curl Course

Flavio Copes

Updated August 3, 2026

Contents

Use curl to inspect, reproduce, debug, and automate HTTP and API transfers with clear security and reliability boundaries.

What you'll learn: Build precise requests, inspect every transfer stage, handle authentication and TLS safely, and turn a request into a reliable smoke test.

Request basics

Install curl, make requests, inspect the exchange, and save response data.

Inspect your curl installation

Check the installed curl version, supported protocols, TLS backend, and features before relying on an option.

curl is both a command-line tool and a transfer library (libcurl) that other programs embed. The tool you type is built from many optional pieces, and builds differ by operating system, TLS backend, protocol support, and enabled features. The curl on a fresh macOS is not the same build as the curl in an Alpine container, even when the version numbers look close.

That is why this course starts here. When an option misbehaves, the first question is always: what exact curl am I running?

Read the version output

Inspect the build you will use for every lab:

```
curl --version
```

You get something like:

```
curl 8.7.1 (x86_64-apple-darwin24.0) libcurl/8.7.1 (SecureTransport) LibreSSL/3.3.6
Release-Date: 2024-03-27
Protocols: dict file ftp ftps http https imap imaps ...
Features: alt-svc AsynchDNS HSTS HTTP2 IPv6 SSL ...
```

Read it line by line. The first line names the version, the platform, and the TLS backend — the library curl uses for HTTPS. The TLS backend explains real behavioral differences, like where curl looks for trusted certificates. The `Protocols` line lists every URL scheme this build accepts. The `Features` line tells you whether things like HTTP/2 support were compiled in.

Save the output with your lab notes so later behavior has a concrete version attached. "It failed on curl 8.7.1 with LibreSSL" is evidence. "It failed on my laptop" is not.

Match the documentation to the build

Options get added in specific versions. `--json`, for example, only exists since curl 7.82.0, and plenty of servers still run older builds. Do not assume an online man page exactly matches an older system curl.

Ask your own build instead:

```
curl --help all | grep json
man curl
```

`curl --help all` lists every option this binary understands, and `man curl` opens the manual that shipped with it. If an option from a tutorial is missing from that list, the tutorial is not wrong — your curl is older, and now you know before wasting time on mystery errors.

Make a GET request

Request one URL, see the response body, and separate curl output from the HTTP exchange.

With no method option, an HTTP URL normally produces a GET request: the plain ”give me this resource” verb of the web. This is the command you will run more than any other, so it is worth understanding exactly what appears on your screen and why.

Request a small public page:

```
curl https://example.org/
```

The terminal should contain the returned HTML, starting with `<!doctype html>`. That text is the response body, exactly as the server sent it. curl adds nothing and interprets nothing — no rendering, no styling, just bytes.

Two output streams, one terminal

curl writes the response body to standard output and progress information to standard error. The progress meter is not part of that HTML, even though both may appear in one terminal window. You can prove the separation by sending each stream somewhere different:

```
curl https://example.org/ > page.html
```

The redirect captures only stdout, so `page.html` contains pure HTML while any progress output stays on screen. This separation is what makes curl composable: you can pipe the body into another tool without progress noise corrupting it.

When you want no progress output at all, add `--silent`:

```
curl --silent https://example.org/ | wc -c
```

The number printed is the body size in bytes, and nothing else leaked into the pipe.

Quote your URLs

Quote URLs containing `?`, `&`, or shell wildcard characters. The shell processes those characters before curl sees them. An unquoted `&` puts curl in the background mid-URL, and an unquoted `?` can match filenames in your current directory:

```
curl 'https://example.org/search?q=curl&page=2'
```

Single quotes hand the URL to curl untouched. Make this a habit now, before it costs you a confusing afternoon.

If you run a URL and get nothing back, check the exit status with `echo $?`. Zero means the transfer worked and the body was genuinely empty or went where you redirected it. Non-zero means the transfer itself failed, and the code tells you how — `6` is a DNS failure, `7` means the connection was refused.

Inspect a request with verbose output

Use verbose mode to separate DNS, connection, TLS, request headers, and response headers.

A single curl command hides a lot of work: a DNS lookup, a TCP connection, a TLS handshake, a request, a response. When something misbehaves, you need to see which of those stages went wrong. **Verbose mode** exposes all of them.

Inspect one HTTPS request:

```
curl -v https://example.org/ -o /dev/null
```

The body is discarded with `-o /dev/null` for a cleaner trace, so everything left on screen is diagnostics.

Read the prefixes

Every verbose line starts with a marker that tells you who said what. Lines beginning with `>` are sent headers, lines beginning with `<` are received headers, and lines beginning with `*` describe curl events like name resolution and TLS negotiation.

```
* Host example.org:443 was resolved.
* Connected to example.org port 443
* SSL connection using TLSv1.3
> GET / HTTP/2
> Host: example.org
> User-Agent: curl/8.7.1
< HTTP/2 200
< content-type: text/html
```

Follow the order: address resolution, connection, TLS negotiation, request, response, then connection reuse or closure. That sequence is your debugging map. A failure during the `*` lines is a network or TLS problem and never reached the server. A `>` request followed by an unexpected `<` status means the server received you and disagreed.

