API Concepts

Verified 2026-08-02

The twenty concepts you need to design, consume, or review any HTTP API.

Contents

Mental model

An API is a contract between programs: one side publishes operations with stable names, inputs, and outputs; the other side calls them without knowing how they are implemented. Almost every concept below answers one of two questions — how do I change my system without breaking someone else's, and what happens when the network fails mid-call? The network is never an implementation detail: a call can be slow, duplicated, or lost, so an API is designed for failure from the start rather than patched for it later.

Endpoint

Part	Example	Carries
Host	`api.shop.com`	Which service
Path	`/orders/42`	Which resource
Path param	`42`	Which instance
Query	`?limit=20`	How to shape the result
Fragment	`#top`	Nothing — never sent

An endpoint is one addressable operation: a URL plus a method. Name paths after resources (nouns, usually plural), not actions — the verb is the method, so `/orders/42` with `DELETE` beats `/deleteOrder`. Use the query string to filter, sort, and paginate a collection; it should never change *what* the resource is.

```
GET /orders/42 HTTP/1.1
Host: api.shop.com

GET /orders?status=open&limit=20 HTTP/1.1
Host: api.shop.com
```

> **Gotcha:** `/orders` and `/orders/` are different URLs to many routers, caches, and signature schemes. Pick one form and redirect the other permanently.

HTTP Methods

Method	Purpose	Safe / Idempotent
`GET`	Read a resource	Yes / Yes
`POST`	Create, or run an action	No / No
`PUT`	Replace the whole resource	No / Yes
`PATCH`	Modify part of it	No / No
`DELETE`	Remove it	No / Yes
`HEAD`	Headers only, no body	Yes / Yes

Safe means the call causes no observable change; *idempotent* means making it twice leaves the server in the same state as making it once. These are promises to everyone in the middle — browsers, proxies, and retry layers prefetch and replay safe methods without asking you.

```
PATCH /orders/42 HTTP/1.1
Content-Type: application/merge-patch+json

{"status": "cancelled"}
```

> **Gotcha:** `DELETE` is idempotent even though the second call usually returns `404`. Idempotency constrains the resulting server state, not the status code.

Request-Response

Part	In a request	In a response
Start line	`POST /orders`	`201 Created`
Headers	`Accept`, `Authorization`	`Content-Type`, `ETag`
Body	What you send	The representation

One request gets one response, and the server remembers nothing between them. That is statelessness: every request carries everything needed to serve it, which is exactly what lets any instance behind a load balancer answer any call.

```
POST /orders HTTP/1.1
Content-Type: application/json
Accept: application/json

{"sku": "A-17", "qty": 2}
```

```
HTTP/1.1 201 Created
Location: /orders/42
```

> **Note:** Statelessness is about server memory, not about the user having no session. Session state is fine — it just travels in a token on every request instead of living in server RAM.

Status Codes

Code	Name	Use when
`200`	OK	Read or update succeeded
`201`	Created	New resource; add `Location`
`204`	No Content	Success, nothing to return
`400`	Bad Request	Malformed or invalid input
`404`	Not Found	Unknown, or hidden on purpose
`409`	Conflict	State clash, e.g. a duplicate

The first digit is the class: `2xx` succeeded, `3xx` look elsewhere, `4xx` the caller must change something, `5xx` the server failed and the identical request might work later. That last split is what tells a client whether retrying is pointless or correct.

```
HTTP/1.1 409 Conflict
Content-Type: application/problem+json
```

```
{"type": "https://api.shop.com/errors/dup",
 "title": "Order already exists",
 "status": 409}
```

> **Gotcha:** `200 OK` with `{"error": ...}` in the body defeats retries, caches, and monitoring at once — every layer between you and the client reads the status line, never the body.

Authentication

Scheme	Sent as	Fits
Bearer token	`Authorization: Bearer …`	Users and apps
API key	`X-API-Key: …`	Server-to-server
Basic	`Authorization: Basic …`	Legacy, internal
mTLS	Client certificate	High-trust B2B

Authentication answers “who is calling?” Because the server keeps no memory between requests, credentials are verified on every single call. Reject unauthenticated requests with `401` and say how to authenticate in `WWW-Authenticate`.

```
GET /me HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9...

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token"
```

> **Warning:** Never accept credentials in the query string. URLs are written to access logs, browser history, and `Referer` headers — a token in a URL is a token in a dozen places you don't control.

