

API Design

Aayush Bajaj

July 10, 2026

Contents

1 rest, actually	1
1.1 resources and verbs	1
1.2 the status codes that matter	2
1.3 url conventions	2
2 pagination	3
3 versioning strategies, honestly	3
4 error design	3
5 auth	4
6 rate limiting	4
6.1 token bucket	4
6.2 the request's whole life	6
7 contract-first with openapi	6
8 when not rest	6
9 see also	7

an api is a promise you cannot take back: the moment a second team depends on it, every quirk you shipped is load-bearing. this page is the checklist for making promises deliberately — resources and verbs, status codes, pagination, versioning, errors, auth, and rate limiting, with the http machinery treated as the application-layer protocol it is (Kurose, James F. and Ross, Keith W., 2020).

hyrum's law: with enough users, every observable behaviour of your system will be depended on by somebody

1 rest, actually

1.1 resources and verbs

rest's one big idea: model the domain as **nouns** (resources with urls) and reuse http's small, fixed verb set — rather than inventing an endpoint per action.

richardson's 'maturity model' ranks apis: level 0 rpc-over-post, level 1 resources, level 2 verbs + status codes, level 3 hypermedia. level 2 is where the industry actually lives.

method	safe	idempotent	meaning
GET	yes	yes	read; never mutate on a GET
HEAD	yes	yes	GET without the body (cheap existence checks)
PUT	no	yes	replace whole resource at this url
DELETE	no	yes	remove; second call may 404, state is the same
POST	no	no	create under a collection / trigger an action
PATCH	no	no	partial update (no idempotency guarantee)

- **safe** = no state change (caches and crawlers rely on this); **idempotent** = $f(f(x)) = f(x)$ — repeating the call leaves the same end state, which is what makes retries legal. clients and proxies *will* retry idempotent methods; if your GET mutates, a prefetcher will mutate it for you.
- POST is the escape hatch. for create-with-retry semantics, accept an **Idempotency-Key** header and dedupe server-side — the same trick as **event-driven** consumers.

1.2 the status codes that matter

return the most specific code you can justify; clients branch on them.

code	meaning	notes
200	ok	body contains the result
201	created	set Location: header to the new resource
202	accepted	async: work queued, poll the job url
204	no content	success with nothing to say (DELETE, some PUTs)
304	not modified	conditional GET hit (ETag / If-None-Match)
400	bad request	malformed syntax, failed validation
401	unauthenticated	misnamed “unauthorized”: who are you?
403	forbidden	i know who you are; the answer is no
404	not found	also the polite mask for 403 when existence is secret
409	conflict	version clash, duplicate create
422	unprocessable content	syntactically fine, semantically wrong
429	too many requests	rate limited; send Retry-After
500	internal server error	your bug; log it, never leak the stack trace
503	service unavailable	overload/maintenance; also deserves Retry-After

two habits worth the pedantry: distinguish 401 (authenticate) from 403 (authenticated but denied), and never return 200 with `{"error": ...}` in the body — it defeats every generic client, cache, and monitor between you and the caller.

1.3 url conventions

- plural nouns for collections, ids for members: `/orders`, `/orders/42`, `/orders/42/items`.
- no verbs in paths — the verb is the method. `POST /orders/42/cancel` is the pragmatic exception when “cancel” is a genuine domain action that PATCH can’t express cleanly; model it as creating a *cancellation* resource if you want to stay pure.
- lowercase, hyphens not underscores, no trailing slash, no file extensions; filtering/sorting/searching live in the query string: `/orders?status=open&sort=-created_at`.
- nesting: one level deep is navigation, two is a smell — `/customers/7/orders/42/items/3` couples the url to a hierarchy that will change; give items their own top-level identity.

2 pagination

never return an unbounded collection; the question is only which cursor you hand back.

- **offset**: `?limit=50&offset=100`. trivial to implement (**LIMIT/OFFSET**), supports “jump to page 7”. two real problems: the database must still *count and skip* 100 rows (cost grows linearly, so deep pages get slow), and concurrent inserts/deletes shift the window — page 3 shows rows you already saw on page 2, or silently skips some.
- **cursor (keyset)**: `?limit=50&after=eyJpZCI6OTgxfQ` — an opaque token encoding the last-seen sort key; the query becomes `WHERE (created_at, id) < (...)` `ORDER BY created_at DESC, id DESC LIMIT 50`, an index seek at constant cost, stable under writes. the price: no random access, and the sort order is baked into the token.
- **default**: cursor for anything user-facing or large; offset is fine for small admin tables. either way return the next cursor *in the response* (and treat it as opaque — clients that parse your cursor format have just extended your public api).

