_version_num;     /* not used, always zero */
11    const char *libz_version; /* human readable string */
12    const char * const *protocols; /* protocols */
13
14    /* when 'age' is 1 or higher, the members below also exist: */
15    const char *ares;          /* human readable string */
16    int ares_num;             /* number */
17
18    /* when 'age' is 2 or higher, the member below also exists: */
19    const char *libidn;        /* human readable string */
20
21    /* when 'age' is 3 or higher (7.16.1 or later), the members below also
22       exist */
23    int iconv_ver_num;        /* '_libiconv_version' if iconv support enabled */
24
25    const char *libssh_version; /* human readable string */
26
27    /* when 'age' is 4 or higher, the member below also exists: */
28    unsigned int brotli_ver_num; /* Numeric Brotli version
29                                (MAJOR << 24) | (MINOR << 12) | PATCH */
30    const char *brotli_version; /* human readable string. */
31
32    /* when 'age' is 5 or higher, the member below also exists: */
33    unsigned int nghttp2_ver_num; /* Numeric nghttp2 version
34                                (MAJOR << 16) | (MINOR << 8) | PATCH */
35    const char *nghttp2_version; /* human readable string. */
36    const char *quic_version;   /* human readable quic (+ HTTP/3) library +
37                                version or NULL */
```

```
38  /* when 'age' is 6 or higher, the member below also exists: */
39  const char *cainfo;           /* the built-in default CURLOPT_CAINFO, might
40                                be NULL */
41  const char *capath;          /* the built-in default CURLOPT_CAPATH, might
42                                be NULL */
43
44
45  /* when 'age' is 7 or higher, the member below also exists: */
46  unsigned int zstd_ver_num;   /* Numeric Zstd version
47                                (MAJOR << 24) | (MINOR << 12) | PATCH */
48  const char *zstd_version;    /* human readable string. */
49
50  /* when 'age' is 8 or higher, the member below also exists: */
51  const char *hyper_version;   /* human readable string. */
52
53 } curl_version_info_data;
```

--libcurl

We actively encourage users to first try out the transfer they want to do with the curl command-line tool, and once it works roughly the way you want it to, you append the `--libcurl [filename]` option to the command line and run it again.

The `--libcurl` command-line option will create a C program in the provided file name. That C program is an application that uses libcurl to run the transfer you just had the curl command-line tool do. There are some exceptions and it is not always a 100% match, but you will find that it can serve as an excellent inspiration source for what libcurl options you want or can use and what additional arguments to provide to them.

If you specify the filename as a single dash, as in `--libcurl -` you will get the program written to stdout instead of a file.

As an example, we run a command to get `http://example.com`:

```
1 curl http://example.com --libcurl example.c
```

This creates `example.c` in the current directory, looking similar to this:

```
1 ***** Sample code generated by the curl command-line tool *****
2 * All curl_easy_setopt() options are documented at:
3 * https://curl.se/libcurl/c/curl_easy_setopt.html
4 ****
5 #include <curl/curl.h>
6
7 int main(int argc, char *argv[])
8 {
9     CURLcode ret;
```

```

11 CURL *hnd;
12 hnd = curl_easy_init();
13 curl_easy_setopt(hnd, CURLOPT_URL, "http://example.com");
14 curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
15 curl_easy_setopt(hnd, CURLOPT_USERAGENT, "curl/7.45.0");
16 curl_easy_setopt(hnd, CURLOPT_MAXREDIRS, 50L);
17 curl_easy_setopt(hnd, CURLOPT_SSH_KNOWNHOSTS, "/home/daniel/.ssh/known_hosts");
18 curl_easy_setopt(hnd, CURLOPT_TCP_KEEPALIVE, 1L);
19
20 /* Here is a list of options the curl code used that cannot get generated
21    as source easily. You may select to either not use them or implement
22    them yourself.
23
24 CURLOPT_WRITEDATA set to a objectpointer
25 CURLOPT_WRITEFUNCTION set to a functionpointer
26 CURLOPT_READDATA set to a objectpointer
27 CURLOPT_READFUNCTION set to a functionpointer
28 CURLOPT_SEEKDATA set to a objectpointer
29 CURLOPT_SEEKFUNCTION set to a functionpointer
30 CURLOPT_ERRORBUFFER set to a objectpointer
31 CURLOPT_STDERR set to a objectpointer
32 CURLOPT_HEADERFUNCTION set to a functionpointer
33 CURLOPT_HEADERDATA set to a objectpointer
34
35 */
36
37 ret = curl_easy_perform(hnd);
38
39 curl_easy_cleanup(hnd);
40 hnd = NULL;
41
42 return (int)ret;
43 }
44 **** End of sample code ****/

```

Header files

There is only ever one header your libcurl using application needs to include:

```
1 #include <curl/curl.h>
```

That file in turn includes a few other public header files but you can pretend they do not exist. (Historically speaking, we started out slightly different but over time we have stabilized around this form of only using a single one for includes.)

Global initialization

Before you do anything libcurl related in your program, you should do a global libcurl initialize call with `curl_global_init()`. This is necessary because some underlying libraries that libcurl might be using need a call ahead to get setup and initialized properly.

`curl_global_init()` is, unfortunately, not thread safe, so you must ensure that you only do it once and never simultaneously with another call. It initializes global state so you should only call it once, and once your program is completely done using libcurl you can call `curl_global_cleanup()` to free and clean up the associated global resources the init call allocated.

libcurl is built to handle the situation where you skip the `curl_global_init()` call, but it does so by calling it itself instead (if you did not do it before any actual file transfer starts) and it then uses its own defaults. But beware that it is still not thread safe even then, so it might cause some "interesting" side effects for you. It is much better to call `curl_global_init()` yourself in a controlled manner.

multi-threading

libcurl is thread safe but has no internal thread synchronization. You may have to provide your own locking or change options to properly use libcurl threaded. Exactly what is required depends on how libcurl was built. Please refer to the [libcurl thread safety](#) web page, which contains the latest information.

curl easy options

You set options in the easy handle to control how that transfer is going to be done, or in some cases you can actually set options and modify the transfer's behavior while it is in progress. You set options with `curl_easy_setopt()` and you provide the handle, the option you want to set and the argument to the option. All options take exactly one argument and you must always pass exactly three parameters to the `curl_easy_setopt()` calls.

Since the `curl_easy_setopt()` call accepts several hundred different options and the various options accept a variety of different types of arguments, it is important to read up on the specifics and provide exactly the argument type the specific option supports and expects. Passing in the wrong type can lead to unexpected side-effects or hard to understand hiccups.

The perhaps most important option that every transfer needs, is the URL. libcurl cannot perform a transfer without knowing which URL it concerns so you must tell it. The URL option name is `CURLOPT_URL` as all options are prefixed with `CURLOPT_` and then the descriptive name — all using uppercase letters. An example line setting the URL to get the `http://example.com` HTTP contents could look like:

```
1 CURLcode ret = curl_easy_setopt(easy, CURLOPT_URL, "http://example.com");
```

Again: this only sets the option in the handle. It will not do the actual transfer or anything. It will just tell libcurl to copy the string and if that works it returns OK.

It is, of course, good form to check the return code to see that nothing went wrong.

Get options

There is no way to extract the values previously set with `curl_easy_setopt()`. If you need to be able to extract the information again that you set earlier, we encourage you to keep track of that data yourself in your application.

Set numerical options

Since `curl_easy_setopt()` is a vararg function where the 3rd argument can use different types depending on the situation, normal C language type conversion cannot be done. So you **must** make sure that you truly pass a `long` and not an `int` if the documentation tells you so. On architectures where they are the same size, you may not get any problems but not all work like that. Similarly, for options that accept a `curl_off_t` type, it is **crucial** that you pass in an argument using that type and no other.

Enforce a `long`:

```
1 curl_easy_setopt(handle, CURLOPT_TIMEOUT, 5L); /* 5 seconds timeout */
```

Enforce a `curl_off_t`:

```
1 curl_off_t no_larger_than = 0x50000;
2 curl_easy_setopt(handle, CURLOPT_MAXFILESIZE_LARGE, no_larger_than);
```

Set string options

There are currently over 80 options for `curl_easy_setopt()` that accept a string as its third argument.

When a string is set in a handle, libcurl immediately copies that data so that the application does not have to keep the data around for the time the transfer is being done - **with one notable exception:** `CURLOPT_POSTFIELDS`.

Set a URL in the handle:

```
1 curl_easy_setopt(handle, CURLOPT_URL, "https://example.com");
```

CURLOPT_POSTFIELDS

The exception to the rule that libcurl always copies data, `CURLOPT_POSTFIELDS` only stores the pointer

to the data, meaning an application using this option **must** keep the memory around for the entire duration of the associated transfer.

If that is problematic, an alternative is to instead use `CURLOPT_COPYPOSTFIELDS` which copies the data. If the data is binary and does not stop at the first presence of a null byte, make sure that `CURLOPT_POSTFIELDSIZE` is set *before* this option is used.

Why?

The reason `CURLOPT_POSTFIELDS` is an exception is due to legacy. Originally (before curl 7.17.0), libcurl did not copy *any* string arguments and when this current behavior was introduced, this option could not be converted over without breaking behavior so it had to keep working like before. Which now sticks out, as no other option does...

C++

If you use libcurl from a C++ program, it is important to remember that you cannot pass in a string object where libcurl expects a string. It has to be a null terminated C string. Usually you can make this happen with the `c_str()` method.

For example, keep the URL in a string object and set that in the handle:

```
1 std::string url("https://example.com/");
2 curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
```

TLS options

At the time of writing this, there are no less than **forty** different options for `curl_easy_setopt` that are dedicated for controlling how libcurl does SSL and TLS.

Transfers done using TLS use safe defaults but since curl is used in many different scenarios and setups, chances are you will end up in situations where you want to change those behaviors.

Protocol version

With `CURLOPT_SSLVERSION'` and `CURLOPT_PROXY_SSLVERSION`` you can specify which SSL or TLS protocol range that is acceptable to you. Traditionally SSL and TLS protocol versions have been found detect and unsuitable for use over time and even if curl itself will raise its default lower version over time you might want to opt for only using the latest and most security protocol versions.

These options take a lowest acceptable version and optionally a maximum. If the server cannot negotiate a connection with that condition, the transfer will fail.

Example:

```
1 curl_easy_setopt(easy, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
```

Protocol details and behavior

You can select what ciphers to use by setting `CURLOPT_SSL_CIPHER_LIST` and `CURLOPT_PROXY_SSL_CIPHER_LIST`.

You can ask to enable SSL "False Start" with `CURLOPT_SSL_FALSESTART`, and there are a few other behavior changes to tweak using `CURLOPT_SSL_OPTIONS`.

Verification

A TLS-using client needs to verify that the server it speaks to is the correct and trusted one. This is done by verifying that the server's certificate is signed by a Certificate Authority (CA) for which curl has a public key for and that the certificate contains the server's name. Failing any of these checks will cause the transfer to fail.

For development purposes and for experimenting, curl allows an application to switch off either or both of these checks for the server or for an HTTPS proxy.

- `CURLOPT_SSL_VERIFYPEER` controls the check that the certificate is signed by a trusted CA.
- `CURLOPT_SSL_VERIFYHOST` controls the check for the name within the certificate.
- `CURLOPT_PROXY_SSL_VERIFYPEER` is the proxy version of `CURLOPT_SSL_VERIFYPEER`.
- `CURLOPT_PROXY_SSL_VERIFYHOST` is the proxy version of `CURLOPT_SSL_VERIFYHOST`.

Optionally, you can tell curl to verify the certificate's public key against a known hash using `CURLOPT_PINNEDPUBLICKEY` or `CURLOPT_PROXY_PINNEDPUBLICKEY`. Here too, a mismatch will cause the transfer to fail.

Authentication

TLS Client certificates

When using TLS and the server asks the client to authenticate using certificates, you typically specify the private key and the corresponding client certificate using `CURLOPT_SSLKEY` and `CURLOPT_SSLCERT`. The password for the key is usually also required to be set, with `CURLOPT_SSLKEYPASSWD`.

Again, the same set of options exist separately for connections to HTTPS proxies:

`CURLOPT_PROXY_SSLKEY`, `CURLOPT_PROXY_SSLCERT` etc.

TLS auth

TLS connections offer a (rarely used) feature called Secure Remote Passwords. Using this, you authenticate the connection for the server using a name and password and the options are called `CURLOPT_TLSAUTH_USERNAME` and `CURLOPT_TLSAUTH_PASSWORD`.

STARTTLS

For protocols that are using the STARTTLS method to upgrade the connection to TLS (FTP, IMAP, POP3, and SMTP), you usually tell curl to use the non-TLS version of the protocol when specifying a URL and then ask curl to enable TLS with the `CURLOPT_USE_SSL` option.

This option allows a client to let curl continue if it cannot upgrade to TLS, but that is not a recommend path to walk as then you might be using an insecure protocol without properly noticing.

```
1 /* require use of SSL for this, or fail */
2 curl_easy_setopt(curl, CURLOPT_USE_SSL, CURLUSESSL_ALL);
```

All options

This is a table of a complete list of all available options for `curl_easy_setopt()` as of what will be present in the 7.83.0 release, April 2022.

Option	Purpose
<code>CURLOPT_ABSTRACT_UNIX_SOCKET</code>	abstract Unix domain socket
<code>CURLOPT_ACCEPTTIMEOUT_MS</code>	timeout waiting for FTP server to connect back
<code>CURLOPT_ACCEPT_ENCODING</code>	automatic decompression of HTTP downloads
<code>CURLOPT_ADDRESS_SCOPE</code>	scope id for IPv6 addresses
<code>CURLOPT_ALTSVC</code>	alt-svc cache file name
<code>CURLOPT_ALTSVC_CTRL</code>	control alt-svc behavior
<code>CURLOPT_APPEND</code>	append to the remote file
<code>CURLOPT_AUTOREFERER</code>	automatically update the referer header
<code>CURLOPT_AWS_SIGV4</code>	V4 signature
<code>CURLOPT_BUFFERSIZE</code>	receive buffer size

CURLOPT_CAINFO	path to Certificate Authority (CA) bundle
CURLOPT_CAINFO_BLOB	Certificate Authority (CA) bundle in PEM format
CURLOPT_CAPATH	directory holding CA certificates
CURLOPT_CERTINFO	request SSL certificate information
CURLOPT_CHUNK_BGN_FUNCTION	callback before a transfer with FTP wildcardmatch
CURLOPT_CHUNK_DATA	pointer passed to the FTP chunk callbacks
CURLOPT_CHUNK_END_FUNCTION	callback after a transfer with FTP wildcardmatch
CURLOPT_CLOSESOCKETDATA	pointer passed to the socket close callback
CURLOPT_CLOSESOCKETFUNCTION	callback to socket close replacement
CURLOPT_CONNECTTIMEOUT	timeout for the connect phase

Get option information

curl offers an API, a set of functions really, that allow applications to get information about all currently supported *options*. It does not return *the values* for the options, but it rather informs about name, ID and type of the option.

Iterate over all options

A modern libcurl supports nearly 300 different options. With the use of `curl_easy_option_by_next()` an application can iterate over all the known options and return a pointer to a `struct curl_easyoption` for them.

This function only returns information about options that this exact libcurl build knows about. Other options may exist in newer libcurl builds, or in builds that enable/disable options differently at build-time.

Example, iterate over all available options:

```
1 const struct curl_easyoption *opt;
2 opt = curl_easy_option_by_next(NULL);
3 while(opt) {
4     printf("Name: %s\n", opt->name);
5     opt = curl_easy_option_by_next(opt);
6 }
```

Find a specific option by name

Given a specific easy option name, you can ask libcurl to return a pointer to a `struct curl_eeasyoption` for it. The name should be provided without the `CURLOPT_` prefix.

As an example, an application can ask libcurl about the `CURLOPT_VERBOSE` option like this:

```
1 const struct curl_easyoption *opt = curl_easy_option_by_name("VERBOSE");
2 if(opt) {
3     printf("This option wants CURLOPT %x\n", (int)opt->id);
4 }
```

Find a specific option by ID

Given a specific easy option ID, you can ask libcurl to return a pointer to a `struct curl_eeasyoption` for it. The "ID" is the `CURLOPT_-prefixed symbol` as provided in the public `curl/curl.h` header file.

An application can ask libcurl for the name of the `CURLOPT_VERBOSE` option like this:

```
1 const struct curl_easyoption *opt =
2     curl_easy_option_by_id(CURLOPT_VERBOSE);
3 if(opt) {
4     printf("This option has the name: %s\n", opt->name);
5 }
```

The `curl_easyoption` struct

