You said:

I am learning to become a strong backend engineer. I want to deeply understand the topic: [### **API Design & Fundamentals** - REST best practices (naming, status codes, error handling) - API versioning strategies - Pagination (offset, cursor-based, keyset pagination) - Filtering, sorting, searching in APIs]. Please explain it in the following way: 1. Concept & Fundamentals – clear definition, why it matters, and real-world examples. 2. Core Subtopics – break down all key concepts, strategies, and best practices under this topic. 3. Step-by-Step Implementation – show how to implement it in code (preferably in JavaScript/TypeScript with Node.js or Next.js context). 4. Common Mistakes & Pitfalls – what beginners usually do wrong and how to avoid it. 5. Scaling & Production Considerations – how this topic works when handling large-scale apps and many users. 6. Comparison – how this differs from related approaches (if applicable). 7. Interview/Practical Questions – questions I should be able to answer about this topic. 8. Mini Project / Exercise – suggest a practical exercise to solidify my learning. Make the explanation detailed but practical, focusing on production-level knowledge, not just theory.

ChatGPT said:

Thought for 6s

Great — this is an excellent topic to master. I'll walk you through **API Design & Fundamentals** with production-focused, practical details and working TypeScript/Node examples you can copy. I'll keep it forward-looking and encouraging — you'll be able to apply these patterns immediately to real systems.

1. Concept & Fundamentals

Definition: An API (Application Programming Interface) is the contract between clients and servers describing how clients may request resources and how the server responds. Good API **design** makes that contract consistent, predictable, and evolvable.

Why it matters

- Predictability → easier client development & fewer bugs.
- Extensibility → you can add features without breaking clients.
- Performance & security → correct patterns avoid bottlenecks and vulnerabilities.
- Maintainability → consistent patterns reduce developer cognitive load.

Real-world examples

- Public web APIs: GitHub REST API (clear resources, rate limits, pagination), Stripe API (versioning, strong backwards compatibility promises).

- Internal microservices: Product / Order / Auth services with well-defined REST endpoints and consistent error formats.

---

2. Core Subtopics (Breakdown & Best Practices)

REST best practices (naming, verbs, status codes, error handling)

- **Resource naming**
    - Use **nouns** (not verbs): /users, /products.
    - Use **plural**: /users/123/orders.
    - Hierarchical: /stores/{storeId}/products.
- **HTTP methods**
    - GET — read
    - POST — create
    - PUT — full replace (idempotent)
    - PATCH — partial update (prefer JSON Patch or JSON Merge Patch semantics)
    - DELETE — delete (idempotent)
- **Status codes**
    - 200 OK — success with body
    - 201 Created — resource created (include Location header)
    - 204 No Content — success, no body (common for deletes)
    - 400 Bad Request — client-side validation / malformed input
    - 401 Unauthorized — missing/invalid credentials
    - 403 Forbidden — authenticated but not allowed
    - 404 Not Found
    - 409 Conflict — conflict (e.g., unique key violation)
    - 422 Unprocessable Entity — semantic validation failure (optional)
    - 429 Too Many Requests — rate limiting
    - 500/502/503 — server errors
- **Error responses**

  - Use a consistent error schema. Prefer RFC 7807 (Problem Details) or a small JSON object:
  - {
  - "type": "https://example.com/probs/out-of-credit",
  - "title": "You do not have enough credit.",
  - "status": 403,
  - "detail": "Your current balance is X",
  - "instance": "/account/123/msgs/abc"
  - }
  - Include machine-readable error code, human-friendly message, and optionally errors[] for field-level errors.

API versioning strategies

- **URI versioning (v1)**: /api/v1/products — very explicit, easy to cache/route. Most common.
- **Header versioning**: Accept: application/vnd.myapp.v2+json — cleaner URIs, content negotiation-friendly.
- **Query param**: /api/products?version=2 — simple but noisy and often abused.
- **No version + backward compatible changes**: evolve carefully with feature flags; use semantic versioning for public SDKs.
- **Strategy:** prefer URI versioning for public APIs; for internal APIs you may use header or evolve without versions if you can guarantee compatibility.

Pagination

- **Offset pagination** (?limit=20&offset=40) — easy but bad for large offsets (slow scans, inconsistent if data mutates).
- **Cursor (keyset) pagination** — use a cursor (opaque token) representing last item (e.g., ?limit=20&cursor=eyJsYXN0SWQiOjEwMH0=). Fast and consistent. Use stable ordering key(s) (created_at + id).
- **Keyset pagination** — same idea as cursor but expressed as SQL WHERE (created_at, id) < ($cursor_created_at, $cursor_id) ORDER BY created_at DESC, id DESC LIMIT $limit.
- Choose cursor/keyset for high-scale systems.