3 versioning strategies, honestly

strategy	example	for	against
uri path	<code>/v2/orders</code>	visible, curl-able	one bump forks every resource
custom header	<code>Api-Version: 2024-06-01</code>	urls stay clean	invisible in logs, forgettable
media type	<code>Accept: ...vnd.co.v2+json</code>	the “correct” answer	tooling makes it unpleasant
dated versions	stripe-style pinning	gradual migration	a translator per date, forever

the honest advice:

- versioning is a defeat you plan for, not a feature. spend the effort on **additive evolution** first — new optional fields and endpoints break nobody, and clients must be written to ignore unknown fields (the same tolerant-reader rule as **event schema evolution**).
- when you must break: `/v2` in the path is the least-clever option and therefore the best default — visible, testable, obvious in every access log.
- version bumps are migrations, not releases: you will run v1 and v2 side by side for years; deprecation needs headers (**Deprecation, Sunset**), metrics on v1 traffic, and emails to the last three laggards. budget accordingly.

4 error design

errors are part of the contract — clients write code against them.

- use **problem+json** (rfc 9457, formerly 7807): a standard error envelope with media type `application/problem+json`:

```
1 {
2   "type": "https://api.example.com/errors/insufficient-funds",
3   "title": "insufficient funds",
4   "status": 422,
5   "detail": "balance is 30.00, transfer wanted 120.50",
6   "instance": "/transfers/7f3c",
7   "balance": 30.00
```

- **type** is a stable identifier clients can branch on (the url need not resolve, but it's polite if it documents the error); **detail** is for humans; extra members (like **balance**) are allowed and encouraged.
- return validation failures *in bulk* — an **errors** array with one entry per field — not one 400 per round trip.
- never leak internals: stack traces, sql fragments and hostnames in error bodies are a gift to attackers and a lawsuit to you.

5 auth

- **api keys**: a static bearer secret identifying a *machine* caller. fine for server-to-server integrations and internal tooling. notes: transmit only over tls in a header (never the query string — urls end up in logs and browser history), store hashed like passwords, support rotation with overlapping validity, and scope each key to the minimum surface.
- **oauth2**: a delegation protocol — the user grants your app limited access to their resources without sharing a password. the flow that matters today is **authorization code** + **pkce** (browser redirects to the provider, app exchanges a one-time code for tokens; pkce binds the code to the initiating client so an intercepted code is useless). **client credentials** covers machine-to-machine; the old **implicit** and **password** flows are deprecated — do not build new systems on them.
- **jwt**: a token *format*, not a protocol — a signed (not encrypted) json payload of claims. the win: stateless verification, any service can check the signature locally without a session-store round trip. the costs: revocation is hard (tokens are valid until **exp**, so keep lifetimes short — minutes — and pair with refresh tokens), clock skew matters, and the payload is readable by anyone who holds it. security notes: verify **alg** against an allowlist (the **alg: none** and rs256-to-hs256 confusion attacks are real and historic), validate **iss**, **aud** and **exp** always, and never put secrets in claims.

6 rate limiting

protects you from abuse, bugs, and enthusiastic customers — and protects tenants from each other.

6.1 token bucket

the standard algorithm because it permits *bursts* while capping the *average*: a bucket holds at most b tokens, refilled at r tokens per second; each request spends one. sustained throughput converges to r , and a client that has been quiet can burst b requests at once. over any window of length t the bucket admits at most $b + rt$ requests.

the leaky bucket is the strict cousin: constant outflow, no bursts. token bucket is almost always what an api wants.

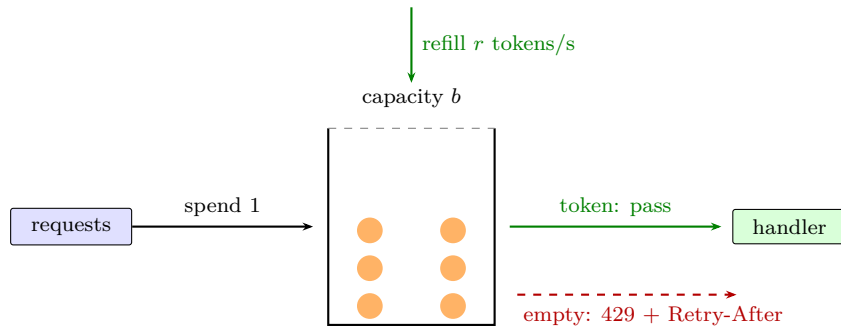


Figure 1: token bucket: refill at rate r up to capacity b ; each admitted request spends a token, arrivals finding the bucket empty get 429.