```
1 struct curl_easyoption {  
2     const char *name;  
3     CURLoption id;  
4     curl_easystyle type;  
5     unsigned int flags;  
6 };
```

There is only one bit with a defined meaning in 'flags': if `CURLOPT_FLAG_ALIAS` is set, it means that that option is an "alias". A name provided for backwards compatibility that is nowadays rather served by an option with another name. If you lookup the ID for an alias, you will get the new canonical name for that option.

CURLcode return codes

Many libcurl functions return a CURLcode. That is a special libcurl typedefed variable for error codes. It returns `CURLE_OK` (which has the value zero) if everything is fine and dandy and it returns a non-zero number if a problem was detected. There are almost one hundred `CURLcode` errors in use, and you can find them all in the `curl/curl.h` header file and documented in the libcurl-errors man page.

You can convert a CURLcode into a human readable string with the `curl_easy_strerror()` function—but be aware that these errors are rarely phrased in a way that is suitable for anyone to expose in a UI or to an end user:

```
1 const char *str = curl_easy_strerror( error );  
2 printf("libcurl said %s\n", str);
```

Another way to get a slightly better error text in case of errors is to set the `CURLOPT_ERRORBUFFER` option to point out a buffer in your program and then libcurl will store a related error message there before it returns an error:

```
1 char error[CURL_ERROR_SIZE]; /* needs to be at least this big */  
2 CURLcode ret = curl_easy_setopt(handle, CURLOPT_ERRORBUFFER, error);
```

Verbose operations

Okay, we just showed how to get the error as a human readable text as that is an excellent help to figure out what went wrong in a particular transfer and often explains why it can be done like that or what the problem

is for the moment.

The next lifesaver when writing libcurl applications that everyone needs to know about and needs to use extensively, at least while developing libcurl applications or debugging libcurl itself, is to enable "verbose mode" with `CURLOPT_VERBOSE`:

```
1 CURLcode ret = curl_easy_setopt(handle, CURLOPT_VERBOSE, 1L);
```

When libcurl is told to be verbose it will mention transfer-related details and information to stderr while the transfer is ongoing. This is awesome to figure out why things fail and to learn exactly what libcurl does when you ask it different things. You can redirect the output elsewhere by changing stderr with `CURLOPT_STDERR` or you can get even more info in a fancier way with the debug callback (explained further in a later section).

Trace everything

Verbose is certainly fine, but sometimes you need more. libcurl also offers a trace callback that in addition to showing you all the stuff the verbose mode does, it also passes on *all* data sent and received so that your application gets a full trace of everything.

The sent and received data passed to the trace callback is given to the callback in its unencrypted form, which can be handy when working with TLS or SSH based protocols when capturing the data off the network for debugging is not practical.

When you set the `CURLOPT_DEBUGFUNCTION` option, you still need to have `CURLOPT_VERBOSE` enabled but with the trace callback set libcurl will use that callback instead of its internal handling.

The trace callback should match a prototype like this:

```
1 int my_trace(CURL *handle, curl_infotype type, char *data, size_t size,
2                 void *user);
```

handle is the easy handle it concerns, **type** describes the particular data passed to the callback (data in/out, header in/out, TLS data in/out and "text"), **data** is a pointer pointing to the data being **size** number of bytes. **user** is the custom pointer you set with `CURLOPT_DEBUGDATA`.

The data pointed to by **data** *will not* be zero terminated, but will be exactly of the size as told by the **size** argument.

The callback must return 0 or libcurl will consider it an error and abort the transfer.

On the curl website, we host an example called [debug.c](#) that includes a simple trace function to get inspiration from.

There are also additional details in the [CURLOPT_DEBUGFUNCTION man page](#).

Caches

libcurl caches different information in order to help subsequent transfers to perform faster. There are three key caches: DNS, connections and TLS sessions.

When the multi interface is used, these caches are by default shared among all the easy handles that are added to that single multi handle, and when the easy interface is used they are kept within that handle.

You can instruct libcurl to share some of the caches with the [share interface](#).

DNS cache

When libcurl resolves a host name to one or more IP addresses, that is stored in the DNS cache so that subsequent transfers in the near term will not have to redo the same resolve again. A name resolve can easily take several hundred milliseconds and sometimes even much longer.

By default, each such host name is stored in the cache for 60 seconds (changeable with `CURLOPT_DNS_CACHE_TIMEOUT`).

libcurl does in fact not usually know what the TTL (Time To Live) value is for DNS entries, as that is generally not exposed in the system function calls it uses for this purpose, so increasing this value come with a risk that libcurl keeps using stale addresses longer periods than necessary.

Connection cache

Also sometimes referred to as the connection pool. This is a collection of previously used connections that instead of being closed after use, are kept around alive so that subsequent transfers that are targeting the same host name and have several other checks also matching, can use them instead of creating a new connection.

A reused connection usually saves having to a DNS lookup, setting up a TCP connection, do a TLS handshake and more.

Connections are only reused if the name is identical. Even if two different host names resolve to the same IP addresses, they will still always use two separate connections with libcurl.

Since the connection reuse is based on the host name and the DNS resolve phase is entirely skipped when a connection is reused for a transfer, libcurl will not know the current state of the host name in DNS as it can in fact change IP over time while the connection might survive and continue to get reused over the original IP address.

The size of the connection cache - the number of live connections to keep there - can be set with `CURLOPT_MAXCONNECTS` (default is 5) for easy handles and `CURLMOPT_MAXCONNECTS` for multi handles. The default size for multi handles is 4 times the number of easy handles added.

TLS session cache

TLS session IDs and tickets are special TLS mechanisms that a client can pass to a server to shortcut subsequent TLS handshakes to a server it previously established a connection to.

libcurl caches session IDs and tickets associated with host names and port numbers, so if a subsequent connection attempt is made to a host for which libcurl has a cached ID or ticket, using that can greatly decrease the TLS handshake process and therefore the time needed until completion.

libcurl examples

The native API for libcurl is in C so this chapter is focused on examples written in C. But since many language bindings for libcurl are thin, they usually expose more or less the same functions and thus they can still be interesting and educational for users of other languages, too.

- [Get a simple HTTP page](#)
- [Get a page in memory](#)
- [Submit a login form over HTTP](#)
- [Get an FTP directory listing](#)
- [Non-blocking HTTP form-post](#)

Get a simple HTTP page

This example just fetches the HTML from a given URL and sends it to stdout. Possibly the simplest libcurl program you can write.

By replacing the URL this will of course be able to get contents over other supported protocols as well.

Getting the output sent to stdout is a default behavior and usually not what you actually want. Most applications will instead install a [write callback](#) to have receive the data that arrives.

```
1 #include <stdio.h>
2 #include <curl/curl.h>
3
4 int main(void)
5 {
6     CURL *curl;
7     CURLcode res;
8
9     curl = curl_easy_init();
10    if(curl) {
11        curl_easy_setopt(curl, CURLOPT_URL, "http://example.com/");
12
13        /* Perform the request, res will get the return code */
14        res = curl_easy_perform(curl);
```

```

15  /* Check for errors */
16  if(res != CURLE_OK)
17      fprintf(stderr, "curl_easy_perform() failed: %s\n",
18              curl_easy_strerror(res));
19
20  /* always cleanup */
21  curl_easy_cleanup(curl);
22 }
23 return 0;
24 }
```

Get a response into memory

This example is a variation of the former that instead of sending the received data to stdout (which often is not what you want), this example instead stores the incoming data in a memory buffer that is enlarged as the incoming data grows.

It accomplishes this by using a [write callback](#) to receive the data.

This example uses a fixed URL string with a set URL scheme, but you can of course change this to use any other supported protocol and then get a resource from that instead.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include <curl/curl.h>
6
7 struct MemoryStruct {
8     char *memory;
9     size_t size;
10 };
11
12 static size_t
13 WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)
14 {
15     size_t realsize = size * nmemb;
16     struct MemoryStruct *mem = (struct MemoryStruct *)userp;
17
18     mem->memory = realloc(mem->memory, mem->size + realsize + 1);
19     if(mem->memory == NULL) {
20         /* out of memory */
21         printf("not enough memory (realloc returned NULL)\n");
22         return 0;
23     }
24
25     memcpy(&(mem->memory[mem->size]), contents, realsize);
26     mem->size += realsize;
27     mem->memory[mem->size] = 0;
28 }
```

```
29     return realsize;
30 }
31
32 int main(void)
33 {
34     CURL *curl_handle;
35     CURLcode res;
36
37     struct MemoryStruct chunk;
38
39     chunk.memory = malloc(1); /* will be grown as needed by the realloc above */
40     chunk.size = 0; /* no data at this point */
41
42     curl_global_init(CURL_GLOBAL_ALL);
43
44     /* init the curl session */
45     curl_handle = curl_easy_init();
46
47     /* specify URL to get */
48     curl_easy_setopt(curl_handle, CURLOPT_URL, "https://www.example.com/");
49
50     /* send all data to this function */
51     curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
52
53     /* we pass our 'chunk' struct to the callback function */
54     curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
55
56     /* some servers do not like requests that are made without a user-agent
57      field, so we provide one */
58     curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
59
60     /* get it! */
61     res = curl_easy_perform(curl_handle);
62
63     /* check for errors */
64     if(res != CURLE_OK) {
65         fprintf(stderr, "curl_easy_perform() failed: %s\n",
66                 curl_easy_strerror(res));
67     }
68     else {
69         /*
70          * Now, our chunk.memory points to a memory block that is chunk.size
71          * bytes big and contains the remote file.
72          *
73          * Do something nice with it!
74         */
75
76         printf("%lu bytes retrieved\n", (long)chunk.size);
77     }
78
79     /* cleanup curl stuff */
80     curl_easy_cleanup(curl_handle);
81 }
```

```

82 free(chunk.memory);
83
84 /* we are done with libcurl, so clean it up */
85 curl_global_cleanup();
86
87 return 0;
88 }

```

Submit a login form over HTTP

A login submission over HTTP is usually a matter of figuring out exactly what data to submit in a POST and to which target URL to send it.

Once logged in, the target URL can be fetched if the proper cookies are used. As many login-systems work with HTTP redirects, we ask libcurl to follow such if they arrive.

Some login forms will make it more complicated and require that you got cookies from the page showing the login form etc, so if you need that you may want to extend this code a little bit.

By passing in a non-existing cookie file, this example will enable the cookie parser so incoming cookies will be stored when the response from the login response arrives and then the subsequent request for the resource will use those and prove to the server that we are in fact correctly logged in.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <curl/curl.h>
4
5 int main(void)
6 {
7     CURL *curl;
8     CURLcode res;
9
10    static const char *postthis = "user=daniel&password=monkey123";
11
12    curl = curl_easy_init();
13    if(curl) {
14        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/login.cgi");
15        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, postthis);
16        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L); /* redirects! */
17        curl_easy_setopt(curl, CURLOPT_COOKIEFILE, ""); /* no file */
18        res = curl_easy_perform(curl);
19        /* Check for errors */
20        if(res != CURLE_OK)
21            fprintf(stderr, "curl_easy_perform() failed: %s\n",
22                    curl_easy_strerror(res));
23    } else {
24        /*
25         * After the login POST, we have received the new cookies. Switch
26         * over to a GET and ask for the login-protected URL.

```

```

28     curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/file");
29     curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L); /* no more POST */
30     res = curl_easy_perform(curl);
31     /* Check for errors */
32     if(res != CURLE_OK)
33         fprintf(stderr, "second curl_easy_perform() failed: %s\n",
34                 curl_easy_strerror(res));
35 }
36 /* always cleanup */
37 curl_easy_cleanup(curl);
38 }
39 return 0;
40 }
```

Get an FTP directory listing

This example just fetches the FTP directory output from the given URL and sends it to stdout. The trailing slash in the URL is what makes libcurl treat it as a directory.

```

1 #include <curl/curl.h>
2
3 int main(void)
4 {
5     CURL *curl;
6     CURLcode res;
7
8     curl_global_init(CURL_GLOBAL_DEFAULT);
9
10    curl = curl_easy_init();
11    if(curl) {
12        /*
13         * Make the URL end with a trailing slash!
14         */
15        curl_easy_setopt(curl, CURLOPT_URL, "ftp://ftp.example.com/");
16
17        res = curl_easy_perform(curl);
18
19        /* always cleanup */
20        curl_easy_cleanup(curl);
21
22        if(CURLE_OK != res) {
23            /* we failed */
24            fprintf(stderr, "curl told us %d\n", res);
25        }
26    }
27
28    curl_global_cleanup();
29
30    return 0;
31 }
```

```
31 }
```

Non-blocking HTTP form-post

This examples makes a multipart form-post using the multi interface.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <sys/time.h>
4
5 #include <curl/curl.h>
6
7 int main(void)
8 {
9     CURL *curl;
10
11    CURLM *multi_handle;
12    int still_running = 0;
13
14    curl_mime *form = NULL;
15    curl_multipart *field = NULL;
16    struct curl_slist *headerlist = NULL;
17    static const char buf[] = "Expect:";
18
19    curl = curl_easy_init();
20    multi_handle = curl_multi_init();
21
22    if(curl && multi_handle) {
23        /* Create the form */
24        form = curl_mime_init(curl);
25
26        /* Fill in the file upload field */
27        field = curl_mime_addpart(form);
28        curl_mime_name(field, "sendfile");
29        curl_mime_filedata(field, "multi-post.c");
30
31        /* Fill in the filename field */
32        field = curl_mime_addpart(form);
33        curl_mime_name(field, "filename");
34        curl_mime_data(field, "multi-post.c", CURL_ZERO_TERMINATED);
35
36        /* Fill in the submit field too, even if this is rarely needed */
37        field = curl_mime_addpart(form);
38        curl_mime_name(field, "submit");
39        curl_mime_data(field, "send", CURL_ZERO_TERMINATED);
40
41        /* initialize custom header list (stating that Expect: 100-continue is not
42           wanted */
```

```

44     headerlist = curl_slist_append(headerlist, buf);
45     /* what URL that receives this POST */
46     curl_easy_setopt(curl, CURLOPT_URL, "https://www.example.com/upload.cgi");
47     curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
48
49     curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headerlist);
50     curl_easy_setopt(curl, CURLOPT_MIMEPOST, form);
51
52     curl_multi_add_handle(multi_handle, curl);
53
54     do {
55         CURLMcode mc = curl_multi_perform(multi_handle, &still_running);
56
57         if(still_running)
58             /* wait for activity, timeout or "nothing" */
59             mc = curl_multi_poll(multi_handle, NULL, 0, 1000, NULL);
60
61         if(mc)
62             break;
63     } while(still_running);
64
65     curl_multi_cleanup(multi_handle);
66
67     /* always cleanup */
68     curl_easy_cleanup(curl);
69
70     /* then cleanup the form */
71     curl_mime_free(form);
72
73     /* free slist */
74     curl_slist_free_all(headerlist);
75 }
76 return 0;
77 }
```

for C++ programmers

libcurl provides a C API. C and C++ are similar but not the same. There are a few things to keep in mind when using libcurl in C++.

Strings are C strings, not C++ string objects

When you pass strings to libcurl's APIs that accept `char *` that means you cannot pass in C++ strings or objects to those functions.

For example, if you build a string with C++ and then want that string used as a URL:

```
1 std::string url = "https://example.com/foo.asp?name=" + i;
2 curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
```

Callback considerations

Since libcurl is a C library, it does not know anything about C++ member functions or objects. You can overcome this "limitation" with relative ease using for a static member function that is passed a pointer to the class.

Here's an example of a write callback using a C++ method as callback:

```
1 // f is the pointer to your object.
2 static size_t YourClass::func(void *buffer, size_t sz, size_t n, void *f)
3 {
4     // Call non-static member function.
5     static_cast<YourClass*>(f)->nonStaticFunction();
6 }
7
8 // This is how you pass pointer to the static function:
9 curl_easy_setopt(hcurl, CURLOPT_WRITEFUNCTION, YourClass::func);
10 curl_easy_setopt(hcurl, CURLOPT_WRITEDATA, this);
```

HTTP with libcurl

HTTP is by far the most commonly used protocol by libcurl users and libcurl offers countless ways of modifying such transfers. See the [HTTP protocol basics](#) for some basics on how the HTTP protocol works.

HTTPS

Doing HTTPS is typically done the same way as for HTTP as the extra security layer and server verification etc is done automatically and transparently by default. Just use the `https://` scheme in the URL.

HTTPS is HTTP with TLS on top. See also the [TLS options](#) section.