Filtering, sorting, searching

- **Filtering**: GET /products?category=shoes&color=black&min_price=10&max_price=100
    - Validate allowed filters (allow-list).
    - Map query params to SQL WHERE clauses using prepared statements.
- **Sorting**: ?sort=-price,name where - denotes descending.
    - Validate allowed sort fields and enforce deterministic tie-breakers (like id).
- **Searching**
    - Simple substring search: ILIKE or CONTAINS (for small datasets).
    - Full-text search: PostgreSQL tsvector, or external search (Elasticsearch, Typesense, Meilisearch) for complicated queries, relevance ranking, fuzzy search.
- **Security & performance**
    - Avoid SQL injection by parameterized queries and never interpolating user input.
    - Use indexes for filter/sort columns.
    - Prefer controlled query builders, ORMs, or parameterized raw queries.

---

3. Step-by-Step Implementation (TypeScript + Node/Express + PostgreSQL examples)

I'll show:

1. REST routes
2. Validation (zod)
3. Error middleware
4. Cursor pagination implementation
5. Filtering + sorting safely

Note: these are minimal examples — adapt to your stack (Next.js API routes are structurally similar).

A. Setup (abbreviated)

```
npm init -y
npm i express pg zod dotenv
npm i -D typescript @types/express ts-node-dev
```

tsconfig.json basic assumed.

B. Database access (using pg with parameterized queries)

db.ts

```
import { Pool } from "pg";
import dotenv from "dotenv";
dotenv.config();

export const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  // in prod, set pool size, ssl, etc.
});
```

C. Zod schemas (validation)

schemas.ts

```
import { z } from "zod";

export const productQuerySchema = z.object({
  limit: z.string().optional(),        // we'll parse to number
  cursor: z.string().optional(),       // base64 encoded
  sort: z.string().optional(),         // e.g., "-price,name"
  category: z.string().optional(),
  min_price: z.string().optional(),
  max_price: z.string().optional(),
  q: z.string().optional(),            // search query
});
```

D. Cursor helper

cursor.ts

```
export function encodeCursor(obj: object) {
  return Buffer.from(JSON.stringify(obj)).toString("base64");
}
export function decodeCursor(cursor?: string) {
  if (!cursor) return null;
  try {
    return JSON.parse(Buffer.from(cursor, "base64").toString("utf8"));
  } catch {
    return null;
  }
}
```

