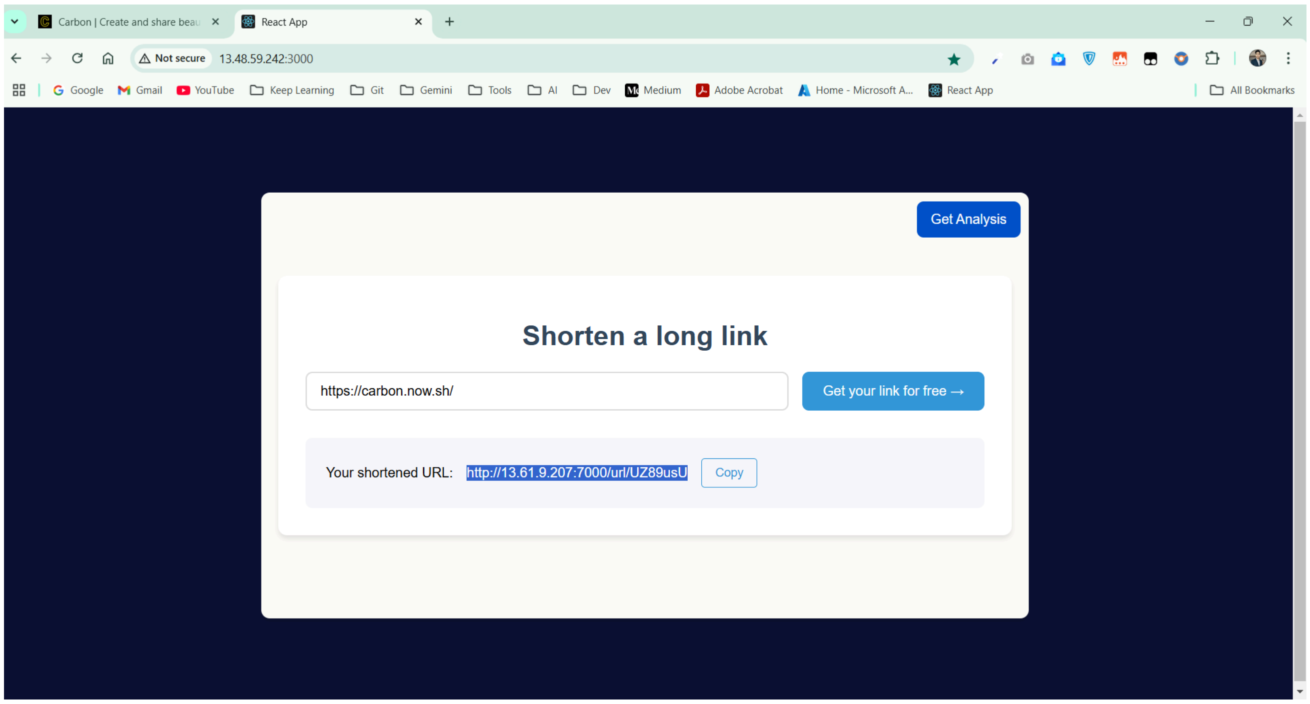
System Design: URL Shortener

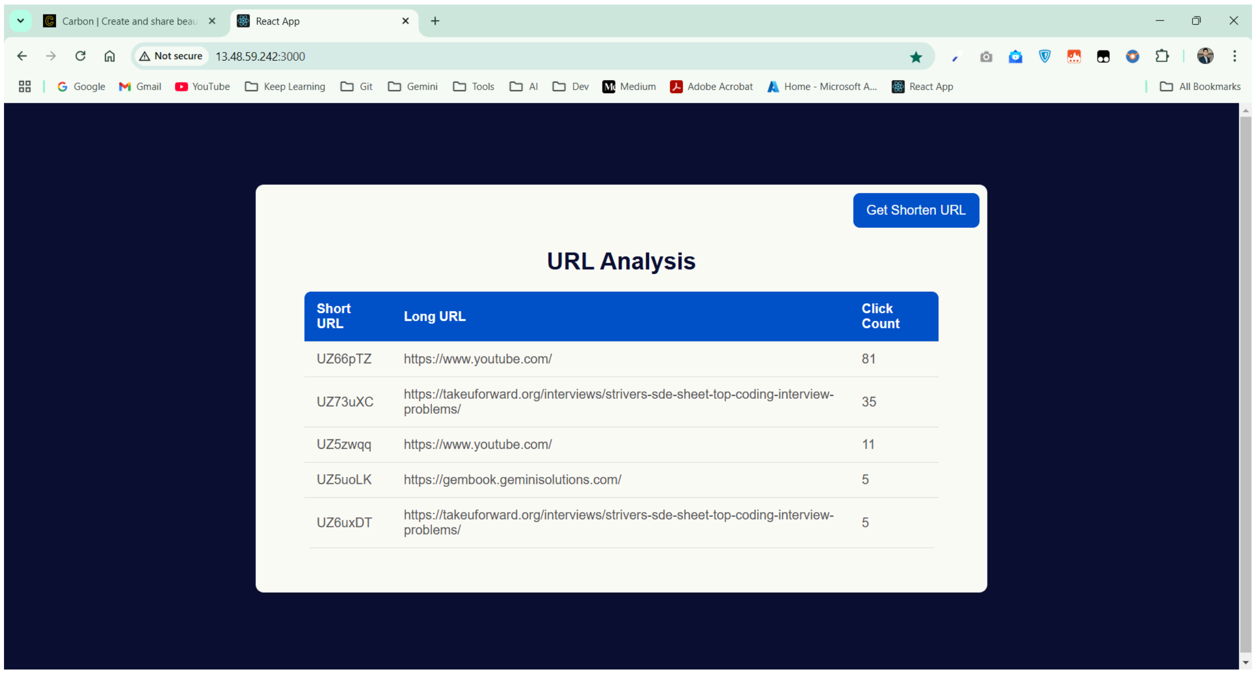
Deployed Application on AWS EC2



Shorten a long link

https://carbon.now.sh/

Your shortened URL: http://13.61.9.207:7000/url/UZ89usU



URL Analysis

Short URL	Long URL	Click Count
UZ66pTZ	https://www.youtube.com/	81
UZ73uXC	https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/	35
UZ5zwqq	https://www.youtube.com/	11
UZ5uoLK	https://gembook.geminisolutions.com/	5
UZ6uxDT	https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/	5

Link: http://13.48.59.242:3000/ (**Because of the restrictions on the free AWS account, the hosting will be temporary.**)

What is a URL Shortener?

A URL shortener service creates an alias or a short URL for a long URL. Users are redirected to the original URL when they visit these short links.

Requirements

Our URL shortening system should meet the following requirements:

Functional requirements

1. Shorten URL:

- Accept a long URL and return a shortened URL.
- Validate input URLs (e.g., proper format).

2. Redirect URL:

- Redirect users to the original URL when the shortened URL is accessed.

3. Analytics:

- Track and display click counts for each shortened URL.

4. Frontend:

- Provide a user-friendly interface for shortening URLs and viewing analytics.

5. Backend:

- Provide RESTful APIs for the frontend to interact with.

Non Functional requirements

1. Scalability:

- Design the system to handle at least 10 million requests per day.

2. Availability:

- Ensure high uptime and reliability.

3. Performance:

- Optimize for low latency in redirections.

4. Security:

- Prevent malicious URLs and ensure safe handling of user data.

