nexmedis

🧠 Backend Engineer Technical Test (TypeScript / Go)

Duration: 3 Days
Difficulty: Mid–Senior Level
Submission: GitHub repository link with a clear README.md
Languages Allowed: TypeScript **or** Go

📋 Overview

This technical test is designed to evaluate your ability to **design, implement, and optimize backend services** for high-traffic and high-availability applications. You'll be working on a **User Activity Tracking System**, which will involve **advanced caching strategies**, **high concurrency**, and **secure API handling**.

You may choose **TypeScript** (Node.js) or **Go** to implement the solution. The goal is to **demonstrate a scalable and fault-tolerant system** capable of handling large traffic loads while ensuring data consistency, performance, and security.

You may use any framework (e.g., Express/Fiber/Fastify/Gin/Echo) and database (e.g., PostgreSQL, MongoDB).

🎯 Core Requirements

1. API Design & Implementation

Create RESTful APIs for the following:

Endpoint	Method	Description

/api/register	POST	Register a new client (store client_id, name, email, api_key)
/api/logs	POST	Record an API hit (includes api_key, ip, endpoint, and timestamp)
/api/usage/daily	GET	Fetch total daily requests per client for the last 7 days
/api/usage/top	GET	Fetch top 3 clients with the highest total requests in the last 24 hours

Advanced Requirements:

- Implement authentication and authorization: Use API keys and **JWT** for token-based authentication.
- Secure all /api/usage/* endpoints with proper token verification.
- Implement input validation and error handling (e.g., 400 for bad requests, 401 for unauthorized access).

---

2. Caching & Performance (Important)

- **Redis Caching**: Implement Redis caching with the following:
    - Cache /api/usage/* responses for 1 hour, including cache invalidation after updates to ensure freshness.
    - Use **Redis TTL** effectively for cache expiration. **Avoid stale cache reads** by implementing cache invalidation strategies (e.g., **cache versioning**).
    - Simulate **cache pre-warming** for high-traffic API endpoints.
    - Handle **cache failures** gracefully (fallbacks to database).

- **Redis Pub/Sub**: Use Redis Pub/Sub for real-time updates to notify other services when usage data is updated.
- **Caching Alternatives**: Use **local in-memory cache** (e.g., LRU cache) for fast reads if Redis is unavailable.

BONUS: Implement a **cache prefetch** mechanism for the /api/usage/top endpoint to ensure it's always fresh without unnecessary DB calls.

---

3. Database Design

- **Database Sharding/Replication**: Propose and design a database schema (SQL or NoSQL) capable of handling millions of records, ensuring scalability and high availability.
- **Indexing**: Ensure that database queries are optimized for performance, including **indexing** commonly queried fields like api_key, timestamp, and client_id.
- **Partitioning**: Design a strategy for **horizontal partitioning (sharding)** and **replication** (read replicas) for handling high-traffic systems.

---

4. Concurrency & Fault Tolerance

- Handle **high-frequency API hits** in a concurrent environment.
- Implement **atomic operations** (e.g., Redis **INCRBY** for counting hits) to ensure data consistency in the event of multiple simultaneous writes.
- Use **batching** for logging API hits to avoid overwhelming the database.
- Implement **retry logic** for transient failures (e.g., Redis down, DB unresponsive).
- **Graceful degradation**: If Redis or the database is down, the system should still function, albeit at a lower performance (using in-memory or temporary storage).

---

5. Security and Data Protection

- **JWT Authentication**: Secure the /api/* endpoints with **JWT** tokens.

- **Rate Limiting**: Implement **rate limiting** for each client, ensuring they cannot make excessive API requests (e.g., 1000 requests per hour).
- **SQL Injection Protection**: Ensure all database queries are safe from SQL injection attacks (use parameterized queries, ORMs, etc.).
- **Data Encryption**: Ensure that sensitive data like api_key and email are encrypted at rest and transmitted securely (HTTPS, AES encryption).

BONUS: Implement **IP Whitelisting**: Only allow requests from specific IPs to hit the most sensitive endpoints.

---

6. Bonus Challenges (Optional but Highly Valued)

- Add **WebSocket or SSE endpoint** to stream real-time usage updates.
- Add **rate limiter** (per client per minute/hour).
- Implement **Dockerfile** and optional **docker-compose** for local setup.
- Implementing the Solution in Both **TypeScript** and **Go**:
  - If you can demonstrate competency in **both TypeScript and Go**, a **bonus will be awarded** for your ability to implement the solution in both languages. This shows **flexibility, depth in backend development**, and the ability to adapt to different environments.
  - You may implement the core functionality in **one language** and the other in **another language**, comparing your design and performance optimizations between the two.
- Add **Swagger/OpenAPI** documentation.