FREE · LEARN.AIWITHPRIYABRAT.COM

The REST API Design Cheat Sheet

Design APIs that are easy to use and hard to break: naming, methods, status codes, errors, pagination, versioning, idempotency, security, APIs for LLM features, and three APIs designed end to end.

By **Priyabrat Dalbehera** · GenAI engineer

6 pages | 7 parts | 57 rules, patterns & checklists | 3 worked API designs

What's inside

1 **Resources & methods** p. 2
naming, HTTP methods, CRUD, relationships, filters, JSON bodies

2 **Status codes & errors** p. 2
14 status codes, one error format, 401 vs 403

3 **Robust APIs** p. 3
pagination, versioning, idempotency keys, ETags, long jobs, rate limits

4 **Security & performance** p. 3
auth options, JWT pitfalls, CORS, security checklist

5 **AI APIs & interview** p. 4
streaming, webhooks, a RAG API, REST vs GraphQL vs gRPC, Q&A

6 **Key points in depth** p. 5
the 10 rules, anti-patterns, breaking changes, rate-limit algorithms, uploads

7 **Interview deep dive** p. 6
design recipe, URL shortener, payments, state machines, caching, API keys

How to read it
Each box is one design decision: the rule, an example request or response, and why.

Tip & Watch out
Green boxes are good habits. Amber boxes are mistakes that cause bugs or breaches.

Any language
The rules apply to FastAPI, Express, Spring or Go; code examples use Python.

learn.aiwithpriyabrat.com

© 2026 Priyabrat Dalbehera · Free to download and share unchanged, with credit · Not for resale

REST API Design Cheat Sheet

Parts 1–2 · naming resources · methods · CRUD · filters · bodies · status codes · errors

1.1 Resources, not actions

Avoid	Prefer
`GET /getOrders`	`GET /orders`
`POST /createOrder`	`POST /orders`
`GET /order/42`	`GET /orders/42` (plural, always)
`GET /Orders_List`	`GET /orders` (lowercase, hyphens: `/order-items`)
`POST /orders/42/cancelOrder`	`POST /orders/42/cancel` (a verb only for real actions)

1.2 HTTP methods

Method	Use for	Safe	Idempotent
`GET`	read	yes	yes
`POST`	create, or run an action	no	no
`PUT`	replace the whole resource	no	yes
`PATCH`	change some fields	no	not always
`DELETE`	remove	no	yes

Idempotent: sending it twice has the same effect as once, so clients can retry safely.

1.3 CRUD for one resource

```
GET    /orders           list (filter, sort, page)
POST   /orders           create         → 201 + Location
GET    /orders/42        read one       → 200 or 404
PATCH  /orders/42        update fields  → 200
DELETE /orders/42        delete         → 204
```

1.4 Relationships

```
GET /customers/7/orders        orders of one customer
GET /orders?customer_id=7      same, as a filter
GET /orders/42/items           items inside an order
```

Tip: nest one level at most. Deeper paths (`/a/1/b/2/c/3`) are hard to use; use filters instead.

1.5 Filter, sort, search, choose fields

```
GET /orders?status=paid&created_after=2026-10-01
GET /orders?sort=-created_at,total     (- = desc)
GET /products?q=keyboard               search
GET /orders?fields=id,total,status     sparse fields
```

1.6 Request and response bodies

- JSON with **one naming style** everywhere (`snake_case` or `camelCase`).
- Dates in ISO 8601 with time zone: `2026-10-15T09:30:00Z` .
- Money as integer minor units or a decimal string, never a float: `{"amount": 14900, "currency": "INR"}` .
- IDs as strings if they may not fit in JavaScript numbers.
- Return the created or updated resource in the response.