Estimation and Constraints

When designing a scalable and reliable URL shortener application, it's crucial to understand the expected system requirements and limitations. Below is an analysis of **estimations** and **constraints** for the system:

1. Traffic Analysis

The URL shortener will remain a **read-heavy system**, with a **100:1 read-to-write ratio**. Based on the assumptions:

- **Total Requests per Day:** 10 million.
- **New Links Generated:** 10 million / 100 = **100,000 write requests/day**.
- **Read Requests (Redirection):** 10 million−100,000 = **9.9 million read requests/day**

2. Requests Per Second (RPS)

- **Writes (New URLs):**
  (100,000 requests/day) / (24 hours×3600 seconds) ≈ **1.2 writes/second**
- **Reads (Redirection):**
  (9.9 million/day) / (24 hours×3600 seconds) ≈ **115 reads/second**

3. Bandwidth Estimation

- **Write Requests:**
  Assume each write request has a payload size of 500 bytes.
  1.2 writes/second × 500 bytes = **600 bytes/second (incoming)**
- **Read Requests:**
  Assume each read request response is 500 bytes.
  115 reads/second×500 bytes = **57.5 KB/second (outgoing)**

4. Storage Requirements

- **Total Links to Store Over 100 Years:**
  100,000 new links/day × 365 days/year ×100 years = **3.65 billion links**
- **Storage Per Link:**
  Each link (original URL, shortened URL, click count) requires approximately **500 bytes**.
  3.65 billion links × 500 bytes/link = **1.825 TB** of storage required

5. Cache Requirements

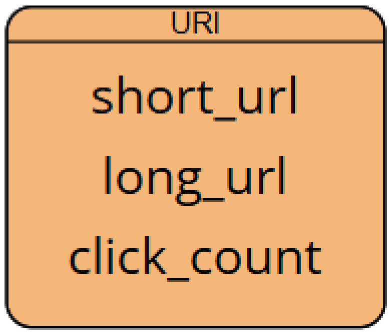
Using the **80/20 rule (Pareto principle)** for caching:

- **Cache 20% of the Data:**
  20% of the most-accessed URLs account for 80% of the requests.
- **Daily Read Requests:**
  9.9 million requests/day
- **Cached Data Size:**
  20% × 9.9 million requests/day × 500 bytes/request = **990 MB of memory required/day**