Use it to answer real questions

Verbose output settles arguments that guessing cannot. Did curl actually send the header you added? Look for it in the `>` lines. Which protocol version was negotiated? It is in the `*` and `>` lines. Is the connection being reused across requests? Run two URLs in one command and look for a `Re-using existing connection` event.

Redact before you share

Verbose output can expose authorization headers, cookies, and other secrets, because it prints every header exactly as sent. Before pasting a trace into a ticket, chat, or blog post, redact those values. The habit matters: the whole point of `-v` is that nothing is hidden, and that includes things that should stay hidden from other people.

Save and name response files

Save a response under an explicit name and understand when the server-provided name is safe to use.

By default curl prints the response body to your terminal. For downloads you want a file, and there are two ways to name it. Use `--output` when the destination filename matters. `--remote-name` (or `-O`) derives a local name from the URL path, which is convenient only when you trust that name.

Name the file yourself

Download a text file with an explicit destination:

```
curl --fail --output robots.txt https://curl.se/robots.txt
```

Check the exit status and inspect the saved file:

```
echo $?        # 0 means the transfer and the HTTP status were both fine
cat robots.txt
```

`--fail` is the option people forget, and it matters. Without it, a 404 page is a "successful" transfer: curl happily saves the HTML error page as `robots.txt` and exits with 0. Your script continues, and the corruption surfaces much later, far from its cause. `--fail` makes HTTP responses at or above 400 produce a failed command instead of silently becoming the saved content — exit code 22, no misleading file.

Try it against a URL that does not exist. `example.org` serves no robots.txt, so `curl --fail --output robots.txt https://example.org/robots.txt` prints `curl: (22) The requested URL returned error: 404` and leaves nothing on disk. Some HTTP/2 transfers report the same failure as exit code 56 — either way, non-zero.

Let the URL name the file

When mirroring a file whose name you already know and trust:

```
curl --fail -O https://curl.se/robots.txt
```

curl takes the last path segment, `robots.txt`, and writes to that name in the current directory. The name comes from a URL you typed, so the trust question is easy here. It stops being easy with `--remote-header-name` (`-J`), which lets the *server* pick the filename via a response header. A hostile or compromised server could suggest a name you did not expect, so treat that option with care and never combine it with directories you care about.

For large downloads that get interrupted, resume instead of restarting:

```
curl --fail -C - --output ubuntu.iso https://releases.ubuntu.com/24.04/ubuntu-24.04.4-desktop
```

`-C -` tells curl to look at the partial file and continue from where it stopped.

Do not execute what you have not read

Do not pipe an unverified download directly into a shell. The popular `curl ... | bash` pattern runs whatever bytes arrive, instantly, with your permissions. Save it, inspect it, and verify its expected source or checksum first. The two extra commands cost seconds; running an attacker's script costs considerably more.

Check your understanding: request basics

Check the commands, decisions, safety boundaries, and failure cases from request basics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Send request data

Build query strings, headers, JSON bodies, forms, uploads, and deliberate methods.

Build a query string safely

Encode query values with curl instead of manually replacing spaces and reserved characters.

Query values can contain spaces, ampersands, Unicode, and other characters with URL meaning. A URL cannot carry those characters raw. A space must become `%20`, an `&` inside a value must become `%26`, and so on. Doing that by hand is tedious and easy to get wrong. One missed character and the server parses your query differently than you intended.

Let curl encode each value from its original form instead.

Ask curl to create the query string

```
curl --get --data-urlencode 'q=network tools' https://httpbin.org/get
```

`--data-urlencode` takes a `name=value` pair and percent-encodes the value. `--get` moves the encoded data to the query string instead of sending a request body. Without `--get`, the data options would turn this into a POST.

Inspect the returned URL and arguments. httpbin echoes what it received:

```
{
  "args": {
    "q": "network tools"
  },
  "url": "https://httpbin.org/get?q=network+tools"
}
```

The value arrived intact. curl turned the space into a safe form on the wire, and the server decoded it back to `network tools`. You wrote the value once, in its natural form.

Multiple parameters

Repeat the option for each pair:

```
curl --get \
  --data-urlencode 'q=curl & friends' \
  --data-urlencode 'page=2' \
  https://httpbin.org/get
```

curl joins the pairs with `&` in the final URL. The `&` inside the first value gets encoded as `%26`, so it stays part of the value instead of splitting the query.

The mistake to avoid

Quote each value so the shell cannot split it or interpret `&` as a background operator. This command looks close but breaks twice:

```
curl --get --data-urlencode q=network tools https://httpbin.org/get
```

The shell splits on the space, so `tools` becomes a separate argument that curl tries to use as a URL. You'll see an error like `Could not resolve host: tools`. That message is your signal: the shell carved up your command before curl ever saw it. Put the whole `name=value` pair in single quotes and the problem disappears.

Send JSON

Send a JSON request body with the correct content type and preserve the exact bytes from a file when needed.

JSON is the default language of HTTP APIs, and for years sending it with curl meant three options: `--data` for the body, a `--header` for the content type, and careful quoting in between. Modern curl (7.82.0 and later) provides `--json`, which sends JSON data and adds suitable `Content-Type` and `Accept` headers in one move.

