URADICAL

POCKET REFERENCE

cURL Pocket Reference

Alan Bradley

cURL Pocket Reference

A quick reference guide for command-line data transfer

Alan Bradley

uradical.io

Table of Contents

cURL Pocket Reference

A comprehensive quick reference guide for cURL, the command-line tool for transferring data with URLs.

Author: Alan Bradley

Chapter 1: Introduction

What is cURL?

cURL (Client URL) is a command-line tool and library for transferring data using URLs. It supports a vast array of protocols including HTTP, HTTPS, FTP, SFTP, SCP, LDAP, and many more. First released in 1998, cURL has become the de facto standard for making HTTP requests from the command line and is installed by default on most Unix-like systems, macOS, and recent versions of Windows.

cURL is:

- Ubiquitous — Available on virtually every platform and often pre-installed
- Protocol-agnostic — Supports dozens of protocols beyond HTTP
- Scriptable — Perfect for automation, testing, and CI/CD pipelines
- Well-documented — Extensive man pages and online resources

When to Use cURL

cURL excels when you need to:

- Test and debug APIs during development
- Download files from the command line
- Automate HTTP requests in shell scripts
- Inspect HTTP headers and response details
- Transfer files via FTP, SFTP, or SCP
- Test authentication mechanisms
- Debug SSL/TLS certificate issues
- Simulate browser requests for scraping or testing

Installation

macOS:

cURL is pre-installed. To get a newer version:

```
brew install curl
```

Debian / Ubuntu:

```
sudo apt install curl
```

Fedora:

```
sudo dnf install curl
```

Arch Linux:

```
sudo pacman -S curl
```

Windows:

cURL is included in Windows 10 (build 1803+) and Windows 11. For older versions or updates:

```
# Using Chocolatey
choco install curl

# Using Scoop
scoop install curl
```

From source:

```
git clone https://github.com/curl/curl.git
cd curl
autoreconf -fi
./configure --with-openssl
make
sudo make install
```

Verifying Installation

```
curl --version
```

This displays the version number, supported protocols, and features compiled into your build. Pay attention to the features list — it tells you whether HTTP/2, HTTP/3, and other capabilities are available.

Basic Usage

The simplest cURL command fetches a URL and prints the response to stdout:

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

Save output to a file:

```
curl -o page.html https://example.com
```

Use the remote filename:

```
curl -O https://example.com/file.tar.gz
```

Follow redirects:

```
curl -L https://example.com/redirecting-url
```

Getting Help

```
# Brief help
curl -h

# Full manual
```

```
curl --manual

# Man page
man curl

# List all options
curl --help all
```

Chapter 2: Command-Line Options

cURL has over 200 command-line options. This chapter covers the most frequently used ones, organised by purpose. Options can use short form (`-X`) or long form (`--request`).

Request Method Options

Option	Long Form	Description
`-X`	`--request`	Specify HTTP method (GET, POST, PUT, DELETE, etc.)
`-G`	`--get`	Force GET method even with `-d` data
`-I`	`--head`	Fetch headers only (HEAD request)

Data & Body Options

Option	Long Form	Description
`-d`	`--data`	Send data in POST request body
`-F`	`--form`	Submit multipart form data
`--data-raw`		Send data without special interpretation
`--data-binary`		Send binary data exactly as specified
`--data-urlencode`		URL-encode the data before sending
`--json`		Shortcut for JSON content-type and data

Header Options

Option	Long Form	Description
`-H`	`--header`	Add or modify a request header
`-A`	`--user-agent`	Set the User-Agent header
`-e`	`--referer`	Set the Referer header
`-i`	`--include`	Include response headers in output
`-D`	`--dump-header`	Write response headers to a file

Output Options

Option	Long Form	Description
`-o`	`--output`	Write output to a file
`-O`	`--remote-name`	Save with remote filename
`-s`	`--silent`	Suppress progress meter and errors
`-S`	`--show-error`	Show errors even in silent mode
`-w`	`--write-out`	Custom output format after transfer

Authentication Options

Option	Long Form	Description
`-u`	`--user`	Username and password for authentication
`--basic`		Use HTTP Basic authentication
`--digest`		Use HTTP Digest authentication
`--ntlm`		Use NTLM authentication
`--negotiate`		Use HTTP Negotiate (SPNEGO) authentication
`--oauth2-bearer`		Specify OAuth 2.0 Bearer token

Connection Options

Option	Long Form	Description
`-L`	`--location`	Follow redirects
`--max-redirs`		Maximum number of redirects to follow
`-m`	`--max-time`	Maximum time for the entire operation
`--connect-timeout`		Maximum time for connection phase
`--retry`		Number of retries on transient errors
`-x`	`--proxy`	Use specified proxy

SSL/TLS Options

Option	Long Form	Description
`-k`	`--insecure`	Allow insecure SSL connections
`--cacert`		CA certificate bundle file
`--cert`		Client certificate file
`--key`		Private key file
`--tlsv1.2`		Use TLS 1.2 or later
`--tlsv1.3`		Use TLS 1.3 or later

Debugging Options

Option	Long Form	Description
`-v`	`--verbose`	Verbose output showing request/response details
`--trace`		Full trace dump to file
`--trace-ascii`		Like `--trace` but without hex dump
`--trace-time`		Add timestamps to trace output

Cookie Options

Option	Long Form	Description
`-b`	`--cookie`	Send cookies from string or file
`-c`	`--cookie-jar`	Save cookies to file after request

Multiple URLs

cURL can handle multiple URLs in a single invocation:

```
curl -O https://example.com/file1.txt -O https://example.com/file2.txt
```

Option Order

Options generally apply to the next URL specified. For complex scenarios, use `--next` to reset options between URLs:

```
curl -X POST https://api.example.com/data --next https://example.com/page
```

Chapter 3: Basic Requests

HTTP defines several request methods, each serving a different purpose. cURL defaults to GET but can use any method.

