Pagination

APIs often return large datasets (e.g., all orders, all products, all posts).

If the API dumps everything in one response:

Too slow → takes time to fetch.

Too heavy → wastes bandwidth.

Too costly → server load increases.

Solution → break data into pages (like in a book or search results).

In short: Pagination = splitting large results into smaller, manageable chunks

Why Pagination Matters

Performance: Faster responses, less memory usage.

User experience: You don't overload users with 10,000 results at once.

Cost: Saves network bandwidth and API costs (some APIs charge per call/data size).

Control: Lets client apps choose how much data to fetch.

Pagination in REST

Offset + Limit (Page-based Pagination)

Most common, especially in SQL-based systems.

how it looks (api)

page style: `GET /products?page=3&limit=20`

offset style: `GET /products?limit=20&offset=40`

(page 3 ↔ offset = (3-1)*20 = 40)

db query (sql)

```
SELECT id, name, price
FROM products
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 40;
```

response shape (good practice)

```
{
  "data": [ /* 20 products */ ],
  "pagination": {
    "type": "offset",
    "page": 3,
    "limit": 20,
    "total_items": 1243,
    "total_pages": 63,
    "next": "/products?page=4&limit=20",
    "prev": "/products?page=2&limit=20"
  }
}
```

pros

-> easy to implement & debug
-> supports jump-to-page UIs ("page 17")
-> most ORMs support .limit().offset()
-> great for small / mostly-static lists (admin tables, reports)

cons

-> deep offsets are slow (DB still scans/steps past skipped rows)
-> inconsistent under writes:
inserts/deletes before your offset shift rows → duplicates/missing
-> expensive to return exact total counts on huge tables

Table

created_at	id	name
~~2025-09-27~~	~~11~~	~~product 11~~
~~2025-09-27~~	~~10~~	~~Newest~~
2025-09-26	9	Product 9
2025-09-25	8	Product 8
2025-09-24	7	Product 7
2025-09-23	6	Product 6
2025-09-22	5	Product 5

Best practices:

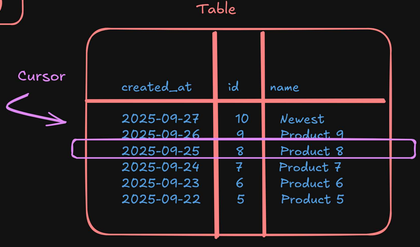
-> cap limit (e.g., max 100)
-> avoid deep offsets (e.g., forbid offset > 10k or switch to cursor after page N)
-> compute has_more via LIMIT (limit+1) (don't COUNT(*) every time)
-> cache total_items or return "estimated_total"

when to use
-> small/medium lists
-> mostly read-only data during a session
-> you need jump-to-page

DEMO

cursor-based pagination (opaque token)

Instead of offset, API returns a cursor (base64 or signed token).
like a bookmark



how it looks (api)

```
GET /products?limit=20&
cursor=eyJjcmVhdGVkQXQiOjE3MTYzMDAwMDAwMCwiaWQiOjUwMH0
```

token contents (server-side)

```
{ "createdAt": 1716300000000, "id": 500,
"filters": { "q": "phone", "brand": "Acme" },
"sort": "createdAt.desc,id.desc" }
```

include sort + filters used to create it
→ reject cursor if client changes them

(good practice)

-> opaque + signed (HMAC) so clients can't forge cursors
-> include expiry in the cursor if data moves fast
-> implement reverse pagination by generating a
previous_cursor and using > vs < with reversed order

pros

consistent with concurrent inserts/deletes

fast (seek via index)

hides internal keys; token can be signed/expiring

you can tuck filters/sort inside for validation

cons

needs encoding/decoding + signing

harder to support jump-to-page

tokens are tied to a particular sort/filter;
must return 400 if they change

response

```
{
  "data": [ /* 20 products */ ],
  "pagination": {
    "type": "cursor",
    "limit": 20,
    "next_cursor": "eyJjcmVhdGVkQXQiOjE3MTYyOTg4MDAwMCwiaWQiOjQ4MH0",
    "has_more": true
  }
}
```

db query (sql)

```
SELECT id, name, price, created_at
FROM products
WHERE (created_at, id) < (:cursor_created_at, :cursor_id)
ORDER BY created_at DESC, id DESC
LIMIT :limit;
```

DEMO

keyset (seek) pagination

paginate by values, not positions. "give me items after this key."

```
GET /products?limit=20&after_id=500&after_ts=1716300000000
```

time-window pagination (temporal slices)

page by time ranges; great for logs/analytics.

```
GET /events?start=2025-09-01T00:00:00Z&end=2025-09-01T01:00:00Z&limit=1000
```