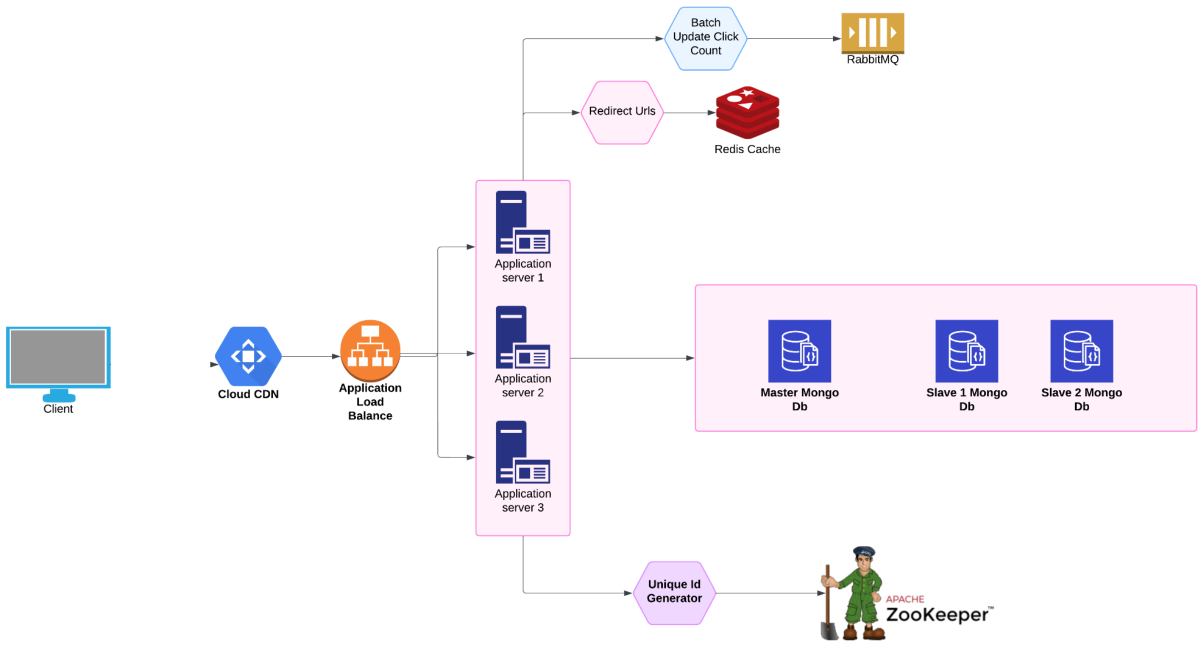
6. High-Level Estimates

Type	Estimate
Writes (New URLs)	~1.2 requests/second
Reads (Redirection)	~115 requests/second
Bandwidth (Incoming)	~600 bytes/second
Bandwidth (Outgoing)	~57.5 KB/second
Storage (100 years)	~1.825 TB
Cache Memory (Daily)	~990 MB

Data Model



Application Architecture Diagram



Why CDN?

CDNs are designed to deliver content with high availability and low latency by caching data in geographically distributed servers. For a URL shortener, this can provide several advantages:

1. **Caching**: CDN caches short URL mappings, reducing backend load.
2. **Low Latency**: Nearest edge servers provide faster redirection for users worldwide.
3. **High Availability**: Redundant servers ensure reliability, even during traffic spikes or outages.
4. **Scalability**: Offloads traffic from the origin server, handling high loads efficiently.

Why LoadBalancer?

A **Load Balancer** is essential for distributing incoming traffic across multiple servers or instances, ensuring high availability, scalability, and reliability in a system. Here’s why it's crucial:

1. **Traffic Distribution**:
   - Distributes user requests evenly across backend servers to prevent any single server from being overwhelmed.
2. **High Availability**:
   - If a server goes down, the load balancer reroutes traffic to healthy servers, minimizing downtime.
3. **Scalability**:
   - Supports horizontal scaling by adding or removing servers dynamically as traffic fluctuates.
4. **Improved Performance**:
   - Balances load to avoid bottlenecks, ensuring optimal response times for users.
5. **Fault Tolerance**:
   - Detects and removes faulty servers from the pool until they are healthy again.
6. **Security**:
   - Acts as a shield by terminating SSL/TLS connections and protecting backend servers from direct exposure.
7. **Session Persistence** (Sticky Sessions):
   - Ensures users are directed to the same server for a consistent experience when required.

Why Redis Cache?

Redis is a critical component in the system to enhance performance and handle high traffic efficiently. Here's why Redis is suitable:

1. **Caching for Performance**:
   - Redirection requests are read-heavy. Using Redis as a cache reduces database queries for frequently accessed URLs.
   - This aligns with the **80/20 rule**, where 20% of URLs handle 80% of the traffic.
2. **Low Latency**:
   - Redis operates in-memory, providing sub-millisecond response times, essential for optimizing redirection performance.
3. **High Throughput**:
   - Redis can handle millions of requests per second, ensuring the system can handle peak traffic efficiently.
4. **Scalability**:
   - Redis supports clustering, allowing the system to scale horizontally to handle growing traffic.
5. **Persistence**:
   - Redis provides optional persistence (e.g., RDB snapshots, AOF logs) to ensure data durability while maintaining performance.
6. **Cost-Effective Cache Size**:
   - With ~990 MB of daily cache memory estimated, Redis can easily accommodate the most frequently accessed URLs in memory.
7. **Justification Based on Estimations**:
   - Redis can cache the **20% most accessed URLs** (~990 MB daily) and serve **80% of daily traffic (~115 requests/sec)** without hitting the database.

Why RabbitMq?

RabbitMQ plays a pivotal role in optimizing the performance of the click count update process in your URL shortener system. Here's how it is effectively utilized:

1. Asynchronous Processing of Updates

- **Problem Without RabbitMQ**:
  - Directly writing to the database for each redirection leads to high latency and puts immense pressure on the database.
- **Solution with RabbitMQ**:
  - Instead of immediate writes, click counts are stored in memory (e.g., Redis) and pushed to RabbitMQ in batches every 10 minutes via a **scheduled job**.