Send an object

Send a small JSON object:

```
curl --json '{"name":"Mina","active":true}' https://httpbin.org/anything
```

Inspect the echoed headers and parsed body:

```
{
  "headers": {
    "Accept": "application/json",
    "Content-Type": "application/json"
  },
  "json": {
    "active": true,
    "name": "Mina"
  },
  "method": "POST"
}
```

Three things to verify in that echo. The method became POST, because sending data implies it. `Content-Type: application/json` tells the server how to parse the body — without it, many frameworks refuse the request or ignore the body entirely. And the `json` field proves the server parsed your object, not a mangled version of it.

Note the single quotes around the JSON. They protect the double quotes inside from the shell. This is the most common failure: with the quoting wrong, the server receives `{name:Mina}` or a shell error appears before curl even runs.

Send a file

For larger documents, use `--json @request.json` instead of fighting shell quoting:

```
curl --json @request.json https://httpbin.org/anything
```

The `@` prefix reads the body from the file, byte for byte. No escaping, no quoting puzzles, and the file can be validated first with a tool like `jq . request.json`, which fails loudly on broken JSON before the server ever sees it.

On an older curl without `--json`, the equivalent is explicit:

```
curl --data '{"name":"Mina","active":true}' --header 'Content-Type: application/json' https:/
```

Same request on the wire, just assembled by hand.

Keep credentials out of examples

Request bodies get pasted into tickets, committed in test scripts, and stored in shell history. Never place long-lived API credentials inside the JSON example, shell history, or a committed script. A well-formed request with a real secret in it is still a leak.

The server still decides whether the JSON shape is valid. `--json` guarantees correct headers and intact bytes — the schema contract is between you and the API.

Submit forms and files

Choose URL-encoded form data or multipart form data and upload a local file deliberately.

HTML-style forms commonly send data in one of two encodings. **URL-encoded** data packs fields into `name=value` pairs, the same format as a query string, sent as the request body. **Multipart** data splits the body into parts separated by a boundary, each with its own headers. Multipart is the normal choice when a field contains a file, because files are binary and do not survive URL encoding gracefully.

curl has one option for each: `--data` for URL-encoded, `--form` for multipart.

A simple form

For text-only fields, URL-encoded is what most login and search forms use:

```
curl --data 'title=Lab notes' --data 'author=mina' https://httpbin.org/post
```

The echo shows both fields under `form`, and the `Content-Type` header reads `application/x-www-form-urlencoded`.

A form with a file

Create a small file, then send one text field and one file:

```
echo 'first line of my notes' > notes.txt
curl --form 'title=Lab notes' --form 'attachment=@notes.txt' https://httpbin.org/post
```

The response should show a form value and file content separately:

```
{
  "files": {
    "attachment": "first line of my notes\n"
  },
  "form": {
    "title": "Lab notes"
  }
}
```

That split is your verification. `title` arrived as an ordinary field, and `attachment` arrived as a file part with its content intact. curl builds the multipart boundary and per-part headers for you — the fiddly parts of the format you never want to write by hand.

The `@` prefix reads a local file and uploads it as a file part. If you want the file's *content* as a plain text field instead, use `<` rather than `@`. And when the server cares about the file's MIME type, state it: `--form 'attachment=@notes.txt;type=text/plain'`.

The mistake to avoid

Check the path and destination before running a command copied from elsewhere. `@` means ”read this file from my disk and send it over the network” — pasted blindly, a command like `--form 'config=@~/.ssh/id_ed25519'` uploads your private key to whoever runs the server. curl also

fails with `26` when the file does not exist, so a typo in the path shows up as a read error, not a silent empty upload.

One more trap: mixing `--data` and `--form` in the same command does not work. A request body has one content type. Pick the encoding the server expects and stick to it.

Choose a method deliberately

Understand when curl infers POST or HEAD and when an explicit custom method changes only the method token.

You rarely tell curl which HTTP method to use. Some curl options imply one. A plain URL produces GET. `--data` implies POST, because sending a body is what POST is for. `--head` requests headers through HEAD. `--upload-file` implies PUT. Each option switches the method because the method matches what the option does.

Then there's `--request` (short form `-X`). It changes the method string but does not add the behavior normally associated with that method. It swaps one word in the request line and nothing else.

See the difference

Compare a HEAD request with an explicit method:

```
curl --head https://example.org/
curl --request DELETE https://httpbin.org/anything
```

The first transfer expects no response body. That's real behavior: curl knows HEAD responses carry headers only, so it prints them and doesn't wait for content.

The second sends DELETE, and httpbin's echo confirms it:

```
{
  "method": "DELETE",
  "data": "",
  "json": null
}
```

But curl does not invent authentication, a JSON body, or application semantics. It sent an ordinary request whose method token happens to say `DELETE`. Whether anything gets deleted is entirely the server's decision.

Where -X goes wrong

The classic mistake is `-X GET` combined with `--data`. The request line says GET, but a body still goes out, because `-X` only renames the method. Some servers ignore bodies on GET, others reject them. If you want a GET with data in the URL, the right tool is `--get` with the data options, not `-X`.