1.7 A complete request and response

```
POST /v1/orders HTTP/1.1
Authorization: Bearer eyJhbGciOi…
Content-Type: application/json
Idempotency-Key: 7c1e3b2a-…

{"product_id": "prd_9", "quantity": 2}

HTTP/1.1 201 Created
Location: /v1/orders/ord_982

{"id": "ord_982", "status": "pending",
 "total": {"amount": 29800, "currency": "INR"},
 "created_at": "2026-10-15T09:30:00Z"}
```

2.1 Status codes to know

200 OK	success with a body
201 Created	new resource; send `Location`
202 Accepted	started, finishing later (3.5)
204 No Content	success, no body (DELETE)
304 Not Modified	cached copy is still valid (3.4)
400 Bad Request	malformed request
401 Unauthorized	not signed in / bad token
403 Forbidden	signed in, not allowed
404 Not Found	no such resource
409 Conflict	clashes with current state (duplicate, version)
422 Unprocessable	valid JSON, invalid values
429 Too Many Requests	rate limited; send `Retry-After`
500 Internal Error	a bug on the server
502 / 503 / 504	upstream failed / overloaded / timed out

2.2 One error format for every endpoint

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

{
  "type": "https://api.example.com/errors/validation",
  "title": "Invalid request",
  "status": 422,
  "detail": "2 fields are invalid",
  "errors": [
    {"field": "email", "message": "not a valid email"},
    {"field": "quantity", "message": "must be 1–100"}
  ],
  "request_id": "req_8f2a91"
}
```

Based on RFC 9457 (Problem Details). The `request_id` lets support find the log line.

2.3 Errors done right

- Use the right status code; never `200` with `{"error": …}` .
- Messages for humans, codes for machines.
- No stack traces, SQL or internal hostnames in responses.
- `404` instead of `403` when even revealing that a resource exists is a leak.
- Map exceptions in one place: NotFound → 404, PermissionDenied → 403, AlreadyExists → 409, anything unexpected → 500 with a logged request ID.

2.4 401 or 403?

No token, expired token, bad signature	**401**
Valid user, but not their order	**403** (or 404)
Valid user, plan doesn't include the feature	**403**

2.5 Validation for free with Pydantic + FastAPI

```python
from pydantic import BaseModel, Field

class OrderIn(BaseModel):
    product_id: str = Field(pattern=r"^prd_\w+$")
    quantity: int = Field(ge=1, le=100)

@app.post("/v1/orders", status_code=201)
def create_order(order: OrderIn): ...
# a bad body gets a 422 with every invalid field
```

learn.aiwithpriyabrat.com

3.1 Pagination: offset or cursor?

Type	Good	Bad
Offset `?page=3&limit=20`	simple, jump to any page	slow on big tables; rows shift when data changes
Cursor `?after=ord_982&limit=20`	fast, stable for feeds and syncs	no jumping to page 50

```
{
  "data": [ … 20 orders … ],
  "next_cursor": "ord_962",
  "has_more": true
}
```

3.2 Versioning

URL `/v1/orders`	most common, visible, easy to route
Header `API-Version: 2026-10-01`	clean URLs; dated versions (Stripe style)

- Adding fields or endpoints is **not** breaking; removing or renaming is.
- Announce deprecations with a date; send a `Deprecation` header.

3.3 Safe retries: idempotency keys

```
POST /payments
Idempotency-Key: 7c1e3b2a-…   (a UUID from the client)
```

The server stores the key with the result. A retry with the same key returns the **same** result instead of charging twice. Essential for payments and anything non-idempotent.

3.4 Caching and concurrent edits

```
GET /products/9          → 200, ETag: "v7"
GET /products/9
If-None-Match: "v7"      → 304 Not Modified

PATCH /products/9
If-Match: "v7"           → 412 if someone changed it
```

3.5 Long-running work

```
POST /reports            → 202 Accepted
                           Location: /jobs/51
GET  /jobs/51            → {"status": "running"}
GET  /jobs/51            → {"status": "done",
                            "result_url": "/reports/88"}
```

Tip: or let the client register a **webhook** and call it when the job finishes.

3.6 Rate limits