Authorization

Model	Decision from	Example
RBAC	The caller's role	`admin` may delete
ABAC	Attributes	Only the record's owner
Scopes	Grants in the token	`orders:read`
ACL	Per-object list	A shared document

Authorization answers "may this caller do this, to *this* object?" It is two checks, not one: does the token grant the operation, and does this subject have rights to that specific instance. Enforce it server-side on every request — hiding a button changes nothing.

```
GET /orders/42 HTTP/1.1
Authorization: Bearer <token with orders:read>


HTTP/1.1 403 Forbidden
```

> **Gotcha:** A valid token with the right scope is still not permission for someone else's row. Skipping the per-object owner check is the most common API vulnerability there is — `/orders/43` quietly returns another customer's order.

Access Tokens

Token	Lifetime	Sent where
Access	Minutes	`Authorization`, every call
Refresh	Days to months	Token endpoint only
ID token	Minutes	Client only — identity

An access token is a bearer credential: whoever holds it can use it, so it is as sensitive as a password. A JWT is self-contained — the signature is verified locally with no lookup, which is fast but makes revocation hard; an opaque token needs introspection but dies the moment you delete it. Keep access tokens short-lived either way.

```
{
  "sub": "user_991",
  "scope": "orders:read orders:write",
  "iss": "https://auth.shop.com",
  "aud": "https://api.shop.com",
  "exp": 1785000000
}
```

> **Warning:** A JWT is signed, not encrypted — anyone holding it can read every claim. Put no secrets in it, and on the server verify `iss`, `aud`, `exp`, and the algorithm, not just the signature.

OAuth 2.0

Grant	Use for
Authorization code + PKCE	Web, mobile, SPA
Client credentials	Machine-to-machine
Device code	TVs, CLIs, no browser
Refresh token	Renewing access

OAuth 2.0 is delegated authorization: a user lets an app call an API on their behalf without giving it their password. Four roles — the resource owner (user), the client (app), the authorization server (issues tokens), and the resource server (your API). Authorization code with PKCE is the default for anything user-facing; the implicit and password grants are deprecated, so treat them as legacy only.

```
POST /oauth/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=SplxlOBeZQ
&redirect_uri=https://app.example/cb
&code_verifier=dBjftJeZ4CVP
```

> **Gotcha:** OAuth is authorization, not authentication. An access token says an app may act — it does not tell you who the user is. Use OpenID Connect and an ID token when you need identity.

Rate-Limiting

Header	Meaning
`RateLimit-Limit`	Ceiling for the window
`RateLimit-Remaining`	Calls left in it
`RateLimit-Reset`	Seconds until it resets
`Retry-After`	Wait this long (with `429`)

Rate limiting caps how many calls a client may make per window, protecting shared capacity and enforcing pricing tiers. Pick the key deliberately: per token or per user is fair, per IP alone punishes everyone behind one NAT. Publish the numbers in headers so clients can pace themselves instead of discovering the limit by failing.

```
HTTP/1.1 429 Too Many Requests
RateLimit-Limit: 100
RateLimit-Remaining: 0
RateLimit-Reset: 42
Retry-After: 42
```

> **Tip:** Retry with exponential backoff *plus jitter*. Fixed intervals re-synchronize every blocked client into one spike at the moment the window resets.

Throttling

Strategy	Behavior	Client sees
Reject	Drop the excess	`429`
Shape	Queue and delay	Slower responses
Degrade	Cheaper answer	Partial data

Rate limiting is the policy — the number in the contract. Throttling is the enforcement: what actually happens to the call that crosses it. Token bucket allows bursts up to the bucket size then refills steadily; leaky bucket smooths output to a fixed rate; a sliding window avoids the double-burst at fixed-window boundaries.

```
throttle:
  key: token
  algorithm: token_bucket
  rate: 10/s
  burst: 50
  on_exceed: reject   # or: queue
```

> **Gotcha:** Queueing instead of rejecting converts overload into latency. Clients time out, retry, and add more load — shed traffic early and cheaply rather than holding it.