GET Requests

GET is the default method. It retrieves a resource without modifying server state.

```
# Simple GET
curl https://api.example.com/users

# GET with query parameters
curl "https://api.example.com/users?limit=10&offset=0"

# Force GET even with data (adds to URL as query string)
curl -G -d "limit=10" -d "offset=0" https://api.example.com/users
```

POST Requests

POST submits data to create or update a resource.

```
# POST with form data
curl -X POST -d "name=Alice&email=alice@example.com" https://api.example.com/users

# POST with JSON
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com"}' \
  https://api.example.com/users

# Shorthand for JSON (cURL 7.82+)
curl --json '{"name": "Alice"}' https://api.example.com/users
```

PUT Requests

PUT replaces a resource entirely at the specified URL.

```
curl -X PUT \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Updated", "email": "alice@example.com"}' \
  https://api.example.com/users/123
```

PATCH Requests

PATCH applies partial modifications to a resource.

```
curl -X PATCH \
  -H "Content-Type: application/json" \
  -d '{"email": "newemail@example.com"}' \
  https://api.example.com/users/123
```

DELETE Requests

DELETE removes the specified resource.

```
curl -X DELETE https://api.example.com/users/123
```

HEAD Requests

HEAD retrieves only the headers, not the body. Useful for checking if a resource exists or inspecting metadata.

```
# Using -I flag
curl -I https://example.com

# Using explicit method
curl -X HEAD https://example.com
```

OPTIONS Requests

OPTIONS queries the server for supported methods and CORS headers.

```
curl -X OPTIONS -i https://api.example.com/users
```

Custom Methods

Some APIs use non-standard methods. cURL can send any method string:

```
curl -X PURGE https://cdn.example.com/cached-resource
```

Chapter 4: Headers

HTTP headers carry metadata about the request and response. cURL provides fine-grained control over headers in both directions.

Setting Request Headers

Use `-H` or `--header` to add or modify headers:

```
# Single header
curl -H "Authorization: Bearer token123" https://api.example.com

# Multiple headers
curl -H "Authorization: Bearer token123" \
     -H "Accept: application/json" \
     -H "X-Request-ID: abc-123" \
     https://api.example.com
```

Common Request Headers

```
# Content-Type for JSON
curl -H "Content-Type: application/json" -d '{"key":"value"}' https://api.example.com

# Accept header to request specific format
curl -H "Accept: application/xml" https://api.example.com

# Custom User-Agent
curl -A "MyApp/1.0" https://api.example.com

# Or using -H
curl -H "User-Agent: MyApp/1.0" https://api.example.com

# Referer header
curl -e "https://google.com" https://example.com

# Or using -H
curl -H "Referer: https://google.com" https://example.com
```

Removing Default Headers

cURL sends certain headers automatically. Remove them by setting an empty value:

```
# Remove User-Agent
curl -H "User-Agent:" https://example.com

# Remove Accept header
curl -H "Accept:" https://example.com
```

Viewing Response Headers

```
# Include headers in output
curl -i https://example.com
```

```
# Headers only (HEAD request)
curl -I https://example.com

# Save headers to a file
curl -D headers.txt -o body.txt https://example.com

# Headers to stderr, body to stdout
curl -v https://example.com 2>&1 | grep "^<"
```

Header Case Sensitivity

HTTP header names are case-insensitive, but values are case-sensitive. cURL preserves the case you specify:

```
# These are equivalent
curl -H "content-type: application/json" https://api.example.com
curl -H "Content-Type: application/json" https://api.example.com
```

Reading Headers from a File

For complex header sets, store them in a file:

```
# headers.txt
Authorization: Bearer token123
Accept: application/json
X-Custom-Header: value
```

```
curl -H @headers.txt https://api.example.com
```

Chapter 5: Request Body & Data

Sending data with requests is fundamental to working with APIs. cURL offers several ways to include data, each suited to different content types.

Form Data

Traditional HTML form encoding (`application/x-www-form-urlencoded`):

```
# Using -d (data is URL-encoded automatically for special chars in values)
curl -d "name=Alice&email=alice@example.com" https://api.example.com/users

# Multiple -d flags are joined with &
curl -d "name=Alice" -d "email=alice@example.com" https://api.example.com/users

# Explicitly URL-encode a value
curl --data-urlencode "message=Hello World!" https://api.example.com/send
```

JSON Data

Most modern APIs expect JSON:

```
# Manual JSON with Content-Type header
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "age": 30}' \
  https://api.example.com/users

# Using --json shorthand (cURL 7.82+)
# Automatically sets Content-Type and Accept headers
curl --json '{"name": "Alice", "age": 30}' https://api.example.com/users
```

Reading Data from Files

For large payloads, read from a file:

```
# Read entire file as body
curl -d @data.json -H "Content-Type: application/json" https://api.example.com

# Read from stdin
echo '{"name": "Alice"}' | curl -d @- -H "Content-Type: application/json" https://api.ex
```

Binary Data

For binary content, preserve the data exactly:

```
# Send binary file
curl --data-binary @image.png -H "Content-Type: image/png" https://api.example.com/uploa
```

