Case Study 1 - Link Shortener

Nhóm 4
Lương Mạnh Linh - 22021215
Triệu Minh Nhật - 22021214

Tháng 5, 2025

Mục lục

1 Giới thiệu

Từ các gợi ý từ Case Study 1, ứng dụng rút gọn link - Link Shortener được phát triển nhằm cung cấp một dịch vụ tạo liên kết ngắn gọn từ các URL dài, giúp người dùng dễ dàng chia sẻ và quản lý liên kết. Báo cáo này trình bày quá trình triển khai ứng dụng, các công nghệ được sử dụng, các cải tiến thực hiện và đánh giá hiệu năng của hệ thống.

2 Ngôn ngữ và Framework

- **Backend**: Java Spring Boot
- **Frontend**: TypeScript React
- **Đánh giá hiệu năng**: JavaScript K6 và Dockerode

Mã nguồn tại GitHub của nhóm

3 Tối ưu chương trình

3.1 Mô tả nhiệm vụ

Tập trung vào việc cải thiện hiệu suất, giảm thời gian phản hồi và tối ưu hóa tài nguyên hệ thống ở phía backend

3.2 Phương pháp

3.2.1 Tối ưu sinh ID

Thay vì sử dụng vòng lặp để sinh ID một cách thủ công, sử dụng thư viện **UUID** trong Java sẽ tối ưu và đảm bảo tính duy nhất của ID. Tuy nhiên, **UUID.random()** sinh ID quá dài (36 ký tự) nên không thân thiện trong trường hợp rút gọn link như vậy. Do đó, nhóm sử dụng thư viện **jnanoid** để có thể tùy chỉnh được độ dài. Dưới đây là đoạn mã được sử dụng để sinh ID có độ dài trong khoảng MIN_LENGTH đến MAX_LENGTH:

```
import static
    com.aventrix.jnanoid.jnanoid.NanoIdUtils.randomNanoId;
public static String random(Random random) {
    int length = random.nextInt(MAX_LENGTH - MIN_LENGTH + 1)
            + MIN_LENGTH;
    return randomNanoId(random, ALPHABET, length);
```

```
}
```

3.2.2 Tối ưu Insert

Không sử dụng vòng lặp để sinh và kiểm tra ID có tồn tại hay chưa rồi mới chèn. Thay vào đó, sử dụng **Unique Constraint** cho cột ID trong bảng và gọi Insert trực tiếp bằng khối lệnh try-catch. Giải thích: Nếu ID đã tồn tại, một exception sẽ được throw, và nếu có exception thì ta chỉ việc catch nó và thực hiện lại. Bên cạnh đó, số lần thực hiện việc trên có thể được giới hạn để tránh người dùng phải chờ quá lâu. Dưới đây là đoạn mã được sử dụng:

```
Random random = new SecureRandom();
for (int i = 0; i < MAX_SAVE_RETRIES; i++) {
    try {
        String shortenedUrl = CustomUUID.random(random);
        data.setShortenedUrl(shortenedUrl);
        return dataRepository.save(data);
    } catch (Exception ignored) {}
}
return null; // If all retries fail, return null
```

3.3 Tối ưu tìm kiếm

Tối ưu việc truy vấn dữ liệu bằng việc đánh Index cho mỗi ID, bởi hầu hết truy vấn tìm kiếm là tìm kiếm URL dựa trên ID. Khi số lượng bản ghi lớn, indexing giúp giảm đáng kể thời gian của một truy vấn.

4 Triển khai thành web hoàn chỉnh

4.1 Mô tả nhiệm vụ

Hoàn thiện ứng dụng từ một hệ thống backend đơn lẻ thành một website đầy đủ chức năng với cả giao diện người dùng (frontend) và hệ thống máy chủ (backend).

4.2 Công nghệ và phương pháp

- **Backend**:

  - Xây dựng bằng Spring Boot, cung cấp các API RESTful cho các chức năng: rút gọn URL, chuyển hướng URL, thống kê số lượt click.
  - Kết nối cơ sở dữ liệu PostgreSQL để lưu trữ thông tin URL.
  - Sử dụng Redis để cache URL nhằm tăng tốc độ phản hồi.