```
HTTP/1.1 429 Too Many Requests
Retry-After: 30
RateLimit-Limit: 100
RateLimit-Remaining: 0
```

Limit per API key or user, not only per IP. Clients should back off and retry after the given time.

3.7 Clients: timeouts and retries

- Always set a **timeout** on outgoing calls.
- Retry only on timeouts, 429, 502, 503 and 504, and only idempotent requests (or with a key).
- Exponential backoff **with jitter**: ~1 s, 2 s, 4 s ± random, then give up.
- Respect `Retry-After`.
- A circuit breaker stops calling a service that keeps failing.

4.1 Authentication options

API key	server-to-server, simple; send in a header, never in the URL
OAuth 2 + OIDC	users sign in with Google/GitHub; access tokens with scopes
JWT	self-contained signed token; short expiry + refresh token
Session cookie	browser apps on the same site; `HttpOnly`, `Secure`, `SameSite`

4.2 Security checklist

- ☐ HTTPS everywhere, HSTS on
- ☐ Check **authorisation on every request**: does this user own order 42?
- ☐ Validate every input: types, ranges, lengths, enums
- ☐ Rate-limit logins and expensive endpoints
- ☐ Never trust IDs or roles from the request body
- ☐ CORS: allow only your own front-end origins
- ☐ Secrets in a secret store, rotated
- ☐ Log who did what, without passwords or tokens

Watch out: broken object-level authorisation (checking login but not ownership) is the #1 API vulnerability (OWASP API Top 10).

4.3 Performance

- Paginate every list; set a maximum `limit`.
- Avoid N+1 queries: load related rows in one query.
- Compress responses (gzip or br).
- Cache reads: `Cache-Control`, ETags, a CDN for public data.
- Move slow work to background jobs (3.5).

4.4 JWT pitfalls

- Verify the signature and pin the algorithm; reject `alg: none`.
- Short expiry (minutes) + refresh tokens that rotate.
- The payload is only encoded, not encrypted: no secrets in it.
- In browsers, keep tokens in `HttpOnly` cookies, not `localStorage`.
- Check `iss`, `aud` and `exp` on every request.

4.5 CORS for your front-end

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://app.example.com"],
    allow_methods=["GET", "POST", "PATCH", "DELETE"],
    allow_headers=["Authorization", "Content-Type"],
    allow_credentials=True,
)
```

Watch out: never combine `allow_origins=["*"]` with credentials. CORS protects browsers, not your API: server-side auth still decides.

learn.aiwithpriyabrat.com

REST API Design Cheat Sheet

Part 5 · APIs for LLM features · streaming · webhooks · REST vs GraphQL vs gRPC · checklist · interview

5.1 APIs for LLM and AI features

Streaming replies	Server-Sent Events: `Content-Type: text/event-stream`
Slow generation	202 + job URL, or stream; don't hold a request for minutes
Cost control	token-based rate limits and budgets per key
Retries	idempotency keys; LLM calls are slow and costly to repeat
Model upgrades	a `model` field, or version the endpoint

5.2 Stream tokens with FastAPI (SSE)

```python
from fastapi import FastAPI
from fastapi.responses import StreamingResponse

app = FastAPI()

@app.post("/v1/chat")
async def chat(req: ChatRequest):
    async def events():
        async for token in llm_stream(req.message):
            yield f"data: {token}\n\n"
        yield "data: [DONE]\n\n"
    return StreamingResponse(
        events(), media_type="text/event-stream")
```

5.3 Webhooks you can trust

- Sign each payload (HMAC-SHA256) and verify the signature before reading it.
- Send an event ID; receivers ignore duplicates.
- Retry with backoff until the receiver returns 2xx.
- Receivers reply fast (200) and do the work in the background.

5.4 REST, GraphQL or gRPC?

	Best for	Why
REST	public APIs, CRUD, caching over HTTP	simple, universal
GraphQL	many front-ends needing different data shapes	one request, no over-fetching
gRPC	service-to-service, high throughput, streaming	fast, typed, not for browsers

5.5 Document it: OpenAPI

- FastAPI generates an OpenAPI spec and docs at `/docs` from your Pydantic models.
- Give every endpoint a summary, examples and its error responses.
- Generate client SDKs and contract tests from the spec.