    - RabbitMQ processes these updates **asynchronously**, ensuring user-facing redirection operations are not delayed.

2. Batch Updates Reduce Database Load

- **How It Works**:
    - The scheduled job collects multiple click counts, aggregates them, and sends a single batch message to RabbitMQ.
    - RabbitMQ forwards this batch to consumer services that handle **bulk writes** to the database.
- **Optimization**:
    - Bulk updates are significantly faster and more efficient than multiple small writes, reducing strain on the database.

3. High Reliability for Critical Data

- **Problem Without RabbitMQ**:
    - If the system crashes, in-memory click count data might be lost before it can be written to the database.
- **Solution with RabbitMQ**:
    - RabbitMQ ensures data persistence. All messages (click count updates) are stored reliably in queues until processed, even during system failures.
    - This guarantees no loss of critical click count data.

What kind of database should we use?

For a URL shortener, a NoSQL database is an excellent choice due to the nature of the application, which benefits from high performance, scalability, and simplicity in schema design. Here's a detailed analysis:

Why MongoDB?

- **Horizontal Scalability**: MongoDB supports sharding, which allows you to scale horizontally by distributing data across multiple nodes.
- **Schema Flexibility**: The flexible document model is ideal for a URL shortener, as the schema is simple and might evolve over time (e.g., expiration metadata).
- **High Read/Write Throughput**: MongoDB's architecture supports high-speed read and write operations, critical for a URL shortener with a high volume of requests.
- **Indexing and Querying**: MongoDB provides robust indexing capabilities for fast lookups, and you can query by either the key (short URL) or value (long URL)

Advantages of MongoDB for a URL Shortener

- **Fast Reads/Writes**:
  - Key-value access pattern ensures efficient lookups and storage.
- **Scalability**:
  - Sharding enables **horizontal scaling** to handle millions or billions of records.
- **Fault Tolerance**:
  - Replica sets ensure high availability and data durability.

URL Encoding

What is Base62?

Base62 encoding uses a set of **62 unique characters** to represent values. These characters are:

- **Uppercase letters**: A-Z (26 characters)
- **Lowercase letters**: a-z (26 characters)
- **Digits**: 0-9 (10 characters)

So, the character set for **Base62** is:
“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789”

1. Why Base62?

Our Assumptions:

1. **10 million requests per day**.
2. **3650 million requests per year** (10 million requests/day * 365 days/year).
3. **365,000 million requests over 100 years** (10 million requests/day * 365 days/year * 100 years).

1. Key Space Calculation:

The number of unique combinations possible with **Base62 encoding** for **N characters** is 62^N, where 62 is the base (A-Z, a-z, 0-9).

- **For 6 characters** in Base62:

62^6 ≈ **56.8 billion** unique combinations

- **6 characters** will give us **56.8 billion** unique URLs, which is **insufficient** for the required **365 billion URLs** over 100 years.

- **For 7 characters** in Base62:

627≈3.5 trillion unique combinations

- **7 characters** will give us **3.5 trillion** unique URLs, which is **more than sufficient** for the **365 billion URLs** over 100 years.

So, based on the **365 billion unique URLs** required over **100 years**, we can safely use **7 characters** for Base62 encoding.

Why ZooKeeper?

ZooKeeper can be critical when handling the unique generation of short URLs in a distributed system. Here's why **ZooKeeper** would be needed despite the large keyspace offered by **Base62 encoding**:

1. Distributed System Coordination:

- If your **URL shortener** service is running on multiple **distributed servers**, you need a way to coordinate the generation of unique keys (short URLs) across those servers. Without a centralized coordination mechanism, each server might generate the same key for different long URLs, leading to collisions.
- **ZooKeeper** provides distributed synchronization, ensuring that only one server generates a particular key at a time, preventing collisions even if multiple servers are working in parallel.