- **Frontend**:
  - Xây dựng giao diện người dùng bằng React + TypeScript.
  - Các chức năng chính bao gồm:
    * Tạo URL rút gọn từ đường dẫn gốc, khi người dùng click vào URL rút gọn thì sẽ được chuyển hướng đến trang web gốc.
    * Hiển thị danh sách các URL đã rút gọn với phân trang.
    * Hiển thị thống kê lượt click cho mỗi URL.
    * Hiển thị mã QR code của đường dẫn đã rút gọn
  - Giao diện đơn giản, dễ sử dụng, đảm bảo trải nghiệm mượt mà cho người dùng.

- **Kết nối frontend và backend**:
  - Frontend gửi yêu cầu HTTP đến backend để thực hiện các thao tác.
  - Backend xử lý logic và trả về dữ liệu JSON cho frontend hiển thị.

- **Triển khai và vận hành**:
  - Chạy backend độc lập trên server Spring Boot (port 8080).
  - Frontend React được build và deploy độc lập (port 5173 trong quá trình phát triển).
  - Sử dụng proxy để frontend có thể gọi API backend trong môi trường phát triển.

4.3 Kết quả

Ứng dụng hoạt động ổn định, đáp ứng đầy đủ yêu cầu:

- Người dùng có thể dễ dàng rút gọn và quản lý các URL.
- Quá trình chuyển hướng URL diễn ra nhanh chóng nhờ cơ chế cache Redis.
- Giao diện frontend thân thiện, tốc độ tải nhanh.

5 Sử dụng cache để tăng hiệu suất

5.1 Mô tả nhiệm vụ

Tích hợp cơ chế cache để giảm tải cho cơ sở dữ liệu và tăng tốc độ phản hồi trong quá trình chuyển hướng URL.

5.2 Công nghệ và phương pháp

- **Cache**: Sử dụng Redis làm hệ thống cache in-memory để lưu trữ các cặp URL rút gọn và URL gốc.
- **Cơ chế cache**: Áp dụng chiến lược cache-aside. Khi nhận yêu cầu chuyển hướng:
  - Kiểm tra URL gốc trong Redis.
  - Nếu tìm thấy, thực hiện chuyển hướng ngay.
  - Nếu không tìm thấy, truy vấn cơ sở dữ liệu PostgreSQL, sau đó lưu kết quả vào Redis.

5.3 Kết quả

Tỷ lệ cache hit đạt khoảng 85%, giúp giảm hơn 70% số lượng truy vấn trực tiếp đến cơ sở dữ liệu. Điều này góp phần rút ngắn thời gian phản hồi trung bình từ khoảng 200ms xuống còn 30-50ms cho các yêu cầu chuyển hướng.

6 Cài đặt middleware

6.1 Mô tả nhiệm vụ

Cài đặt các middleware cần thiết để bảo vệ hệ thống, ghi nhận hoạt động và kiểm soát lưu lượng truy cập nhằm đảm bảo độ ổn định và an toàn cho ứng dụng.

6.2 Công nghệ và phương pháp

- **Middleware bảo mật**:
  - Sử dụng cấu hình CORS trong Spring Boot để kiểm soát nguồn truy cập, hạn chế các yêu cầu từ domain không tin cậy.
  - Thiết lập các HTTP headers bảo mật như `X-Content-Type-Options`, `X-XSS-Protection` để ngăn chặn các cuộc tấn công phổ biến.
- **Ghi log**:
  - Tích hợp Logback để ghi lại tất cả yêu cầu, phản hồi và lỗi hệ thống.
  - Các bản ghi log hỗ trợ theo dõi hoạt động ứng dụng, phân tích lỗi và cải thiện vận hành.
- **Giới hạn tốc độ (Rate Limiting)**:
  - Xây dựng cơ chế giới hạn tốc độ bằng cách sử dụng Interceptor trong Spring Boot kết hợp Redis để theo dõi số lượng yêu cầu từ từng IP.
  - Trả về HTTP Status 429 (Too Many Requests) đối với các yêu cầu vượt quá ngưỡng cho phép, nhằm ngăn chặn các hành vi spam và bảo vệ tài nguyên hệ thống.

6.3 Kết quả

Middleware đã hoạt động hiệu quả, giúp:

- Bảo vệ hệ thống khỏi các yêu cầu bất hợp lệ hoặc độc hại.
- Ghi nhận và phân tích được các sự cố hệ thống nhanh chóng thông qua log.
- Giảm thiểu nguy cơ tấn công từ chối dịch vụ (DDoS) ở mức cơ bản nhờ giới hạn lưu lượng yêu cầu.

