EXPERIMENT: 1

NAME: NIKHIL VERMA

BRANCH: BE-CSE

SEMESTER: 6TH

SUBJECT NAME:SYSTEMDESIGN

UID: 23BCS14095

SECTION/GROUP: KRG_1B

SUBJECT CODE: 23CSH-314

1. Aim:

Tostudy and design a scalable URL Shortener system by analysing its functional requirements, non-functional requirements, API design, database schema, and low-level design approaches including counter-based short URL generation for distributed systems.

2. Objective:

1. To understand the working of a URL shortening service.
2. To design REST APIs for URL creation and redirection.
3. To identify suitable database and server choices.
4. To analyse different short URL generation techniques.
5. To understand scalability challenges and their solutions.

3. Tools Used: Postman,Draw.io,Lucidchart, Excalidraw

4. Requirements Specification:

4.1. Functional Requirements:

- ☐ Generate a unique short alias for a given long URL.
- ☐ Redirect users from a short URL to the original long URL with minimum latency.
- ☐ Basic Sign-up and Login functionality.
- ☐ Premium Features:
  - ⬚ Custom URLs: Users can pick their own short string (e.g., bit.ly/Aman).
  - ⬚ Expiration: Set aTime-to-Live (TTL) for links.

4.2. Non-Functional Requirements:

- ☐ High Availability: The systemmust be up 24/7, a broken redirect is a bad user experience.
- ☐ Low Latency: Redirection should happen in real-time (< 50ms).
- ☐ Scale Estimation:
  - ⬚ Read QPS: 100 million requests/day.
  - ⬚ Write QPS: 1 million requests/day.
  - ⬚ Read/Write Ratio: 100:1 (Read-heavy system).

5. API Design:

5.1.Generate Short URL:

Method: POST

Endpoint: /api/shorten

Request Body:

```
{
            "longUrl": "any long url",
            "customUrl":            "?",
            "expiryDate": "?"

    }
```

Response:

```
    {
            "shortUrl": "generated short url",
            "shortCode": "generated short code"
    }
```

5.2. Redirect:

Method:GET

Endpoint:/api/<shortCode>

Action:Redirects to corresponding Long URL mapped with the short code.

6. Database Schema:

Forhighscalability,aNoSQL database (like MongoDB or Cassandra) is preferred, but Relational (PostgreSQL) works for structured metadata.

Table: Users

☐ UserId(PK): VARCHAR (8)

☐ UserName: TEXT

☐ UserEmail: TEXT UNIQUE

☐ UserPassword: TEXT ENCRYPTED

Table: URL_Mappings

☐ ShortCode (PK): VARCHAR (8)

☐ LongURL: TEXT

☐ CustomURL: TEXT

☐ UserID: INT (FK)

☐ ExpirationDate: TIMESTAMP

7. High Level Design (HLD):

TheHLDfocusesontherequest-response lifecycle for two primary actions: Creation and Redirection.

7.1. Creation:

- ☐ The user sends an HTTPS POST request containing the Long_Url.
- ☐ The application server performs a logical computation (hashing MD5) on the long URL to generate a unique short code.
- ☐ The server checks for/stores the Long_Url ☐ Short_Url mapping in the database.
- ☐ The server returns the Short_url in the response body.

7.2. Redirection:

- ☐ The user sends an HTTPS GET request using the Short Code.
- ☐ The server performs a database lookup to find the corresponding original URL.
- ☐ The server returns an HTTPS Redirect response (e.g., 301/302) to the original Long_URL.

8. Low Level Design (LLD):

8.1.LatencyBreakdown:

- ☐ Hashing Computation: 1–2ms.
- ☐ Server-to-DB Request: ~4ms.
- ☐ Database Lookup: 20ms+ (Slow process).
- ☐ Total Latency: Approximately 25ms+ per single hashing attempt.

8.2. High Latency and Collision Handling:

- ☐ Problem:

    Asthe userbasegrows to 1 million active "write" users, the probability of hash collisions increases. If a collision occurs, the system must re-hash, adding another 25ms+ of latency. Therefore,this system is notfeasible.

- ☐ Solution: Use Counters to handle collisions. By appending a counter value to similar short codes and incrementing it, we can resolve duplicates more efficiently than repeated hashing. Essentially reducing database lookups

8.3. Single Point of Failure (SPOF):

- ☐ Problem:

    Relying on a single application server or a single database instance makes the system fragile.

- ☐ Solution:

    Horizontal Scaling. Deploy multiple server instances behind a Load Balancer. Load Balancing Strategy: Use algorithms like Round-Robin to decide which server instance handles an incoming request.

8.4. The "Dirty Read" Problem in Distributed Counters:
- Problem:
  When multiple horizontally scaled servers maintain their own local counters (e.g., Counter 1, Counter 2, Counter 3), they may become out of sync. This leads to "dirty reads" where different servers might attempt to assign the same counter value to different URLs.
- Solution:
  Global State Management via Redis. Store the counter value globally in a fast, in-memory cache like Redis. This ensures all server instances read the same incremented value, maintaining consistency while preserving low latency.

8.5. SPOF in REDIS:
- Problem:
  While Redis solves the consistency issue, a single Redis instance could become a new SPOF or a performance bottleneck.
- Solution:
  Vertical Scaling of Redis. Increase the resources (CPU/RAM) of the Redis instance to handle higher throughput and ensure high availability.

9. LearningOutcomes:
9.1. Understanding Distributed System Latency
9.2. Collision Resolution Strategies
9.3. Architectural Scalability and Reliability
9.4. Distributed State Management
9.5. Single Points of Failure (SPOF)