## Complete Beginner Guide

curl · HTTP · Annotated Outputs · Plain-English Explanations

# ■ 0. What is curl? — Start Here!

■ **Beginner Tip** If you are completely new, read this section FIRST before anything else. Once you understand curl, every example in this guide will click immediately.

**curl** (Client URL) is a free command-line tool pre-installed on macOS, Linux, and Windows 10+. It lets you send HTTP requests from your terminal — exactly what a web browser does when you type a URL, except you have full control over every detail.

**Simple analogy:** Your browser is like ordering food through an app — it handles everything for you. curl is like calling the restaurant directly — you speak to them yourself, ask exactly what you want, and hear exactly what they say back. Same result, full control.

### Is curl installed on your machine?

**$ TERMINAL — type this** — Check if curl is installed — type this in your terminal

```
curl --version
```

**■ SERVER RESPONSE** — Expected output — if you see this, you are ready to go!

```
curl 8.4.0 (x86_64-apple-darwin23.0)
Release-Date: 2023-10-11
Protocols: dict file ftp ftps http https ...
Features: alt-svc AsynchDNS brotli HSTS HTTP2
```

- ■ *version number — anything 7.x or 8.x is fine*
- ■ *when this version was released*
- ■ *curl can speak many protocols — we use http/https*
- ■ *capabilities — HTTP2 means faster requests*

■ **Note** Not installed? On Ubuntu/Debian: **sudo apt install curl** | Download for all OS: **curl.se/download.html**

### Your Very First API Call — Try it Right Now

Let's call a real, live, public API. This one returns a random joke. No sign-up needed. Type this in your terminal:

**$ TERMINAL — type this** — Your first ever API call — paste this and press Enter

```
curl https://official-joke-api.appspot.com/random_joke
```

**■ SERVER RESPONSE** — The server replies with JSON — a REST API response!

```
{
"id": 327,
"type": "general",
"setup": "Why do programmers prefer dark mode?",
"punchline": "Because light attracts bugs!"
}
```

- ■ *the joke's unique ID in the database*
- ■ *category of this joke*
- ■ *the setup line*
- ■ *the punchline*

■ **Beginner Tip** You just made a REST API call! You sent a request, the server found data, and sent it back as JSON. That is literally how EVERY REST API works — just with different URLs and different data.

## The Anatomy of a curl Command

Every curl command follows the same pattern. Let's break it apart:

**$ TERMINAL — type this** — Full curl command — every flag explained

```
curl                                                    ■ the tool itself
--request POST                                          ■ -X POST — which HTTP method to use
--header "Content-Type: application/json"               ■ -H — add a header (can repeat -H multiple times)
--header "Authorization: Bearer mytoken"                ■ -H again — to add your auth token
--data '{"name":"Alice","age":25}'                      ■ -d — the request body (data you are sending)
https://api.example.com/users                           ■ the URL — always comes last
```

| Flag | Short form | What it does | Example |
|---|---|---|---|
| --request | -X | Set HTTP method | `-X POST` |
| --header | -H | Add a request header | `-H "Content-Type: application/json"` |
| --data | -d | Send a request body | `-d '{"name":"Alice"}'` |
| --user | -u | Basic auth (user:password) | `-u admin:secret123` |
| --include | -i | Show response headers in output | `curl -i https://...` |
| --verbose | -v | Show FULL request AND response | `curl -v https://...` |
| --silent | -s | Hide the progress bar | `curl -s https://...` |
| --output | -o | Save response to a file | `-o result.json` |
| --location | -L | Follow redirects (3xx) | `curl -L https://...` |
| --insecure | -k | Skip SSL check (dev only!) | `curl -k https://...` |

■ **Note** You can combine short flags: `-s -i -X POST` can be written as `-siX POST`

## curl with -v (Verbose) — See the Full HTTP Conversation

The **-v flag** is the most educational tool available. It reveals every step of the HTTP process — the connection, encryption handshake, headers sent, headers received — everything that is normally invisible.

**$ TERMINAL — type this** — Add -v to see everything curl does

```
curl -v https://jsonplaceholder.typicode.com/users/1
```

**■ SERVER RESPONSE** — Complete verbose output — every line explained

| | |
|---|---|
| `* Trying 104.21.45.109:443...` | ■ *DNS resolved the hostname to an IP. Connecting...* |
| `* Connected to ...typicode.com port 443` | ■ *TCP 3-way handshake complete. We have a connection!* |
| `* SSL connection using TLSv1.3 / AES_256_GCM` | ■ *TLS handshake done. Channel is now encrypted.* |
| `* Server certificate: *.typicode.com verified OK` | ■ *Server proved its identity. Safe to continue.* |
| `> GET /users/1 HTTP/2` | ■ *→ curl SENDS this. GET = read, /users/1 = which resource* |
| `> Host: jsonplaceholder.typicode.com` | ■ *→ which server (needed because one IP can host many sites)* |
| `> Accept: */*` | ■ *→ we accept any format back (JSON, HTML, etc.)* |
| `>` | ■ *→ blank line marks end of request headers* |
| `< HTTP/2 200` | ■ *← SERVER RESPONDS. 200 = success!* |
| `< content-type: application/json` | ■ *← response body will be in JSON format* |
| `< x-powered-by: Express` | ■ *← server runs on Express.js (Node.js framework)* |
| `<` | ■ *← blank line marks end of response headers* |
| `{ "id": 1, "name": "Leanne Graham" ... }` | ■ *← the actual response body / data* |

■ **Beginner Tip** Lines with * = curl info. Lines with > = what curl SENDS. Lines with < = what server SENDS BACK. This is the entire HTTP conversation made visible!

# ■ 1. What is REST & REST API?

**REST** (Representational State Transfer) is a set of design rules for building APIs. Created by Roy Fielding in 2000. Most APIs you use every day — GitHub, Stripe, Google Maps, Instagram, Spotify — are REST APIs.