Multipart Form Data

For file uploads and mixed content (`multipart/form-data`):

```
# Upload a file
```

```
curl -F "file=@document.pdf" https://api.example.com/upload

# File with explicit content type
curl -F "file=@photo.jpg;type=image/jpeg" https://api.example.com/upload

# File with custom filename
curl -F "file=@localname.txt;filename=remote.txt" https://api.example.com/upload

# Mix files and fields
curl -F "file=@document.pdf" \
     -F "description=My document" \
     -F "tags=important,work" \
     https://api.example.com/upload

# Multiple files
curl -F "files=@file1.txt" -F "files=@file2.txt" https://api.example.com/upload
```

Raw Data

Send data without any processing:

```
curl --data-raw '$pecial&characters=not&encoded' https://api.example.com
```

Empty Body

Some requests need an explicit empty body:

```
curl -X POST -d "" https://api.example.com/trigger
```

Checking What Was Sent

Use verbose mode to verify the request body:

```
curl -v -d '{"test": true}' -H "Content-Type: application/json" https://api.example.com
```

Chapter 6: Authentication

APIs and web services use various authentication mechanisms. cURL supports most common schemes natively.

Basic Authentication

HTTP Basic auth sends base64-encoded credentials:

```
# Username and password
curl -u username:password https://api.example.com

# Prompt for password (more secure)
curl -u username https://api.example.com

# Explicit Basic auth (usually automatic)
curl --basic -u username:password https://api.example.com
```

Bearer Tokens

OAuth 2.0 and JWT tokens use the Authorization header:

```
# Using -H
curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIs..." https://api.example.com

# Using --oauth2-bearer (cURL 7.81+)
curl --oauth2-bearer "eyJhbGciOiJIUzI1NiIs..." https://api.example.com
```

API Keys

APIs often use custom headers or query parameters for keys:

```
# Header-based
curl -H "X-API-Key: your-api-key" https://api.example.com

# Query parameter
curl "https://api.example.com?api_key=your-api-key"
```

Digest Authentication

More secure than Basic auth; uses challenge-response:

```
curl --digest -u username:password https://api.example.com
```

NTLM Authentication

Used in Windows/Microsoft environments:

```
curl --ntlm -u domain\\username:password https://intranet.example.com
```

Negotiate (Kerberos/SPNEGO)

For enterprise single sign-on:

```
curl --negotiate -u : https://intranet.example.com
```

The `-u :` with empty credentials tells cURL to use existing Kerberos tickets.

AWS Signature

For AWS services, use the `--aws-sigv4` option (cURL 7.75+):

```
curl --aws-sigv4 "aws:amz:eu-west-1:s3" \
     -u "ACCESS_KEY:SECRET_KEY" \
     https://bucket.s3.eu-west-1.amazonaws.com/object
```

Client Certificates

Some APIs require mutual TLS:

```
curl --cert client.crt --key client.key https://api.example.com
```

Storing Credentials

Never put passwords in shell history. Use a netrc file:

```
# ~/.netrc
machine api.example.com
login username
password secret
```

```
curl --netrc https://api.example.com
```

Secure the file:

```
chmod 600 ~/.netrc
```

Chapter 7: Cookies & Sessions

Cookies maintain state across HTTP requests. cURL can send, receive, and persist cookies for session management.

Sending Cookies

```
# Single cookie
curl -b "session=abc123" https://example.com

# Multiple cookies
curl -b "session=abc123; user=alice" https://example.com

# From a file (Netscape cookie format)
curl -b cookies.txt https://example.com
```

Saving Cookies

Use `-c` to write cookies to a file after the request:

```
# Save cookies received in response
curl -c cookies.txt https://example.com/login -d "user=alice&pass=secret"
```

Cookie Jar (Session Persistence)

Combine `-b` and `-c` to maintain a session across requests:

```
# Login and save session
curl -c cookies.txt -d "user=alice&pass=secret" https://example.com/login

# Use saved session for subsequent requests
curl -b cookies.txt https://example.com/dashboard

# Both read and write in one command
curl -b cookies.txt -c cookies.txt https://example.com/page
```

Cookie File Format

cURL uses the Netscape cookie format:

```
# Netscape HTTP Cookie File
.example.com␉TRUE␉/␉FALSE␉1735689600␉session␉abc123
```

Fields: domain, include subdomains, path, secure only, expiration (Unix timestamp), name, value.

Session Example: Login Flow

```
# Step 1: Get login page (may set initial cookies)
curl -c cookies.txt -b cookies.txt https://example.com/login

# Step 2: Submit credentials
curl -c cookies.txt -b cookies.txt \
     -d "username=alice&password=secret" \
```

```
    https://example.com/login

# Step 3: Access protected resource
curl -b cookies.txt https://example.com/protected
```

Ignoring Cookies from Response

To send cookies but not save any returned:

```
curl -b cookies.txt https://example.com
```

Viewing Cookie Headers

Use verbose mode to see cookie exchange:

```
curl -v -b cookies.txt -c cookies.txt https://example.com 2>&1 | grep -i cookie
```

Clearing Cookies

Simply delete or truncate the cookie file:

```
> cookies.txt
```

Chapter 8: SSL/TLS

cURL uses TLS for secure HTTPS connections. Understanding SSL/TLS options is essential for debugging certificate issues and configuring secure connections.

Verifying Certificates

By default, cURL verifies the server's certificate against the system CA bundle:

```
# Default behaviour - verifies certificate
curl https://example.com
```

Skipping Verification (Development Only)

For self-signed certificates in development:

```
curl -k https://localhost:8443
curl --insecure https://localhost:8443
```

Warning: Never use `-k` in production scripts — it defeats the security of TLS.