My advice: reach for `-X` only when the API genuinely requires a method curl has no native option for, like DELETE or PATCH. Let curl's own options pick the method everywhere else, because they adjust the rest of the request to match.

Use the method required by the API contract, and be careful with the destructive ones. A successful connection does not mean a destructive method is authorized or safe. Point DELETE at test endpoints like `httpbin.org/anything` until you've read what the real API does with it.

Check your understanding: send request data

Check the commands, decisions, safety boundaries, and failure cases from send request data.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

State and authentication

Send credentials, bearer tokens, cookies, redirects, and sensitive data with clear boundaries.

Use Basic authentication

Send Basic authentication over verified HTTPS without embedding credentials in a shared URL.

HTTP Basic authentication is the oldest and simplest auth scheme on the web: the client sends a username and password in an `Authorization` header with every request. The pair is joined with a colon and Base64-encoded — encoded, not encrypted. Base64 is trivially reversible, so the scheme itself protects nothing. HTTPS must protect the connection, or you are broadcasting a password.

You still meet Basic auth constantly: staging environments behind a shared password, internal dashboards, webhooks, and countless APIs that use it for token-plus-empty-password schemes.

Authenticate a request

Use a disposable lab credential against an endpoint built for practice:

```
curl --user 'student:practice-only' https://httpbin.org/basic-auth/student/practice-only
```

A matching credential returns success:

```
{
  "authenticated": true,
  "user": "student"
}
```

Change the password and run it again. The server answers `401 Unauthorized` and curl prints nothing useful — add `--write-out '%{response_code}\n'` to see the status, or `--fail` to turn the 401 into exit code 22 a script can act on.

What actually happened: curl took `student:practice-only`, Base64-encoded it to `c3R1ZGVudDpwcmFjdGljZS1vbmx5`, and sent `Authorization: Basic c3R1ZGVudDpwcmFjdGljZS1vbmx5`. Run `echo 'c3R1ZGVudDpwcmFjdGljZS1vbmx5' | base64 -d` and the password stares back at you. That is why the HTTPS requirement is absolute.

Add `-v` only in a private terminal, because verbose output can reveal the `Authorization` header.

Keep the password out of history

The command above stores the password in your shell history. For real credentials, give `--user` only the username and let curl prompt:

```
curl --user student https://httpbin.org/basic-auth/student/practice-only
# Enter host password for user 'student':
```

The password never touches history, process listings, or your screen.

Avoid the other tempting shortcut too: credentials embedded in the URL, like `https://student:practice-only@httpbin.org/...`. URLs end up in logs, browser history, and pasted messages, and a URL-shaped secret gets copied around without anyone noticing it contains a password.

My advice is a simple hierarchy. Prefer a prompt for interactive use, a protected configuration file or secret manager for automation, and never a credential typed inline in a shared terminal or committed script.

Send a bearer token

Place an API token in the Authorization header and keep it out of source, logs, and command history.

Most APIs today authenticate with a **bearer token**: a string you send in the `Authorization` header, prefixed with the word `Bearer`. The name is literal. Whoever bears the token gets the access — there is no password prompt, no second factor, nothing else. A bearer token grants its holder authority, so treat it like a password with the exact scope and lifetime assigned by the service.

That framing drives everything in this lesson. Sending the token is one line. Keeping it from leaking is the actual skill.

Send the header

Read a disposable token from an environment variable:

```
export LAB_API_TOKEN='demo-token-for-practice'
curl --header "Authorization: Bearer $LAB_API_TOKEN" https://api.lab.test/profile
```

The shell expands the variable before curl runs, so the header arrives as `Authorization: Bearer demo-token-for-practice`. The token is absent from the script file, which is the point: scripts get committed, and grepping a repository for `Bearer` finds leaked credentials depressingly often.

You can verify the header reaches a server with a public echo endpoint:

```
curl --header "Authorization: Bearer $LAB_API_TOKEN" https://httpbin.org/bearer
```

A response of `{"authenticated": true, "token": "demo-token-for-practice"}` proves the exact header the server saw. If the API you are really calling returns `401 Unauthorized`, and this echo check shows the header intact, the problem is the token itself — expired, wrong scope, wrong environment — not your curl command.

Where tokens still leak

The environment variable keeps the token out of your script, but it may still appear in process inspection or debug output. Know the escape routes:

Shell history records commands where you pasted the token literally. `set -x` in a script prints every expanded command, token included. Verbose curl output (`-v`) prints the `Authorization` header. Screenshots and pasted terminal output carry whatever was on screen.

Avoid all of those around real credentials. And do not use a real token in this lab — the echo endpoint reflects the token back in plain text, which is exactly what you never want happening to a credential that works somewhere.

When a token does leak, revoke it at the service and issue a new one. Rotating is cheap. Hoping nobody saw it is not a strategy.

Store and reuse cookies

Use a cookie jar to preserve server state across separate curl commands and inspect what is stored.

HTTP is stateless, so servers use cookies to recognize you across requests: log in once, and a session cookie proves who you are on every request after. Your browser handles this invisibly. curl does not persist cookies between commands unless you provide a **cookie jar** — each curl invocation is a stranger by default.