**What is an API?** API stands for Application Programming Interface. It is a way for two programs to talk to each other. A REST API uses the web (HTTP) as the communication channel.

**Restaurant analogy:** You = client. Waiter = API. Kitchen = server/database. You tell the waiter what you want (request). Waiter goes to the kitchen, brings back your food (response). You never enter the kitchen yourself.

| Term | Plain English | Real Example |
|---|---|---|
| Resource | The thing you want — always a noun | A user, a tweet, a product, an order |
| Endpoint | The URL address of that resource | /api/users or /api/products/42 |
| Request | You asking the server for something | curl https://api.github.com/users/torvalds |
| Response | Server's answer to your request | {"id":1, "name":"Linus Torvalds"} |
| JSON | The format data is usually sent in | {"key": "value", "number": 42} |
| HTTP Method | What action you want to perform | GET=read, POST=create, DELETE=remove |
| Status Code | 3-digit number saying how it went | 200=OK, 404=not found, 500=server error |

### Live Example — Call a Real Public API

**$ TERMINAL — type this** — Get user #1 from a free test API (no auth required)

```
curl https://jsonplaceholder.typicode.com/users/1
```

**■ SERVER RESPONSE** — Full response — a JSON object describing user #1

```
{
"id": 1,                                    ■ unique identifier — every resource has one
"name": "Leanne Graham",                    ■ full name field
"username": "Bret",                         ■ login username
"email": "Sincere@april.biz",               ■ email address
"address": {                                ■ nested object — address has its own sub-fields
"street": "Kulas Light",
"city": "Gwenborough",
"zipcode": "92998-3874"
},
"phone": "1-770-736-0988 x56442",           ■ phone number
"website": "hildegard.org"
}
```

# ■ 2. The Six Architectural Constraints

Roy Fielding said: for an API to be called 'RESTful' it must follow 6 rules. As a beginner, focus on **Statelessness** — it is the most important and affects every API call you write.

| # | Rule | What it means in plain English | Your action |
|---|---|---|---|
| 1 | Client–Server | Frontend and backend are separate teams. Each can change without breaking the other. | Always call the server through the API, never access DB directly. |
| 2 | Stateless ■ | Server has NO memory between requests. Every call must be completely self-contained. | Always send your auth token on EVERY request — not just the first one! |
| 3 | Cacheable | Server can say 'cache this response for 1 hour' to reduce load and speed things up. | Respect Cache-Control headers; don't over-call for stable data. |
| 4 | Uniform Interface | All resources follow the same URL + method patterns. Predictable and consistent. | Stick to the patterns: /resources for list, /resources/id for one item. |
| 5 | Layered System | You don't know if you're talking to the real server, a CDN, or a load balancer. | You don't need to care — the API behaves the same regardless. |
| 6 | Code on Demand | Server can optionally send JavaScript to the client. Rarely used in practice. | Mostly ignore this one — it is the only optional constraint. |

### Statelessness — The Most Important Constraint to Understand

The server forgets you the instant each request ends. There is no 'session' on the server side. This means your auth token must travel with EVERY request — not just at login.

**$ TERMINAL — type this** ■ WRONG idea — thinking server remembers you after login

```
# Request 1: Login
curl -X POST -d '{"email":"alice@ex.com","password":"abc"}' \
https://api.example.com/login
→ server sends back a token...

# Request 2 (WRONG): Calling without the token
curl https://api.example.com/profile
→ you get 401 Unauthorized! Server forgot you.
```

**$ TERMINAL — type this** ■ CORRECT — send your token on EVERY single request

```
# Step 1: Login and SAVE the token
curl -X POST -d '{"email":"alice@ex.com","password":"abc"}' \
https://api.example.com/login
→ saves: {"access_token": "eyJhbGci..."}

# Step 2: Use the token in EVERY request after that
curl -H "Authorization: Bearer eyJhbGci..." \
https://api.example.com/profile
→ server reads token, finds you, returns profile ✓
```

**■ SERVER RESPONSE** — Profile response when token is correctly sent

```
{
"id": 1,                                  ■ server decoded the token to find your user ID
"name": "Alice Johnson",                  ■ your profile data
"email": "alice@example.com",
"role": "admin"                           ■ your permissions level
}
```

# ■ 3. HTTP Methods — The Verbs

HTTP methods tell the server **what action to perform**. URLs are the nouns (what thing), methods are the verbs (what to do). Always use the right method — this is what makes an API 'RESTful'.

| Method | Meaning | curl flag | Returns | Real example |
|---|---|---|---|---|
| GET | Read / fetch | `-X GET (default)` | 200 OK + data | Get your profile, list all posts |
| POST | Create something new | `-X POST` | 201 Created + new item | Register, submit a form, upload |
| PUT | Replace entire record | `-X PUT` | 200 OK + updated item | Overwrite all fields of a user |
| PATCH | Update specific fields | `-X PATCH` | 200 OK + updated item | Change only your email address |
| DELETE | Remove a resource | `-X DELETE` | 204 No Content | Delete your account, remove a post |

■ **Beginner Tip** GET is curl's default. You never need to write -X GET. `curl https://...` and `curl -X GET https://...` are exactly the same.

## GET — Read Data

GET is the most common method. It never changes any data on the server — it only reads. Safe to call as many times as you want.

**$ TERMINAL — type this** — Get a list of posts (limit to 3 results)

```
curl "https://jsonplaceholder.typicode.com/posts?_limit=3"
```

**■ SERVER RESPONSE** — 200 OK — an array of 3 post objects

```
[                                                          ■ [ = this is an array (list) of items
{
"userId": 1,                                               ■ which user wrote this post
"id": 1,                                                   ■ this post's unique ID
"title": "sunt aut facere...",                             ■ the post title
"body": "quia et suscipit..."                              ■ the post content
},
{ "userId": 1, "id": 2, "title": "qui est esse" ... },     ■ post #2
{ "userId": 1, "id": 3, "title": "ea molestias..." ... }   ■ post #3
]                                                          ■ ] = end of the array
```

## POST — Create New Data

POST sends data TO the server. You must set Content-Type so the server knows how to read your body. Server creates the resource and assigns it an ID.