Specifying CA Certificates

Point to a custom CA bundle:

```
# CA bundle file
curl --cacert /path/to/ca-bundle.crt https://example.com

# CA certificate directory
curl --capath /path/to/certs/ https://example.com
```

Client Certificates

For mutual TLS (mTLS):

```
# Separate certificate and key files
curl --cert client.crt --key client.key https://api.example.com

# Combined PEM file
curl --cert client.pem https://api.example.com

# With passphrase
curl --cert client.crt --key client.key --pass "keypassphrase" https://api.example.com

# PKCS#12 format
curl --cert client.p12 --cert-type P12 https://api.example.com
```

TLS Version Control

Force specific TLS versions:

```
# TLS 1.2 minimum
curl --tlsv1.2 https://example.com

# TLS 1.3 only
curl --tlsv1.3 https://example.com

# Maximum TLS version
curl --tls-max 1.2 https://example.com
```

Cipher Suites

Restrict or specify cipher suites:

```
curl --ciphers "ECDHE-RSA-AES256-GCM-SHA384" https://example.com
```

Certificate Information

View certificate details:

```
# Show certificate chain
curl -v https://example.com 2>&1 | grep -A 20 "Server certificate"

# Using openssl for more detail
echo | openssl s_client -connect example.com:443 2>/dev/null | openssl x509 -text
```

Certificate Pinning

Pin to a specific certificate hash:

```
# Get the hash first
openssl s_client -connect example.com:443 2>/dev/null | \
  openssl x509 -pubkey -noout | \
  openssl rsa -pubin -outform DER 2>/dev/null | \
  openssl dgst -sha256 -binary | base64

# Pin to that hash
curl --pinnedpubkey "sha256//YhKJG..." https://example.com
```

Common SSL Errors

Error	Cause	Solution
`certificate verify failed`	CA not trusted	Add CA to bundle or use `--cacert`
`certificate has expired`	Server cert expired	Contact server admin
`self signed certificate`	No trusted CA	Use `--cacert` with the self-signed cert
`unable to get local issuer`	Missing intermediate	Server needs to send full chain

Chapter 9: HTTP/2 and HTTP/3

Modern HTTP protocols offer significant performance improvements. cURL supports HTTP/2 and experimental HTTP/3 (QUIC).

Checking Protocol Support

First, verify your cURL build supports these protocols:

```
curl --version
```

Look for `HTTP2` and `HTTP3` in the features list. HTTP/2 requires nghttp2; HTTP/3 requires a QUIC library (ngtcp2, quiche, or msh3).

HTTP/2

HTTP/2 provides multiplexing, header compression, and server push over a single TCP connection.

```
# Request HTTP/2 (falls back to HTTP/1.1 if unavailable)
curl --http2 https://example.com

# Require HTTP/2 (fail if not supported)
curl --http2-only https://example.com

# HTTP/2 without TLS (h2c) - rare
curl --http2-prior-knowledge http://localhost:8080
```

Checking Protocol Used

Use verbose mode or write-out to see which protocol was negotiated:

```
# Verbose shows protocol
curl -v --http2 https://example.com 2>&1 | grep "using HTTP"

# Write-out format
curl -w "%{http_version}\n" -o /dev/null -s --http2 https://example.com
```

HTTP/3 (QUIC)

HTTP/3 uses QUIC instead of TCP, offering faster connection establishment and better performance on unreliable networks.

```
# Request HTTP/3 (falls back if unavailable)
curl --http3 https://example.com

# Require HTTP/3
curl --http3-only https://example.com
```

ALPN (Application-Layer Protocol Negotiation)

ALPN negotiates the protocol during the TLS handshake. cURL handles this automatically:

```
# See ALPN negotiation in verbose output
curl -v --http2 https://example.com 2>&1 | grep ALPN
```

Protocol Comparison

Feature	HTTP/1.1	HTTP/2	HTTP/3
Transport	TCP	TCP	QUIC (UDP)
Multiplexing	No	Yes	Yes
Header Compression	No	HPACK	QPACK
TLS Required	No	Effectively yes	Yes (TLS 1.3)
Head-of-line Blocking	Yes	Partially	No

Multiple Requests

HTTP/2 multiplexing shines with parallel requests:

```
# Sequential in HTTP/1.1, multiplexed in HTTP/2
curl --http2 https://example.com/page1 https://example.com/page2 https://example.com/pag
```

Alt-Svc Header

Servers advertise HTTP/3 support via the `Alt-Svc` header. cURL can use this for subsequent requests:

```
# Enable Alt-Svc caching
curl --alt-svc altsvc.txt https://example.com
```

Troubleshooting

If HTTP/2 or HTTP/3 fails:

```
# Check server support
curl -I --http2 https://example.com | grep -i "http/"

# Force HTTP/1.1 as fallback
curl --http1.1 https://example.com
```

Chapter 10: WebSockets

WebSockets provide full-duplex communication channels over a single TCP connection. cURL has added WebSocket support, though it remains somewhat experimental.

Checking WebSocket Support

```
curl --version | grep -i websocket
```

Look for `ws` and `wss` in the protocols list. WebSocket support requires cURL 7.86+ built with WebSocket enabled.