E. Create the route with keyset (cursor) pagination + filtering + sorting

server.ts

```
import express from "express";
import { pool } from "./db";
import { productQuerySchema } from "./schemas";
import { decodeCursor, encodeCursor } from "./cursor";
import { z } from "zod";

const app = express();
app.use(express.json());

function parseLimit(s?: string, defaultVal=20, maxVal=100) {
  const n = s ? parseInt(s, 10) : defaultVal;
  return Math.min(Math.max(1, isNaN(n) ? defaultVal : n), maxVal);
}
```

```
app.get("/api/v1/products", async (req, res, next) => {
  try {
    const parsed = productQuerySchema.parse(req.query);
    const limit = parseLimit(parsed.limit);
    const cursor = decodeCursor(parsed.cursor);
    // Allowed sorts
    const ALLOWED_SORTS = new Set(["price", "created_at", "name"]);
    const sortFields = (parsed.sort || "-created_at").split(",").map(s => s.trim()).filter(Boolean);

    // Build ORDER BY and WHERE conditions safely
    // For keyset, we enforce ordering to be created_at DESC, id DESC as primary if not requested.
    // For demo: we'll only support single sort field + deterministic tie break.
    let orderBy = "created_at DESC, id DESC";
    let whereClauses: string[] = [];
    const params: any[] = [];

    if (parsed.category) {
      params.push(parsed.category);
      whereClauses.push(` category = $${params.length}` );
    }
    if (parsed.min_price) {
      params.push(parseFloat(parsed.min_price));
      whereClauses.push(` price >= $${params.length}` );
    }
    if (parsed.max_price) {
      params.push(parseFloat(parsed.max_price));
      whereClauses.push(` price <= $${params.length}` );
    }
    if (parsed.q) {
      params.push(`%${parsed.q}%`);
      whereClauses.push(`(name ILIKE $${params.length} OR description ILIKE $${params.length})`);
    }

    // Cursor-based keyset condition
    if (cursor && cursor.created_at && cursor.id) {
      // For created_at DESC ordering:
      params.push(cursor.created_at);
      params.push(cursor.id);
      whereClauses.push(`(created_at, id) < ($${params.length - 1}::timestamptz, $${params.length}::bigint)`);
    }

    const whereSQL = whereClauses.length ? "WHERE " + whereClauses.join(" AND ") : "";

    // Fetch limit + 1 to know if there's a next page
    params.push(limit + 1);
    const sql = `
      SELECT id, name, price, category, created_at
      FROM products
      ${whereSQL}
      ORDER BY ${orderBy}
      LIMIT $${params.length}
    `;

    const { rows } = await pool.query(sql, params);
    let nextCursor = null;
    if (rows.length > limit) {
      const last = rows[limit - 1];
      nextCursor = encodeCursor({ created_at: last.created_at.toISOString(), id: last.id });
      rows.splice(limit); // trim the extra row
    }

    res.json({ data: rows, nextCursor });
   } catch (err) {
    next(err);
   }
  });

  // Basic error middleware
  app.use((err: any, req: any, res: any, next: any) => {
   console.error(err);
   if (err instanceof z.ZodError) {
    return res.status(400).json({
      type: "https://example.com/probs/validation",
      title: "Invalid request",
      status: 400,
      detail: "Validation failed",
      errors: err.errors,
    });
   }
   res.status(500).json({
    type: "about:blank",
    title: "Internal server error",
    status: 500,
    detail: "Something went wrong",
  });
});

app.listen(3000, () => console.log("Listening on 3000"));
```

Notes on the implementation

- We used parameterized queries ($1, $2) to avoid SQL injection.
- Cursor encodes {created_at, id} as base64; on the server we decode and use as keyset condition.
- We fetch limit + 1 to determine if there’s more data.
- Sort fields and filters are allow-listed. Never allow arbitrary SQL from client input.

4. Common Mistakes & Pitfalls (and how to avoid them)

1. **Using OFFSET for very large datasets**
    - Problem: OFFSET 100000 is expensive and slow.
    - Fix: Use cursor/keyset pagination and appropriate indexes.
2. **Not validating user input**
    - Problem: SQL injection, crashes from invalid types.
    - Fix: Use zod/Joi, parameterized queries, typed request parsing.
3. **Inconsistent resource naming**
    - Problem: Clients confused by /user & /users endpoints.
    - Fix: Define style guide and follow it.
4. **Unclear or inconsistent error formats**
    - Problem: Clients have to handle many error shapes.
    - Fix: Use a consistent error schema (RFC7807 or similar).
5. **Breaking changes without versioning**

  - Problem: Clients break when responses change.
  - Fix: Use semantic versioning and API versioning strategy.

6. **Allowing arbitrary sorts/filters**
   - Problem: Open to slow queries and SQL injection.
   - Fix: Allow-list fields and validate values.
7. **Relying on timestamps alone for keyset pagination**
   - Problem: Duplicate timestamps cause skipped/duplicated items.
   - Fix: Use composite tie-breaker (timestamp + id).
8. **Returning internal error messages**
   - Problem: Leaks internal implementation details.
   - Fix: Log internally; return sanitized error messages to clients.

---

5. Scaling & Production Considerations

DB & Query optimization

- **Indexes**: create indexes for columns used in WHERE and ORDER BY (e.g., (created_at DESC, id DESC)).
- **Read replicas**: use read replicas for read-heavy endpoints.
- **Partitioning**: partition large tables (by date, tenant).
- **Connection pooling**: use a pool (pg Pool, PgBouncer) and tune pool sizes to avoid overload.

Caching & CDN

- **HTTP caching**: Use Cache-Control, ETag, conditional requests for resources that are cacheable.
- **Reverse proxies**: Varnish or CDN (Fastly, CloudFront).
- **Application cache**: Redis for frequently accessed data and for storing cursor state or precomputed search results.

Rate limiting & throttling

- Protect critical endpoints using rate limits (per IP, per user).

- Use token buckets or fixed window algorithms (libraries or API gateways like Kong, Envoy).

Observability

- **Metrics**: request latency, error rate, DB query times.
- **Distributed tracing**: OpenTelemetry (trace requests across services).
- **Logging**: structured logs with request IDs. Don’t log sensitive PII.