**$ TERMINAL — type this** — Create a new post — send JSON data with -d

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"title":"My First Post","body":"Hello World","userId":1}' \
https://jsonplaceholder.typicode.com/posts
```

**■ SERVER RESPONSE** — 201 Created — the new resource, with its server-assigned ID

```
HTTP/1.1 201 Created
Location: /posts/101

{
"id": 101,
"title": "My First Post",
"body": "Hello World",
"userId": 1
}
```

- *201 (not 200!) = new resource was created*
- *URL of the new resource — save this!*
- *server assigned this ID automatically*
- *your data echoed back*

## PATCH — Update Only Specific Fields

PATCH is for partial updates. Only send the fields you want to change. Everything else stays exactly as it was. This is different from PUT, which replaces the entire object.

**$ TERMINAL — type this** — Only change the title — keep body and userId untouched

```
curl -X PATCH \
-H "Content-Type: application/json" \
-d '{"title":"Updated Title Only"}' \
https://jsonplaceholder.typicode.com/posts/1
```

**■ SERVER RESPONSE** — 200 OK — only title changed, other fields preserved

```
{
"userId": 1,
"id": 1,
"title": "Updated Title Only",
"body": "quia et suscipit..."
}
```

- *unchanged — we did not include it in PATCH*
- *unchanged*
- *CHANGED — this is what we sent*
- *unchanged — still the original text*

## DELETE — Remove a Resource

**$ TERMINAL — type this** — Delete post with ID 1

```
curl -X DELETE https://jsonplaceholder.typicode.com/posts/1 -i
```

**■ SERVER RESPONSE** — 204 No Content — deletion successful (empty body is expected!)

```
HTTP/1.1 204 No Content
Date: Mon, 24 Feb 2025 10:30:00 GMT
```

- *204 = success with no body to return*
- *when the response was sent*
- *no body after this — that is correct for DELETE*

■ **Beginner Tip** 204 No Content confuses beginners. It is NOT an error! It means: 'it worked, and there is nothing to return to you.'

# ■ 4. HTTP Status Codes

Every response starts with a 3-digit status code. The **first digit** tells you the category. Learn the first digit rule and you can immediately understand any status code you encounter.

| Family | Range | Category | Think of it as... |
|---|---|---|---|
| 1xx | 100–199 | Informational — still working | Server: 'Hold on, I'm processing...' |
| 2xx | 200–299 | Success — everything worked | Server: 'Done! Here is your result.' |
| 3xx | 300–399 | Redirect — resource moved | Server: 'Not here. Try this URL instead.' |
| 4xx | 400–499 | Client error — YOU made a mistake | Server: 'That's on you. Fix your request.' |
| 5xx | 500–599 | Server error — THEY have a problem | Server: 'Our bug. Please try again later.' |

| Code | Name | When you see it | What to do |
|---|---|---|---|
| 200 | OK | GET/PUT/PATCH succeeded | Read the response body — your data is there |
| 201 | Created | POST created a new resource | Check the Location header for the new URL |
| 204 | No Content | DELETE or action with nothing to return | Do nothing — empty body is expected and correct |
| 400 | Bad Request | You sent malformed or missing data | Check your JSON syntax and required fields |
| 401 | Unauthorized | No token, wrong token, expired token | Login again / send your token in every request |
| 403 | Forbidden | Token valid but you lack permission | Check your user role — you need higher access |
| 404 | Not Found | URL or resource ID doesn't exist | Check spelling of URL and verify the ID exists |
| 409 | Conflict | Duplicate (email already taken etc.) | Use a different value (email, username, etc.) |
| 422 | Unprocessable Entity | Validation errors in your request body | Fix the specific fields listed in the error details |
| 429 | Too Many Requests | You called the API too many times | Wait the number of seconds in Retry-After header |
| 500 | Internal Server Error | The server crashed — their bug | Retry after a delay; report to the API provider |
| 503 | Service Unavailable | Server down / overloaded | Retry with exponential backoff |

## See Status Codes with curl — Use the -i and -s Flags

By default curl shows only the response body. Add **-i** to also see the status code and headers:

**$ TERMINAL — type this** — Add -i to see headers + status code + body together

```
curl -i https://jsonplaceholder.typicode.com/posts/1
```

**■ SERVER RESPONSE** — Output with -i — headers come BEFORE the body

```
HTTP/2 200                                    ■ ← STATUS CODE — first thing to read
content-type: application/json                ■ ← body format: JSON
cache-control: max-age=43200                  ■ ← cache for 43200 seconds = 12 hours
etag: W/"124-yiKdLzqO5gfBrJFrcdJ8Yq0"         ■ ← fingerprint for cache validation
                                              ■ ← blank line = headers are done, body begins below
{
"userId": 1,
"id": 1,
"title": "sunt aut facere repellat provident...",
"body": "quia et suscipit suscipit recusandae..."
}
```

**$ TERMINAL — type this** — Just get the status code — useful in scripts

```
curl -o /dev/null -s -w "%{http_code}\n" https://jsonplaceholder.typicode.com/posts/1
```

**■ SERVER RESPONSE** — Only prints the 3-digit status code