WebSocket URLs

cURL recognises WebSocket URL schemes:

```
# Unencrypted WebSocket
curl ws://echo.websocket.org

# Encrypted WebSocket (TLS)
curl wss://echo.websocket.org
```

Basic WebSocket Connection

```
# Connect and receive messages
curl --include wss://echo.websocket.org
```

Sending Messages

Use `--data` to send a message:

```
curl --include \
     --data "Hello, WebSocket!" \
     wss://echo.websocket.org
```

Manual WebSocket Upgrade (Legacy Approach)

Before native WebSocket support, you could manually perform the upgrade handshake:

```
curl --include \
     --no-buffer \
     --header "Connection: Upgrade" \
     --header "Upgrade: websocket" \
     --header "Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==" \
     --header "Sec-WebSocket-Version: 13" \
     https://echo.websocket.org
```

The `--no-buffer` flag is essential for streaming responses.

Streaming Responses

For long-lived WebSocket connections:

```
curl --no-buffer wss://stream.example.com
```

WebSocket Limitations in cURL

cURL's WebSocket support is basic compared to dedicated clients:

- Limited control over message framing
- No built-in ping/pong handling
- Awkward for interactive use
- Better suited for simple testing than production use

Alternative Tools

For serious WebSocket work, consider dedicated tools:

```
# websocat - versatile WebSocket client
websocat wss://echo.websocket.org

# wscat - Node.js WebSocket client
wscat -c wss://echo.websocket.org
```

Testing WebSocket Endpoints

cURL is useful for basic connectivity tests:

```
# Check if WebSocket upgrade succeeds
curl -I \
     -H "Connection: Upgrade" \
     -H "Upgrade: websocket" \
     -H "Sec-WebSocket-Key: test" \
     -H "Sec-WebSocket-Version: 13" \
     https://api.example.com/ws

# Look for: HTTP/1.1 101 Switching Protocols
```

Chapter 11: Proxies & Tunnelling

cURL can route requests through various types of proxies, essential for corporate environments, debugging, and privacy.

HTTP Proxy

```
# Using -x or --proxy
curl -x http://proxy.example.com:8080 https://api.example.com

# With authentication
curl -x http://user:pass@proxy.example.com:8080 https://api.example.com

# Proxy authentication separately
curl -x http://proxy.example.com:8080 --proxy-user user:pass https://api.example.com
```

HTTPS Proxy

For proxies that accept HTTPS connections:

```
curl -x https://proxy.example.com:8443 https://api.example.com
```

SOCKS Proxy

SOCKS proxies work at a lower level and support more protocols:

```
# SOCKS4
curl --socks4 proxy.example.com:1080 https://api.example.com

# SOCKS4a (proxy resolves DNS)
curl --socks4a proxy.example.com:1080 https://api.example.com

# SOCKS5
curl --socks5 proxy.example.com:1080 https://api.example.com

# SOCKS5 with proxy DNS resolution
curl --socks5-hostname proxy.example.com:1080 https://api.example.com
```

Environment Variables

Set default proxies via environment variables:

```
export http_proxy="http://proxy.example.com:8080"
export https_proxy="http://proxy.example.com:8080"
export no_proxy="localhost,127.0.0.1,.internal.com"

curl https://api.example.com  # Uses proxy automatically
```

Bypassing Proxy

```
# Ignore proxy for specific request
curl --noproxy "*" https://api.example.com

# Bypass for specific hosts
```

```
curl --noproxy "localhost,.local" https://api.example.com
```

CONNECT Tunnelling

For HTTPS through an HTTP proxy, cURL uses the CONNECT method:

```
# This happens automatically
curl -x http://proxy:8080 https://secure.example.com

# Force tunnel for HTTP too
curl -x http://proxy:8080 --proxytunnel http://example.com
```

Proxy Headers

Add headers specifically for the proxy:

```
curl -x http://proxy:8080 \
     --proxy-header "X-Proxy-Auth: token123" \
     https://api.example.com
```

Debugging Proxy Connections

```
curl -v -x http://proxy:8080 https://api.example.com 2>&1 | grep -i proxy
```

PAC Files

cURL doesn't support PAC files directly. Parse them externally or extract the proxy URL manually.

Chapter 12: Timeouts & Retries

Network operations can hang or fail. Proper timeout and retry configuration makes scripts robust.

Connection Timeout

Maximum time to establish the TCP connection:

```
# 10 seconds to connect
curl --connect-timeout 10 https://api.example.com
```

Maximum Time

Total time limit for the entire operation:

```
# 30 seconds total
curl -m 30 https://api.example.com
curl --max-time 30 https://api.example.com
```

Combining Timeouts

```
# 5 seconds to connect, 30 seconds total
curl --connect-timeout 5 --max-time 30 https://api.example.com
```

DNS Timeout

Limit DNS resolution time (requires c-ares):

```
curl --dns-servers 8.8.8.8 --connect-timeout 5 https://api.example.com
```

Retry on Failure

Automatically retry on transient errors:

```
# Retry up to 3 times
curl --retry 3 https://api.example.com

# Retry with delay
curl --retry 3 --retry-delay 2 https://api.example.com

# Maximum retry time
curl --retry 3 --retry-max-time 60 https://api.example.com
```

Retry Conditions

Control what triggers a retry:

```
# Retry on connection problems only (default)
curl --retry 3 --retry-connrefused https://api.example.com

# Retry on all errors including HTTP errors
curl --retry 3 --retry-all-errors https://api.example.com
```

Exponential Backoff

cURL uses exponential backoff by default. Override with:

```
# Fixed 5-second delay between retries
curl --retry 3 --retry-delay 5 https://api.example.com
```

Speed Limits

Abort if transfer is too slow:

```
# Abort if slower than 1000 bytes/sec for 30 seconds
curl --speed-limit 1000 --speed-time 30 https://example.com/largefile
```

Expect Timeout

For POST requests, cURL sends `Expect: 100-continue` and waits for server acknowledgment:

```
# Don't wait for 100-continue
curl --no-expect -d "data" https://api.example.com
```

Keep-Alive

Control connection persistence:

```
# Set keep-alive time
curl --keepalive-time 60 https://api.example.com

# Disable keep-alive
curl --no-keepalive https://api.example.com
```

Chapter 13: Output Control

cURL offers extensive control over what output is produced and where it goes.