5.6 Design review checklist

- ☐ Plural nouns, lowercase, hyphens
- ☐ Right method and status code
- ☐ One error format with a request ID
- ☐ Pagination on every list
- ☐ Versioning plan
- ☐ Idempotency keys on POSTs that cost money
- ☐ Auth and object-level permissions
- ☐ Rate limits and timeouts
- ☐ OpenAPI docs with examples

5.7 Quick interview answers

PUT vs PATCH	replace the whole resource vs change some fields
Idempotent?	repeat = same effect: GET, PUT, DELETE yes; POST no
401 vs 403	who are you? vs you can't do that
Offset vs cursor	simple, any page vs fast, stable for large or live data
Avoid double charges	idempotency key stored with the result
Breaking change?	removing/renaming fields or changing types; adding is fine
Stateless	each request carries everything needed; any server can answer

5.8 Worked example: a RAG API

```
POST   /v1/documents          upload → 202 + job URL
GET    /v1/jobs/{id}          parsing / indexing status
GET    /v1/documents?cursor=  list, cursor-paginated
DELETE /v1/documents/{id}     remove from the index
POST   /v1/query              question → streamed answer
                              with cited document IDs
GET    /v1/usage?month=2026-10  tokens and cost per key
```

5.9 Health, readiness and batches

`GET /health`	the process is alive (cheap, no dependencies)
`GET /ready`	database, cache and model reachable: ready for traffic
`POST /orders/batch`	many items in one call; return a result per item

5.10 HTTP headers worth knowing

`Authorization`	`Bearer <token>`
`Content-Type / Accept`	what you send / want back
`Location`	URL of a created resource or a job
`Cache-Control`	who may cache, and for how long
`ETag / If-None-Match`	cache validation (3.4)
`Idempotency-Key`	safe retries (3.3)
`Retry-After`	when to try again after 429 or 503
`X-Request-ID`	trace one request across services and logs

5.11 Test the contract

```python
from fastapi.testclient import TestClient

client = TestClient(app)

def test_create_order():
    r = client.post("/v1/orders",
                    json={"product_id": "prd_9", "quantity": 2})
    assert r.status_code == 201
    assert r.headers["location"].startswith("/v1/orders/")

def test_rejects_bad_quantity():
    r = client.post("/v1/orders",
                    json={"product_id": "prd_9", "quantity": 0})
    assert r.status_code == 422
```

5.12 Keep learning, free

- Notes: FastAPI for GenAI, request and response models
- Docker for AI & Python Developers
- Developer Templates Pack: a FastAPI skeleton ready to copy

learn.aiwithpriyabrat.com

learn.aiwithpriyabrat.com

6.1 The 10 rules that matter most

1. Nouns, not verbs	the method is the verb; URLs name things, so they stay stable as features grow
2. Status codes are the contract	clients branch on the code, not on the message text
3. One error shape	one parser on the client; a request ID in every error
4. Paginate every list	a list that is small today is 2 million rows next year
5. Never break clients	add freely; remove or rename only in a new version
6. Safe retries	networks fail mid-request; idempotency keys stop double charges
7. Check ownership	authenticated ≠ authorised: check every object, every call
8. Validate at the edge	reject bad input with 422 before it reaches the database
9. Be boring and consistent	same naming, dates, money and IDs everywhere: fewer docs to read
10. Design for the caller	write the client code first; awkward calls mean an awkward API

6.2 Anti-patterns and their fixes

Avoid	Do this
`200` with `{"success": false}`	real status codes (2.1)
`GET /deleteUser?id=7`	`DELETE /users/7`; GET must never change data
Returning the whole table	pagination with a max `limit` (3.1)
Leaking DB columns as-is	a response model; hide internal fields
Auto-increment IDs in URLs	random IDs (`ord_8f2a`): not guessable, no count leak
Booleans that grow (`is_paid`, `is_shipped`)	one `status` enum (7.4)
Chatty: 10 calls per screen	include related data: `?expand=customer`
Different date formats	ISO 8601 UTC everywhere

6.3 Breaking or safe?

Add an endpoint, an optional field or param	safe
Add a value to an enum	risky: old clients may crash on it
Make an optional field required	breaking
Remove or rename a field, change its type	breaking
Change a status code or the error shape	breaking
Tighten validation (max 100 → max 50)	breaking

Tip: tell clients to **ignore unknown fields** and handle unknown enum values: then adding is always safe.