```
200                                           ■ that is it — just the status code, nothing else
```

■ **Beginner Tip** -o /dev/null throws away the body, -s hides the progress bar, -w prints custom info. Together they give you just the status code — great for health checks.

# ■ 5. REST vs SOAP vs GraphQL

There are multiple ways to build APIs. REST is the most common. Here is how they compare:

| Feature | REST | SOAP | GraphQL |
|---|---|---|---|
| Learning curve | ■ Easy | Hard (lots of XML) | Medium |
| Data format | JSON (usually) | XML only | JSON |
| Flexibility | High | Very rigid | Very high |
| Over-fetching? | Yes (gets all fields) | Yes | No — ask for exact fields |
| Multiple requests? | Yes (one endpoint/resource) | Yes | No — one query for everything |
| Caching | Easy (built into HTTP) | Hard | Complex |
| Used by | GitHub, Stripe, Twitter | Banks, enterprise | GitHub v4, Shopify, Facebook |
| With curl | ■ Very simple | Complex XML required | POST with JSON query body |

### The Problem curl Shows Clearly: REST Over-fetching vs GraphQL

REST returns all fields even if you only need one. GraphQL lets you ask for exactly what you need:

**$ TERMINAL — type this** — REST — must fetch the entire user object just to get the name

```
curl https://jsonplaceholder.typicode.com/users/1
# Returns: id, name, username, email, address, phone, website, company
# Even if you only wanted the name!
```

**$ TERMINAL — type this** — GraphQL — ask for ONLY what you need in one query

```
curl -X POST https://api.github.com/graphql \
-H "Authorization: Bearer YOUR_GITHUB_TOKEN" \
-H "Content-Type: application/json" \
-d '{"query":"{ viewer { name email } "}'
```

**■ SERVER RESPONSE** — GraphQL gives you ONLY the 2 fields you asked for

```
{
"data": {                          ■ GraphQL always wraps data in a 'data' key
"viewer": {
"name": "Your Name",               ■ only name — nothing else
"email": "you@example.com"         ■ only email — nothing else
}
}
}
```

# ■ 6. Authentication & Security

Authentication answers: **Who are you?** Authorization answers: **What are you allowed to do?** Because REST is stateless, you must prove your identity on EVERY request.

| Method | How it works | curl syntax | Security level |
|---|---|---|---|
| API Key | Send a static secret key | `-H "X-API-Key: abc123"` | Medium — rotate regularly |
| Basic Auth | Send username + password encoded | `curl -u user:pass https://...` | Low alone — always use HTTPS |
| JWT Bearer | Send a signed token each request | `-H "Authorization: Bearer eyJ..."` | High — token expires automatically |
| OAuth 2.0 | Login via a third-party provider | Multi-step browser flow | Very high — delegated access |

### Method 1: API Key — Simplest

You get a static key from the API provider. Include it in every request header. Never put it in the URL (it gets logged).

**$ TERMINAL — type this** — API key in a header — the correct way

```
# Correct — key goes in a header
curl -H "X-API-Key: sk_live_abc123xyz" \
https://api.example.com/data

# WRONG — key in the URL gets saved in server logs!
curl "https://api.example.com/data?api_key=sk_live_abc123xyz"
```

**■ SERVER RESPONSE** — 200 OK when key is valid

```
{ "data": [...] }
```

■ *your data — key was accepted*

### Method 2: Basic Auth — curl Makes it Easy with -u

**$ TERMINAL — type this** — curl encodes username:password automatically with -u

```
curl -u alice@example.com:mypassword \
https://api.example.com/profile

# What curl actually sends (Base64 encoded):
# Authorization: Basic YWxpY2VAZXhhbXBsZS5jb206bXlwYXNzd29yZA==
```

### Method 3: JWT Bearer Token — Most Common for REST APIs

JWT (JSON Web Token) is 3 base64 parts joined by dots: header.payload.signature. You login once, receive a token, and include it in every request. The server verifies the signature — no database lookup needed.

**$ TERMINAL — type this** — Step 1 — Login to get your token

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"email":"alice@example.com","password":"secret123"}' \
https://api.example.com/auth/login
```

**■ SERVER RESPONSE** — Login response — save the access_token immediately!

```
{
"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9      ■ ← SAVE THIS — it is your key
.eyJzdWIiOiIxIiwiZXhwIjoxNzA1OTAwMDAwfQ                   ■ ← contains your user ID and expiry
.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_a",                    ■ ← cryptographic signature
"expires_in": 900,                                         ■ ← valid for 900 seconds = 15 minutes
"token_type": "Bearer",                                    ■ ← always use the word Bearer before the token
"refresh_token": "rt_xyz789abc..."                         ■ ← use this to get a new access_token later
}
```

**$ TERMINAL — type this** — Step 2 — Use the token in every subsequent request

```
# On Linux/Mac — save token to a variable:
TOKEN="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

# Use the variable in your curl command:
curl -H "Authorization: Bearer $TOKEN" \
https://api.example.com/profile
```

**■ SERVER RESPONSE** — 200 OK — server decoded your token and found your account

```
{
"id": 1,                                                   ■ server found your user ID inside the token
"name": "Alice Johnson",                                   ■ your profile
"email": "alice@example.com",
"role": "admin"                                            ■ your permissions
}
```

**$ TERMINAL — type this** — What happens when token expires — you get a 401

```
curl -H "Authorization: Bearer EXPIRED_TOKEN_HERE" \
https://api.example.com/profile
```

**■ SERVER RESPONSE** — 401 Unauthorized — your token has expired

```
HTTP/1.1 401 Unauthorized                          ■ 401 = authentication failed
WWW-Authenticate: Bearer error="invalid_token"     ■ server tells you WHY it rejected you

{
"error": "TOKEN_EXPIRED",                          ■ machine-readable error code
"message": "Token expired. Use refresh_token to get a new
one."                                              ■ human-readable message
}
```

■ **Beginner Tip** When you see 401: your token is missing, wrong, or expired. Re-login or use your refresh_token. When you see 403: your token is VALID but you don't have permission for that action.

# ■ 7. Request & Response Structure

Every HTTP request and response has the same structure. Understanding it makes reading curl output easy.