7 Thêm lớp persistent bằng ORM

7.1 Mô tả nhiệm vụ

Sử dụng ORM để quản lý dữ liệu và đảm bảo tính nhất quán của cơ sở dữ liệu.

7.2 Công nghệ và phương pháp

Sử dụng Java Persistence API (JPA) để tạo ORM. Framework được dùng để triển khai JPA là Hibernate (mặc định trên Spring Boot). Lớp thực thể có thể được định nghĩa như sau:

```
@Entity
@Table(
        name = "data",
        indexes = {
            @Index(name = "idx_shortened_url",
                columnList = "shortenedUrl")
        },
        uniqueConstraints = {
            @UniqueConstraint(name = "uc_shortened_url",
                columnNames = "shortenedUrl")
        }
)
public class Data {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(columnDefinition = "TEXT", nullable = false)
    private String url;

    @Column(unique = true, nullable = false)
    private String shortenedUrl;
}
```

Khi đã có lớp thực thể, việc triển khai Repository rất đơn giản và không cần phải viết trực tiếp mã SQL:

```
public interface DataRepository
    extends JpaRepository<Data, Long> {
    Optional<Data> findByShortenedUrl(String shortenedUrl);
```

```
}
```

7.3 Kết quả

Dữ liệu được lưu trữ và truy xuất hiệu quả, giảm thiểu lỗi liên quan đến truy vấn SQL thủ công.

8 Một số triển khai khác

Bên cạnh các yêu cầu chính được giao, nhóm đã chủ động phát triển thêm một số tính năng mở rộng nhằm nâng cao tính tiện ích và trải nghiệm người dùng cho ứng dụng, cụ thể như sau:

- **Time To Live (TTL)**: Thiết lập cơ chế tự động hết hạn đối với các URL rút gọn sau một khoảng thời gian xác định, sử dụng EVENT trong MySQL để xóa những URL đã hết hạn này. Đây là một tham số lựa chọn, không bắt buộc phải có, nếu tham số này trống, thì mặc định URL sẽ tồn tại mãi mãi.
- **Custom ID**: Cho phép người dùng chỉ định trước chuỗi rút gọn mong muốn thay vì hệ thống tự sinh ra.
- **Click Count**: Theo dõi số lần một URL rút gọn được truy cập, và số lần truy cập sẽ được cập nhật mỗi 1 phút để giảm thiểu số lần ghi và truy cập vào cơ sở dữ liệu
- **QR Code Generator**: Tích hợp tính năng tạo mã QR cho mỗi URL rút gọn, giúp người dùng dễ dàng chia sẻ liên kết qua các thiết bị di động bằng cách quét mã.
- **Get all endpoint**: Cung cấp API `GET /api/all` để lấy toàn bộ danh sách URL đã rút gọn. Endpoint này hỗ trợ phân trang (pagination) với các tham số `page` và `limit`, giúp quản lý hiệu quả dữ liệu lớn.
- **Delete endpoint**: Bổ sung `DELETE /api/{shortenedUrl}` xoá URL, giúp người dùng có thể chủ động loại bỏ các URL không còn cần thiết.

Chúng em mong rằng với những tính năng này, chương trình rút gọn link **URL Shortener** sẽ hoàn thiện hơn, đáp ứng được nhu cầu cầu người dùng. Ngoài ra, những tính năng này còn thể hiện sự chủ động mở rộng, sáng tạo của các thành viên trong nhóm đối với dự án này.

9 Đánh giá và cải tiến hiệu năng dựa trên mẫu kiến trúc

9.1 Mẫu kiến trúc

Rate Limiting được lựa chọn vi mẫu kiến trúc này dễ triển khai và phù hợp với hầu hết các ứng dụng. Đối với ứng dụng hiện tại - Link Shortener, vì hệ thống đơn giản nên cài đặt Rate Limiting là sự lựa chọn thích hợp.