That default is why "it works in the browser but not in curl" so often comes down to a missing cookie. To script anything session-based, you need the jar.

Write, then read

Write and reuse a temporary cookie jar:

```
curl --cookie-jar cookies.txt 'https://httpbin.org/cookies/set?theme=dark'
curl --cookie cookies.txt https://httpbin.org/cookies
```

The first command hits an endpoint that sets a cookie, and `--cookie-jar cookies.txt` saves everything the server set. The second command sends the jar back with `--cookie cookies.txt`. The second response should show the stored cookie:

```
{
  "cookies": {
    "theme": "dark"
  }
}
```

Two separate processes, one continuous session. That is the whole mechanism.

For a realistic login flow you usually want both options on every command — read existing cookies, and save any updates the server sends:

```
curl --cookie cookies.txt --cookie-jar cookies.txt https://httpbin.org/cookies
```

Look inside the jar

The jar is a plain text file in the Netscape cookie format, one cookie per line:

```
httpbin.org FALSE   /   FALSE   0   theme   dark
```

Open it and connect each field to the request logic: the domain and path decide which requests the cookie accompanies, the secure flag restricts it to HTTPS, and the expiry (`0` means a session cookie) decides how long it lives. When a cookie mysteriously fails to be sent, the answer is almost always in one of these fields — a domain that does not match, a path that is too narrow, or an expiry already passed.

The jar is a credential

Cookie jars can contain session credentials. A saved session cookie is login-equivalent: anyone who reads the file can be you until the session expires. Keep jars out of Git, restrict permissions with `chmod 600 cookies.txt`, and remove the lab file when finished:

```
rm cookies.txt
```

Treat the jar with exactly the care you would give the password that created it.

Follow redirects with boundaries

Follow redirect chains, inspect every hop, and understand why credentials need special care across hosts.

A redirect tells the client to request another URL. The server answers with a 3xx status and a `Location` header pointing somewhere else. Browsers follow that pointer automatically. curl reports it but follows it only when you enable location handling with `--location` (short form `-L`).

That default is a feature. It means curl never visits a URL you didn't ask about unless you opted in.

Inspect a chain

Inspect a two-hop chain:

```
curl --location --verbose https://httpbin.org/redirect/2 -o /dev/null
```

In the verbose output you'll see the pattern repeat: a request, a `< HTTP/2 302` response, a `< location:` header, then a `* Issue another request` line as curl moves to the next URL. Read every status and `Location` value. Notice the final URL and whether the hostname changes.

Two write-out variables summarize a chain without the noise:

```
curl --location --silent --output /dev/null \
  --write-out 'hops=%{num_redirects} final=%{url_effective}\n' \
  https://httpbin.org/redirect/2
# hops=2 final=https://httpbin.org/get
```

If `num_redirects` surprises you, something in the chain is doing more than you thought.

Put a ceiling on it

Two pages redirecting to each other create a loop. curl gives up after 50 hops by default, but a diagnostic command deserves a tighter bound:

```
curl --location --max-redirs 3 https://httpbin.org/redirect/5
```

This exits with code 47, "maximum redirects followed", after the third hop. In a script, a low `--max-redirs` turns a misconfigured redirect loop into a fast, clear failure instead of a slow crawl through 50 requests.

The credential boundary

Here's where redirects get dangerous. A chain can hop from the host you trust to one you never intended, and any credentials attached to the request could travel with it. curl protects you: when a redirect changes the hostname, it stops sending `--user` credentials and the `Authorization` header. The `--location-trusted` option disables that protection.

Do not use options that forward credentials to unrelated hosts unless you have verified the complete redirect boundary. Inspect the whole chain first, then decide.

Check your understanding: state and authentication

Check the commands, decisions, safety boundaries, and failure cases from state and authentication.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

TLS and network control

Verify servers, override name resolution, use proxies, and bound slow or unreliable transfers.

Keep TLS verification enabled

Understand certificate and hostname verification before reaching for the insecure option.

For HTTPS, curl verifies two things about the server's certificate: that it chains to a trusted authority and that it matches the requested hostname. Both checks protect the server identity. The chain check proves someone trusted vouched for the certificate. The hostname check proves the certificate was issued for the name you asked for, not some other site. Skip either one and encryption still happens, but you no longer know who you're encrypting to.

Watch a verification succeed

Inspect a normal verified connection:

```
curl --verbose https://example.org/ -o /dev/null
```

In the `*` lines, find the certificate subject, issuer, and verification result:

```
* Server certificate:
*  subject: CN=example.org
*  subjectAltName: host "example.org" matched cert's "example.org"
*  issuer: C=US; O=SSL Corporation; CN=Cloudflare TLS Issuing ECC CA 3
*  SSL certificate verify ok.
```

A successful check means the requested name and trust chain passed for this connection. The `subjectAltName ... matched` line is the hostname check happening, and `verify ok` is the chain check passing.