lazy-refill implementation (no background thread — tokens are computed from elapsed time on each call), driven with a fake clock at 2 tokens/s, capacity 5:

```

1 class TokenBucket:
2     """capacity b tokens, refill rate r tokens/second, lazily refilled."""
3     def __init__(self, rate, capacity, clock):
4         self.rate, self.capacity, self.clock = rate, capacity, clock
5         self.tokens, self.last = capacity, clock()
6
7     def allow(self, cost=1.0):
8         now = self.clock()
9         self.tokens = min(self.capacity, self.tokens + (now - self.last) *
10             ↪ self.rate)
11         self.last = now
12         if self.tokens >= cost:
13             self.tokens -= cost
14             return True
15         return False
16
17 # fake clock: 2 tokens/s refill, burst capacity 5
18 now = [0.0]
19 tb = TokenBucket(rate=2.0, capacity=5, clock=lambda: now[0])
20
21 # a burst of 7 requests at t=0, then one request every 0.25 s
22 times = [0.0] * 7 + [1.0 + 0.25 * i for i in range(8)]
23 for t in times:
24     now[0] = t
25     verdict = "200" if tb.allow() else "429"
26     print(f"t={t:5.2f}s  tokens_left={tb.tokens:4.2f}  -> {verdict}")

```

```

t= 0.00s  tokens_left=4.00  -> 200
t= 0.00s  tokens_left=3.00  -> 200
t= 0.00s  tokens_left=2.00  -> 200
t= 0.00s  tokens_left=1.00  -> 200
t= 0.00s  tokens_left=0.00  -> 200
t= 0.00s  tokens_left=0.00  -> 429
t= 0.00s  tokens_left=0.00  -> 429
t= 1.00s  tokens_left=1.00  -> 200
t= 1.25s  tokens_left=0.50  -> 200
t= 1.50s  tokens_left=0.00  -> 200
t= 1.75s  tokens_left=0.50  -> 429
t= 2.00s  tokens_left=0.00  -> 200
t= 2.25s  tokens_left=0.50  -> 429
t= 2.50s  tokens_left=0.00  -> 200
t= 2.75s  tokens_left=0.50  -> 429

```

read the trace: the cold bucket absorbs a 5-request burst, rejects the 6th and 7th; afterwards the client offers 4 requests/s against a 2/s refill and gets exactly every other one — the long-run rate is r , as designed. production notes: key buckets per api-key or per user *and* per ip, keep them in redis (one **EVAL** for atomicity), and always send **429** with **Retry-After** plus the draft **RateLimit-*** headers so well-behaved clients back off — see the retry-with-jitter discussion in [microservices](#) for the client side.

6.2 the request's whole life

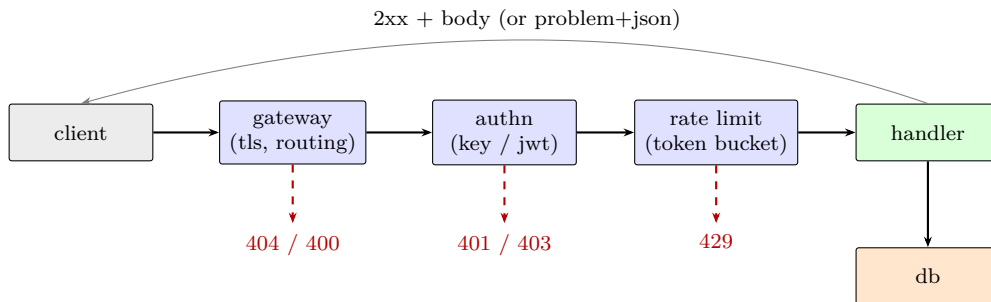


Figure 2: lifecycle of an api request: each gate can end the conversation early with a specific status code; only survivors reach the handler.

7 contract-first with openapi

- write the openapi (swagger) document *before* the handlers: the spec is the contract, reviewed like code, and everything else is generated from it — server stubs, typed clients, docs, request validators, and contract tests in ci.
- the alternative (annotations generating the spec from code) drifts exactly when you are busiest; a handwritten spec that ci enforces cannot drift silently.
- the same discipline that [testing](#) calls design-by-contract, applied at the network boundary.

8 when not rest

- **grpc**: internal service-to-service where you own both ends — typed protobuf contracts, http/2 streaming, an order of magnitude less serialisation overhead than json. wrong for public browser-facing apis (tooling, debuggability, proxies).
- **graphql**: many heterogeneous clients composing views over a rich graph (the classic: mobile apps that can't afford six round trips). you trade the $n+1$ problem, per-query cost analysis and cache complexity for client-shaped responses. wrong as a default for simple resource crud — you inherit its whole complexity budget on day one.
- **webhooks**: the api calling *you* — reversed direction, push not poll. design notes: sign every delivery (hmac over body + timestamp header; receivers must verify both, the timestamp kills replay), deliver at-least-once with exponential backoff so receivers must be idempotent, and give subscribers a dead-letter view of failed deliveries. the event-carried counterpart of everything in [event driven](#).

9 see also

- [microservices](#) — gateways, retries and the resilience machinery around these contracts
- [event driven](#) — the asynchronous complement: events, schemas, idempotency
- [internet networks](#) — tcp, tls and http from the wire up
- [design principles](#) — interface segregation and least knowledge, at network scale

references

Kurose, James F. and Ross, Keith W. (2020). *Computer Networking: A Top-Down Approach*, Pearson.