Silent Mode

Suppress progress meter and error messages:

```
# Silent
curl -s https://api.example.com

# Silent but show errors
curl -sS https://api.example.com
```

Progress Meter

```
# Default progress bar (when output is not terminal)
curl -O https://example.com/file.zip

# Simple progress bar
curl -# -O https://example.com/file.zip

# No progress
curl -s -O https://example.com/file.zip
```

Output to File

```
# Specify filename
curl -o output.html https://example.com

# Use remote filename
curl -O https://example.com/file.tar.gz

# Multiple files
curl -O https://example.com/file1.txt -O https://example.com/file2.txt
```

Write-Out Format

The `-w` option formats custom output after transfer:

```
# HTTP status code
curl -s -o /dev/null -w "%{http_code}" https://api.example.com

# Response time
curl -s -o /dev/null -w "%{time_total}s\n" https://api.example.com

# Multiple variables
curl -s -o /dev/null -w "Code: %{http_code}\nTime: %{time_total}s\nSize: %{size_download
```

Common Write-Out Variables

Variable	Description
%{http_code}	HTTP response code
%{http_version}	HTTP version used
%{time_total}	Total time in seconds
%{time_connect}	Time to establish connection
%{time_starttransfer}	Time to first byte
%{time_namelookup}	DNS lookup time
%{size_download}	Bytes downloaded
%{size_upload}	Bytes uploaded
%{speed_download}	Download speed (bytes/sec)
%{url_effective}	Final URL after redirects
%{redirect_url}	Redirect destination
%{content_type}	Content-Type header value
%{remote_ip}	Server IP address
%{local_ip}	Local IP address

Write-Out from File

Store format string in a file:

```
# format.txt
Code: %{http_code}
Time: %{time_total}
```

```
curl -w @format.txt -o /dev/null -s https://api.example.com
```

JSON Output

cURL can output response info as JSON (cURL 7.70+):

```
curl -w "%{json}" -o /dev/null -s https://api.example.com | jq .
```

Separating Headers and Body

```
# Headers to file, body to stdout
curl -D headers.txt https://example.com

# Headers to stderr
curl -v https://example.com 2>headers.txt 1>body.txt
```

Chapter 14: Debugging & Verbose Output

When requests fail or behave unexpectedly, cURL's debugging options reveal what's happening under the hood.

Verbose Mode

The `-v` flag shows request and response headers:

```
curl -v https://api.example.com
```

Output markers:

- `>` — Data sent to server (request)
- `<` — Data received from server (response)
- `*` — Additional info (TLS, connection)

Trace Output

Full hex dump of all data:

```
# Trace to file
curl --trace trace.txt https://api.example.com

# Trace without hex (ASCII only)
curl --trace-ascii trace.txt https://api.example.com

# With timestamps
curl --trace trace.txt --trace-time https://api.example.com
```

Timing Breakdown

See where time is spent:

```
curl -w "\
   namelookup:  %{time_namelookup}s\n\
      connect:  %{time_connect}s\n\
   appconnect:  %{time_appconnect}s\n\
  pretransfer:  %{time_pretransfer}s\n\
     redirect:  %{time_redirect}s\n\
starttransfer:  %{time_starttransfer}s\n\
        total:  %{time_total}s\n" \
-o /dev/null -s https://api.example.com
```

Filtering Verbose Output

```
# Request headers only
curl -v https://api.example.com 2>&1 | grep "^>"

# Response headers only
curl -v https://api.example.com 2>&1 | grep "^<"
```

```
# TLS/SSL info
curl -v https://api.example.com 2>&1 | grep "^\*"
```

Debugging TLS Issues

```
# Show TLS handshake details
curl -v https://example.com 2>&1 | grep -A 10 "SSL connection"

# Force specific TLS version to test
curl --tlsv1.2 -v https://example.com
```

Debugging DNS

```
# Show DNS resolution
curl -v https://example.com 2>&1 | grep "Trying\|Connected"

# Use specific DNS server (requires c-ares)
curl --dns-servers 8.8.8.8 https://example.com
```

Debugging Redirects

```
# Show redirect chain
curl -v -L https://example.com 2>&1 | grep -i "location\|following"
```

Request Inspection

See exactly what cURL sends:

```
# Using netcat to inspect
curl -v --resolve api.example.com:80:127.0.0.1 http://api.example.com
# In another terminal: nc -l 80
```

Comparing Requests

Useful for debugging why cURL behaves differently from browsers:

```
# Export browser request as cURL from DevTools
# Compare with your command using -v
```

Chapter 15: Following Redirects

HTTP redirects (3xx responses) are common. cURL can follow them automatically or let you handle them manually.