When verification fails

A failure looks like this:

```
curl: (60) SSL certificate problem: unable to get local issuer certificate
```

The important part: this error is information, not an obstacle. Each cause has a real fix. A **hostname mismatch** means you're connecting to a name the certificate doesn't cover — fix the URL or the certificate. An **incomplete chain** means the server forgot to send an intermediate certificate — fix the server config. An **unknown authority** on a corporate network or private CA means curl needs that CA added:

```
curl --cacert internal-ca.pem https://intranet.corp.test/
```

And a **wrong system clock** makes valid certificates look expired, because validity is checked against your local time. Check `date` before blaming the server.

The option to refuse

`--insecure` (short form `-k`) turns both checks off. It ”fixes” every error above by ignoring the problem, which is exactly why it's dangerous: it also ignores an attacker sitting between you and the server.

Do not normalize `--insecure` in scripts. A script that always runs with `-k` has silently given up authentication forever, and nobody will remember why. Fix the hostname, trust store, system clock, or certificate chain instead. Every failure above is diagnosable, and the fix keeps verification working for every future request.

Override resolution with --resolve

Test a hostname against a chosen address while preserving the Host header and TLS server name.

Sometimes you need to send a request for one hostname to an address DNS would not give you. The classic case is testing a new server before switching DNS over to it: you want to know that `example.org` works on the new machine while the public record still points at the old one.

Editing a hosts file is global and easy to forget. Every application on your machine inherits the override, and a week later you are debugging ”weird DNS” you caused yourself. `--resolve` gives one curl command a temporary hostname, port, and address mapping, and it evaporates when the command ends.

Map the hostname for one command

The format is `hostname:port:address`, like `example.org:443:93.184.216.34`. Look up the current address first, then aim curl at it:

```
addr=$(dig +short example.org | head -1)
curl --resolve "example.org:443:$addr" --verbose https://example.org/ -o /dev/null
```

In the verbose output you can confirm the override took effect: curl reports connecting to the address you supplied, not one from a fresh DNS lookup. In a real pre-migration test, you would put the new server's address there instead.

Why this beats connecting to the IP

You might wonder why not just `curl https://93.184.216.34/`. Because that changes the request itself. The `Host` header becomes the IP, the TLS **server name** (SNI) becomes the IP, and certificate verification fails, since certificates are issued for hostnames.

With `--resolve`, curl still requests `example.org` and verifies that hostname. Only address selection changes for this transfer. The `Host` header, SNI, and TLS verification all behave exactly as they will after the DNS switch. That fidelity is the entire point: you are rehearsing production behavior, not an approximation of it.

If the certificate on the target address does not cover `example.org`, curl fails with a verification error. That is a finding, not an obstacle — it means the new server is not ready.

Stay authorized

Use an address you control or are authorized to test. An override can direct credentials and requests to a different server, and sending real session cookies or tokens to a machine that merely claims to be your API is how credentials leak in test environments.

Send requests through a proxy

Configure an HTTP or SOCKS proxy and identify the additional trust boundary it introduces.

A **proxy** is a server that forwards your requests for you. Instead of connecting to the destination, curl connects to the proxy and asks it to reach the destination on its behalf. You meet proxies in corporate networks, in debugging tools like mitmproxy that run on your own machine, and in scrapers that route traffic through specific egress addresses.

The essential thing to understand: a proxy becomes an intermediary in the connection path. That has protocol consequences and trust consequences.

Point curl at a proxy

Point a lab request at a local proxy:

```
curl --proxy http://127.0.0.1:8080 --verbose https://example.org/ -o /dev/null
```

For HTTPS through an HTTP proxy, curl normally uses CONNECT before negotiating TLS with the destination. You can watch it happen in the verbose output:

```
* Connected to 127.0.0.1 port 8080
> CONNECT example.org:443 HTTP/1.1
< HTTP/1.1 200 Connection established
* SSL connection using TLSv1.3
```

Read that sequence carefully, because it separates the two relationships. First, a plain connection to the proxy. Then `CONNECT`, which asks the proxy to open a raw tunnel to `example.org:443`. The proxy answers `200 Connection established`, and only then does curl negotiate TLS — with the destination, through the tunnel. The proxy shuttles encrypted bytes it cannot read.

Inspect both proxy logs and curl verbose output when testing. Separate the connection to the proxy from the protected connection to the target: a failure before `CONNECT` is a proxy problem, a failure after is between you and the destination.

SOCKS proxies

For a SOCKS proxy, such as the one `ssh -D 1080` gives you, change the scheme:

```
curl --proxy socks5h://127.0.0.1:1080 https://example.org/
```

The `h` in `socks5h` matters: it makes the proxy resolve the hostname, so no DNS query leaks from your machine. Plain `socks5` resolves locally first.

curl also honors environment variables like `https_proxy`, which is worth remembering when curl uses a proxy you never asked for. `--noproxy '*'` disables that for one command.

The trust boundary

A proxy can observe destinations and, without end-to-end TLS, content. Even with TLS it sees every hostname you visit and when. If the proxy asks you to install its own CA certificate to inspect

HTTPS, it can read everything. Use only a proxy you trust and understand — you are adding a party to every conversation.

Set timeouts and safe retries

Bound connection and total time, then retry only operations whose repetition is acceptable.

By default, curl waits a long time for a server that never answers. In a terminal that costs you patience. In automation it costs you a hung script, a stuck cron job, or a CI pipeline that blocks for minutes on a dead host. Network operations can hang or fail temporarily, so automation needs time bounds, clear exit codes, and a retry policy connected to what the operation actually does.