## Every HTTP Request Has These 4 Parts

| Part | Where it goes | curl flag | Example |
|---|---|---|---|
| Request line | First line: method + path | (implicit) | POST /api/users HTTP/1.1 |
| Headers | Meta-information lines | -H (one per header) | Content-Type: application/json |
| Blank line | Separator — always present | (automatic) | (empty line) |
| Body | The data you send (optional) | -d | {"name":"Alice"} |

## Fully Annotated curl Request — Every Flag Explained

**$ TERMINAL — type this** — Create a new user — complete command

```
curl                                          ■ the tool
-X POST                                       ■ HTTP method: POST = create something
-H "Content-Type: application/json"           ■ tell server: my body is in JSON format
-H "Authorization: Bearer eyJhbGci..."        ■ prove who you are with your token
-H "Accept: application/json"                 ■ tell server: send response in JSON
-H "X-Request-ID: my-trace-12345"             ■ optional: your ID for tracking this request
-d '{"name":"Alice","email":"a@b.com"}' \     ■ the JSON body data you are sending
https://api.example.com/v1/users              ■ the URL — v1 = API version
```

## What curl Sends Over the Wire (HTTP request)

**■ HTTP REQUEST (what curl sends)** — The actual HTTP request bytes curl transmits

```
POST /v1/users HTTP/1.1                       ■ method + path + HTTP version
Host: api.example.com                         ■ required — which server to talk to
Content-Type: application/json                ■ format of the body we are sending
Authorization: Bearer eyJhbGci...             ■ our auth token
Accept: application/json                      ■ format we want back
Content-Length: 40                            ■ how many bytes in the body (curl calculates this)
                                              ■ ← blank line: marks end of headers
{"name":"Alice","email":"alice@example.com"}  ■ ← request body starts here
```

## Fully Annotated Server Response

**■ SERVER RESPONSE** — 201 Created — complete response with all parts explained

```
HTTP/1.1 201 Created                    ■ status line: HTTP version + code + reason text
Content-Type: application/json          ■ body format of the response
Location: /v1/users/44                  ■ URL where the new resource lives — save this!
X-Request-ID: my-trace-12345            ■ your request ID echoed back — for debugging
Cache-Control: no-store                 ■ don't cache this response (has private data)
X-RateLimit-Remaining: 98               ■ how many API calls you have left this window
                                        ■ ← blank line: headers are done, body begins
{
"id": 44,                               ■ server assigned this ID to the new user
"name": "Alice",                        ■ data that was saved
"email": "alice@example.com",
"created_at": "2025-02-24T10:30:00Z"    ■ ISO 8601 timestamp — always UTC (the Z means UTC)
}
```

# ■ 8. REST API Design Best Practices

These rules make APIs consistent and easy to use. As a consumer of APIs, knowing these helps you predict how any well-built API works. As a builder, they are your blueprint.

| Rule | ■ Bad | ■ Good | Why |
|---|---|---|---|
| Use nouns, not verbs | /getUsers | /users | The method (GET) is already the verb |
| Always use plural | /user | /users | Consistent — even for single items: /users/1 |
| Lowercase + hyphens | /userProfiles | /user-profiles | URLs are case-insensitive; hyphens are readable |
| Nest related resources | /getOrdersByUser | /users/42/orders | Hierarchy shows relationship |
| No file extensions | /users.json | /users | Use Accept header to specify format |
| No verbs in URL | /deletePost/3 | DELETE /posts/3 | Method = verb, URL = noun |

**$ TERMINAL — type this** ■ BAD API — verbs in URLs, POST for everything

```
curl "https://api.example.com/getUser?id=42"
curl -X POST "https://api.example.com/createUser"
curl -X POST "https://api.example.com/updateUser?id=42"
curl "https://api.example.com/deleteUser?id=42"
```

**$ TERMINAL — type this** ■ GOOD REST API — nouns + correct HTTP verbs

```
curl https://api.example.com/users/42
curl -X POST https://api.example.com/users
curl -X PATCH https://api.example.com/users/42
curl -X DELETE https://api.example.com/users/42
```

### Always Wrap List Responses in an Envelope Object

■ SERVER RESPONSE ■ Bad — raw array. No metadata, no pagination info

```
[{"id":1,"name":"Alice"},{"id":2,"name":"Bob"}]
```

■ *raw array — no way to know total count or pages*

■ SERVER RESPONSE ■ Good — envelope with data + pagination metadata

```
{
"data": [                                   ■ your results are inside 'data'
{"id":1,"name":"Alice"},
{"id":2,"name":"Bob"}
],
"meta": {                                   ■ pagination info always present
"total": 250,                               ■ total records in the database
"page": 1, "per_page": 10                   ■ current page info
},
"links": {                                  ■ ready-made URLs for navigation
"next": "/users?page=2",                    ■ click to go to next page
"prev": null                                ■ null = we are on the first page
}
}
```

## ■ 9. Versioning Strategies

APIs change over time — fields get renamed, new fields are added, behaviour changes. Versioning lets you roll out breaking changes without breaking existing clients who use the old version.

| Strategy | Example URL/Header | Pros | Cons |
|---|---|---|---|
| URL path (most common) | /api/v1/users → /api/v2/users | Visible, easy to test in browser | Slightly 'dirty' URLs |
| Query param | /users?version=2 | Simple, no URL change | Easy to forget, cache issues |
| Custom header | API-Version: 2 | Clean URLs | Not visible in browser bar |
| Accept header | Accept: application/vnd.api.v2+json | Truly RESTful standard | Complex, rarely used in practice |

**$ TERMINAL — type this** — v1 vs v2 — same resource, different response shapes

```
# v1 — old response format
curl https://api.example.com/v1/users/1

# v2 — new response format (breaking change)
curl https://api.example.com/v2/users/1
```

**■ SERVER RESPONSE** — v1 response — flat name field

```
{ "id":1, "full_name":"Alice Johnson",
"email":"alice@example.com" }
```

■ *v1: full_name as one string*

**■ SERVER RESPONSE** — v2 response — name is now an object, new fields added

```
{
"id": 1,
"name": { "first":"Alice", "last":"Johnson" },
"email": "alice@example.com",
"verified": true,
"created_at": "2024-01-01T00:00:00Z"
}
```

■ *BREAKING CHANGE: name is now nested*

■ *NEW field — only in v2*

■ *NEW field — only in v2*

**Deprecation:** Send a `Sunset: Sat, 01 Jun 2026 00:00:00 GMT` header in v1 responses to warn clients before removing the old version.