6.4 Retiring an endpoint politely

```
Deprecation: true
Sunset: Sat, 01 May 2027 00:00:00 GMT
Link: </v2/orders>; rel="successor-version"
```

- Announce in the changelog and email key users.
- Track who still calls the old version, by API key.
- Give at least 6 months; then return `410 Gone`.

6.5 Rate-limit algorithms

	How	Trade-off
Fixed window	count per minute; simple	bursts at the window edge (2× limit)
Sliding window	weighted current + previous window	smooth, a little more state
Token bucket	tokens refill at a rate; each call takes one	allows short bursts: the usual choice
Leaky bucket	queue drains at a fixed rate	smooth output, adds waiting

With many servers, keep the counters in Redis (`INCR` + `EXPIRE`), not in memory.

6.6 Token bucket in a few lines

```python
import time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate, self.cap = rate, capacity
        self.tokens, self.t = capacity, time.monotonic()

    def allow(self) -> bool:
        now = time.monotonic()
        self.tokens = min(self.cap, self.tokens
                          + (now - self.t) * self.rate)
        self.t = now
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False            # → 429 + Retry-After
```

6.7 File uploads: don't proxy the bytes

```
POST /v1/uploads  {"filename": "cv.pdf",
                   "size": 182044}
→ 201 {"upload_id": "upl_7",
       "url": "https://s3…?X-Amz-Signature=…",
       "expires_in": 900}
PUT  <url>  (client sends the file to storage)
POST /v1/uploads/upl_7/complete  → 202 (scan, parse)
```

A pre-signed URL lets the client upload straight to S3/GCS: your API stays fast and never holds 100 MB in memory. Check size and type on `complete`.

6.8 Batch calls with partial failure

```
POST /v1/orders/batch   → 200
{"results": [
  {"index": 0, "status": 201, "id": "ord_1"},
  {"index": 1, "status": 422,
   "error": {"field": "quantity", …}}
]}
```

Tip: one bad item shouldn't fail the whole batch; cap the batch size (e.g. 100).

6.9 Observe every request

- **Logs**: one structured JSON line per request: method, route, status, ms, user, request ID.
- **Metrics (RED)**: Rate, Errors, Duration (p50/p95/p99) per endpoint.
- **Traces**: pass `traceparent` through every service (OpenTelemetry).
- **Alerts** on error rate and p99 latency, not on single errors.

6.10 What an API gateway handles

At the gateway	TLS, API-key/JWT check, rate limits, routing, CORS, request size, logging
In your service	business rules, ownership checks, validation of meaning

learn.aiwithpriyabrat.com

REST API Design Cheat Sheet

Part 7 · design-an-API recipe · URL shortener · payments · state machines · real-time · caching · API keys

Part 7 · Interview deep dive

7.1 "Design an API for X": the interview recipe

1. Users & use cases	who calls it, the top 3–5 actions, read vs write ratio
2. Resources	the nouns and how they relate
3. Endpoints	method + path + status codes for each use case
4. Payloads	request and response JSON for the main one or two
5. Hard parts	pagination, idempotency, auth, rate limits, async work
6. Evolve	versioning, what you'd add next, how you'd scale it

7.2 Worked: a URL shortener

```
POST   /v1/links   {"url": "https://…",
                    "alias": "sale"?, "expires_at"?}
       → 201 {"code": "sale", "short_url": "…"}
       → 409 alias already taken
GET    /{code}     → 301/302 Location: <long url>
                   → 404 unknown · 410 expired
GET    /v1/links/{code}/stats?from=&to=
DELETE /v1/links/{code}   → 204
```

302 if you want to count every click (301 is cached by browsers). Reads dominate, so cache `code → url` in Redis.