2. Unique Key Generation Across Multiple Servers:

- Let's say your system needs to handle billions of unique URLs across many servers. If you're generating short URLs based on an **incrementing counter** (or similar logic), multiple servers trying to generate the next short URL may conflict if they access the counter independently.
- ZooKeeper can manage a **distributed counter** where each server requests the next available counter value. ZooKeeper ensures that each server gets a unique value without duplication. It can also distribute the key generation across **multiple nodes** to increase throughput.

3. Handling Key Allocation Ranges:

- To avoid a **single point of failure**, you can use **ZooKeeper** to assign different **ranges** of keys to different servers. This approach ensures that each server operates on a separate subset of keys, reducing the chances of collisions and improving efficiency.
- For example, ZooKeeper can allocate key ranges like:
    - **Server 1**: Handles keys from `1` to `100,000,000`.
    - **Server 2**: Handles keys from `100,000,001` to `200,000,000`, and so on.
- This distributed management of key ranges helps balance the load, prevents collisions, and ensures that multiple servers can generate unique keys in parallel without conflicting.

4. Scalability and High Availability:

- As your system scales, **ZooKeeper** allows you to add more **servers or nodes** to handle the increased load of key generation. It will automatically handle **coordination** and **synchronization**, ensuring that your system remains reliable even as you scale horizontally.
- ZooKeeper also ensures **high availability** of the coordination service, which is crucial in a **high-traffic environment** like a URL shortener.

5. Fault Tolerance:

- If a server goes down while generating keys, ZooKeeper ensures that the system continues to function correctly by assigning new keys from the pool of available key ranges, preventing loss of data or collisions.
- ZooKeeper maintains consistency even if nodes fail, by **replicating** the data across different nodes, making sure the state is never lost.

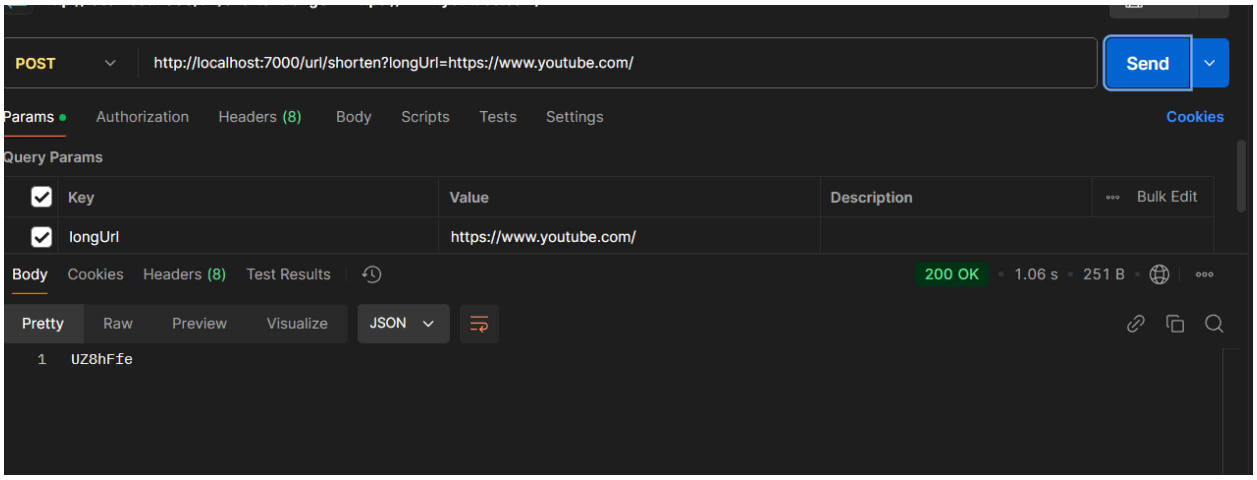
API Endpoints

Here are the detailed API endpoints for our URL shortener service:

1. POST /url/shorten

- **Description**: This endpoint accepts a long URL from the user and returns the shortened version of it.
- **Method**: `POST`
- **URL Path**: `/url/shorten`
- **Parameters**:

    - `longUrl`: A `String` representing the long URL that needs to be shortened. This is provided as a query parameter.
- **Response**:
    - **Success (200 OK)**: A response body containing the shortened URL.
    - **Failure (400 Bad Request)**: If the input URL is invalid.