# ■ 10. TCP Connection & Handshaking

Before your curl command can send a single byte of your REST request, the computer must first shake hands with the server to establish a secure connection. This all happens in milliseconds, completely invisible to you — unless you use -v.

| Step | Layer | What Happens | Visible with -v? |
|---|---|---|---|
| 1 | TCP | 3-way handshake: SYN → SYN-ACK → ACK (opens a connection) | ■ 'Connected to...' |
| 2 | TLS | Certificates exchanged, encryption agreed (secures the channel) | ■ 'SSL connection using' |
| 3 | HTTP | Your REST request travels through the secure tunnel | ■ Lines starting with > |
| 4 | HTTP | Server's response arrives back through the tunnel | ■ Lines starting with < |
| 5 | TCP | Connection is kept alive for next requests (HTTP/2 reuses it) | ■ 'Re-using connection' |

■ **Beginner Tip** HTTP/2 (default on most modern APIs) reuses ONE TCP+TLS handshake for ALL your requests to the same server. This is why APIs today feel so much faster than old HTTP/1.0 where every request paid the full handshake cost.

**$ TERMINAL — type this** — See the full handshake with -v (redirect stderr to stdout with 2>&1)

```
curl -v https://jsonplaceholder.typicode.com/posts/1 2>&1
```

**■ SERVER RESPONSE** — Handshake steps visible in verbose output

```
* Trying 104.21.45.109:443...                    ■ Step 1a: DNS resolved, opening TCP socket
* Connected to ...typicode.com port 443          ■ Step 1b: TCP 3-way handshake complete ✓
* ALPN: curl offers h2,http/1.1                  ■ Negotiating HTTP version
* SSL connection using TLSv1.3                   ■ Step 2a: TLS cipher agreed — AES-256 encryption
* Server certificate: *.typicode.com             ■ Step 2b: Certificate identity verified ✓
* using HTTP/2                                   ■ HTTP/2 — this connection can be reused!
> GET /posts/1 HTTP/2                           ■ Step 3: YOUR request finally sent
> Host: jsonplaceholder.typicode.com
< HTTP/2 200                                    ■ Step 4: Server response received
< content-type: application/json
* Connection #0 to host kept alive              ■ Step 5: Connection saved for reuse
```

# ■ 11. CORS & Same-Origin Policy

■ **Warning** CORS only applies to requests made from JavaScript in a browser. curl is NOT a browser, so curl NEVER encounters CORS errors — even if the API has strict CORS settings.

Browsers enforce the **Same-Origin Policy**: JavaScript on https://myapp.com can only call APIs on https://myapp.com. CORS is the mechanism that lets server explicitly permit calls from other origins.

| Caller | CORS applies? | Why |
|---|---|---|
| curl from terminal | ■ Never | curl is not a browser — no same-origin policy |
| JavaScript fetch() / axios | ■ Always | Browser enforces same-origin policy |
| Postman / Insomnia / Bruno | ■ Never | API testing tools, not browsers |
| Mobile app (Swift/Kotlin) | ■ Never | Native apps, not browsers |
| Server-side code (Node/Python) | ■ Never | Servers don't enforce browser policies |

## How CORS Preflight Works

Before a cross-origin POST/PUT/PATCH/DELETE, the browser automatically sends an OPTIONS request to ask: 'May I?' The server responds with what it allows.

**$ TERMINAL — type this** — Simulate a CORS preflight OPTIONS request with curl

```
curl -X OPTIONS \
-H "Origin: https://myapp.com" \
-H "Access-Control-Request-Method: POST" \
-H "Access-Control-Request-Headers: Authorization, Content-Type" \
-i https://api.example.com/users
```

**■ SERVER RESPONSE** — Server says YES — browser now proceeds with the real request

```
HTTP/1.1 204 No Content                          ■ 204 = preflight approved (no body needed)
Access-Control-Allow-Origin: https://myapp.com   ■ ONLY myapp.com can call this API
Access-Control-Allow-Methods: GET, POST, PUT, DELETE   ■ these methods are allowed
Access-Control-Allow-Headers: Authorization, Content-Type   ■ these headers are allowed
Access-Control-Allow-Credentials: true           ■ cookies and auth headers are permitted
Access-Control-Max-Age: 86400                    ■ browser caches this for 24h — no preflight tomorrow
```

**■ SERVER RESPONSE** — CORS error in browser console (when server says NO)

```
Access to fetch at "https://api.example.com/users"
from origin "https://evil.com" has been blocked by CORS
policy:
No "Access-Control-Allow-Origin" header is present.

# To fix: add your frontend origin to the server's CORS
allow-list
```

■ *which API was blocked*

■ *CORS rejected it*

■ *server did not give permission*

■ *server-side config change*

# ■ 12. Rate Limiting & Throttling

Every public API limits how many requests you can make per time window to prevent abuse and ensure fair usage. When you exceed the limit you get a **429 Too Many Requests** response.

| Header | What it tells you | Example |
|---|---|---|
| X-RateLimit-Limit | Max requests allowed per window | 100 |
| X-RateLimit-Remaining | How many you have left right now | 43 |
| X-RateLimit-Reset | Unix timestamp when your allowance resets | 1705329600 |
| Retry-After | Seconds to wait before retrying (with 429) | 47 |

**$ TERMINAL — type this** — Check your rate limit status — look at the response headers

```
curl -I -H "Authorization: Bearer TOKEN" \
https://api.github.com/users/torvalds
# -I = HEAD request — gets headers only, no body
```