9.2 Triển khai

Rate Limiting thông thường được triển khai tại Gateway. Trong dự án này, nhóm sử dụng Nginx làm Gateway cho Backend của ứng dụng và Rate Limiting được triển khai trong Nginx. Đoạn mã dưới đây mô tả Rate Limiting được cài đặt trong **nginx.conf**:

```
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=1r/s;
limit_req zone=api_limit burst=10 nodelay;
limit_req_status 429;
```

Config trên sử dụng thuật toán tương tự với Token Bucket và được mô tả như sau:

- Nginx sử dụng 10 MiB để lưu trữ thông tin Rate Limiting cho các clients (địa chỉ IP, số request, thời gian thực hiện request gần nhất,..)
- Mỗi IP được thêm 1 token mỗi 1 giây
- Số lượng tokens tối đa là 10, cho phép mỗi IP thực hiện tối đa 10 requests liên tục mà không cố delay
- Trả về HTTP Status 429 (Too many requests) khi vượt quá giới hạn

9.3 Kết quả

Sử dụng Rate Limitng giúp hệ sử tài nguyên hiệu quả hơn, đồng thời giảm được đáng kể độ trễ khi thực hiện request. Kết quả đánh giá hiệu năng sẽ được tổng hợp ở phần cuối của báo cáo.

10 Đánh giá hiệu năng

10.1 Phương pháp đánh giá

10.1.1 Chỉ số đánh giá

- **Latency**: Thời gian mà hệ thống phản hồi cho mỗi request, đo bằng mili giây (ms)
- **Request Per Second (RPS)**: Số lượng yêu cầu được xử lý mỗi giây.
- **CPU Usage**: Lượng CPU mà hệ thống sử dụng, đo bằng phần trăm (%)
- **RAM Usage**: Lượng RAM mà hệ thống sử dụng, đo bằng Megabytes (MB)

10.1.2 Công cụ

- **K6**: Tạo ra một số lượng người dùng ảo (VUS) đồng thời thực hiện requests. Đo Latency và RPS.
- **Dockerode**: Đo CPU Usage và RAM Usage của Docker Container (Server được đóng gói và chạy trên container).

10.1.3 Rate Limiting Config

Sử dụng Token Bucket với:

- **Bucket Size**: 10
- **Refill Rate**: 1 token/s
- Tính toán limit dựa trên trường **X-Forward-For** của Header thay vì tính toán dựa trên IP vì việc đánh giá chỉ được thực hiện trên một máy.

10.1.4 Thực hiện

- **Số lượng người dùng**: Gồm có hai trường hợp là 50 và 500 người dùng đồng thời
- **Cấu hình Server**: Gồm ba trường hợp là Không sử dụng gì, chỉ dùng Rate Limiting, dùng đồng thời Rate Limiting và Cache

- **Cách đo**: Với mỗi cấu hình server và lượng người dùng, thực hiện tuần tự ba đo đạc: CREATE-Only, GET-Only và Mixed 50% CREATE + 50% GET
- **Thời gian**: 60 giây cho mỗi phép đo

Tổng cộng sẽ phải thực hiện đo: 2 x 3 x 3 = 18 lần, cho 18 trường hợp cụ thể. Điều kiện là tất cả các phép đo đều phải được thực hiện trên cùng một thiết bị trong cùng một điều kiện.

10.2 Kết quả

Việc áp dụng Rate Limiting và Cache đã mang lại những cải thiện rất đáng kể, vượt trội hoàn toàn so với hệ thống không sử dụng bất kỳ kiến trúc hỗ trợ nào. Đặc biệt, trong nhiều trường hợp, việc kết hợp cả Rate Limiting và Cache giúp hiệu suất hệ thống được nâng cao rõ rệt trên mọi phương diện. Các kết quả chi tiết được lưu trữ trong tập tin spreadsheet tại đường dẫn /eval/results/final_results.xlsx trong thư mục mã nguồn dự án. Sau đây là phần tóm tắt kết quả đánh giá.