Automatic Redirect Following

```
# Follow redirects
curl -L https://example.com/short-url
curl --location https://example.com/short-url
```

Redirect Limit

Prevent infinite redirect loops:

```
# Maximum 5 redirects (default is 50)
curl -L --max-redirs 5 https://example.com
```

Viewing Redirect Chain

```
# Show each redirect
curl -L -v https://example.com 2>&1 | grep -i "location"

# Get final URL
curl -L -w "%{url_effective}\n" -o /dev/null -s https://example.com
```

Redirect Types

Code	Meaning	Method Preserved
301	Moved Permanently	No (becomes GET)
302	Found	No (becomes GET)
303	See Other	No (becomes GET)
307	Temporary Redirect	Yes
308	Permanent Redirect	Yes

Preserving Method on Redirect

By default, POST becomes GET after redirect. To preserve:

```
# Keep POST method
curl -L --post301 --post302 --post303 -X POST -d "data" https://example.com
```

Redirect Headers

Control whether to send headers to redirect targets:

```
# Send Authorization header to redirects (be careful with cross-domain)
curl -L -H "Authorization: Bearer token" https://example.com
```

```
# Headers are NOT sent to different hosts by default (security)
```

Manual Redirect Handling

Get the redirect location without following:

```
# See redirect response
curl -i https://example.com/redirect

# Extract Location header
curl -I -s https://example.com/redirect | grep -i "^location:"
```

Redirect to File

When downloading, follow redirects to the final resource:

```
curl -L -O https://example.com/download
```

Protocol Downgrades

By default, cURL won't redirect from HTTPS to HTTP:

```
# Allow HTTPS to HTTP redirect (insecure)
curl -L --proto-redir -all,http,https https://example.com
```

Chapter 16: FTP, SFTP & SCP

cURL supports various file transfer protocols beyond HTTP.

FTP Basics

```
# List directory
curl ftp://ftp.example.com/

# Download file
curl -O ftp://ftp.example.com/pub/file.txt

# Upload file
curl -T localfile.txt ftp://ftp.example.com/uploads/

# With credentials
curl -u user:password ftp://ftp.example.com/
```

FTP Commands

```
# Create directory
curl -Q "MKD /newdir" ftp://ftp.example.com/

# Delete file
curl -Q "DELE /path/to/file.txt" ftp://ftp.example.com/

# Rename file
curl -Q "RNFR /old.txt" -Q "RNTO /new.txt" ftp://ftp.example.com/
```

FTPS (FTP over TLS)

```
# Explicit FTPS (starts plain, upgrades)
curl --ftp-ssl ftp://ftp.example.com/

# Require FTPS
curl --ftp-ssl-reqd ftp://ftp.example.com/

# Implicit FTPS (TLS from start)
curl ftps://ftp.example.com:990/
```

SFTP (SSH File Transfer)

```
# Download via SFTP
curl -u user sftp://server.example.com/home/user/file.txt

# With private key
curl --key ~/.ssh/id_rsa sftp://server.example.com/path/

# Known hosts verification
curl --hostpubsha256 "SHA256fingerprint" sftp://server.example.com/
```

SCP (Secure Copy)

```
# Download via SCP
```

```
curl -u user scp://server.example.com/home/user/file.txt

# Upload
curl -T localfile.txt -u user scp://server.example.com/uploads/

# With key
curl --key ~/.ssh/id_rsa scp://server.example.com/path/
```

Passive vs Active FTP

```
# Use passive mode (default, usually works through firewalls)
curl --ftp-pasv ftp://ftp.example.com/

# Use active mode
curl --ftp-port - ftp://ftp.example.com/
```

Resume Transfers

```
# Resume interrupted download
curl -C - -O ftp://ftp.example.com/largefile.zip

# Resume upload
curl -C - -T largefile.zip ftp://ftp.example.com/
```

Recursive Download

cURL doesn't support recursive FTP downloads directly. Use wget or lftp for that:

```
wget -r ftp://ftp.example.com/directory/
```

Chapter 17: Common Recipes

Practical patterns for everyday cURL usage.

API Testing

```
# GET request with JSON response formatting
curl -s https://api.example.com/users | jq .

# POST JSON data
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com"}' \
  https://api.example.com/users

# PUT to update
curl -X PUT \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Updated"}' \
  https://api.example.com/users/123

# DELETE
curl -X DELETE https://api.example.com/users/123
```

Downloading Files

```
# Simple download
curl -O https://example.com/file.zip

# Download with progress bar
curl -# -O https://example.com/file.zip

# Resume interrupted download
curl -C - -O https://example.com/largefile.zip

# Download multiple files
curl -O https://example.com/file[1-10].txt

# Limit download speed
curl --limit-rate 1M -O https://example.com/file.zip
```

Health Checks

```
# Simple health check
curl -s -o /dev/null -w "%{http_code}" https://api.example.com/health

# Health check with timeout
curl -s -o /dev/null -w "%{http_code}" --max-time 5 https://api.example.com/health

# Full health check script
check_health() {
  code=$(curl -s -o /dev/null -w "%{http_code}" --max-time 5 "$1")
  if [ "$code" = "200" ]; then
    echo "OK: $1"
  else
```

```
    echo "FAIL: $1 (HTTP $code)"
  fi
}
```

Polling / Waiting for Service

```
# Wait for service to be ready
until curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/health | grep -q "200
  echo "Waiting for service..."
  sleep 2
done
echo "Service is up!"
```

Uploading Files

```
# Form upload
curl -F "file=@document.pdf" https://api.example.com/upload

# Binary upload
curl --data-binary @image.png -H "Content-Type: image/png" https://api.example.com/image

# Multiple files
curl -F "files[]=@file1.txt" -F "files[]=@file2.txt" https://api.example.com/upload
```