Bound the time

Add a connection timeout, total timeout, and limited retry:

```
curl --connect-timeout 3 --max-time 10 --retry 2 https://example.org/
```

The two timeouts answer different questions. `--connect-timeout 3` bounds how long curl waits to establish the connection: DNS plus TCP plus the handshake. `--max-time 10` bounds the entire transfer, including the download. A server that accepts connections instantly but dribbles out one byte per second sails past the first limit and hits the second.

Test once against an unreachable lab address:

```
time curl --connect-timeout 3 --max-time 10 https://10.255.255.1/
# curl: (28) Failed to connect to 10.255.255.1 port 443 after 3002 ms: Timeout was reached
echo $?   # 28
```

Record the elapsed time and final exit code instead of waiting indefinitely. Exit code 28 is curl's timeout signal, and the roughly three-second duration proves the connection timeout did the work. Without these options the same command can hang for minutes.

Retry what deserves retrying

`--retry 2` makes curl attempt the transfer up to two extra times, but only for failures curl considers **transient**: timeouts, plus HTTP responses like 408, 429, 500, 502, 503, and 504. A 404 does not get retried, and that is correct — the resource will still be missing on attempt three. curl also waits between attempts, backing off progressively, and you can cap the whole budget with `--retry-max-time 30`.

The safety rule

Retries can repeat side effects. Retrying a GET is harmless: reading twice changes nothing. Retrying a POST that creates an order can create two orders, because ”the response timed out” does not mean ”the server did nothing” — the request may have succeeded just as the connection dropped.

Use retries freely only for operations designed to be idempotent or protected by an idempotency key, where the server recognizes a repeated request and refuses to apply it twice. For everything else, a failure should surface to something that can decide, not silently fire the same mutation again.

Check your understanding: tls and network control

Check the commands, decisions, safety boundaries, and failure cases from tls and network control.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Debug and automate

Measure transfers, capture traces, use configuration files, and build a repeatable API smoke test.

Measure a transfer

Print DNS, connection, TLS, first-byte, total time, status, and downloaded size as structured evidence.

"The API is slow" is a complaint, not a diagnosis. Slow where? DNS? The TLS handshake? The server thinking? The download itself? curl can tell you, because it timestamps every stage of a transfer and exposes those numbers through `--write-out` variables. They turn one transfer into useful timing evidence. Measure stages separately before deciding that a server or network is slow.

Print a timing record

Print a compact timing record:

```
curl --silent --output /dev/null --write-out 'dns=%{time_namelookup} connect=%{time_connect}
```

The body goes to `/dev/null`; only your formatted line prints:

```
dns=0.012 connect=0.048 tls=0.115 first=0.234 total=0.236 code=200
```

One thing trips everyone up: these values are cumulative from the start of the transfer, not per-stage durations. `time_connect` includes the DNS time before it. To get the cost of a single stage, subtract the previous value. In the sample above, TLS negotiation took roughly 0.115 minus 0.048, about 67 milliseconds.

The most diagnostic gap is usually `first` minus `tls`: the wait between sending the request and receiving the first response byte. That's the server working. A large gap there with fast numbers before it means the network is fine and the backend is slow.

Interpret with care

Compare repeated runs. Connection reuse, DNS caching, server load, and network path can change each stage:

```
for i in 1 2 3 4 5; do
  curl --silent --output /dev/null --write-out 'total=%{time_total} code=%{response_code}\n'
done
```

The first run often pays for a cold DNS cache. Later runs may hit a warm resolver and look faster for reasons that have nothing to do with the server.

Add `%{size_download}` when byte counts matter — a "fast" response that returned 87 bytes of error JSON is not a healthy response, and the status code alone can miss that.

One request is not a benchmark. Collect enough samples and preserve the target, location, and time. Five requests from your laptop describe your laptop's path to the server at that moment. That's real evidence, as long as you label it as exactly that.

Capture a curl trace

Record protocol bytes and timing details when verbose output does not explain a transfer failure.

Verbose mode shows headers and events. Sometimes that's not enough: a transfer dies mid-body, a server misbehaves on specific bytes, or you need to hand a complete record to someone else. A curl trace contains more detail than verbose mode, including data exchanged by the transfer. It is useful for a small, controlled reproduction.

Record a trace

Capture an ASCII trace with timestamps:

```
curl --trace-ascii trace.txt --trace-time https://example.org/ -o /dev/null
```

Nothing extra appears on screen. The record lands in `trace.txt`:

```
17:20:31.234567 == Info: Connected to example.org (104.20.26.136) port 443
17:20:31.301234 => Send header, 76 bytes (0x4c)
0000: GET / HTTP/2
000e: Host: example.org
...
17:20:31.398765 <= Recv header, 13 bytes (0xd)
0000: HTTP/2 200
17:20:31.401122 <= Recv data, 1256 bytes (0x4e8)
```

Open the trace and follow connection setup, headers, and data. The direction markers do the navigation: `=>` is data curl sent, `<=` is data received, `== Info` lines are curl's own commentary. Every block shows its byte count, and `--trace-time` stamps each line so you can see exactly where time went.