Pagination

Style	Params	Trade-off
Offset	`?offset=40&limit=20`	Simple; drifts, slow deep
Cursor	`?cursor=abc&limit=20`	Stable; no page jumps
Page	`?page=3&size=20`	Familiar; offset in disguise

Never return an unbounded collection: set a default limit and a hard maximum. Offset paging re-reads every skipped row and duplicates or drops items when rows are inserted mid-scan; cursor (keyset) paging encodes the last-seen sort key, so it stays correct and fast at any depth. Return the next cursor in the payload — clients should treat it as opaque and never construct one.

```
{
  "data": [{"id": 44}, {"id": 43}],
  "page": {
    "next": "eyJpZCI6NDN9",
    "limit": 20
  }
}
```

> **Gotcha:** Cursor paging needs a total order. Sort by a unique tiebreaker such as `(created_at, id)`, or rows sharing a timestamp fall between pages and are never returned.

Caching

Header	Role
`Cache-Control`	Who may store it, how long
`ETag`	Version tag to validate against
`If-None-Match`	Client's copy; may get `304`
`Vary`	Which headers change the answer

Caching buys two different things. Freshness (`max-age`) skips the request entirely; validation (`ETag` plus `If-None-Match`) still makes the round trip but skips the body via `304 Not Modified`. Mark per-user responses `private` and shared ones `public`, and set both deliberately — the default is whatever your framework guessed.

```
GET /orders/42 HTTP/1.1
If-None-Match: "v7"

HTTP/1.1 304 Not Modified
ETag: "v7"
Cache-Control: private, max-age=60
```

> **Gotcha:** A shared cache keyed without `Vary: Authorization` will hand one user's response to the next user. If a response depends on who asked, say so in `Vary` or mark it `private`.

Idempotency

Method	Safe to repeat	Why
`GET`, `HEAD`	Yes	Changes nothing
`PUT`, `DELETE`	Yes	State converges
`POST`	No	Creates each time

On the wire, a retry is indistinguishable from a new call: a client that times out has no idea whether the server processed the request. Make unsafe operations replay-safe with an idempotency key — the client generates one key per logical operation, and the server stores key → result and replays the stored response for repeats.

```
POST /payments HTTP/1.1
Idempotency-Key: 7c1f-4b2a-9de3
Content-Type: application/json

{"amount": 4200, "currency": "EUR"}
```

> **Gotcha:** Generate the key once per operation and reuse it across every retry. Generating it inside the retry loop makes each attempt a new operation — and charges the customer twice.

Webhooks

Pattern	Who calls	Latency
Polling	Client → API	One interval
Webhook	API → client	Near-instant
Streaming	Held open	Continuous

A webhook inverts the direction: your API `POST`s an event to a URL the consumer registered. The receiver must verify the signature — anyone on the internet can post to a public URL — return `2xx` quickly and process asynchronously, and tolerate duplicates and out-of-order arrival, since delivery is at-least-once.

```
POST /hooks/shop HTTP/1.1
X-Signature: sha256=9f86d0818...
X-Event-Id: evt_8891

{"type": "order.paid",
 "data": {"id": 42}}
```

> **Warning:** Deliveries repeat, including for events you already handled. Store the event id and ignore ones you have seen, or a single retried delivery ships the order twice.

API Versioning

Where	Example	Note
URL path	`/v2/orders`	Visible, trivial to route
Header	`API-Version: 2`	Clean URLs, easy to miss
Media type	`…+json;v=2`	Most correct, least used

Version only when you must break the contract: removing a field, tightening validation, changing a type, or changing a status code. Additive changes need no version — provided clients ignore fields they don't recognize, which is a rule you publish on day one. Announce removal in the response, not only in a changelog.

```
GET /v1/orders/42 HTTP/1.1


HTTP/1.1 200 OK
Deprecation: Sat, 01 Nov 2026 00:00:00 GMT
Sunset: Sun, 01 Feb 2027 00:00:00 GMT
Link: </v2/orders/42>; rel="successor-version"
```

> **Gotcha:** Every live version is code you must keep, test, and patch for security. Two supported versions is a policy; five is a maintenance backlog you'll never finish.