HTTP proxy

See [using Proxies with libcurl](#)

Sections

- [HTTP responses](#)
- [HTTP requests](#)
- [HTTP versions](#)
- [HTTP ranges](#)
- [HTTP authentication](#)
- [Cookies with libcurl](#)
- [Download](#)
- [Upload](#)
- [Multiplexing](#)
- [HSTS](#)

HTTP responses

Every HTTP request includes an HTTP response. An HTTP response is a set of metadata and a response body, where the body can occasionally be zero bytes and thus nonexistent. An HTTP response will however always have response headers.

Response body

The response body will be passed to the [write callback](#) and the response headers to the [header callback](#).

Virtually all libcurl-using applications need to set at least one of those callbacks instructing libcurl what to do with received headers and data.

Response meta-data

libcurl offers the `curl_easy_getinfo()` function that allows an application to query libcurl for information from the previously performed transfer.

Sometimes an application just want to know the size of the data. The size of a response as *told by the server headers* can be extracted with `curl_easy_getinfo()` like this:

```
1 curl_off_t size;
2 curl_easy_getinfo(curl, CURLINFO_CONTENT_LENGTH_DOWNLOAD_T, &size);
```

If you can wait until after the transfer is already done, which also is a more reliable way since not all URLs

will provide the size up front (like for example for servers that generate content on demand) you can instead ask for the amount of downloaded data in the most recent transfer.

```
1 curl_off_t size;
2 curl_easy_getinfo(curl, CURLINFO_SIZE_DOWNLOAD_T, &size);
```

HTTP response code

Every HTTP response starts off with a single line that contains the HTTP response code. It is a three digit number that contains the server's idea of the status for the request. The numbers are detailed in the HTTP standard specifications but they are divided into ranges that work like this:

Code	Meaning
1xx	Transient code, a new one follows
2xx	Things are OK
3xx	The content is somewhere else
4xx	Failed because of a client problem
5xx	Failed because of a server problem

You can extract the response code after a transfer like this

```
1 long code;
2 curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &code);
```

About HTTP response code "errors"

While the response code numbers can include numbers (in the 4xx and 5xx ranges) which the server uses to signal that there was an error processing the request, it is important to realize that this will not cause libcurl to return an error.

When libcurl is asked to perform an HTTP transfer it will return an error if that HTTP transfer fails. However, getting an HTTP 404 or the like back is not a problem for libcurl. It is not an HTTP transfer error. A user might be writing a client for testing a server's HTTP responses.

If you insist on curl treating HTTP response codes from 400 and up as errors, libcurl offers the `CURLOPT_FAILONERROR` option that if set instructs curl to return `CURLE_HTTP_RETURNED_ERROR` in this case. It will then return error as soon as possible and not deliver the response body.

HTTP requests

An HTTP request is what curl sends to the server when it tells the server what to do. When it wants to get data or send data. All transfers involving HTTP start with an HTTP request.

An HTTP request contains a method, a path, HTTP version and a set of request headers. And of course a libcurl using application can tweak all those fields.

Request method

Every HTTP request contains a "method", sometimes referred to as a "verb". It is usually something like GET, HEAD, POST or PUT but there are also more esoteric ones like DELETE, PATCH and OPTIONS.

Usually when you use libcurl to set up and perform a transfer the specific request method is implied by the options you use. If you just ask for a URL, it means the method will be `GET` while if you set for example `CURLOPT_POSTFIELDS` that will make libcurl use the `POST` method. If you set `CURLOPT_UPLOAD` to true, libcurl will send a `PUT` method in its HTTP request and so on. Asking for `CURLOPT_NOBODY` will make libcurl use `HEAD`.

However, sometimes those default HTTP methods are not good enough or simply not the ones you want your transfer to use. Then you can instruct libcurl to use the specific method you like with `CURLOPT_CUSTOMREQUEST`. For example, you want to send a `DELETE` method to the URL of your choice:

```
1 curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "DELETE");
2 curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/file.txt");
```

The `CURLOPT_CUSTOMREQUEST` setting should only be the single keyword to use as method in the HTTP request line. If you want to change or add additional HTTP request headers, see the following section.

Customize HTTP request headers

When libcurl issues HTTP requests as part of performing the data transfers you have asked it to, it will of course send them off with a set of HTTP headers that are suitable for fulfilling the task given to it.

If just given the URL `http://localhost/file1.txt`, libcurl sends the following request to the server:

```
1 GET /file1.txt HTTP/1.1
2 Host: localhost
3 Accept: */*
```

If you instruct your application to also set `CURLOPT_POSTFIELDS` to the string "foobar" (6 letters, the

number of bytes is included in the definition below), libcurl would send the following headers:

```
1 POST /file1.txt HTTP/1.1
2 Host: localhost
3 Accept: /*
4 Content-Length: 6
5 Content-Type: application/x-www-form-urlencoded
```

If you are not pleased with the default set of headers libcurl sends, the application has the power to add, change or remove headers in the HTTP request.

Add a header

To add a header that would not otherwise be in the request, add it with `CURLOPT_HTTPHEADER`. Suppose you want a header called `Name:` that contains `Mr. Smith`:

```
1 struct curl_slist *list = NULL;
2 list = curl_slist_append(list, "Name: Mr Smith");
3 curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
4 curl_easy_perform(curl);
5 curl_slist_free_all(list); /* free the list again */
```

Change a header

If one of those default headers are not to your satisfaction you can alter them. Like if you think the default `Host:` header is wrong (even though it is derived from the URL you give libcurl), you can tell libcurl your own:

```
1 struct curl_slist *list = NULL;
2 list = curl_slist_append(list, "Host: Alternative");
3 curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
4 curl_easy_perform(curl);
5 curl_slist_free_all(list); /* free the list again */
```

Remove a header

When you think libcurl uses a header in a request that you really think it should not, you can easily tell it to just remove it from the request. Like if you want to take away the `Accept:` header. Just provide the header name with nothing to the right sight of the colon:

```
1 struct curl_slist *list = NULL;
2 list = curl_slist_append(list, "Accept:");
3 curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
4 curl_easy_perform(curl);
5 curl_slist_free_all(list); /* free the list again */
```

Provide a header without contents

As you may then have noticed in the above sections, if you try to add a header with no contents on the right side of the colon, it will be treated as a removal instruction and it will instead completely inhibit that header from being sent. If you instead *truly* want to send a header with zero contents on the right side, you need to use a special marker. You must provide the header with a semicolon instead of a proper colon. Like `Header; .` So if you want to add a header to the outgoing HTTP request that is just `Moo:` with nothing following the colon, you could write it like:

```
1 struct curl_slist *list = NULL;
2 list = curl_slist_append(list, "Moo;");
3 curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
4 curl_easy_perform(curl);
5 curl_slist_free_all(list); /* free the list again */
```

Referrer

The `Referer:` header (yes, it is misspelled) is a standard HTTP header that tells the server from which URL the user-agent was directed from when it arrived at the URL it now requests. It is a normal header so you can set it yourself with the `CURLOPT_HEADER` approach as shown above, or you can use the shortcut known as `CURLOPT_REFERER`. Like this:

```
1 curl_easy_setopt(curl, CURLOPT_REFERER, "https://example.com/fromhere/");
2 curl_easy_perform(curl);
```

Automatic referrer

When libcurl is asked to follow redirects itself with the `CURLOPT_FOLLOWLOCATION` option, and you still want to have the `Referer:` header set to the correct previous URL from where it did the redirect, you can ask libcurl to set that by itself:

```
1 curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
2 curl_easy_setopt(curl, CURLOPT_AUTOREFERER, 1L);
3 curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/redirected.cgi");
4 curl_easy_perform(curl);
```

HTTP versions

As any other Internet protocol, the HTTP protocol has kept evolving over the years and now there are clients and servers distributed over the world and over time that speak different versions with varying levels of success. So in order to get libcurl to work with the URLs you pass in libcurl offers ways for you to specify

which HTTP version that request and transfer should use. libcurl is designed in a way so that it tries to use the most common, the most sensible if you want, default values first but sometimes that is not enough and then you may need to instruct libcurl what to do.

Since mid 2016, libcurl defaults to use HTTP/2 for HTTPS servers if you have a libcurl that has HTTP/2 abilities built-in, libcurl will attempt to use HTTP/2 automatically or fall back to 1.1 in case the negotiation failed. Non-HTTP/2 capable libcurls get 1.1 over HTTPS by default. Plain HTTP requests still default to HTTP/1.1.

If the default behavior is not good enough for your transfer, the `CURLOPT_HTTP_VERSION` option is there for you.

Option	Description
<code>CURL_HTTP_VERSION_NONE</code>	Reset back to default behavior
<code>CURL_HTTP_VERSION_1_0</code>	Enforce use of the legacy HTTP/1.0 protocol version
<code>CURL_HTTP_VERSION_1_1</code>	Do the request using the HTTP/1.1 protocol version
<code>CURL_HTTP_VERSION_2_0</code>	Attempt to use HTTP/2
<code>CURL_HTTP_VERSION_2TLS</code>	Attempt to use HTTP/2 on HTTPS connections only, otherwise do HTTP/1.1
<code>CURL_HTTP_VERSION_2_PRIOR_KNOWLEDGE</code>	Use HTTP/2 straight away without "upgrading" from 1.1. It requires that you know that this server OK with it.
<code>CURL_HTTP_VERSION_3</code>	Use HTTP/3 straight away. It requires that you know that this server is OK with it.

Version 2 not mandatory

When asking libcurl to use HTTP/2, it is an ask not a requirement. libcurl will then allow the server to select to use HTTP/1.1 or HTTP/2 and that is what decides which protocol that is ultimately used.

Version 3 is mandatory

When asking libcurl to use HTTP/3 with the `CURL_HTTP_VERSION_3` option, that makes libcurl use QUIC to connect to the server and unless the server supports HTTP/3 the transfer will fail - there is no fallback to earlier versions from this in the way the HTTP/2 option works.

HTTP ranges

What if the client only wants the first 200 bytes out of a remote resource or perhaps 300 bytes somewhere in the middle? The HTTP protocol allows a client to ask for only a specific data range. The client asks the server for the specific range with a start offset and an end offset. It can even combine things and ask for several ranges in the same request by just listing a bunch of pieces next to each other. When a server sends back multiple independent pieces to answer such a request, you will get them separated with mime boundary strings and it will be up to the user application to handle that accordingly. curl will not further separate such a response.

However, a byte range is only a request to the server. It does not have to respect the request and in many cases, like when the server automatically generates the contents on the fly when it is being asked, it will simply refuse to do it and it then instead respond with the full contents anyway.

You can make libcurl ask for a range with `CURLOPT_RANGE`. Like if you want the first 200 bytes out of something:

```
1 curl_easy_setopt(curl, CURLOPT_RANGE, "0-199");
```

Or everything in the file starting from index 200:

```
1 curl_easy_setopt(curl, CURLOPT_RANGE, "200-");
```

Get 200 bytes from index 0 *and* 200 bytes from index 1000:

```
1 curl_easy_setopt(curl, CURLOPT_RANGE, "0-199,1000-199");
```

HTTP authentication

libcurl supports a wide variety of HTTP authentication schemes.

Note that this way of authentication is different than the otherwise widely used scheme on the web today where authentication is performed by an HTTP POST and then keeping state in cookies. See [Cookies with libcurl](#) for details on how to do that.

User name and password

libcurl will not try any HTTP authentication without a given user name. Set one like:

```
1 curl_easy_setopt(curl, CURLOPT_USERNAME, "joe");
```

and of course most authentications also require a set password that you set separately:

```
1 curl_easy_setopt(curl, CURLOPT_PASSWORD, "secret");
```

That is all you need. This will make libcurl switch on its default authentication method for this transfer: *HTTP Basic*.

Authentication required

A client does not itself decide that it wants to send an authenticated request. It is something the server requires. When the server has a resource that is protected and requires authentication, it will respond with a 401 HTTP response and a `WWW-Authenticate:` header. The header will include details about what specific authentication methods it accepts for that resource.

Basic

Basic is the default HTTP authentication method and as its name suggests, it is indeed basic. It takes the name and the password, separates them with a colon and base64 encodes that string before it puts the entire thing into a `Authorization:` HTTP header in the request.

If the name and password is set like the examples shown above, the exact outgoing header looks like this:

```
1 Authorization: Basic am9lOnNlY3JldA==
```

This authentication method is totally insecure over HTTP as the credentials will then be sent in plain-text over the network.

You can explicitly tell libcurl to use Basic method for a specific transfer like this:

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
```

Digest

Another HTTP authentication method is called Digest. One advantage this method has compared to Basic, is that it does not send the password over the wire in plain text. This is however an authentication method that is rarely spoken by browsers and consequently is not a frequently used one.

You can explicitly tell libcurl to use the Digest method for a specific transfer like this (it still needs user name and password set as well):

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
```

NTLM

Another HTTP authentication method is called NTLM.

You can explicitly tell libcurl to use the NTLM method for a specific transfer like this (it still needs user name and password set as well):

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_NTLM);
```

Negotiate

Another HTTP authentication method is called Negotiate.

You can explicitly tell libcurl to use the Negotiate method for a specific transfer like this (it still needs user name and password set as well):

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_NEGOTIATE);
```

Bearer

To pass on an OAuth 2.0 Bearer Access Token in a request, use `CURLOPT_XOAUTH2_BEARER` for example:

```
1 CURL *curl = curl_easy_init();
2 if(curl) {
3     curl_easy_setopt(curl, CURLOPT_URL, "pop3://example.com/");
4     curl_easy_setopt(curl, CURLOPT_XOAUTH2_BEARER, "1ab9cb22ba269a7");
5     ret = curl_easy_perform(curl);
6     curl_easy_cleanup(curl);
7 }
```

Try-first

Some HTTP servers allow one out of several authentication methods, in some cases you will find yourself in a position where you as a client does not want or is not able to select a single specific method before-hand and for yet another subset of cases your application does not know if the requested URL even require authentication or not!

libcurl covers all these situations as well.

You can ask libcurl to use more than one method, and when doing so, you imply that curl first tries the request without any authentication at all and then based on the HTTP response coming back, it selects one of the methods that both the server and your application allow. If more than one would work, curl will pick them in a order based on how secure the methods are considered to be, picking the safest of the available methods.

Tell libcurl to accept multiple method by bitwise ORing them like this:

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_BASIC | CURLAUTH_DIGEST);
```

If you want libcurl to only allow a single specific method but still want it to probe first to check if it can possibly still make the request without the use of authentication, you can force that behavior by adding `CURLAUTH_ONLY` to the bitmask.

Ask to use digest, but nothing else but digest, and only if proven really necessary:

```
1 curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST | CURLAUTH_ONLY);
```

Cookies with libcurl

By default and by design, libcurl makes transfers as basic as possible and features need to be enabled to get used. One such feature is HTTP cookies, more known as just plain and simply "cookies".

Cookies are name/value pairs sent by the server (using a `Set-Cookie:` header) to be stored in the client, and are then supposed to get sent back again in requests that matches the host and path requirements that were specified along with the cookie when it came from the server (using the `Cookie:` header). On the modern web of today, sites are known to sometimes use large numbers of cookies.

Cookie engine

When you enable the "cookie engine" for a specific easy handle, it means that it will record incoming cookies, store them in the in-memory "cookie store" that is associated with the easy handle and subsequently send the proper ones back if an HTTP request is made that matches.

There are two ways to switch on the cookie engine:

Enable cookie engine with reading

Ask libcurl to import cookies into the easy handle from a given file name with the `CURLOPT_COOKIEFILE` option:

```
1 curl_easy_setopt(easy, CURLOPT_COOKIEFILE, "cookies.txt");
```

A common trick is to just specify a non-existing file name or plain "" to have it just activate the cookie engine with a blank cookie store to start with.

This option can be set multiple times and then each of the given files will be read.

Enable cookie engine with writing

Ask for received cookies to get stored in a file with the `CURLOPT_COOKIEJAR` option:

```
1 curl_easy_setopt(easy, CURLOPT_COOKIEJAR, "cookies.txt");
```

when the easy handle is closed later with `curl_easy_cleanup()`, all known cookies will be written to the given file. The file format is the well-known "Netscape cookie file" format that browsers also once used.

Setting custom cookies

A simpler and more direct way to just pass on a set of specific cookies in a request that does not add any cookies to the cookie store and does not even activate the cookie engine, is to set the set with `'CURLOPT_COOKIE'`:

```
1 curl_easy_setopt(easy, CURLOPT_COOKIE, "name=daniel; present=yes");
```

The string you set there is the raw string that would be sent in the HTTP request and should be in the format of repeated sequences of `NAME=VALUE`; - including the semicolon separator.

Import export

The cookie in-memory store can hold a bunch of cookies, and libcurl offers very powerful ways for an application to play with them. You can set new cookies, you can replace an existing cookie and you can extract existing cookies.

Add a cookie to the cookie store

Add a new cookie to the cookie store by simply passing it into curl with `CURLOPT_COOKIELIST` with a new cookie. The format of the input is a single line in the cookie file format, or formatted as a `Set-Cookie:` response header, but we recommend the cookie file style:

```
1 #define SEP "\t" /* Tab separates the fields */
2
3 char *my_cookie =
4   "example.com"    /* Hostname */
5   SEP "FALSE"      /* Include subdomains */
```

```
6 SEP "/"          /* Path */
7 SEP "FALSE"      /* Secure */
8 SEP "0"          /* Expiry in epoch time format. 0 == Session */
9 SEP "foo"         /* Name */
10 SEP "bar";       /* Value */
11
12 curl_easy_setopt(curl, CURLOPT_COOKIELIST, my_cookie);
```

If that given cookie would match an already existing cookie (with the same domain and path, etc.), it would overwrite the old one with the new contents.

Get all cookies from the cookie store

Sometimes writing the cookie file when you close the handle is not enough and then your application can opt to extract all the currently known cookies from the store like this:

```
1 struct curl_slist *cookies
2 curl_easy_getinfo(easy, CURLINFO_COOKIELIST, &cookies);
```

This returns a pointer to a linked list of cookies, and each cookie is (again) specified as a single line of the cookie file format. The list is allocated for you, so do not forget to call `curl_slist_free_all` when the application is done with the information.

Cookie store commands

If setting and extracting cookies is not enough, you can also interfere with the cookie store in more ways:

Wipe the entire in-memory storage clean with:

```
1 curl_easy_setopt(curl, CURLOPT_COOKIELIST, "ALL");
```

Erase all session cookies (cookies without expiry date) from memory:

```
1 curl_easy_setopt(curl, CURLOPT_COOKIELIST, "SESS");
```

Force a write of all cookies to the file name previously specified with `CURLOPT_COOKIEJAR`:

```
1 curl_easy_setopt(curl, CURLOPT_COOKIELIST, "FLUSH");
```

Force a reload of cookies from the file name previously specified with `CURLOPT_COOKIEFILE`:

```
1 curl_easy_setopt(curl, CURLOPT_COOKIELIST, "RELOAD");
```

Cookie file format

The cookie file format is text based and stores one cookie per line. Lines that start with `#` are treated as comments.

Each line that each specifies a single cookie consists of seven text fields separated with TAB characters.

Field	Example	Meaning
0	example.com	Domain name
1	FALSE	Include subdomains boolean
2	/foobar/	Path
3	FALSE	Set over a secure transport
4	1462299217	Expires at – seconds since Jan 1st 1970, or 0
5	person	Name of the cookie
6	daniel	Value of the cookie

Download

The GET method is the default method libcurl uses when an HTTP URL is requested and no particular other method is asked for. It asks the server for a particular resource—the standard HTTP download request:

```
1 easy = curl_easy_init();
2 curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
3 curl_easy_perform(easy);
```

Since options set in an easy handle are sticky and remain until changed, there may be times when you have asked for another request method than GET and then want to switch back to GET again for a subsequent request. For this purpose, there is the `CURLOPT_HTTPGET` option:

```
1 curl_easy_setopt(easy, CURLOPT_HTTPGET, 1L);
```

Download headers too

An HTTP transfer also includes a set of response headers. Response headers are metadata associated with the actual payload, called the response body. All downloads will get a set of headers too, but when using libcurl you can select whether you want to have them downloaded (seen) or not.

You can ask libcurl to pass on the headers to the same "stream" as the regular body is, by using

`CURLOPT_HEADER`:

```
1 easy = curl_easy_init();
2 curl_easy_setopt(easy, CURLOPT_HEADER, 1L);
3 curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
4 curl_easy_perform(easy);
```

Or you can opt to store the headers in a separate download file, by relying on the default behaviors of the [write](#) and [header callbacks](#):

```
1 easy = curl_easy_init();
2 FILE *file = fopen("headers", "wb");
3 curl_easy_setopt(easy, CURLOPT_HEADERDATA, file);
4 curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
5 curl_easy_perform(easy);
6 fclose(file);
```

If you only want to casually browse the headers, you may even be happy enough with just setting verbose mode while developing as that will show both outgoing and incoming headers sent to stderr:

```
1 curl_easy_setopt(easy, CURLOPT_VERBOSE, 1L);
```

Upload

Uploads over HTTP can be done in many different ways and it is important to notice the differences. They can use different methods, like POST or PUT, and when using POST the body formatting can differ.

In addition to those HTTP differences, libcurl offers different ways to provide the data to upload.

HTTP POST

POST is typically the HTTP method to pass data to a remote web application. A common way to do that in browsers is by filling in an HTML form and pressing submit. It is the standard way for an HTTP request to pass on data to the server. With libcurl you normally provide that data as a pointer and a length:

```
1 curl_easy_setopt(easy, CURLOPT_POSTFIELDS, dataptr);
2 curl_easy_setopt(easy, CURLOPT_POSTFIELDSIZE, (long)datalength);
```

Or you tell libcurl that it is a post but would prefer to have libcurl instead get the data by using the regular [read callback](#):

```
1 curl_easy_setopt(easy, CURLOPT_POST, 1L);
```

```
2 curl_easy_setopt(easy, CURLOPT_READFUNCTION, read_callback);
```

This "normal" POST will also set the request header `Content-Type: application/x-www-form-urlencoded`.

HTTP multipart formposts

A multipart formpost is still using the same HTTP method POST; the difference is only in the formatting of the request body. A multipart formpost is a series of separate "parts", separated by MIME-style boundary strings. There is no limit to how many parts you can send.

Each such part has a name, a set of headers and a few other properties.

libcurl offers a set of convenience functions for constructing such a series of parts and to send that off to the server, all prefixed with `curl_mime`. Create a multipart post and for each part in the data you set the name, the data and perhaps additional meta-data. A basic setup might look like this:

```
1 /* Create the form */
2 form = curl_mime_init(curl);
3
4 /* Fill in the file upload field */
5 field = curl_mime_addpart(form);
6 curl_mime_name(field, "sendfile");
7 curl_mime_filedata(field, "photo.jpg");
```

Then you pass that post to libcurl like this:

```
1 curl_easy_setopt(easy, CURLOPT_MIMEPOST, form);
```

(`curl_formadd` is the former API to build multi-part formposts with but we no longer recommend using that)

HTTP PUT

A PUT with libcurl will assume you pass the data to it using the read callback, as that is the typical "file upload" pattern libcurl uses and provides. You set the callback, you ask for PUT (by asking for `CURLOPT_UPLOAD`), you set the size of the upload and you set the URL to the destination:

```
1 curl_easy_setopt(easy, CURLOPT_UPLOAD, 1L);
2 curl_easy_setopt(easy, CURLOPT_INFILESIZE_LARGE, (curl_off_t) size);
3 curl_easy_setopt(easy, CURLOPT_READFUNCTION, read_callback);
4 curl_easy_setopt(easy, CURLOPT_URL, "https://example.com/handle/put");
```

If you for some reason do not know the size of the upload before the transfer starts, and you are using HTTP 1.1 you can add a `Transfer-Encoding: chunked` header with [CURLOPT_HTTPHEADER](#). For HTTP 1.0 you must provide the size before hand and for HTTP 2 and later, neither the size nor the extra header is needed

Expect: headers

When doing HTTP uploads using HTTP 1.1, libcurl will insert an `Expect: 100-continue` header in some circumstances. This header offers the server a way to reject the transfer early and save the client from having to send a lot of data in vain before the server gets a chance to decline.

The header is added by libcurl if HTTP uploading is done with `CURLOPT_UPLOAD` or if it is asked to do an HTTP POST for which the body size is either unknown or known to be larger than 1024 bytes.

A libcurl-using client can explicitly disable the use of the `Expect:` header with the [CURLOPT_HTTPHEADER](#) option.

This header is not used with HTTP/2 or HTTP/3.

Uploads also downloads

HTTP is a protocol that can respond with contents back even when you upload data to it - it is up to the server to decide. The response data may even start getting sent back to the client before the upload has completed.

Multiplexing

The HTTP versions 2 and 3 offer "multiplexing". Using this protocol feature, an HTTP client can do several concurrent transfers to a server *over the same single connection*. This feature does not exist in earlier versions of the HTTP protocol. In earlier HTTP versions, the client would either have to create multiple connections or do the transfers in a serial manner, one after the other.

libcurl supports HTTP multiplexing for both HTTP/2 and HTTP/3.

Make sure you do multiple transfers using the multi interface to a server that supports HTTP multiplexing. libcurl can only multiplex transfers when the same host name is used for subsequent transfers.

For all practical purposes and API behaviors, an application does not have to care about if multiplexing is done or not.

libcurl enables multiplexing by default, but if you start multiple transfers at the same time they will prioritize short-term speed to a connection so they might then rather open new connections than waiting for a connection to get created by another transfer to be able to multiplex over. To tell libcurl to prioritize multiplexing, set the `CURLOPT_PIPEWAIT` option for the transfer with `curl_easy_setopt()`.

With `curl_multi_setopt()`'s option `CURLMOPT_PIPELINING`, you can disable multiplexing for a

specific multi handle.

HSTS

HSTS is short for HTTP Strict-Transport-Security. It is a defined way for a server to tell a client that the client should prefer to use HTTPS with that site for a specified period of time into the future.

Here is how you use HSTS with libcurl.

In-memory cache

libcurl primarily features an in-memory cache for HSTS hosts, so that subsequent HTTP-only requests to a host name present in the cache will get internally "redirected" to the HTTPS version. Assuming you have this feature enabled.

Enable HSTS for a handle

HSTS is enabled by setting the correct bitmask using the `CURLOPT_HSTS_CTRL` option with `curl_easy_setopt()`. The bitmask has two separate flags that can be used, but `CURLHSTS_ENABLE` is the primary one. If that is set, then this easy handle now has HSTS support enabled.

The second flag available for this option is `CURLHSTS_READONLYFILE`, which if set, tells libcurl that the file name you specify for it to use as a HSTS cache is only to be read from, and not write anything back to.

Set a HSTS cache file

If you want to persist the HSTS cache on disk, then set a file name with the `CURLOPT_HSTS` option. libcurl will read from this file at start of a transfer and write to it (unless it was set read-only) when the easy handle is closed.

alt-svc

Alternative Services, aka alt-svc, is an HTTP header that lets a server tell the client that there is one or more *alternatives* for that server at "another place" with the use of the `Alt-Svc:` response header.

The *alternatives* the server suggests can include a server running on another port on the same host, on another completely different host name and it can also offer the service *over another protocol*.

Enable

To make libcurl consider any offered alternatives by servers, you must first enable it in the handle. You do this by setting the correct bitmask to the `CURLOPT_ALTSVC_CTRL` option. The bitmask allows the application to limit what HTTP versions to allow, and if the "cache" file on disk should only be used to read from (not write).

Enable alt-svc and allow it to switch to either HTTP/1 or HTTP/2:

```
1 curl_easy_setopt(curl, CURLOPT_ALTSVC_CTRL, CURLALTSVC_H1|CURLALTSVC_H2);
```

Tell libcurl to use a specific alt-svc cache file like this:

```
1 curl_easy_setopt(curl, CURLOPT_ALTSVC, "altsvc-cache.txt");
```

libcurl holds the list of alternatives in a memory-based cache, but will load all already existing alternative service entries from the alt-svc file at start-up and consider those when doing its subsequent HTTP requests. If servers responds with new or updated `Alt-Svc:` headers, libcurl will store those in the cache file at exit (unless the `CURLALTSVC_READONLYFILE` bit was set).

The alt-svc cache

The alt-svc cache is similar to a cookie jar. It is a text based file that stores one alternative per line and each entry also has an expiry time for which duration that particular alternative is valid.

HTTPS only

`Alt-Svc:` is only trusted and parsed from servers when connected to over HTTPS.

HTTP/3

The use of `Alt-Svc:` headers is as of March 2022 still the only defined way to bootstrap a client and server into using HTTP/3. The server then hints to the client over HTTP/1 or HTTP/2 that it also is available over HTTP/3 and then curl can connect to it using HTTP/3 in the subsequent request if the alt-svc cache says so.

Bindings

Creative people have written bindings or interfaces for various environments and programming languages. Using one of these allows you to take advantage of the power of curl from within your favorite language or system. This is a list of all known interfaces, as of the time of this writing.

The bindings listed below are not part of the curl/libcurl distribution archives. They must be downloaded and installed separately.

Language	Site	Author(s)
Script Basic	https://scriptbasic.com/	Peter Verhas
C++	http://curlpp.org/	Jean-Philippe, Barrette-LaPier
C++	https://github.com/JosephP91/curlcpp	Giuseppe Persico
C++	https://github.com/libcpr/cpr	Huu Nguyen
Ch/C++	https://chcurl.sourceforge.io/	Stephen Nestinger, Jonathan Rogado
Cocoa (BBHTTP)	https://github.com/biasedbit/BBHTTP	Bruno de Carvalho
Cocoa (CURLHandle)	https://github.com/karelia/curlhandle	Dan Wood
Clojure	https://github.com/lsevero/clj-curl	Lucas Severo
D	https://dlang.org/library/std/net/url.html	Kenneth Bogert
Delphi	https://github.com/Mercury13/curl4delphi	Mikhail Merkuryev
Dylan	https://opendylan.org/	Chris Double
Eiffel	https://iron.eiffel.com/repository/20.11/package/ABEF6975-37AC-45FD-9C67-52D10BA0669B	Eiffel Software
Erlang	https://github.com/puzza007/katipo	Paul Oliver
Falcon	http://www.falconpl.org/	Falcon
Gambas	https://gambas.sourceforge.io/	Gambas

glib/GTK+	http://atterer.org/glibcurl	Richard Atterer
Go	https://github.com/andelf/go-curl	ShuYu Wang
Guile	http://www.lonelycactus.com/guile-curl.html	Michael L. Gran
Harbour	https://github.com/vszakats/harbour/tree/master/contrib/hbcurl	Viktor Szakáts
Haskell	https://hackage.haskell.org/package/curl	Galois, Inc

libcurl internals

libcurl is never finished and is not just an off-the-shelf product. It is a living project that is improved and modified on almost a daily basis. We depend on skilled and interested hackers to fix bugs and to add features.

This chapter is meant to describe internal details to aid keen libcurl hackers to learn some basic concepts on how libcurl works internally and thus possibly where to look for problems or where to add things when you want to make the library do something new.

- [Easy handles and connections](#)
- [Everything is multi](#)
- [State machines](#)
- [Protocol handler](#)
- [Backends](#)
- [Windows vs Unix](#)
- [Memory debugging](#)
- [Content Encoding](#)

- [Structs](#)
- [Resolving host names](#)

Easy handles and connections

When reading the source code there are some useful basics that are good to know and keep in mind:

- 'data' is the variable name we use all over to refer to the easy handle (`struct Curl_easy`) for the transfer being worked on. No other name should be used for this and nothing else should use this name. The easy handle is the main object identifying a transfer. A transfer typically uses a connection at some point and typically only one at a time. There is a `data->conn` pointer that identifies the connection that is currently used by this transfer. A single connection can be used over time and even concurrently by several transfers (and thus easy handles) when multiplexed connections are used.
- `conn` is the variable name we use all over the internals to refer to the current *connection* the code works on (`struct connectdata`).
- `result` is the usual name we use for a `CURLcode` variable to hold the return values from functions and if that return value is different than zero, it is an error and the function should clean up and return (usually passing on the same error code to its parent function).

Everything is multi

libcurl offers a few different APIs to do transfers; where the primary differences are the synchronous easy interface versus the non-blocking multi interface. The multi interface itself can then be further used either by using the event-driven socket interface or the "normal" perform interface.

Internally however, everything is written for the event-driven interface. Everything needs to be written in non-blocking fashion so that functions are never waiting for data in loop or similar. Unless they are the "surface" functions that have that expressed functionality.

The function `curl_easy_perform()` which performs a single transfer synchronously, is itself just a wrapper function that internally will setup and use the multi interface itself.

State machines

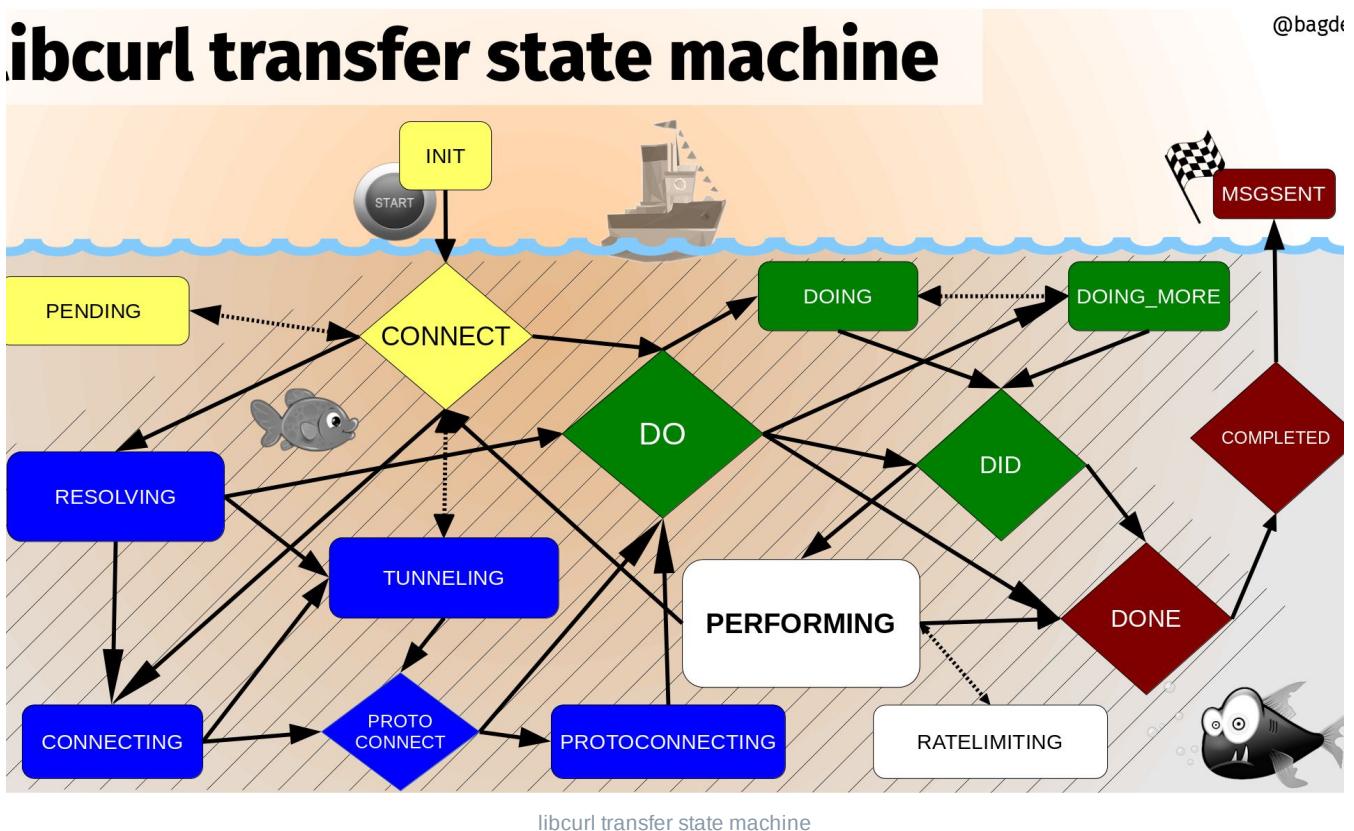
To facilitate non-blocking behavior all through, the curl source is full of state machines. Work on as much data as there is and drive the state machine to where it can go based on what's available and allow the functions to continue from that point later on when more data arrives that then might drive the state machine further.

There are such states in many different levels for a given transfer and the code for each particular protocol may have its own set of state machines.

mstate

One of the primary states is the main transfer "mode" the easy handle holds, which says if the current transfer is resolving, waiting for a resolve, connecting, waiting for a connect, issuing a request, doing a transfer etc (see the `CURLMstate` enum in `lib/multihandle.h`). Every transfer done with libcurl has an associated easy handle and every easy handle will exercise that state machine.

The image below shows all states and possible state transitions. See further explanation below.



All transfers start in **INIT** and they end in **MSGSENT**

yellow: the initial setting-up states

blue: resolving names and setting up the connection

green: initiating and setting up the transfer

white: the transfer

red: post-transfer

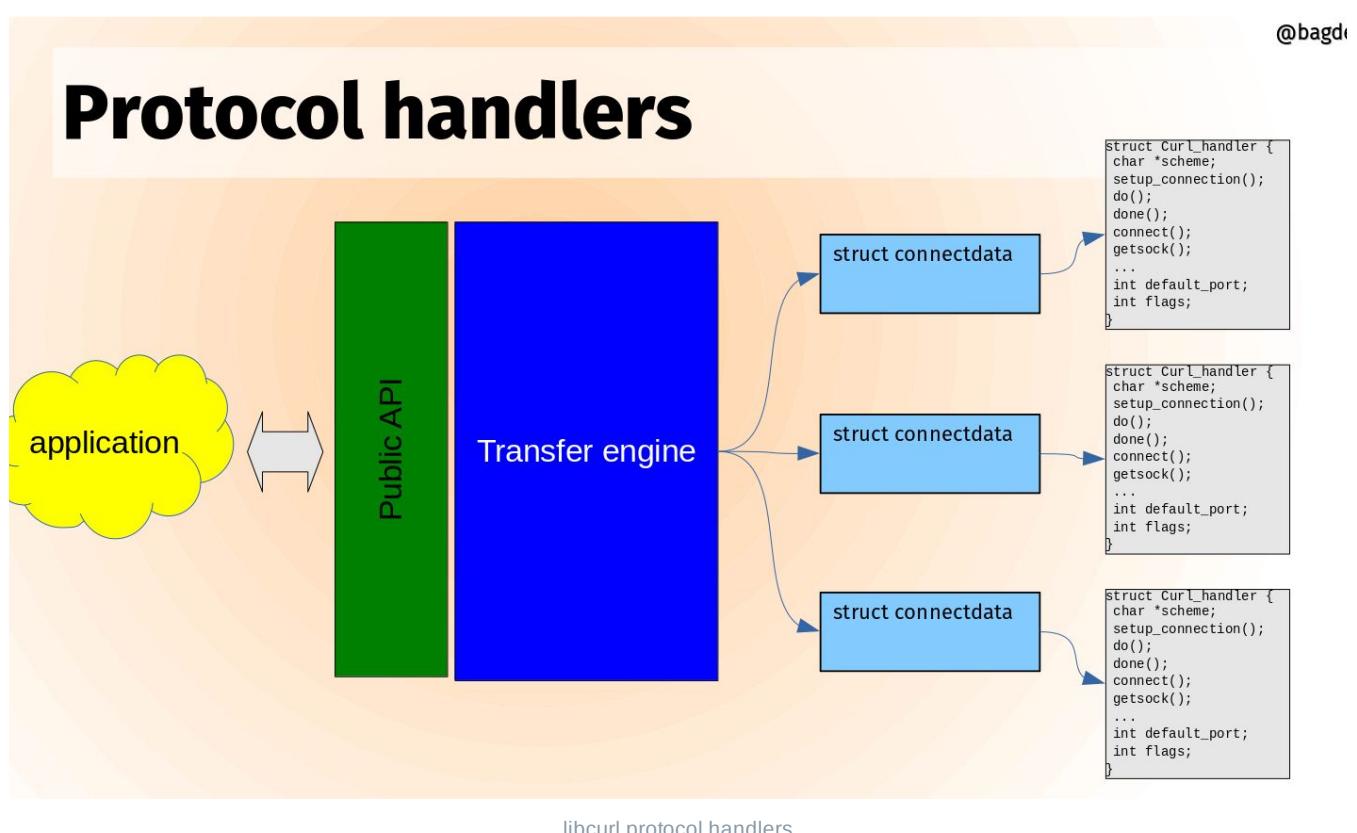
All states within the striped area have an associated connection.

Protocol handler

curl is a multi-protocol transfer library. The core of the code is a set of generic functions that are used for transfers in general and will mostly work the same for all protocols. The main state machine described above for example is there and works for all protocols - even though some protocols may not make use of all states for all transfers.

However, each different protocol curl speaks also has its unique particularities and specialties. In order to not have the code littered with conditions in the style "if the protocol is XYZ, then do...", we instead have the concept of `Curl_handler`. Each supported protocol defines one of those in `lib/url.c` there is an array of pointers to such handlers called `protocols[]`.

When a transfer is about to be done, curl parses the URL it is about to operate on and among other things it figures out what protocol to use. Normally this can be done by looking at the scheme part of the URL. For `https://example.com` that is `https` and for `imaps://example.com` it is `imaps`. Using the provided scheme, curl sets the `conn->handler` pointer to the handler struct for the protocol that handles this URL.



The handler struct contains a set of function pointers that can be NULL or set to point to a protocol specific function to do things necessary for that protocol to work for a transfer. Things that not all other protocols need. The handler struct also sets up the name of the protocol and describes its feature set with a bitmask.

A libcurl transfer is built around a set of different "actions" and the handler can extend each of them. Here are some example function pointers in this struct and how they are used:

Setup connection

If a connection cannot be reused for a transfer, it needs to setup a connection to the host given in the URL

and when it does, it can also call the protocol handler's function for it. Like this:

```
1 if(conn->handler->setup_connection)
2   result = conn->handler->setup_connection(data, conn);
```

Connect

After a connection has been established, this function gets called

```
1 if(conn->handler->connect_it)
2   result = conn->handler->connect_it(data, &done);
```

Do

"Do" is simply the action that issues a request for the particular resource the URL identifies. All protocol has a do action so this function must be provided:

```
1 result = conn->handler->do_it(data, &done);
```

Done

When a transfer is completed, the "done" action is taken:

```
1 result = conn->handler->done(data, status, premature);
```

Disconnect

The connection is about to be taken down.

```
1 result = conn->handler->disconnect(data, conn, dead_connection);
```

Backends

A backend in curl is a **build-time selectable alternative implementation**.

When you build curl, you can select alternative implementations for several different things. Different providers of the same feature set. You select which backend or backends (plural) to use when you build curl.

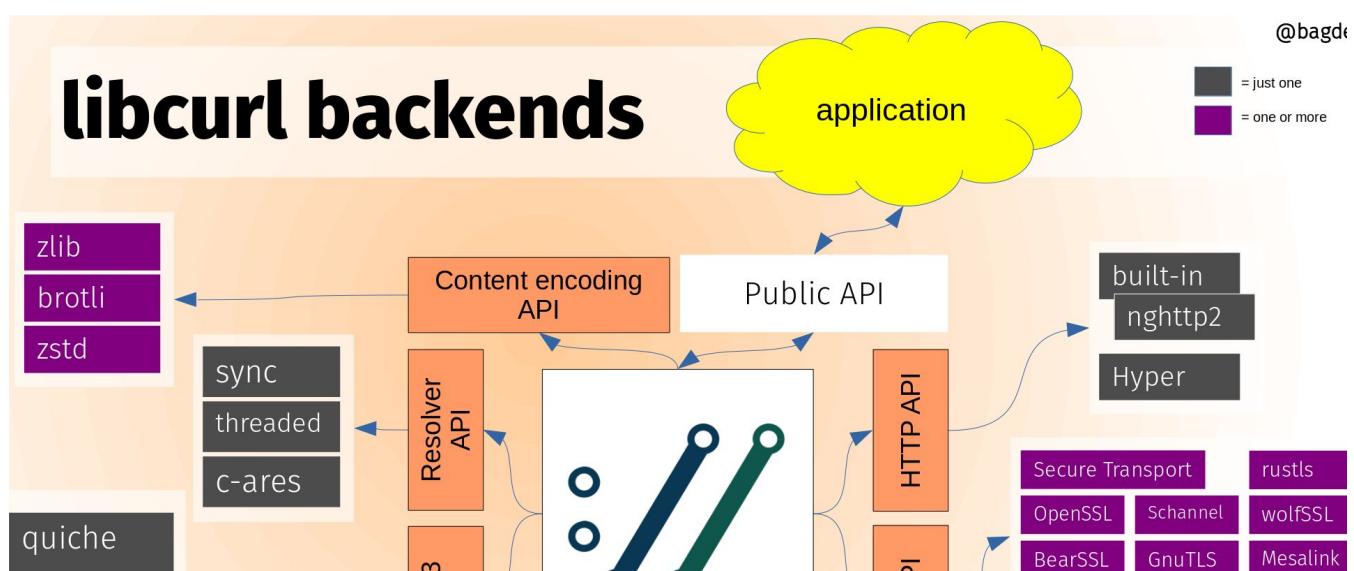
- Backends are selectable and deselectable
- Often platform dependent
- Can differ in features
- Can differ in 3rd party licenses
- Can differ in maturity
- The internal APIs are never exposed externally

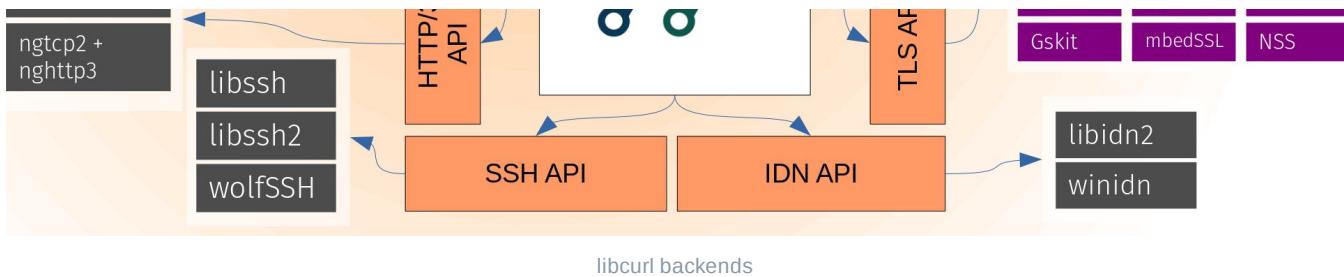
Different backends

In the libcurl source code, there are internal APIs for providing functionality. In these different areas there are multiple different providers:

1. IDN
2. Name resolving
3. TLS
4. SSH
5. HTTP/1 and HTTP/2
6. HTTP/3
7. HTTP content encoding

Backends visualized





Applications (in the upper yellow cloud) access libcurl through the public API. The API is fixed and stable.

Internally, the "core" of libcurl uses internal APIs to perform the different duties it needs to do. Each of these internal APIs are powered by alternative implementations, in many times powered by different third party libraries.

The image above shows the different third party libraries powering different internal APIs. The purple boxes are "one or more" and the dark gray ones are "one of these".

Windows vs Unix

There are a few differences in how to program curl the Unix way compared to the Windows way. Perhaps the four most notable details are:

Different function names for socket operations

In curl, this is solved with defines and macros, so that the source looks the same in all places except for the header file that defines them. The macros in use are `sclose()`, `sread()` and `swrite()`.

Init calls

Windows requires a couple of init calls for the socket stuff.

That is taken care of by the `curl_global_init()` call, but if other libs also do it etc there might be reasons for applications to alter that behavior.

We require WinSock version 2.2 and load this version during global init.

File descriptors

File descriptors for network communication and file operations are not as easily interchangeable as in Unix.

We avoid this by not trying any funny tricks on file descriptors.

Stdout

When writing data to stdout, Windows makes end-of-lines the DOS way, thus destroying binary data, although you do want that conversion if it is text coming through... (sigh)

We set stdout to binary under windows

Ifdefs

Inside the source code, We make an effort to avoid `#ifdef [Your OS]`. All conditionals that deal with features *should* instead be in the format `#ifdef HAVE_THAT_WEIRD_FUNCTION`. Since Windows cannot run configure scripts, we maintain a `curl_config-win32.h` file in the lib directory that is supposed to look exactly like a `curl_config.h` file would have looked like on a Windows machine.

Generally speaking: curl is frequently compiled on several dozens of operating systems. Do not walk on the edge.

Memory debugging

The file `lib/memdebug.c` contains debug-versions of a few functions. Functions such as `malloc()`, `free()`, `fopen()`, `fclose()`, etc that somehow deal with resources that might give us problems if we "leak" them. The functions in the memdebug system do nothing fancy, they do their normal function and then log information about what they just did. The logged data can then be analyzed after a complete session,

`memanalyze.pl` is the perl script present in `tests/` that analyzes a log file generated by the memory tracking system. It detects if resources are allocated but never freed and other kinds of errors related to resource management.

Internally, the definition of the preprocessor symbol `DEBUGBUILD` restricts code which is only compiled for debug enabled builds. And symbol `CURLDEBUG` is used to differentiate code which is *only* used for memory tracking/debugging.

Use `-DCURLDEBUG` when compiling to enable memory debugging, this is also switched on by running configure with `--enable-curldebug`. Use `-DDEBUGBUILD` when compiling to enable a debug build or run configure with `--enable-debug`.

`curl --version` will list 'Debug' feature for debug enabled builds, and will list 'TrackMemory' feature for curl debug memory tracking capable builds. These features are independent and can be controlled when running the configure script. When `--enable-debug` is given both features will be enabled, unless some restriction prevents memory tracking from being used.

Track Down Memory Leaks

... using the memory debug system. In general, we suggest using valgrind a the first choice.

Single-threaded

Please note that this memory leak system is not adjusted to work in more than one thread. If you want/need to use it in a multi-threaded app. Please adjust accordingly.

Build

Rebuild libcurl with `-DCURLDEBUG` (usually, rerunning configure with `--enable-debug` fixes this). `make clean` first, then `make` so that all files are actually rebuilt properly. It will also make sense to build libcurl with the debug option (usually `-g` to the compiler) so that debugging it will be easier if you actually do find a leak in the library.

This will create a library that has memory debugging enabled.

Modify Your Application

Add a line in your application code:

```
1 curl_dbg_memdebug("dump");
```

This will make the malloc debug system output a full trace of all resources using functions to the given file name. Make sure you rebuild your program and that you link with the same libcurl you built for this purpose as described above.

Run Your Application

Run your program as usual. Watch the specified memory trace file grow.

Make your program exit and use the proper libcurl cleanup functions etc. So that all non-leaks are returned/freed properly.

Analyze the Flow

Use the `tests/memanalyze.pl` perl script to analyze the dump file:

```
1 tests/memanalyze.pl dump
```

This now outputs a report on what resources that were allocated but never freed etc. This report is fine for posting to the list.

If this does not produce any output, no leak was detected in libcurl. Then the leak is mostly likely to be in your code.

Content Encoding

About content encodings

[HTTP/1.1][4] specifies that a client may request that a server encode its response. This is usually used to compress a response using one (or more) encodings from a set of commonly available compression techniques. These schemes include `deflate` (the zlib algorithm), `gzip`, `br` (brotli) and `compress`. A client requests that the server perform an encoding by including an `Accept-Encoding` header in the request document. The value of the header should be one of the recognized tokens `deflate`, ... (there's a way to register new schemes/tokens, see sec 3.5 of the spec). A server MAY honor the client's encoding request. When a response is encoded, the server includes a `Content-Encoding` header in the response. The value of the `Content-Encoding` header indicates which encodings were used to encode the data, in the order in which they were applied.

It's also possible for a client to attach priorities to different schemes so that the server knows which it prefers. See sec 14.3 of RFC 2616 for more information on the `Accept-Encoding` header. See sec [3.1.2.2 of RFC 7231](#) for more information on the `Content-Encoding` header.

Supported content encodings

The `deflate`, `gzip`, `zstd` and `br` content encodings are supported by libcurl. Both regular and chunked transfers work fine. The zlib library is required for the `deflate` and `gzip` encodings, the brotli decoding library is for the `br` encoding and not too surprisingly libzstd does `zstd`.

The libcurl interface

To cause libcurl to request a content encoding use:

```
[curl_easy_setopt ][1](curl, [ CURLOPT_ACCEPT_ENCODING ][5], string)
```

where `string` is the intended value of the `Accept-Encoding` header.

Currently, libcurl does support multiple encodings but only understands how to process responses that use the `deflate`, `gzip`, `zstd` and/or `br` content encodings, so the only values for `[CURLOPT_ACCEPT_ENCODING][5]` that will work (besides `identity`, which does nothing) are `deflate`, `gzip`, `zstd` and `br`. If a response is encoded using the `compress` or methods, libcurl will return an error indicating that the response could not be decoded. If `<string>` is NULL no `Accept-Encoding` header is generated. If `<string>` is a zero-length string, then an `Accept-Encoding` header containing all supported encodings will be generated.

The `[CURLOPT_ACCEPT_ENCODING][5]` must be set to any non-NULL value for content to be automatically decoded. If it is not set and the server still sends encoded content (despite not having been

asked), the data is returned in its raw form and the `Content-Encoding` type is not checked.

The curl interface

Use the [`--compressed`][6] option with curl to cause it to ask servers to compress responses using any format supported by curl.

Structs

This section documents internal structs. Since they are truly internal, we can and will change them occasionally which might make this section slightly out of date at times.

Curl_easy

The `Curl_easy` struct is the one returned to the outside in the external API as an opaque `CURL *`. This pointer is usually known as an easy handle in API documentations and examples.

Information and state that is related to the actual connection is in the `connectdata` struct. When a transfer is about to be made, libcurl will either create a new connection or re-use an existing one. The current connectdata that is used by this handle is pointed out by `Curl_easy->conn`.

Data and information that regard this particular single transfer is put in the `SingleRequest` sub-struct.

When the `Curl_easy` struct is added to a multi handle, as it must be in order to do any transfer, the `->multi` member will point to the `Curl_multi` struct it belongs to. The `->prev` and `->next` members will then be used by the multi code to keep a linked list of `Curl_easy` structs that are added to that same multi handle. libcurl always uses multi so `->multi` will point to a `Curl_multi` when a transfer is in progress.

`->mstate` is the multi state of this particular `Curl_easy`. When `multi_runsingle()` is called, it will act on this handle according to which state it is in. The mstate is also what tells which sockets to return for a specific `Curl_easy` when `[curl_multi_fdset()][12]` is called etc.

The libcurl source code generally use the name `data` everywhere for the local variable that points to the `Curl_easy` struct.

When doing multiplexed HTTP/2 transfers, each `Curl_easy` is associated with an individual stream, sharing the same connectdata struct. Multiplexing makes it even more important to keep things associated with the right thing!

connectdata

A general idea in libcurl is to keep connections around in a connection "cache" after they have been used in case they will be used again and then re-use an existing one instead of creating a new one as it creates a significant performance boost.

Each `connectdata` struct identifies a single physical connection to a server. If the connection cannot be kept alive, the connection will be closed after use and then this struct can be removed from the cache and freed.

Thus, the same `Curl_easy` can be used multiple times and each time select another `connectdata` struct to use for the connection. Keep this in mind, as it is then important to consider if options or choices are based on the connection or the `Curl_easy`.

As a special complexity, some protocols supported by libcurl require a special disconnect procedure that is more than just shutting down the socket. It can involve sending one or more commands to the server before doing so. Since connections are kept in the connection cache after use, the original `Curl_easy` may no longer be around when the time comes to shut down a particular connection. For this purpose, libcurl holds a special dummy `closure_handle` `Curl_easy` in the `Curl_multi` struct to use when needed.

FTP uses two TCP connections for a typical transfer but it keeps both in this single struct and thus can be considered a single connection for most internal concerns.

The libcurl source code generally uses the name `conn` for the local variable that points to the `connectdata`.

Curl_multi

Internally, the easy interface is implemented as a wrapper around multi interface functions. This makes everything multi interface.

`Curl_multi` is the multi handle struct exposed as the opaque `CURLM *` in external APIs.

This struct holds a list of `Curl_easy` structs that have been added to this handle with `[curl_multi_add_handle()]`^[13]. The start of the list is `->easyp` and `->num_easy` is a counter of added `Curl_easy`s.

`->msglist` is a linked list of messages to send back when `[curl_multi_info_read()]`^[14] is called. Basically a node is added to that list when an individual `Curl_easy`'s transfer has completed.

`->hostcache` points to the name cache. It is a hash table for looking up name to IP. The nodes have a limited lifetime in there and this cache is meant to reduce the time for when the same name is wanted within a short period of time.

`->timetree` points to a tree of `Curl_easy`s, sorted by the remaining time until it should be checked - normally some sort of timeout. Each `Curl_easy` has one node in the tree.

`->sockhash` is a hash table to allow fast lookups of socket descriptor for which `Curl_easy` uses that descriptor. This is necessary for the `multi_socket` API.

`->conn_cache` points to the connection cache. It keeps track of all connections that are kept after use. The cache has a maximum size.

`->closure_handle` is described in the `connectdata` section.

The libcurl source code generally uses the name `multi` for the variable that points to the `Curl_multi` struct.

Curl_handler

Each unique protocol that is supported by libcurl needs to provide at least one `Curl_handler` struct. It defines what the protocol is called and what functions the main code should call to deal with protocol specific issues. In general, there's a source file named `[protocol].c` in which there's a `struct Curl_handler Curl_handler_[protocol]` declared. In `url.c` there's then the main array with all individual `Curl_handler` structs pointed to from a single array which is scanned through when a URL is given to libcurl to work with.

The concrete function pointer prototypes can be found in `lib/urldata.h`.

- `->scheme` is the URL scheme name, usually spelled out in uppercase. That is "HTTP" or "FTP" etc. SSL versions of the protocol need their own `Curl_handler` setup so HTTPS separate from HTTP.
- `->setup_connection` is called to allow the protocol code to allocate protocol specific data that then gets associated with that `Curl_easy` for the rest of this transfer. It gets freed again at the end of the transfer. It will be called before the `connectdata` for the transfer has been selected/created. Most protocols will allocate its private `struct [PROTOCOL]` here and assign `Curl_easy->req.p.[protocol]` to it.
- `->connect_it` allows a protocol to do some specific actions after the TCP connect is done, that can still be considered part of the connection phase. Some protocols will alter the `connectdata->recv[]` and `connectdata->send[]` function pointers in this function.
- `->connecting` is similarly a function that keeps getting called as long as the protocol considers itself still in the connecting phase.
- `->do_it` is the function called to issue the transfer request. What we call the DO action internally. If the DO is not enough and things need to be kept getting done for the entire DO sequence to complete, `->doing` is then usually also provided. Each protocol that needs to do multiple commands or similar for do/doing needs to implement their own state machines (see SCP, SFTP, FTP). Some protocols (only FTP and only due to historical reasons) have a separate piece of the DO state called `DO_MORE`.
- `->doing` keeps getting called while issuing the transfer request command(s)
- `->done` gets called when the transfer is complete and DONE. That is after the main data has been transferred.
- `->do_more` gets called during the `DO_MORE` state. The FTP protocol uses this state when setting up the second connection.
- `->proto_getsock`, `->doing_getsock`, `->domore_getsock`, `->perform_getsock` Functions that return socket information. Which socket(s) to wait for which I/O action(s) during the particular multi state.
- `->disconnect` is called immediately before the TCP connection is shutdown.

- `->readwrite` gets called during transfer to allow the protocol to do extra reads/writes
 - `->attach` attaches a transfer to the connection.
 - `->defport` is the default report TCP or UDP port this protocol uses
 - `->protocol` is one or more bits in the `CURLPROTO_*` set. The SSL versions have their "base" protocol set and then the SSL variation. Like "HTTP|HTTPS".
 - `->flags` is a bitmask with additional information about the protocol that will make it get treated differently by the generic engine:
 - `PROTOPT_SSL` - will make it connect and negotiate SSL
 - `PROTOPT_DUAL` - this protocol uses two connections
 - `PROTOPT_CLOSEACTION` - this protocol has actions to do before closing the connection. This flag is no longer used by code, yet still set for a bunch of protocol handlers.
 - `PROTOPT_DIRLOCK` - "direction lock". The SSH protocols set this bit to limit which "direction" of socket actions that the main engine will concern itself with.
 - `PROTOPT_NONETWORK` - a protocol that does not use the network (read `file:`)
 - `PROTOPT_NEEDSPWD` - this protocol needs a password and will use a default one unless one is provided
 - `PROTOPT_NOURLQUERY` - this protocol cannot handle a query part on the URL (`?foo=bar`)
-

conncache

Is a hash table with connections for later re-use. Each `Curl_easy` has a pointer to its connection cache. Each multi handle sets up a connection cache that all added `Curl_easy`s share by default.

Curl_share

The libcurl share API allocates a `Curl_share` struct, exposed to the external API as `CURLSH *`.

The idea is that the struct can have a set of its own versions of caches and pools and then by providing this struct in the `CURLOPT_SHARE` option, those specific `Curl_easy`s will use the caches/pools that this share handle holds.

Then individual `Curl_easy` structs can be made to share specific things that they otherwise would not, such as cookies.

The `Curl_share` struct can currently hold cookies, DNS cache and the SSL session cache.

CookieInfo

This is the main cookie struct. It holds all known cookies and related information. Each `Curl_easy` has its own private `CookieInfo` even when they are added to a multi handle. They can be made to share cookies by using the share API.

Resolving host names

Aka `hostip.c` explained

The main compile-time defines to keep in mind when reading the `host*.c` source file are these:

CURLRES_IPV6

this host has `getaddrinfo()` and family, and thus we use that. The host may not be able to resolve IPv6, but we do not really have to take that into account. Hosts that are not IPv6-enabled have `CURLRES_IPV4` defined.

CURLRES_ARES

is defined if libcurl is built to use c-ares for asynchronous name resolves. This can be Windows or *nix.

CURLRES_THREADED

is defined if libcurl is built to use threading for asynchronous name resolves. The name resolve will be done in a new thread, and the supported asynch API will be the same as for ares-builds. This is the default under (native) Windows.

If any of the two previous are defined, `CURLRES_ASYNCH` is defined too. If libcurl is not built to use an asynchronous resolver, `CURLRES_SYNCH` is defined.

host*.c sources

The `host*.c` sources files are split up like this:

- `hostip.c` - method-independent resolver functions and utility functions
- `hostasyn.c` - functions for asynchronous name resolves
- `hostsyn.c` - functions for synchronous name resolves

- `asyn-ares.c` - functions for asynchronous name resolves using c-ares
- `asyn-thread.c` - functions for asynchronous name resolves using threads
- `hostip4.c` - IPv4 specific functions
- `hostip6.c` - IPv6 specific functions

The `hostip.h` is the single united header file for all this. It defines the `CURLRES_*` defines based on the `config*.h` and `curl_setup.h` defines.

Index

- `.netrc`: [Command line leakage](#), [.netrc](#), [All easy options](#)
-

I

- `/etc/hosts`: [Run a local clone](#), [Host](#), [Edit the hosts file](#)
-

<

- `<curl/curl.h>`: [include/curl](#), [Include files](#), [curl --libcurl](#), [Header files](#), [Get a simple HTML page](#), [Get a response into memory](#), [Submit a login form over HTTP](#), [Get an FTP directory listing](#), [Non-blocking HTTP form-post](#)
-

A

- `--alt-svc`: [Enable](#)
 - `--anyauth`: [HTTP authentication](#)
 - `apt`: [Ubuntu and Debian](#)
 - `Arch Linux`: [Arch Linux](#)
-

-
- -b: [Cookie engine](#), [curl HTTP cheat sheet](#), [Web logins and sessions](#)
 - --basic: [HTTP authentication](#)
 - BearSSL: [libvtls](#), [Build to use a TLS library](#)
 - BoringSSL: [Build to use a TLS library](#), [Build curl with boringssl](#), [Restrictions](#)
 - brotli: [HTTP Compression](#), [Version](#), [Which libcurl version runs](#), [About content encodings](#)
-

C

- -c: [Writing cookies to file](#), [curl HTTP cheat sheet](#), [Web logins and sessions](#)
- c-ares: [c-ares](#), [Line 4: Features](#), [Name resolve tricks with c-ares](#), [Name resolver backends](#), [CURLRES_ARES](#)
- C89: [Comments](#), [Build](#)
- CA: [Verbose mode](#), [MITM proxy](#), [Available exit codes](#), [Verifying server certificates](#), [OCSP stapling](#), [Verification](#), [All easy options](#)
- Chrome: [SSLKEYLOGFILE](#), [Copy as curl](#)
- clone: [Building libcurl on MSYS2](#), [Clone the code](#), [git](#), [Web site source code](#), [build boringssl](#)
- code of conduct: [Trust](#), [Code of conduct](#)
- --compressed: [Compression](#), [Gzipped transfers](#), [curl HTTP cheat sheet](#)
- configure: [root](#), [Handling different build options](#), [Platform dependent code](#), [On Linux and Unix-like systems](#), [Autotools](#), [configure](#), [set up the build tree to get detected by curl's configure](#), [Ifdefs](#), [Memory Debugging](#)
- --connect-timeout: [Connection timeout](#), [Never spend more than this to connect](#)
- --connect-to: [Provide a replacement name](#)
- connection cache: [Persistent connections](#), [Connection reuse](#), [Multi handle](#), [All easy options](#), [Connection cache](#), [connectdata](#)
- connection pool: [Connection reuse](#), [Connection reuse](#), [Connection cache](#)
- Connection reuse: [URLs](#), [Connection reuse](#), [Connection reuse](#), [Connection cache](#)
- content-encoding: [Compression](#), [Transfer encoding](#), [About content encodings](#)
- contribute: [Code of conduct](#), [Contributing](#), [Introduction](#)
- Contributing: [docs](#), [Contributing](#)
- Cookie engine: [Cookie engine](#), [Cookie engine](#)
- Cookies: [docs](#), [libpsl](#), [Line 4: Features](#), [Server differences](#), [Change the Host: header](#), [Not perfect](#), [HTTP authentication](#), [Cookies](#), [Cookie file format](#), [Cookies](#), [Simple by default](#), [more on demand](#), [Available information](#), [Sharing between easy handles](#), [All easy options](#), [Submit a login form over HTTP](#), [Sections](#), [HTTP authentication](#), [Cookies with libcurl](#), [Curl_share](#)
- copyright: [License](#), [Copyright](#)

- curl-announce: [curl-announce](#), Vulnerability handling
- curl-library: [curl-users](#), Make a patch for the mailing list, Vulnerability handling
- curl-users: [curl-users](#), Vulnerability handling
- CURLE_ABORTED_BY_CALLBACK: Progress callback
- CURLHSTS_ENABLE: Enable HSTS for a handle
- CURLHSTS_READONLYFILE: Enable HSTS for a handle
- CURLINFO_CERTINFO: Available information
- CURLINFO_CONTENT_TYPE: Post transfer info
- CURLINFO_EFFECTIVE_URL: Available information
- CURLINFO_FILETIME: Available information
- CURLINFO_TOTAL_TIME_T: Available information
- CURLMOPT_PIPELINING: Multiplexing
- CURLMOPT_SOCKETFUNCTION: [socket_callback](#)
- CURLMOPT_TIMERFUNCTION: [timer_callback](#)
- CURLOPT_ALTSVC: All easy options, Enable
- CURLOPT_ALTSVC_CTRL: All easy options, Enable
- CURLOPT_CLOSESOCKETFUNCTION: Socket close callback, All easy options
- CURLOPT_COOKIE: All easy options, Setting custom cookies
- CURLOPT_COOKIEFILE: All easy options, Submit a login form over HTTP, Enable cookie engine with reading
- CURLOPT_COOKIEJAR: All easy options, Enable cookie engine with writing
- CURLOPT_COOKIELIST: All easy options, Add a cookie to the cookie store
- CURLOPT_CURLU: URL API, CURLOPT_CURLU, All easy options
- CURLOPT_CUSTOMREQUEST: All easy options, Request method
- CURLOPT_DEBUGDATA: Debug callback, All easy options, Trace everything
- CURLOPT_DEBUGFUNCTION: Debug callback, All easy options, Trace everything
- CURLOPT_DNS_CACHE_TIMEOUT: Caching, All easy options, DNS cache
- CURLOPT_DNS_INTERFACE: Name server options, All easy options
- CURLOPT_DNS_LOCAL_IP4: Name server options, All easy options
- CURLOPT_DNS_LOCAL_IP6: Name server options, All easy options
- CURLOPT_DNS_SERVERS: Name server options, All easy options
- CURLOPT_DNS_USE_GLOBAL_CACHE: No global DNS cache, All easy options
- CURLOPT_ERRORBUFFER: curl --libcurl, All easy options, CURLcode return code
- CURLOPT_FAILONERROR: All easy options, About HTTP response code "errors"
- CURLOPT_HEADER: Write callback, All easy options, Referrer, Download headers too
- CURLOPT_HEADERDATA: Header callback, curl --libcurl, All easy options, Download headers too

- CURLOPT_HEADERFUNCTION: Header callback, curl --libcurl, All easy options
- CURLOPT_HSTS: All easy options, Set a HSTS cache file
- CURLOPT_HSTS_CTRL: All easy options, Enable HSTS for a handle
- CURLOPT_HTTPGET: All easy options, Submit a login form over HTTP, libcurl HTTP download
- CURLOPT_HTTPHEADER: All easy options, Non-blocking HTTP form-post, Add a header, HTTP PUT
- CURLOPT_HTTPPOST: All easy options
- CURLOPT_IPRESOLVE: Name resolving, All easy options
- CURLOPT_MAXFILESIZE_LARGE: Set numerical options, All easy options
- CURLOPT_MAXREDIRS: curl --libcurl, All easy options
- CURLOPT_MIMEPOST: All easy options, Non-blocking HTTP form-post, HTTP multipart formposts
- CURLOPT_NOBODY: All easy options, Request method
- CURLOPT_NOPROGRESS: Progress callback, curl --libcurl, All easy options
- CURLOPT_OPENSOCKETDATA: Provide a file descriptor, All easy options
- CURLOPT_OPENSOCKETFUNCTION: Provide a file descriptor, All easy options
- CURLOPT_PIPEWAIT: All easy options, Multiplexing
- CURLOPT_POST: All easy options, HTTP POST
- CURLOPT_POSTFIELDS: Set string options, All easy options, Submit a login form over HTTP, Request method, HTTP POST
- CURLOPT_POSTFIELDSIZE: CURLOPT_POSTFIELDS , All easy options, HTTP POST
- CURLOPT_POSTREDIR: Decide what method to use in redirects, All easy options
- CURLOPT_PROGRESSFUNCTION: Progress callback, All easy options
- CURLOPT_PROXY: Proxy types, All easy options
- CURLOPT_PROXYPORT: Proxy types, All easy options
- CURLOPT_PROXYTYPE: Proxy types, All easy options
- CURLOPT_READDATA: Read callback, curl --libcurl, All easy options
- CURLOPT_READFUNCTION: Read callback, curl --libcurl, All easy options, HTTP POST
- CURLOPT_RESOLVE: Custom addresses for hosts, All easy options
- CURLOPT_SEEKDATA: seek callback, curl --libcurl, All easy options
- CURLOPT_SEEKFUNCTION: seek callback, curl --libcurl, All easy options
- CURLOPT_SOCKOPTDATA: sockopt callback, All easy options
- CURLOPT_SOCKOPTFUNCTION: sockopt callback, All easy options
- CURLOPT_SSH_KNOWNHOSTS: SSH key callback, curl --libcurl, All easy options
- CURLOPT_SSLVERSION: Protocol version, All easy options
- CURLOPT_SSL_VERIFYHOST: Verification, All easy options
- CURLOPT_SSL_VERIFYPEER: HTTPS proxy, Verification, All easy options
- CURLOPT_STDERR: curl --libcurl, All easy options, Verbose operations
-

- **CURLOPT_TCP_KEEPALIVE**: curl --libcurl, All easy options
- **CURLOPT_TIMEOUT**: Set numerical options, All easy options
- **CURLOPT_TLSAUTH_USERNAME**: TLS auth, All easy options
- **CURLOPT_UPLOAD**: All easy options, Request method, HTTP PUT
- **CURLOPT_URL**: Easy handle, CURLOPT_CURLU, curl --libcurl, Set handle options, Set string options, All easy options, Get a simple HTML page, Get a response into memory, Submit a login form over HTTP, Get an FTP directory listing, Non-blocking HTTP form-post, Strings are C strings, not C++ string objects, Request method, Bearer, libcurl HTTP download, HTTP PUT
- **CURLOPT_USERAGENT**: curl --libcurl, All easy options, Get a response into memory
- **CURLOPT_VERBOSE**: All easy options, Find a specific option by name, Verbose operations, Non-blocking HTTP form-post, Download headers too
- **CURLOPT_WRITEDATA**: Write callback, curl --libcurl, All easy options, Get a response into memory, Callback considerations
- **CURLOPT_WRITEFUNCTION**: Write callback, curl --libcurl, All easy options, Get a response into memory, Callback considerations
- **CURLOPT_XFERINFODATA**: Progress callback, All easy options
- **CURLOPT_XFERINFOFUNCTION**: Progress callback, All easy options
- **CURLUPART_FRAGMENT**: Get individual URL parts, Set individual URL parts
- **CURLUPART_HOST**: Get individual URL parts, Set individual URL parts
- **CURLUPART_PASSWORD**: Get individual URL parts, Set individual URL parts
- **CURLUPART_PATH**: Get individual URL parts, Set individual URL parts
- **CURLUPART_PORT**: Get individual URL parts, Set individual URL parts
- **CURLUPART_QUERY**: Get individual URL parts, Set individual URL parts, Append to the query
- **CURLUPART_USER**: Get individual URL parts, Set individual URL parts
- **curl_easy_cleanup**: easy handle, curl --libcurl, Get a simple HTML page, Get a response into memory, Submit a login form over HTTP, Get an FTP directory listing, Non-blocking HTTP form-post, Bearer, Enable cookie engine with writing