**■ SERVER RESPONSE** — Rate limit headers in a normal response

```
HTTP/2 200
x-ratelimit-limit: 5000          ■ you get 5000 requests per hour
x-ratelimit-remaining: 4987      ■ you have used 13, 4987 left
x-ratelimit-used: 13             ■ total used in this window
x-ratelimit-reset: 1740390000    ■ resets at this Unix timestamp
```

**$ TERMINAL — type this** — What happens when you exceed the limit

```
curl -H "Authorization: Bearer TOKEN" https://api.example.com/data
# (imagine this is the 101st request in 1 minute)
```

**■ SERVER RESPONSE** — 429 Too Many Requests — you must wait

```
HTTP/1.1 429 Too Many Requests   ■ 429 = you have been rate limited
Retry-After: 47                  ■ wait exactly 47 seconds, then try again
X-RateLimit-Remaining: 0         ■ zero left — limit exhausted

{
"error": "RATE_LIMIT_EXCEEDED",  ■ machine-readable code for your if/else
"message": "Retry after 47 seconds",  ■ human-readable message
"retry_after": 47                ■ seconds to wait
}
```

■ **Beginner Tip** In a bash script, automatically wait the right amount: `sleep $(curl ... | python3 -c "import sys,json; print(json.load(sys.stdin)['retry_after'])")`

# ■ 13. Pagination & Filtering

APIs never return ALL records at once — an API with 10 million users would crash your connection. Pagination splits data into pages. Filtering narrows what you get back.

| Query Param | Purpose | curl Example |
|---|---|---|
| page / _page | Which page number you want | `?page=3` |
| limit / per_page | Items per page | `?per_page=10` |
| offset | Skip N records (alternative to page) | `?offset=20&limit;=10` |
| sort / _sort | Field to sort by | `?sort=created_at` |
| order / _order | Direction: asc or desc | `?order=desc` |
| filter fields | Narrow by value | `?role=admin&active;=true` |
| q / search | Full text search | `?q=alice+johnson` |
| fields / select | Return only listed fields | `?fields=id,name,email` |

**$ TERMINAL — type this** — Page 2 of posts, 3 per page, sorted by ID descending

```
curl "https://jsonplaceholder.typicode.com/posts?_page=2&_limit=3&_sort=id&_order=desc"

# Note: always quote URLs with & in your shell
# Otherwise the shell splits them into separate commands!
```

**■ SERVER RESPONSE** — Paginated response with metadata

```
{
"data": [                                   ■ your 3 results for this page
{ "id": 100, "title": "..." },
{ "id": 99, "title": "..." },
{ "id": 98, "title": "..." }
],
"meta": {                                   ■ pagination info
"total": 100,                               ■ 100 total posts in the database
"page": 2,                                  ■ you are on page 2
"per_page": 3,                              ■ 3 items per page
"total_pages": 34                           ■ there are 34 pages total
},
"links": {                                  ■ ready-made URLs to navigate
"prev": "/posts?_page=1&_limit=3",          ■ previous page URL
"next": "/posts?_page=3&_limit=3",          ■ next page URL
"first": "/posts?_page=1&_limit=3",
"last": "/posts?_page=34&_limit=3"
}
}
```

■ **Beginner Tip** Always quote URLs with & in the shell: curl 'https://api.com/posts?page=2&limit;=10' (single quotes) or curl "https://..." (double quotes). Without quotes the shell interprets & as 'run in background'.

# ■ 14. Error Handling

A well-designed API always returns structured, helpful error responses — not just a status code. The error body should tell you: what went wrong, which field caused it, and how to fix it.

| Error field | Purpose | Example value |
|---|---|---|
| code | Machine-readable — use in if/else in your code | VALIDATION_ERROR |
| message | Human-readable — safe to display to end users | Email is required |
| field | Which specific field failed | email |
| request_id | Unique ID — give to support team when reporting bugs | req-abc-123-xyz |
| details | Array of ALL errors at once (not one at a time) | [{field, message}, ...] |

## Triggering and Reading Common Errors

**$ TERMINAL — type this** — 400 Bad Request — send malformed JSON (missing closing brace)

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"name":"Alice", "email":"alice@ex.com"' \
https://api.example.com/users
# Missing closing } — JSON is invalid
```

**■ SERVER RESPONSE** — 400 Bad Request — server cannot even parse the body

```
HTTP/1.1 400 Bad Request
{
"error": "BAD_REQUEST",
"message": "Invalid JSON: Unexpected end of input"
}
```

■ *your JSON was unparseable*

■ *category of error*

■ *tells you exactly what is wrong*

**$ TERMINAL — type this** — 422 Unprocessable — valid JSON but fails validation rules

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"name":"","email":"not-valid","age":-5}' \
https://api.example.com/users
```

**■ SERVER RESPONSE** — 422 — all validation errors returned at once (not one by one)

```
HTTP/1.1 422 Unprocessable Entity                    ■ validation failed
X-Request-Id: req-abc-def-123                        ■ save this ID when reporting bugs

{
"error": {
"code": "VALIDATION_ERROR",                          ■ use this in your if/else
"message": "Request validation failed",
"request_id": "req-abc-def-123",                     ■ same ID as in the header
"details": [                                         ■ ALL errors listed at once
{"field":"name", "message":"Name is required"},
{"field":"email","message":"Invalid email format"},
{"field":"age", "message":"Must be 0 or greater"}
]
}
}
```

**$ TERMINAL — type this** — 404 Not Found — resource with that ID does not exist

```
curl https://api.example.com/users/99999
```

**■ SERVER RESPONSE** — 404 Not Found — clear message about what is missing

```
HTTP/1.1 404 Not Found
{
"error": {
"code": "NOT_FOUND",
"message": "User with id 99999 does not exist",      ■ very specific — not just 'not found'
"request_id": "req-xyz-456"
}
}
```

# ■ 15. Real-World API Examples

## JSONPlaceholder — Free Practice API (Perfect for Beginners)

No signup, no API key, no cost. All changes are simulated — nothing is saved. Use this to practice all the concepts in this guide.