OpenAPI

Artifact	What it is
OpenAPI document	The contract, in YAML or JSON
Swagger UI	Renders that document as docs
Generators	Clients, servers, mocks, tests

OpenAPI describes an HTTP API in machine-readable form: paths, methods, parameters, schemas, responses, and security schemes. One document then drives documentation, SDKs, request validation, mock servers, and contract tests. Write it by hand or generate it from code — but check it in CI, because a spec that has drifted from the implementation is worse than no spec at all.

```yaml
paths:
  /orders/{id}:
    get:
      parameters:
        - name: id
          in: path
          required: true
```

```
        schema: { type: string }
    responses:
      '200': { description: An order }
      '404': { description: Not found }
```

> **Note:** OpenAPI 3.1 aligns with JSON Schema 2020-12, so its schemas work in ordinary validators. 3.0's dialect looks the same but is not — `nullable` and `exclusiveMinimum` differ.

REST vs GraphQL

Aspect	REST	GraphQL
Surface	Many URLs	One endpoint
Fields	Server decides	Client asks
Caching	HTTP, for free	App-level, built
Limits	Per request	Per query cost

REST models resources and reuses HTTP itself: methods, status codes, caches, and CDNs all work without extra code. GraphQL exposes one `POST` endpoint plus a type system, letting a client fetch exactly the fields it needs in one round trip — which is worth a lot for complex, client-driven UIs. The cost is that caching, rate limiting, and error semantics move into your application layer.

```
REST:    GET /orders/42?include=customer
```

```
GraphQL: POST /graphql
{ order(id: 42) {
    total
    customer { name }
} }
```

> **Gotcha:** GraphQL returns `200 OK` with an `errors` array even when the query failed. Status-code-based dashboards will show a perfectly healthy API while every request is failing.

API Gateway

Concern	Handled at the gateway
Identity	Token check, key lookup
Traffic	Rate limits, throttling
Routing	Path → service, versions
Insight	Logs, metrics, tracing

A gateway is the single front door: one place to terminate TLS, authenticate, throttle, route, and observe, so no service has to reimplement any of it. It also decouples the public contract from internal topology — you can split a service in two without changing a single URL. Keep business logic out of it; rules that live in the gateway become a shared bottleneck nobody dares to edit.

```yaml
routes:
  - path: /v1/orders/*
    upstream: http://orders.svc:8080
    auth: jwt
    rate_limit: 100/min
    timeout: 3s
```

> **Warning:** The gateway is also a single point of failure and of latency. Set its timeouts below the client's, or one slow upstream stacks connections until the whole front door stops answering.

Microservices

Property	Monolith	Microservices
Deploy	One unit	Independent
Failure	Process-wide	Partial
Call	Function call	Network call
Data	Shared schema	Owned per service

Microservices split a system into independently deployable services, each owning its own data. The payoff is team and release autonomy; the price is that every former function call is now a network call that can be slow, duplicated, or lost. That is why timeouts, bounded retries, and circuit breakers are baseline requirements — and why sharing one database silently re-couples services you meant to split.

```
orders_client:
  timeout: 2s
  retries: 2   # only if idempotent
  backoff: exponential+jitter
  circuit_breaker:
    error_rate: 50%
    open_for: 30s
```

> **Note:** There is no distributed transaction across services. A write spanning two of them needs a saga: local commits plus explicit compensating actions when a later step fails.

Error Handling

Signal	Lives in	Read by
Status code	Start line	Clients, proxies, caches
`type`	Body	Client branching logic
`detail`	Body	A human debugging
Trace id	Body or header	Your support and logs

An error response is part of the contract, so make it as predictable as a success. RFC 9457 problem details standardizes the body around `type`, `title`, `status`, `detail`, and `instance` — return a stable machine-readable `type` the client can branch on, plus a trace id so a bug report maps to one log line. Never leak stack traces, SQL, or internal hostnames; they help attackers more than callers.

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{"type": "https://api.shop.com/errors/qty",
 "title": "Invalid quantity",
 "status": 422,
 "detail": "qty must be 1-99",
 "instance": "/orders",
 "traceId": "b7ad6b71"}
```

> **Gotcha:** Translating `title` server-side breaks every client that matched on the old string. Branch on `type` or a stable `code`; treat all human-readable text as display-only.

Further reading

- RFC 9110 — HTTP Semantics
- RFC 9111 — HTTP Caching
- RFC 6749 — OAuth 2.0
- RFC 9457 — Problem Details
- OpenAPI Specification 3.1
- MDN — HTTP caching