Benchmarking

```
# Single request timing
curl -o /dev/null -s -w "Total: %{time_total}s\n" https://example.com

# Multiple requests (simple)
for i in {1..10}; do
  curl -o /dev/null -s -w "%{time_total}\n" https://example.com
done

# Use Apache Bench or hey for serious benchmarking
ab -n 100 -c 10 https://example.com/
```

Working with APIs that Paginate

```
# Fetch all pages
page=1
while true; do
  response=$(curl -s "https://api.example.com/items?page=$page")
  items=$(echo "$response" | jq '.items | length')

  if [ "$items" -eq 0 ]; then
    break
  fi

  echo "$response" | jq '.items[]'
  ((page++))
done
```

Simulating Browser Requests

```
curl -H "User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36" \
```

```
     -H "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8" \
     -H "Accept-Language: en-US,en;q=0.5" \
     -b cookies.txt \
     https://example.com
```

Sending Email via SMTP

```
curl --url "smtp://smtp.example.com:587" \
     --ssl-reqd \
     --mail-from "sender@example.com" \
     --mail-rcpt "recipient@example.com" \
     --user "sender@example.com:password" \
     -T email.txt
```

Where `email.txt` contains:

```
From: sender@example.com
To: recipient@example.com
Subject: Test Email

This is the email body.
```

Conditional Requests

```
# Only download if modified since
curl -z "2024-01-01" -O https://example.com/file.txt

# Using If-Modified-Since header
curl -H "If-Modified-Since: Mon, 01 Jan 2024 00:00:00 GMT" https://example.com/file.txt

# Using ETag
curl -H "If-None-Match: \"abc123\"" https://example.com/resource
```

Chapter 18: Quick Reference Tables

HTTP Methods

Method	Description	Has Body	Idempotent
GET	Retrieve resource	No	Yes
POST	Create resource	Yes	No
PUT	Replace resource	Yes	Yes
PATCH	Partial update	Yes	No
DELETE	Remove resource	Optional	Yes
HEAD	Get headers only	No	Yes
OPTIONS	Get supported methods	No	Yes

Common Options Quick Reference

Option	Long Form	Purpose
`-X`	`--request`	HTTP method
`-H`	`--header`	Set header
`-d`	`--data`	Send data
`-F`	`--form`	Multipart form
`-o`	`--output`	Output file
`-O`	`--remote-name`	Save as remote name
`-L`	`--location`	Follow redirects
`-i`	`--include`	Include headers
`-I`	`--head`	Headers only
`-s`	`--silent`	Silent mode
`-S`	`--show-error`	Show errors
`-v`	`--verbose`	Verbose output
`-u`	`--user`	Authentication
`-k`	`--insecure`	Skip TLS verify
`-b`	`--cookie`	Send cookies
`-c`	`--cookie-jar`	Save cookies
`-m`	`--max-time`	Timeout
`-w`	`--write-out`	Custom output
`-x`	`--proxy`	Use proxy
`-A`	`--user-agent`	Set User-Agent
`-e`	`--referer`	Set Referer

Write-Out Variables

Variable	Description
`%{http_code}`	HTTP status code
`%{http_version}`	HTTP version
`%{time_total}`	Total time (seconds)
`%{time_namelookup}`	DNS time
`%{time_connect}`	TCP connect time
`%{time_appconnect}`	TLS handshake time
`%{time_starttransfer}`	Time to first byte
`%{size_download}`	Downloaded bytes
`%{size_upload}`	Uploaded bytes
`%{speed_download}`	Download speed
`%{speed_upload}`	Upload speed
`%{url_effective}`	Final URL
`%{redirect_url}`	Redirect target
`%{content_type}`	Content-Type
`%{remote_ip}`	Server IP
`%{local_ip}`	Client IP
`%{scheme}`	URL scheme
`%{json}`	All info as JSON

HTTP Status Codes

Range	Category	Common Codes
1xx	Informational	100 Continue, 101 Switching Protocols
2xx	Success	200 OK, 201 Created, 204 No Content
3xx	Redirection	301 Moved, 302 Found, 304 Not Modified
4xx	Client Error	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
5xx	Server Error	500 Internal Error, 502 Bad Gateway, 503 Unavailable

Exit Codes

Code	Meaning
0	Success
1	Unsupported protocol
2	Failed to initialise
3	Malformed URL
6	Couldn't resolve host
7	Couldn't connect
22	HTTP error (with `--fail`)
28	Operation timed out
35	SSL connect error
51	SSL certificate problem
52	Empty reply from server
56	Network receive error

Protocol Prefixes

Prefix	Protocol
`http://`	HTTP
`https://`	HTTPS
`ftp://`	FTP
`ftps://`	FTPS
`sftp://`	SFTP
`scp://`	SCP
`ws://`	WebSocket
`wss://`	WebSocket Secure
`file://`	Local file

Environment Variables

Variable	Purpose
`http_proxy`	HTTP proxy URL
`https_proxy`	HTTPS proxy URL
`all_proxy`	Proxy for all protocols
`no_proxy`	Hosts to exclude from proxy
`CURL_CA_BUNDLE`	CA certificate bundle path

About This Book

This pocket reference is designed for quick lookup during development and operations work. It covers cURL 7.68+ with notes on newer features where version requirements differ.

For the complete cURL documentation, visit: https://curl.se/docs/

For interactive help:

```
curl --help all
man curl
```

cURL Pocket Reference by Alan Bradley

https://uradical.io