That combination answers questions verbose mode can't. A transfer that hangs shows a timestamp gap at the precise byte where it stalled. A response cut short shows fewer received bytes than the headers promised.

`--trace-ascii` keeps the output readable by replacing non-printable bytes. When the exact byte values matter, like debugging a compressed or binary response, use `--trace trace.bin` instead to get a full hex dump.

Make the trace worth keeping

A trace is a reproduction artifact, so preserve its context. Keep the file with the exact command and curl version:

```
curl --version > repro-notes.txt
echo 'curl --trace-ascii trace.txt --trace-time https://example.org/ -o /dev/null' >> repro-n
```

Whoever picks this up later, including future you, can rerun the same transfer and compare traces line by line.

One serious caution: a trace records everything, by design. Trace files can contain credentials, cookies, URLs, and bodies. Use disposable data and sanitize before sharing. A trace made with a real session cookie is a working credential sitting in a text file.

Write a curl config file

Move stable, non-secret options into a readable configuration file without hiding important request behavior.

Commands you run repeatedly grow options until they no longer fit on a screen, and a fifteen-option command is where mistakes hide. A **curl config file** makes long repeatable commands easier to review: the options live in a file, one per line, where a colleague can actually read them.

Write the file

Create `smoke.curl`:

```
# transfer policy for the API smoke check
url = "https://example.org/"
fail-with-body
show-error
silent
max-time = 10
```

The format rules are few. Use long option names without the leading dashes, one option per line. Options that take a value use `=` (or a space). Lines starting with `#` are comments — use them, since a config file is documentation that happens to execute.

Run it:

```
curl --config smoke.curl
```

Same transfer as typing all five options by hand, but the file describes the transfer policy clearly, lives in version control, and changes show up in diffs and code review.

Command-line options can still add a temporary override:

```
curl --config smoke.curl --verbose
```

That is the intended division of labor: stable policy in the file, situational flags on the command line.

Keep secrets elsewhere

Config files get committed, shared, and copied between machines — that is their value. So keep request-specific secrets out of them. A `header = "Authorization: Bearer ..."` line in a committed file is a leaked credential. Pass secrets at run time from an environment variable or a secret manager, and let the file hold only what is safe for anyone to read.

The invisible config file

Here is what surprises people: curl may load a default user config automatically, `~/.curlrc` on Linux and macOS, even when you never asked. If your curl behaves differently from a colleague's — a proxy that appears from nowhere, an unexpected user agent — check whether a `.curlrc` is silently adding options.

Use `--disable` (or `-q`) first when you need a reproducible command independent of personal settings:

```
curl -q https://example.org/
```

It must be the first option on the command line, or curl ignores it. For scripts you intend to share, starting with `-q` and an explicit `--config` means the command behaves the same on every machine, which is the entire point of writing the policy down.

Build an API smoke test

Combine curl exit status, HTTP status, timing, and a small response assertion into one repeatable check.

A **smoke test** proves one critical path works from the caller's location. It should fail clearly, finish quickly, and avoid mutating production data. Think of it as the question "is it up, really?" asked by a machine: not just "did something answer" but "did the right thing answer with the right content, in time".

This lesson combines what you've built in this module — exit codes, time bounds, and output control — into one script.

The check

Create a small Bash check:

```
#!/bin/bash
body=$(mktemp)
trap 'rm -f "$body"' EXIT

status=$(curl --silent --show-error --fail-with-body --max-time 10 --output "$body" --write-o
test "$status" = 200
grep -q 'Example Domain' "$body"
```

Each line earns its place. `--silent --show-error` removes the progress meter but keeps real errors visible. `--fail-with-body` makes HTTP errors fail the command while still saving the response, so you can read what the server said. `--max-time 10` guarantees the check finishes, up or down, within ten seconds. `--write-out '%{response_code}'` captures the status into a variable while the body lands in a temporary file.

Then two assertions. The status must be exactly `200`, not just "not an error". And the body must contain content we expect — `grep -q` exits non-zero when the text is missing. That last check catches the failure the status code hides: a load balancer answering 200 with an empty or wrong page.

The `trap` line removes the temporary file whenever the script exits, on success or failure. Cleanup you don't have to remember is cleanup that happens.

Prove it fails

A check you've never seen fail is a check you can't trust. Run it with a healthy URL, then break the hostname and expected text:

```
./smoke.sh; echo "exit: $?"        # exit: 0
# now edit the URL to a wrong hostname
./smoke.sh; echo "exit: $?"        # exit: 1, curl reports the resolution failure
# now change the grep text to 'Wrong Text'
./smoke.sh; echo "exit: $?"        # exit: 1, content assertion failed
```

Each failure should stop with a non-zero status. That non-zero exit is the contract: cron, CI, and monitoring systems all understand it without parsing any output.

Two boundaries to respect. Use a read-only endpoint, because this script runs often and unattended. And never print a secret-bearing response into shared logs — if the endpoint needs auth, keep the body check minimal and the output quiet.

Check your understanding: debug and automate

Check the commands, decisions, safety boundaries, and failure cases from debug and automate.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/curl/.