Bài tập: Xây dựng API Blog hiệu năng cao

Context

Bạn được giao nhiệm vụ xây dựng các API lõi cho một hệ thống blog. Để đảm bảo hiệu năng và khả năng mở rộng, bạn cần áp dụng các kỹ thuật đã học về database, caching và search.

Ngôn ngữ

Tự do lựa chọn **Golang (recommend)** hoặc **Node.js**.

Yêu cầu về môi trường

Sử dụng **Docker Compose** để dựng các services cần thiết:

- PostgreSQL
- Redis
- Elasticsearch
- Service API của bạn

Phần 1: PostgreSQL - Tối ưu và Toàn vẹn dữ liệu

Mục tiêu là thiết kế schema và tối ưu truy vấn liên quan đến các đối tượng **post** và **tag**.

Schema: Tạo một bảng `posts` trong PostgreSQL với các trường cơ bản:

- `id` (SERIAL PRIMARY KEY)
- `title` (VARCHAR)
- `content` (TEXT)
- `tags` (TEXT[]) - Mảng các tag dạng text.
- `created_at` (TIMESTAMP)

Coding Task:

1. **Tối ưu tìm kiếm theo Tag:**
    - Trên cột `tags` (kiểu `TEXT[]`), hãy tạo một **`GIN` index** để tăng tốc độ truy vấn các bài viết chứa một tag nhất định.

    - Viết một endpoint: `GET /posts/search-by-tag?tag=<tag_name>` để tìm tất cả các bài viết có chứa `<tag_name>` trong mảng `tags`.
2. **Đảm bảo toàn vẹn dữ liệu với Transaction:**
    - Giả sử chúng ta có thêm một bảng `activity_logs` (`id`, `action`, `post_id`, `logged_at`).
    - Viết một endpoint `POST /posts` để tạo một bài viết mới.
    - **Yêu cầu:** Việc `INSERT` vào bảng `posts` và `INSERT` một dòng log "new_post" vào bảng `activity_logs` phải được thực hiện bên trong **cùng một transaction**. Nếu một trong hai thất bại, cả hai phải được rollback.

Phần 2: Redis - Tăng tốc độ đọc

Mục tiêu là áp dụng chiến lược caching để giảm tải cho database và tăng tốc độ phản hồi.

Coding Task:

1. **Triển khai Cache-Aside Pattern:**
    - Viết một endpoint `GET /posts/:id` để lấy chi tiết một bài viết.
    - **Yêu cầu:** Áp dụng chiến lược **Cache-Aside** như sau:
        1. Khi có request, đầu tiên tìm trong Redis với key là `post:<id>`.
        2. Nếu tìm thấy (cache hit), trả về dữ liệu từ Redis.
        3. Nếu không tìm thấy (cache miss), truy vấn dữ liệu từ PostgreSQL.
        4. Lưu kết quả vừa truy vấn được vào Redis (với key `post:<id>`) và **set một TTL (Time-To-Live) là 5 phút**.
        5. Trả về dữ liệu cho người dùng.
2. **Xử lý Cache Invalidation:**
    - Viết một endpoint `PUT /posts/:id` để cập nhật một bài viết.
    - **Yêu cầu:** Sau khi cập nhật thành công dữ liệu trong PostgreSQL, bạn phải **xóa (invalidate) key `post:<id>` tương ứng trong Redis** để đảm bảo request tiếp theo sẽ đọc được dữ liệu mới nhất.

Phần 3: Elasticsearch - Triển khai Tìm kiếm Full-text

Mục tiêu là tích hợp khả năng tìm kiếm văn bản mạnh mẽ vào hệ thống.

Coding Task:

1. **Đồng bộ hóa dữ liệu (Indexing):**

    - Khi một bài viết được tạo mới (`POST /posts`) hoặc cập nhật (`PUT /posts/:id`), hãy đồng bộ dữ liệu của nó (`id`, `title`, `content`) vào một index trong Elasticsearch tên là `posts`.
2. **Triển khai API tìm kiếm:**
    - Viết một endpoint `GET /posts/search?q=<query_string>`.
    - **Yêu cầu:** Endpoint này phải sử dụng một **`match` query** của Elasticsearch để tìm kiếm `<query_string>` trên cả hai trường `title` và `content`.

Bài tập nâng cao (Bonus - Không bắt buộc)

Triển khai tính năng "Bài viết liên quan".

- Khi xem chi tiết một bài viết (`GET /posts/:id`), dựa vào các `tags` của bài viết đó, hãy trả về thêm một danh sách 5 bài viết khác có tag tương tự.
- **Gợi ý:** Sử dụng `bool` query với mệnh đề `should` trong Elasticsearch để tìm các bài viết khác có chứa bất kỳ tag nào của bài viết hiện tại, và loại trừ chính bài viết đó.

Yêu cầu nộp bài

1. Upload source code lên một repository Git (GitHub, etc.).
2. Repository phải bao gồm:
    - Toàn bộ source code (Go/Node.js).
    - File `docker-compose.yml` để có thể khởi chạy toàn bộ hệ thống.
    - File `README.md` giải thích rõ ràng cách chạy dự án và cách để kiểm tra các endpoint (ví dụ: cung cấp các lệnh `curl` mẫu cho mỗi API).

Chúc các bạn hoàn thành tốt bài tập!