7.3 Worked: a payments API

```
POST /v1/payment_intents
Idempotency-Key: …
{"amount": 49900, "currency": "INR",
 "order_id": "ord_9"}
→ 201 {"id": "pi_3", "status": "requires_payment",
       "client_secret": "…"}
POST /v1/payment_intents/pi_3/confirm  → 200
POST /v1/refunds {"payment": "pi_3"}   → 201
```

- Amounts in paise (integers), never floats.
- The final result arrives by a **signed webhook**, not the browser redirect (users close tabs).
- Never store card numbers: the provider does.

7.4 Status as a state machine

```
pending ──pay──▶ paid ──ship──▶ shipped ──▶ delivered
   │              │
 cancel         refund
   ▼              ▼
cancelled      refunded
```

```
POST /v1/orders/42/ship   (order is 'pending')
→ 409 {"title": "Order must be paid first"}
```

7.5 Real-time: which one?

	How	Use for
Polling	simple; client asks every N s	job status, low traffic
SSE	server → client over HTTP	LLM tokens, live feeds
WebSocket	two-way, long-lived	chat, collaboration, games
Webhook	server → another server	payments, integrations

7.6 Where to cache

Browser	`Cache-Control: private, max-age=60`
CDN	public, same for everyone: catalogue, images
App cache (Redis)	expensive queries, sessions, LLM answers
Database	indexes and read replicas

Watch out: never cache per-user data with `public`: one user's data reaches another.

7.7 API keys done right

- Prefixed and readable: `sk_live_…` / `sk_test_…` (secret scanners spot them).
- Store only a **hash** of the key; show it once at creation.
- Scopes per key (`orders:read`), an owner, a last-used date.
- Allow two active keys so users can rotate without downtime.

7.8 More interview answers

REST vs RPC	resources + standard verbs vs named procedures (`createOrder()`)
POST vs PUT to create	POST when the server picks the ID; PUT when the client knows it
Scale a read-heavy API	cache, CDN, read replicas, pagination, stateless servers behind a load balancer
Slow endpoint?	measure p99, check N+1 queries and indexes, cache, move work to a job
Secure an API	HTTPS, auth, ownership checks, validation, rate limits, logs
Why `/v1` ?	lets old clients keep working while v2 changes the contract
Webhook security	HMAC signature + timestamp to block replays + event ID for duplicates
HTTP/2 benefit	many requests over one connection, compressed headers

7.9 Cursor pagination in SQL

```
-- cursor = (created_at, id) of the last row sent
SELECT id, total, created_at FROM orders
WHERE user_id = :uid
  AND (created_at, id) < (:c_time, :c_id)
ORDER BY created_at DESC, id DESC
LIMIT 21;     -- 20 + 1 tells you has_more
```

Add the `id` as a tie-breaker so rows with the same time are never skipped; index `(user_id, created_at, id)` . Encode the cursor as base64 so clients treat it as opaque.

7.10 Verify a webhook signature

```
import hmac, hashlib, time

def verify(body: bytes, ts: str, sig: str) -> bool:
    if abs(time.time() - int(ts)) > 300:
        return False                  # replayed
    msg = ts.encode() + b"." + body
    good = hmac.new(SECRET, msg,
                    hashlib.sha256).hexdigest()
    return hmac.compare_digest(good, sig)
```

Watch out: verify the **raw** body bytes before parsing JSON, and compare with `compare_digest` , not `==` .

7.11 Key takeaways

- ☐ Predictable beats clever: same patterns on every endpoint
- ☐ Status code + one error format = easy client code
- ☐ Every list paginated, every POST that costs money idempotent
- ☐ Auth on the gateway, ownership in the service
- ☐ Add fields freely; breaking changes need a new version
- ☐ Slow work → 202 + job, or stream it
- ☐ Measure: RED metrics and a request ID in every log

learn.aiwithpriyabrat.com