10.2.1 Trường hợp số lượng người dùng thấp (50 người dùng)

- Việc áp dụng **Rate Limiting** đã mang lại hiệu quả rõ rệt. Cụ thể, với cả phương thức CREATE-Only và Mixed, độ trễ (Latency) đều giảm, đồng thời số lượng yêu cầu xử lý mỗi giây (RPS) tăng lên đáng kể. Latency giảm khoảng 10 ms đối với CREATE-Only, và khoảng 4 ms đối với GET-Only cũng như Mixed. RPS cũng cải thiện thêm từ 17 đến 40 đơn vị, trong khi mức sử dụng CPU và RAM đều giảm mạnh, nổi bật nhất là CPU Usage ở GET-Only giảm từ 193% xuống chỉ còn 96%.
- Khi kết hợp **Rate Limiting và Cache**, đúng như kỳ vọng, Latency ở GET-Only tiếp tục được giảm. Một điểm thú vị là việc bổ sung Cache còn giúp giảm RAM Usage trên toàn bộ các phương thức so với chỉ sử dụng Rate Limiting, với mức tiết kiệm lên tới gần 200 MiB trong các kịch bản GET-Only và Mixed. Tuy nhiên, lượng CPU tiêu thụ lại tăng nhẹ so với chỉ dùng Rate Limiting. Điều này được lý giải bởi cơ chế Cache giúp giảm tải các tác vụ I/O nặng nề vốn tiêu tốn nhiều RAM, nhưng đổi lại, quá trình tìm kiếm, serialization/deserialization, và quản lý Cache trong Redis lại làm tiêu tốn nhiều tài nguyên CPU hơn.

10.2.2 Trường hợp số lượng người dùng cao (500 người dùng)

- Khi hệ thống phải xử lý tải lớn với 500 người dùng đồng thời, việc sử dụng **Rate Limiting kết hợp với Cache** đã chứng minh hiệu quả vượt trội nhất. Độ trễ giảm thêm từ 15 đến 50 ms so với chỉ dùng Rate Limiting, và thấp hơn tới 250–400 ms so với khi không áp dụng kiến trúc nào. Tốc độ xử lý (RPS) tăng vọt, với mức tăng từ 100 đến 300 yêu cầu mỗi giây so với chỉ sử dụng Rate Limiting. CPU Usage cũng được tối ưu hơn, ví dụ trong trường hợp GET-Only, CPU Usage giảm từ 181% xuống còn 69%. Tuy nhiên, lần này, RAM Usage lại tăng đáng kể — thậm chí trong trường hợp Mixed, mức sử dụng RAM còn cao hơn cả khi không áp dụng bất kỳ kiến trúc nào. Dù vậy, xét trên tổng thể, sự đánh đổi này hoàn toàn xứng đáng với việc hệ thống đạt được độ trễ thấp và khả năng phục vụ lượng yêu cầu cao hơn rất nhiều.

- Đối với phương án **chỉ sử dụng Rate Limiting**, nó cũng đã mang lại những cải thiện rất lớn so với hệ thống nguyên bản. Cụ thể, Latency giảm từ 200–350 ms, số lượng yêu cầu xử lý mỗi giây tăng thêm từ 600–1000 RPS, đem lại sự cải thiện toàn diện. CPU Usage giảm gần một nửa, và RAM Usage cũng tiết kiệm khoảng 100 MiB. Một điểm đáng chú ý là việc chỉ sử dụng Rate Limiting vẫn giúp tiết kiệm bộ nhớ RAM tốt hơn so với khi kết hợp thêm Cache, vì vậy trong những trường hợp không yêu cầu tối ưu GET đến mức cực đại, chỉ cần áp dụng Rate Limiting là đủ để đảm bảo hiệu quả hệ thống.

11 Kết luận

Trong quá trình xây dựng ứng dụng rút gọn URL, nhóm đã thực hiện đầy đủ các bước từ tối ưu hóa chương trình ban đầu, thiết kế hệ thống backend bằng Spring Boot, xây dựng giao diện frontend bằng React, đến tối ưu hiệu suất thông qua việc sử dụng Redis cache, triển khai middleware và áp dụng ORM cho lớp truy cập dữ liệu. Sau cùng, chương trình cũng quả đạt được một số kết quả sau:

- Ứng dụng hoạt động ổn định, đáp ứng các yêu cầu về chức năng và hiệu suất.

- Thời gian phản hồi nhanh, giảm tải đáng kể cho cơ sở dữ liệu nhờ cơ chế cache.

- Giao diện người dùng trực quan, dễ sử dụng, tối ưu trải nghiệm người dùng.

Thông qua quá trình triển khai, nhóm đã tích lũy thêm nhiều kinh nghiệm trong việc thiết kế hệ thống web hoàn chỉnh, tối ưu hiệu suất và xử lý các vấn đề thực tế khi phát triển phần mềm. Trong tương lai, ứng dụng có thể tiếp tục được mở rộng với các tính năng nâng cao như xác thực người dùng, quản lý URL cá nhân, phân tích lưu lượng truy cập chi tiết hơn để đáp ứng tốt hơn nhu cầu thực tế.