POST http://localhost:7000/url/shorten?longUrl=https://www.youtube.com/

	Key	Value	Description
✓	longUrl	https://www.youtube.com/	

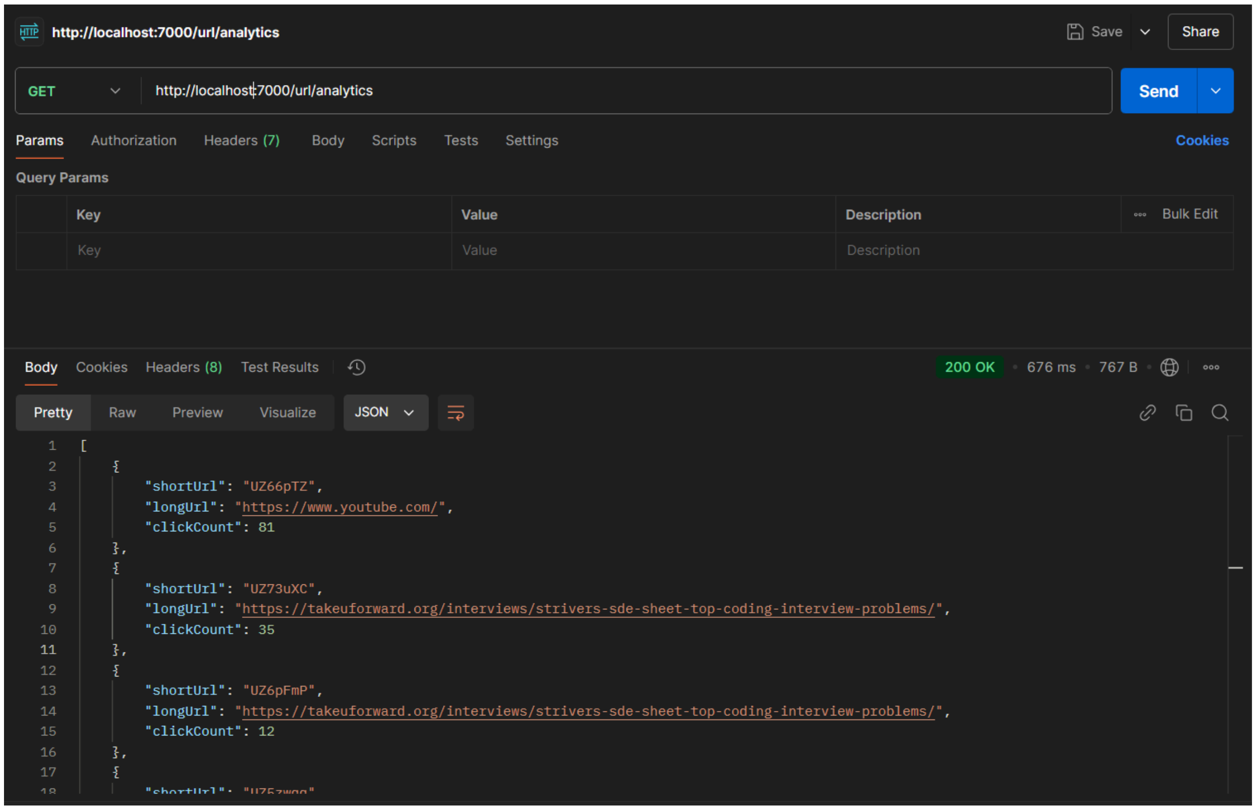
```
UZ8hFfe
```

2. GET /url/{shortUrl}

- **Description**: This endpoint redirects the user to the original long URL when provided with the shortened URL.
- **Method**: `GET`
- **URL Path**: `/url/{shortUrl}`
    - `{shortUrl}` is a path variable representing the shortened URL.
- **Parameters**:
    - `shortUrl`: A `String` representing the shortened URL that the user wants to access.
- **Response**:
    - **Success (301 Redirect)**: The user is redirected to the original long URL.
    - **Failure (404 Not Found)**: If the shortened URL does not exist in the database.
    - **Failure (500 Internal Server Error)**: If an unexpected error occurs while handling the redirect.
- **Rate Limiting**: This endpoint is protected by a rate limiter (`@RateLimiter`) to prevent abuse. If the limit is exceeded, the fallback method `handleShortUrl()` is invoked.

3. GET /url/analytics

- **Description**: This endpoint retrieves the analytics data, it provides top 5 click counts for each shortened URL.
- **Method**: `GET`
- **URL Path**: `/url/analytics`
- **Parameters**: None.
- **Response**:
  - **Success (200 OK)**: A list of `AnalyticsResponse` objects, containing the analytics data (such as click counts, URL data, etc.).
-



```
[
    {
        "shortUrl": "UZ66pTZ",
        "longUrl": "https://www.youtube.com/",
        "clickCount": 81
    },
    {
        "shortUrl": "UZ73uXC",
        "longUrl": "https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/",
        "clickCount": 35
    },
    {
        "shortUrl": "UZ6pFmP",