- **curl_easy_getinfo**: docs/libcurl/opts, Post transfer info, Response meta-data, Get all cookies from the cookie store
- **curl_easy_init**: Easy handle, CURLOPT_CURLU, curl --libcurl, Get a simple HTML page, Get a response into memory, Submit a login form over HTTP, Get an FTP directory listing, Non-blocking HTTP form-post, Bearer, libcurl HTTP download
- **curl_easy_option_by_id**: Find a specific option by ID
- **curl_easy_option_by_next**: Iterate over all options
- **curl_easy_perform**: Driving with the easy interface, Easy API pool, Caching, curl --libcurl, Get a simple HTML page, Get a response into memory, Submit a login form over HTTP, Get an FTP directory listing, Add a header, Bearer, libcurl HTTP download, Everything is multi
- **curl_easy_reset**: Reuse
- **curl_easy_setopt**: docs/libcurl/opts, Easy handle, Write callback, Read callback, Progress callback, Header callback, Debug callback, sockopt callback, Provide a file descriptor, Name resolving, Sharing between easy handles, CURLOPT_CURLU, curl --libcurl, Set handle options, Set numerical options,

Set string options, libcurl TLS options, All easy options, CURLcode return code, Verbose operations, Get a simple HTML page, Get a response into memory, Submit a login form over HTTP, Get an FTP directory listing, Non-blocking HTTP form-post, Strings are C strings, not C++ string objects, Request method, HTTP ranges, User name and password, Enable cookie engine with reading, libcurl HTTP download, HTTP POST, Multiplexing, Enable HSTS for a handle, Enable

- curl_global_cleanup: Global initialization, Get a response into memory, Get an FTP directory listing
 - curl_global_init: Global initialization, Get a response into memory, Get an FTP directory listing, Init calls
 - CURL_IPRESOLVE_V6: Name resolving
 - CURL_MAX_WRITE_SIZE: Write callback
 - curl_mime_addpart: Non-blocking HTTP form-post, HTTP multipart formposts
 - curl_mime_filedata: Non-blocking HTTP form-post, HTTP multipart formposts
 - curl_mime_init: Non-blocking HTTP form-post, HTTP multipart formposts
 - curl_mime_name: Non-blocking HTTP form-post, HTTP multipart formposts
 - curl_multi_add_handle: Driving with the multi interface, Many easy handles, Non-blocking HTTP form-post, Curl_multi
 - curl_multi_cleanup: Multi API, Non-blocking HTTP form-post
 - curl_multi_fdset: Driving with the multi interface, Curl_easy
 - curl_multi_info_read: When is a single transfer done?, When is it done?, Multi API, Curl_multi
 - curl_multi_init: Driving with the multi interface, Non-blocking HTTP form-post
 - curl_multi_remove_handle: Driving with the multi interface, Many easy handles, Multi API
 - curl_multi_setopt: docs/libcurl/opts, Driving with the multi interface, socket_callback, Multiplexing
 - curl_multi_socket_action: socket_callback
 - curl_multi_timeout: Driving with the multi interface
 - curl_multi_wait: Driving with the multi interface
 - curl_off_t: Progress callback, seek callback, Available information, Set numerical options, Response meta-data, HTTP PUT
 - CURL_SOCKET_TIMEOUT: timer_callback
 - CURL_SSL_BACKEND: Line 1: TLS versions, Multiple TLS backends
 - curl_url: Include files, Create, cleanup, duplicate, Parse a URL, Redirect to a relative URL, CURLOPT_CURLU
 - curl_url_cleanup: Create, cleanup, duplicate
 - curl_url_dup: Create, cleanup, duplicate
 - curl_url_get: CURLU_ALLOW_SPACE , Get a URL, Get individual URL parts
 - curl_url_set: Include files, Parse a URL, Redirect to a relative URL, Set individual URL parts, Append to the query, CURLOPT_CURLU
 - curl_version_info: Which libcurl version runs
-

D

- -d: Arguments to options, Separate options per URL, POST, MQTT, HTTP methods, Simple POST, Content-Type, Posting binary, Convert to GET, Expect 100-continue, Chunked encoded POSTs, Hidden form fields, -d vs -F, PUT, curl HTTP cheat sheet, Web logins and sessions
 - --data: Arguments to options, Separate options per URL, POST, Simple POST, JSON, URL encoding
 - --data-binary: Not perfect, Posting binary, URL encoding
 - --data-urlencode: URL encoding, Convert to GET
 - debian: Ubuntu and Debian, Version
 - Debug callback: Debug callback, All easy options, Verbose operations
 - development: Project communication, curl-users, Reporting bugs, Commercial support, Development, The development team, Future, Ubuntu and Debian, Who decides what goes in?, From Safari, Figure out what a browser sends, Which libcurl version runs, Verification
 - DICT: What protocols does curl support?, DICT, Without scheme, Version, DICT, CURLU_GUESS_SCHEME
-

E

- Edge: Copy as curl, Ifdefs
 - environment variables: Proxies, Proxy environment variables, Proxy environment variables
 - ETag: Conditionals
 - --etag-compare: Check by modification of content
 - --etag-save: Check by modification of content
 - etiquette: Mailing list etiquette
 - event-driven: Driving with the "multi_socket" interface, Everything is multi
-

F

- -F: multipart formpost, Not perfect, HTTP methods, HTTP POST, Sending such a form with curl, -d vs -F, curl HTTP cheat sheet
- --fail: Available exit codes, HTTP response codes
- --fail-with-body: HTTP response codes
- Firefox: libvtls, Discover your proxy, SSLKEYLOGFILE, Copy as curl, User-agent
- Fragment: URLs, Fragment, Modify the HTTP request, Anchors or fragments
- --ftp-method: multicwd
- --ftp-pasv: Passive connections

- --ftp-port: Available exit codes, Active connections
 - --ftp-skip-pasv-ip: Passive connections
 - FTPS: What protocols does curl support?, FTPS, Build to use a TLS library, Supported schemes, Network leakage, --trace and --trace-ascii, Version, Protocols allowing upload, Enable TLS, FTPS
 - future: Project communication, Future, What other protocols are there?, docs, curl-security@haxx.se, "Not used", Cookies, Multiplexing, When QUIC is denied, API compatibility, HSTS
-

G

- --get: Convert to GET
 - git: Daily snapshots, Building libcurl on MSYS2, Clone the code, root, git, Web site source code, git vs release tarballs, Notes, build boringssl
 - Globbing: URL globbing, Uploading with FTP
 - GnuTLS: lib/vtls, Select TLS backend, Build to use a TLS library, OCSP stapling, Restrictions, Proxy types
 - GOPHER: How it started, What protocols does curl support?, GOPHER, Supported schemes, Version
 - GOPHERS: What protocols does curl support?, GOPHERS, Supported schemes
-

H

- --header: Server differences, Proxy headers, JSON, Customize headers
- Header callback: Header callback, All easy options, Response body, Download headers too
- homebrew: Get curl for macOS
- Host:: Verbose mode, --trace and --trace-ascii, Change the Host: header, HTTP protocol basics, The HTTP this generates, Customize headers, Customize HTTP request headers
- HSTS: HTTP Strict Transport Security, Callbacks, HSTS, All easy options, Sections, HSTS
- --hsts: HSTS cache
- HTTP ranges: HTTP ranges, Sections, HTTP ranges
- HTTP redirects: Short options, Available exit codes, HTTP redirects, Submit a login form over HTTP
- HTTP Strict Transport Security: HTTP Strict Transport Security, HSTS
- HTTP/1.1: HTTP, Verbose mode, --trace and --trace-ascii, Debugging with TELNET, HTTP protocol basics, HTTP versions, Caveats, The HTTP this generates, GET or POST?, Modify the request method, Modify request target, HTTP/2, Customize HTTP request headers, HTTP versions, About content encodings
- HTTP/2: HTTP, docs, nghttp2, HTTP/2 and HTTP/3, Line 4: Features, Available exit codes, HTTP versions, HTTP/2 and later, GET or POST?, HTTP/2, HTTP/3, DNS over HTTPS, HTTP versions,

- Expect: headers, Multiplexing, HTTP/3, Different backends, Curl_easy
 - HTTP/3: HTTPS, Select HTTP/3 backend, TCP vs UDP, HTTP/2 and HTTP/3, Line 4: Features, Available exit codes, HTTP versions, HTTP/3, HTTP/3, Which libcurl version runs, HTTP versions, Expect: headers, Multiplexing, HTTP/3, Different backends
 - HTTP/3 backend: Select HTTP/3 backend
 - --http0.9: Accepting HTTP/0.9
 - --http1.1: HTTP versions
 - --http2: HTTP versions, HTTP/2
 - --http2-prior-knowledge: HTTP versions, HTTP/2
 - --http3: HTTP versions, Enable
 - HttpGet: How it started
-

I

- IDN: libidn2, International Domain Names (IDN), Version, CURLU_URL_ENCODE , Different backends
 - Indentation: Indentation
 - IPv4: Host, Port number, Available --write-out variables, Name resolving, All easy options, host*.c sources
 - IPv6: Host, Port number, URL globbing, Available --write-out variables, Version, Name resolving, All easy options, CURLRES_IPV6
 - IRC: How it started, Project communication
-

J

- JavaScript: Client differences, PAC, HTTP POST, JavaScript and forms, JavaScript redirects, Figure out what the browser does
 - --json: JSON
 - json: Arguments with spaces, Available --write-out variables, HTTP POST, Content-Type, JSON, POST outside of HTML
-

K

- -K: Command lines, quotes and aliases, Config file
- keep-alive: Keep connections alive, All easy options
-

- `--keepalive-time`: [Keep alive, Keep connections alive](#)
-

L

- `-L`: [Short options, Available --write-out variables, Tell curl to follow redirects, Modify the request method, Cookie engine, curl HTTP cheat sheet, Redirects](#)
 - `--libcurl`: [curl --libcurl](#)
 - `libcurl version`: [The latest version?, Line 1: curl, Available exit codes, Which libcurl version](#)
 - `libidn2`: [libidn2](#)
 - `libpsl`: [libpsl](#)
 - `libressl`: [Build to use a TLS library, Restrictions](#)
 - `librtmp`: [librtmp](#)
 - `libssh`: [SSH libraries, SCP and SFTP](#)
 - `libssh2`: [Running DLL based configurations, SSH libraries, SCP and SFTP](#)
 - `license`: [Finding users, License, root, License](#)
 - `--limit-rate`: [Rate limiting](#)
 - `--location`: [Long options, Separate options per URL, Config file, Tell curl to follow redirects](#)
-

M

- `--max-filesize`: [Maximum filesize](#)
 - `--max-time`: [Tweak your retries, Maximum time allowed to spend](#)
 - `MIT`: [License](#)
 - `MQTT`: [What protocols does curl support?, MQTT, Supported schemes, Line 3: Protocols, MQTT](#)
 - `mTLS`: [Client certificates](#)
 - `multi-threading`: [libcurl multi-threading](#)
-

N

- `name resolving`: [Host name resolving, Handling different build options, Available --write-out variables, Name resolve tricks with c-ares, SOCKS proxy, Connection reuse, Name resolving, Proxy types, Available information, Different backends](#)
- `--negotiate`: [Network leakage, HTTP authentication](#)
-

- --netrc-file: [Enable netrc](#)
 - --netrc-optional: [Enable netrc](#)
 - nghttp2: [nghttp2](#), Which libcurl version runs
 - nix: [nix](#)
 - --no-eprt: [Active connections](#)
 - --no-epsv: [Passive connections](#)
 - NPN: [All easy options](#)
 - NSS: [Build to use a TLS library](#), [OCSP stapling](#), [Restrictions](#), [Proxy types](#)
 - --ntlm: [Network leakage](#), [HTTP authentication](#)
-

O

- -O: Many options and URLs, Numerical ranges, Download to a file named by the URL, Use the target file name from the server, Shell redirects, Multiple downloads, Resuming and ranges, Authentication, Download, Check by modification date, curl HTTP cheat sheet
 - openldap: [openldap](#)
 - OpenSSL: Get curl and libcurl on MSYS2, libvtls, Select TLS backend, Running DLL based configurations, Build to use a TLS library, OCSP stapling, Restrictions, SSL context callback, Proxy types, Available information, All easy options
-

P

- PAC: [Proxies](#), [PAC](#), Which proxy?
- --parallel: [Parallel transfers](#)
- --parallel-immediate: [Connection before multiplex](#)
- --parallel-max: [Parallel transfers](#)
- --path-as-is: [--path-as-is](#)
- Percent-encoding: [URL encoding](#)
- pop3: [What protocols does curl support?](#), [POP3](#), Without scheme, Verbose mode, Version, Available exit codes, Reading email, Secure mail transfer, Enable TLS, `CURLOPT_GUESS_SCHEME`, `STARTTLS`
- port number: Connects to "port numbers", URLs, Port number, Available --write-out variables, Connections, Provide a custom IP address for a name, Local port number, HTTP proxy, Available exit codes, Historic TELNET, The URL converted to a request, Converting a web form, Enable, Implicit FTPS, Connection reuse, Prereq, Custom addresses for hosts, Proxies, Post transfer info, `CURLOPT_DEFAULT_PORT`, Set individual URL parts, All easy options
- --post301: Decide what method to use in redirects
-

- post302: Decide what method to use in redirects
 - post303: Decide what method to use in redirects
 - Progress callback: [timer_callback](#), [Progress callback](#), All easy options
 - pronunciation: [Pronunciation](#)
 - --proxy: [HTTP proxy](#), [HTTP authentication](#)
 - proxy: [How it started](#), [Available --write-out variables](#), [Line 4: Features](#), [Intermediaries' fiddlings](#), [Proxies](#), [Discover your proxy](#), [PAC](#), [Proxy type](#), [HTTP proxy](#), [SOCKS proxy](#), [MITM proxy](#), [Proxy authentication](#), [HTTPS proxy](#), [Proxy environment variables](#), [Proxy headers](#), [Available exit codes](#), [CONNECT response codes](#), [HTTP authentication](#), [Proxies](#), [Available information](#), [Verification](#), All easy options, [HTTP proxy](#)
 - --proxy-user: [Proxy authentication](#), [HTTP authentication](#)
 - --proxy1.0: [HTTP proxy tunneling](#)
 - --proxytunnel: [HTTP proxy tunneling](#)
-

Q

- -Q: [Quote](#)
 - --quote: [Quote](#)
-

R

- ranges: [Numerical ranges](#), [Downloads](#), [Resuming and ranges](#), [HTTP ranges](#), [Provide a file descriptor](#), [Sections](#), [HTTP response code](#), [HTTP ranges](#)
- Read callback: [Read callback](#), All easy options
- redhat: [Redhat and CentOS](#)
- redirects: [Long options](#), [Separate options per URL](#), [Config file](#), [Available --write-out variables](#), [Downloads](#), [Download to a file named by the URL](#), [Shell redirects](#), [Provide a custom IP address for a name](#), [Available exit codes](#), [HTTP redirects](#), [Modify the request method](#), [Redirects](#), [Custom addresses for hosts](#), [Available information](#), All easy options, [Submit a login form over HTTP](#), [Automatic referrer](#)
- RELEASE-NOTES: [scripts](#)
- releases: [1. [The cURL project](#)]([how-to-read.md#1.-the-curl-project](#)), [curl-announce](#), [Releases](#), [scripts](#), [Releases source code](#), [Verbose mode](#), [Which libcurl version](#)
- --remote-name-all: [One output for each given URL](#), [Use the URL's file name part for all URLs](#)
- repository: [Releases](#), [Source code on GitHub](#), [Arch Linux](#), [Building libcurl on MSYS2](#), [Hosting and download](#), [root](#), [What to add](#), [Web site source code](#), [git vs release tarballs](#), [Notes](#), [Content](#)
- --resolve: [Provide a custom IP address for a name](#)
- --retry: [Retrying failed attempts](#)
-

- --retry-all-errors: [Retry on any and all errors](#)
 - --retry-connrefused: [Connection refused](#)
 - --retry-delay: [Tweak your retries](#)
 - --retry-max-time: [Tweak your retries](#)
 - RFC 1436: [GOPHER](#)
 - RFC 1738: [FILE](#), [multicwd](#)
 - RFC 1939: [POP3](#)
 - RFC 1945: [HTTP redirects](#)
 - RFC 2229: [DICT](#)
 - RFC 2246: [TLS versions](#)
 - RFC 2326: [RTSP](#)
 - RFC 2595: [IMAP](#)
 - RFC 2818: [HTTPS](#)
 - RFC 3207: [SMTP](#)
 - RFC 3501: [IMAP](#)
 - RFC 3986: [URLs](#)
 - RFC 4217: [FTPS](#)
 - RFC 4511: [LDAP](#)
 - RFC 5321: [SMTP](#)
 - RFC 7230: [HTTP](#)
 - RFC 7540: [HTTP](#)
 - RFC 7838: [Alternative Services](#)
 - RFC 8314: [IMAPS](#)
 - RFC 8446: [TLS versions](#)
 - RFC 854: [TELNET](#)
 - RFC 959: [FTP](#), [FTP with curl](#), [Quote](#)
 - roadmap: [Future](#)
 - RTMP: [What protocols does curl support?](#), [RTMP](#), [librtmp](#), [Supported schemes](#), [Version](#)
 - RTSP: [What protocols does curl support?](#), [RTSP](#), [Supported schemes](#), [Version](#), [Callbacks](#), [RTSP interleave callback](#), [Available information](#), [All easy options](#)
 - rustls: [Build to use a TLS library](#)
 - rustls-ffi: [Select TLS backend](#), [Rustls](#)
-

S

- Safari: [Copy as curl](#)

- Schannel: Build to use a TLS library, CA store on windows
 - Scheme: Connects to "port numbers", FILE, Naming, librtmp, URLs, Scheme, Name and password, TCP vs UDP, Available --write-out variables, Proxy type, SOCKS proxy, Proxy authentication, Available exit codes, TLS for emails, Proxy types, Available information, CURLU_NON_SUPPORT_SCHEME , CURLU_DEFAULT_PORT , Which libcurl version, Get a response into memory, HTTPS, HTTP authentication, Bindings for libcurl, Protocol handler, Curl_handler
 - SCP: What protocols does curl support?, SCP, SSH libraries, Supported schemes, Version, Protocols allowing upload, Available exit codes, SCP and SFTP, All easy options, Curl_handler
 - security: curl-announce, Commercial support, Security, Trust, TLS, How much do protocols change?, FTPS, docs, Reporting vulnerabilities, http_proxy in lower case only, TLS, TLS ciphers, Enable TLS, TLS versions, HTTP with curl, Accepting HTTP/0.9, HTTP Strict Transport Security, HSTS, URL API, Protocol version, All easy options, HTTPS
 - SFTP: What protocols does curl support?, SFTP, SSH libraries, Supported schemes, --trace and --trace-ascii, Version, Protocols allowing upload, Available exit codes, SCP and SFTP, All easy options, Curl_handler
 - --show-error: Silence
 - --silent: The progress meter, Silence, Error message
 - SMTP: What protocols does curl support?, SMTP, Without scheme, Verbose mode, Version, Protocols allowing upload, Available exit codes, Sending email, Enable TLS, CURLU_GUESS_SCHEME , STARTTLS, All easy options
 - SMTPS: What protocols does curl support?, SMTPS, Build to use a TLS library, Supported schemes, Version, Protocols allowing upload, Enable TLS
 - snapshots: Daily snapshots, root
 - SNI: Change the Host: header
 - --socks4: SOCKS proxy
 - --socks4a: SOCKS proxy
 - --socks5: SOCKS proxy
 - --socks5-hostname: SOCKS proxy
 - --speed-limit: Transfer speeds slower than this means exit
 - --speed-time: Transfer speeds slower than this means exit
 - SSH: SCP, Select SSH backend, SSH libraries, Available exit codes, SCP and SFTP, Historic TELNET, Callbacks, SSH key callback, All easy options, Trace everything, Different backends, Curl_handler
 - SSH backend: Select SSH backend
 - SSL context callback: SSL context callback, All easy options
 - SSLKEYLOGFILE: TLS, SSLKEYLOGFILE
 - STARTTLS: IMAP, TLS for emails, Explicit FTPS, STARTTLS
-

T

- -T: [PUT](#), [Upload](#), [HTTP methods](#), [PUT](#), curl [HTTP cheat sheet](#), [Uploading with FTP](#)
- TELNET: [What protocols does curl support?](#), [TELNET](#), [Supported schemes](#), [Version](#), [Available exit codes](#), [TELNET](#), [All easy options](#)
- testing: [What does curl do?](#), [Reporting bugs](#), [Handling different build options](#), [Contributing](#), [Run a local clone](#), [About HTTP response code "errors"](#)
- TFTP: [What protocols does curl support?](#), [TFTP](#), [Supported schemes](#), [TCP vs UDP](#), [Version](#), [Protocols allowing upload](#), [Available exit codes](#), [TFTP](#), [All easy options](#)
- --tftp-blksize: [TFTP options](#)
- --tftp-no-options: [TFTP options](#)
- --time-cond: [Check by modification date](#)
- TLS: [TLS](#), [How much do protocols change?](#), [GOPHERS](#), [Ubuntu and Debian](#), [libvtls](#), [Handling different build options](#), [Learn more](#), [Select TLS backend](#), [TLS libraries](#), [Build to use a TLS library](#), [Connection reuse](#), [Verbose mode](#), [Line 1: curl](#), [Change the Host: header](#), [MITM proxy](#), [Available exit codes](#), [SCP](#) and [SFTP](#), [TLS for emails](#), [Caveats](#), [TLS](#), [TLS ciphers](#), [Enable TLS](#), [TLS versions](#), [Verifying server certificates](#), [Certificate pinning](#), [OCSP stapling](#), [Client certificates](#), [TLS auth](#), [TLS backends](#), [SSLKEYLOGFILE](#), [HTTP with curl](#), [The URL converted to a request](#), [HTTPS](#), [Figure out what a browser sends](#), [HTTPS only](#), [TLS fingerprinting](#), [FTPS](#), [SSL context callback](#), [Proxy types](#), [Available information](#), [libcurl TLS options](#), [All easy options](#), [Trace everything](#), [Caches](#), [HTTPS](#), [Different backends](#)
- TLS backend: [Ubuntu and Debian](#), [libvtls](#), [Select TLS backend](#), [Line 1: curl](#), [Available exit codes](#), [TLS](#), [Certificate pinning](#), [Client certificates](#), [TLS backends](#), [SSL context callback](#)
- TODO: [Future](#), [Suggestions](#), [Notes](#)
- --tr-encoding: [Compression](#), [Transfer encoding](#)
- --trace: [--trace](#) and [--trace-ascii](#)
- --trace-ascii: [--trace](#) and [--trace-ascii](#), [Server differences](#), curl [HTTP cheat sheet](#)
- --trace-time: [--trace-time](#)
- transfer-encoding: [Pass on transfer encoding](#), [Chunked encoded POSTs](#)

U

- -U: [Building libcurl on MSYS2](#), [Proxy authentication](#)
 - -u: [Building libcurl on MSYS2](#), [Passwords and snooping](#), [URLs](#), [IMAP](#), [HTTP authentication](#), curl [HTTP cheat sheet](#), [Authentication](#)
 - Ubuntu: [Ubuntu and Debian](#)
 - URL Globbing: [URL globbing](#)
-

V

- --verbose: [Long options](#), [Verbose mode](#), [--trace-time](#)
 - --version: [Version](#), [TLS backends](#), [Memory Debugging](#)
 - Vulnerability: [Vulnerability handling](#)
-

W

- Wireshark: [--trace](#) and [--trace-ascii](#), [Available exit codes](#), [SSLKEYLOGFILE](#), [Figure out what a browser sends](#)
 - wolfSSH: [SSH libraries](#), [SCP and SFTP](#)
 - wolfSSL: [Commercial support](#), [lib/vtls](#), [Running DLL based configurations](#), [Build to use a TLS library](#), [Restrictions](#), [SSL context callback](#), [All easy options](#)
 - Write callback: [Write callback](#), [All easy options](#), [Get a response into memory](#), [Callback considerations](#)
 - --write-out: [--write-out](#), [HTTP response codes](#)
-

X

- -X: [Modify the request method](#), [Modify request target](#), [PUT](#), [curl HTTP cheat sheet](#)
 - -x: [HTTP proxy](#), [SOCKS proxy](#), [Proxy authentication](#), [Proxy environment variables](#), [Proxy headers](#), [curl HTTP cheat sheet](#), [Proxy environment variables](#)
-

Y

- yum: [Redhat and CentOS](#)
-

Z

- -z: [Check by modification date](#)
- -Z: [Parallel transfers](#)
- zlib: [HTTP Compression](#), [About content encodings](#)
- zstd: [HTTP Compression](#), [Which libcurl version runs](#), [Supported content encodings](#)