**$ TERMINAL — type this** — Practice these right now in your terminal

```
# Get all users (100 total)
curl https://jsonplaceholder.typicode.com/users

# Get only user #3
curl https://jsonplaceholder.typicode.com/users/3

# Get all posts by user #1
curl "https://jsonplaceholder.typicode.com/posts?userId=1"

# Create a post
curl -X POST -H "Content-Type: application/json" \
-d '{"title":"Test","body":"Hello REST!","userId":1}' \
https://jsonplaceholder.typicode.com/posts

# Delete post #5
curl -X DELETE https://jsonplaceholder.typicode.com/posts/5 -i
```

## GitHub REST API — A Real Production API

**$ TERMINAL — type this** — Public endpoints — no token needed

```
# Get any public GitHub user
curl https://api.github.com/users/torvalds

# Get a public repository
curl https://api.github.com/repos/facebook/react

# List repos for a user (with pagination)
curl "https://api.github.com/users/torvalds/repos?per_page=5&sort;=updated"
```

**■ SERVER RESPONSE** — GitHub user response (simplified)

```
{
"login": "torvalds",          ■ GitHub username
"name": "Linus Torvalds",     ■ display name
"bio": "Nothing.",            ■ profile bio
"public_repos": 8,            ■ number of public repos
"followers": 230000,          ■ follower count
"created_at": "2011-09-03T15:26:22Z"   ■ account creation date
}
```

**$ TERMINAL — type this** — Authenticated endpoint — get YOUR profile (needs GitHub token)

```
# Create a token at: github.com/settings/tokens
curl -H "Authorization: Bearer ghp_YOUR_GITHUB_TOKEN" \
https://api.github.com/user
```

**■ SERVER RESPONSE** — Authenticated response — sees private fields too

```
{
"login": "yourusername",
"email": "you@example.com",
"private_repos": 12,
"plan": {"name":"pro"}
}
```

- ■ *private — only visible when authenticated*
- ■ *private — only you can see this*
- ■ *your subscription level*

## Open-Meteo Weather API — Free, No Signup Required

**$ TERMINAL — type this** — Get current weather for Bengaluru (free API, no key!)

```
curl "https://api.open-meteo.com/v1/forecast
?latitude=12.9716
&longitude;=77.5946
&current;=temperature_2m,wind_speed_10m
&timezone;=Asia/Kolkata"
```

**■ SERVER RESPONSE** — Weather response with current conditions

```
{
"latitude": 12.9716,
"timezone": "Asia/Kolkata",
"current": {
"time": "2025-02-24T15:30",
"temperature_2m": 26.5,
"wind_speed_10m": 8.2
}
}
```

- ■ *coordinates confirmed*
- ■ *timezone used*
- ■ *when this reading was taken*
- ■ *temperature in Celsius at 2m height*
- ■ *wind speed km/h at 10m height*

# ■ 16. curl Cheat Sheet — Complete Reference

## Essential Patterns

| Task | curl Command |
|---|---|
| Simple GET | `curl https://api.example.com/users` |
| GET with auth token | `curl -H "Authorization: Bearer TOKEN" https://...` |
| GET with query parameters | `curl 'https://api.example.com/users?page=2&limit;=10'` |
| See response headers + body | `curl -i https://api.example.com/users` |
| See FULL conversation | `curl -v https://api.example.com/users` |
| Silence progress bar | `curl -s https://api.example.com/users` |
| POST — create resource | `curl -X POST -H 'Content-Type: application/json' -d '{"name":"Alice"}' https://...` |
| PATCH — partial update | `curl -X PATCH -H 'Content-Type: application/json' -d '{"email":"new@x.com"}' https://.../1` |
| PUT — full replace | `curl -X PUT -H 'Content-Type: application/json' -d '{...all fields...}' https://.../1` |
| DELETE a resource | `curl -X DELETE https://api.example.com/users/1` |
| Basic auth | `curl -u username:password https://api.example.com` |
| API key in header | `curl -H 'X-API-Key: mykey123' https://api.example.com` |
| Save response to file | `curl -o response.json https://api.example.com/users` |
| Get only the status code | `curl -o /dev/null -s -w '%{http_code}' https://...` |
| Follow redirects (3xx) | `curl -L https://api.example.com/redirect` |
| Pretty-print JSON output | `curl https://api.example.com/users \| python3 -m json.tool` |
| Time how long request takes | `curl -w 'Time: %{time_total}s\n' -o /dev/null -s https://...` |
| Send a file as body | `curl -X POST -H 'Content-Type: application/json' -d @data.json https://...` |

## Status Code Decision Tree

| Code | Meaning | Most likely cause | What to do |
|---|---|---|---|
| 200 | OK | Normal successful request | Read the body — your data is there |
| 201 | Created | POST was successful | Check Location header for new URL |

| Code | Meaning | Most likely cause | What to do |
|---|---|---|---|
| 204 | No Content | DELETE or action with no return | Success — empty body is correct |
| 400 | Bad Request | Malformed JSON or missing fields | Fix your -d data — check JSON syntax |
| 401 | Unauthorized | Missing, wrong, or expired token | Re-login or add -H Authorization |
| 403 | Forbidden | Valid token, insufficient rights | Need a higher-permission account |
| 404 | Not Found | Wrong URL or wrong ID | Double-check URL spelling and ID value |
| 409 | Conflict | Duplicate value (email in use) | Use a different value |
| 422 | Unprocessable | Validation failed on fields | Fix the specific fields in error.details |
| 429 | Too Many Requests | Rate limit hit | Wait Retry-After seconds |
| 500 | Server Error | Bug on their side | Retry later; report to API team |
| 503 | Service Unavailable | Server overloaded or maintenance | Retry with increasing delays |