        "longUrl": "https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/",
        "clickCount": 12
    },
    {
        "shortUrl": "UZ5zwqq"
```

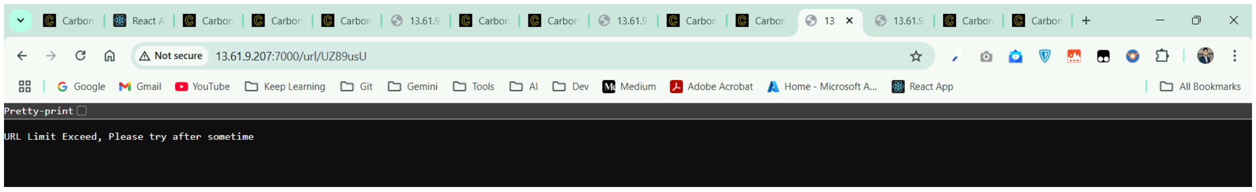
Why Rate Limiting?

1. **Prevent Abuse**: Protect the URL shortening service from excessive requests (e.g., bots, high traffic), ensuring system stability.
2. **Avoid DDoS**: Throttle incoming requests to prevent overwhelming the server or database, mitigating potential Distributed Denial of Service (DDoS) attacks.
3. **Fair Access**: Ensure no single user or client monopolizes the service by limiting the number of requests.

4. **Performance**: Control the rate of requests to reduce load on backend systems and avoid performance degradation.

Steps to Implement Rate Limiting Using Resilience4j in Spring Boot:

1. **Add Dependencies**:
   - Include `resilience4j-spring-boot2` in `pom.xml` to use Resilience4j in Spring Boot.
2. **Configure Rate Limiting in `application.yml`**:
   - Define rate limit parameters like:
     - `limitForPeriod`: Max requests allowed within a period.
     - `limitRefreshPeriod`: Period duration to refresh the count.
     - `timeoutDuration`: Time to wait before rejecting requests.
3. **Apply Rate Limiting with `@RateLimiter` Annotation**:
   - Use `@RateLimiter(name = "shortLinkLimiter")` on endpoints to apply rate limiting to the methods.
4. **Define a Fallback Method**:
   - Define a fallback method to handle requests that exceed the rate limit, e.g., return a message like "Rate limit exceeded."
5. **Optional: Customize Rate Limiting for Other Endpoints**:
   - Apply rate limiting to multiple endpoints as required by adding `@RateLimiter` to each method.



```
URL Limit Exceed, Please try after sometime
```