Fault tolerance

- **Circuit breakers** for downstream services.
- **Bulkheads** to limit failures to parts of the system.
- **Retries** with backoff for transient failures, but be careful to make idempotent operations.

Security

- **Auth & Authorization**: JWT, OAuth2, API Keys; ensure scopes & roles.
- **Input sanitation & validation**
- **CORS** configuration and least-privilege APIs.
- **HTTPS everywhere**, HSTS.

---

6. Comparison — REST vs Other Approaches

REST vs GraphQL

- REST:
    - Strength: simple caching, HTTP semantics, easy logging & monitoring.
    - Weakness: over-fetching/under-fetching possible, multiple round trips.
- GraphQL:
    - Strength: client can request exactly what it needs, single endpoint for complex UIs.
    - Weakness: harder caching, query complexity needs control, learning curve.
- When to choose:
    - Public, resource-centric APIs or simple services → REST.

    - UIs needing flexible queries and aggregation across resources → GraphQL (with persisted queries & complexity limits).

REST vs gRPC

- gRPC:
    - Best for internal, high-performance microservice communication; uses proto contracts, streaming, binary transport.
- REST:
    - Better for public APIs or browser clients (HTTP/JSON).

---

7. Interview / Practical Questions You Should Be Able To Answer

- What are the differences between offset and cursor pagination? When to use each?
- How do you design an API to be backward-compatible? Give examples of non-breaking and breaking changes.
- Explain idempotency. Which HTTP methods are idempotent? How do you implement idempotent POST?
- How would you implement cursor-based pagination? Show the SQL condition for keyset pagination.
- How do you safely allow filtering and sorting? How do you prevent abuse?
- When would you choose GraphQL over REST? Reverse?
- How would you add rate limiting to an API? Where would you enforce it?
- Explain how to design search functionality: DB ILIKE vs full-text vs Elasticsearch.
- How do you version an API? Pros/cons of URI vs header versioning.
- How do you handle long-running requests and asynchronous operations? (webhooks, background jobs)
- How do you scale read-heavy endpoints? (caching, read replicas, query optimization)

---

8. Mini Project / Exercise (Practical)

Build: Product Catalog API — a small production-ready API that supports:

- CRUD for products
- Cursor-based pagination for GET /products
- Filtering by category, price range
- Sorting by price or created_at
- Text search on name and description
- Field-level validation and consistent error responses
- Unit tests + Postman collection

Requirements & steps

1. Use TypeScript + Express + PostgreSQL (or Prisma if you prefer).
2. Schema: products(id pk, name, description, price numeric, category text, created_at timestamptz default now()).
3. Implement:
    - GET /api/v1/products?limit=20&cursor=...&category=...&min_price=..&sort=-price
    - POST /api/v1/products (validation)
    - PATCH /api/v1/products/:id (partial)
    - DELETE /api/v1/products/:id
4. Add:
    - DB indexes: CREATE INDEX ON products (created_at DESC, id DESC), indexes on category, price.
    - A small Redis cache for first page responses.
    - Rate limiting middleware (e.g., in-memory for test or Redis-backed).
    - OpenAPI spec for your endpoints.
5. Tests:
    - Pagination correctness (no duplicates/omissions across pages).
    - Filtering & sorting correctness.
6. Bonus:
    - Add full-text search with PostgreSQL tsvector and GIN index.

    - Add a small front-end simple UI or Postman collection to demo.

Quick Checklist — Best Practices You Should Internalize

- Use nouns & plural names for endpoints.
- Use proper HTTP verbs and appropriate status codes.
- Validate inputs and return consistent error structures (RFC7807).
- Use cursor/keyset pagination for large datasets.
- Allow-list filters/sorts and parameterize DB queries.
- Index columns used for filtering/sorting.
- Use versioning strategy and maintain backwards compatibility.
- Add caching, rate limits, metrics, tracing for production.
- Avoid leaking internal errors or data.

If you want, I can:

- Draft an OpenAPI (Swagger) spec for the Product Catalog API above.
- Provide a complete GitHub-ready starter repo (TS + Express + DB + tests).
- Convert the example to Next.js API Routes or Prisma-based code.

Which of those would help you cement this most? I recommend starting with the mini-project — build the paginated GET /products first (cursor-based) — and I'll help you iterate on it. You're on the right track — once you implement cursor pagination and safe filtering, you'll have tackled one of the core pain points in production APIs.