Implementation Report – URL Shortener Service

> Group 23

1. Implementation Overview

In this report we will discuss the URL shortener that we have designed and programmed. We have implemented a RESTful API that handles the requests. URLs are shortened by using a Snowflake ID algorithm to ensure uniqueness. By using base62 on the generated ID we can shorten the ID and we ensure no URL unsafe characters are included.

The API supports the following endpoints:

Endpoint	Method	Functionality	Expected Response
$/<id>$	GET	Retrieves the full URL for a given short ID	301 Redirect if found, 404 Not Found if missing
$/<id>$	PUT	Updates an existing short ID with a new URL	200 OK if successful, 400 Bad Request for invalid input, 404 Not Found if ID does not exist
$/<id>$	DELETE	Deletes an existing short ID	204 No Content if successful, 404 Not Found if ID does not exist
$/$	GET	Lists all stored short IDs	200 OK, returns all keys
$/$	POST	Creates a new short URL	201 Created, returns a new short ID, 400 Bad Request for invalid input
$/stats/<id>$	GET	Retrieves usage statistics for a short URL	200 OK with statistics if found, 404 Not Found if ID does not exist

2. Snowflake ID Generation Algorithm

The challenge of this task lies in generating globally unique IDs to ensure the uniqueness of the stored URLs while having little characters to represent it. Therefore, we referred to Twitter's Snowflake ID generator algorithm. This algorithm combines a timestamp, a machine identifier, and a sequence number to guarantee the uniqueness of the IDs, and the machine identifier enables our system to scale in a distributed manner.

Structure of the Snowflake ID

1. Timestamp (31 bits) – Ensures uniqueness over time.

2. Machine ID (5 bits) – Differentiates IDs generated by different machines.
3. Sequence Number (5 bits) – Ensures uniqueness within the same millisecond.

By mixing these components, the Snowflake ID generator guarantees that no two IDs are identical. In our code we put the Machine ID to a static value of 1 because it is not on a distributed environment but just our individual laptop. If you plan on deploying this on a distributed environment then you can change this value on every machine.

Concurrency and Thread Safety

To prevent errors while multiple users are generating URLs at the same moment; we have implemented threading locks. We do this in the generating ID function before the code is run that is retrieving the timestamp for example.

Advantages of the Snowflake Approach

- Collision-Free: When the code is deployed in a distributed environment, different machines will not create the same IDs.
- Scalable: The solution works on multiple machines if the Machine IDs are unique.
- Time-Ordered: The timestamp component that only increment can make sorting and querying more efficient.

3. Base62 Encoding for Compact URLs

The ID generated by the Snowflake ID generator is 13 digits long (e.g., 1780836574254). We then perform Base62 encoding on this ID, converting it into a much shorter 7-character string (e.g., VLrUd4Q).

Rationale for Choosing Base62:

- Shorter: Each character in Base62 can represent a larger number, which helps compress the ID length.
- Safe: Other encodings with stronger compression capabilities, such as Base64 and Base85, use special characters like + and /, which have specific meanings in URLs or require additional encoding (e.g., + is converted to %2B, / is converted to %2F), potentially increasing the URL length. In contrast, Base62 only uses letters and numbers, requiring no further encoding, ensuring a fixed final length that better aligns with our objectives.
- Scalable: As the number of stored URLs increases, the larger character space reduces the risk of collisions.

Why not directly use the concatenation of hash values?

1. Length is too long: Common hashes (e.g., SHA-256) have long outputs (usually 64 characters), and concatenating them is not suitable for storage and transmission, which doesn't align with our goals.
2. Performance issues: Calculating hash values incurs computational cost, and when the number of users and URLs increases drastically, it can lead to significant overhead.

3. Character set limitations: Hashes typically use hexadecimal or other character sets, making them difficult to compress effectively, and they are less efficient than Base62.
4. Poor scalability: As the number of IDs increases, managing concatenated hashes becomes more complex, while Base62 scales better and avoids collisions.

4. URL Validation Using Regular Expressions

To prevent storing malformed URLs, the system applies regular expression-based validation before accepting any input.
Reference: HTTPS://STACKOVERFLOW.COM/QUESTIONS/3809401/WHAT-IS-A-GOOD-REGULAR-EXPRESSION-TO-MATCH-A-URL

Regex Pattern Used
`^(https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}`
`www\.[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}`
`https?://(?:www\.|(?!www))[a-zA-Z0-9]+\.[^\s]{2,}`
`www\.[a-zA-Z0-9]+\.[^\s]{2,})$`

Pattern Breakdown

Pattern	Explanation
`^`	Anchors the match to the beginning of the string.
`https?:\/\/`	Matches the `http://` or `https://` protocol. The `s?` makes the "s" optional.
`(?:www\.|(?!www))`	It allows the presence of `"www."` but explicitly prevents a URL from starting with `"www"` if it is not immediately followed by a dot (`"."`). For example, it blocks `"wwwweb.com"` while allowing `"www.example.com"` .
`[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]`	This matches the domain name, which must start and end with an alphanumeric character, and may contain hyphens in between.
`\.[^\s]{2,}`	Matches a dot (`.`) followed by at least two non-space characters, which represent a valid top-level domain (e.g., `.com` , `.org`).
`)$`	Matches a closing parenthesis and asserts that it occurs at the end of the string (due to the `$` anchor).

5. Multi-User URL Shortener Design (Future Implementation)

We have thought of some ideas to implement when the design is for a great group of users:

1. Distributed systems: You can employ the code on a distributed environment so that more users can be handled.
2. Database Storage: By storing the URLs in a database they will be properly stored even if systems are rebooted.
3. Authentication: By implementing authentication users can keep shortened URL ownership and maybe change or remove shortened URLs.

6.Additional Features and /stats Endpoint

The service includes a $/stats/ < id >$ endpoint that allows users to retrieve usage statistics for each shortened URL. This endpoint returns data such as:

- Total Clicks: The number of times the shortened URL was accessed.
- Last Accessed Timestamp: The most recent time the URL was visited.
- Creation Timestamp: The time the short URL was generated.
- This feature provides valuable insights into URL usage and can be extended to track user demographics, referrer data, and geolocation analytics in a full-scale deployment.

7. Contribution Breakdown

Team_Member	Contribution
Michiel	- Implemented Snowflake ID. Base62 encoding
Zifeng Ma	- Implemented Snowflake ID. Wrote regex validation and handled concurrency issues and drafted report
Jay	- Developed initial Flask code and first tested endpoints - Integrated /stats endpoint and finished report - Wrote README file

While tasks were divided, all members collaborated on debugging, testing, and refining the implementation.

8. Conclusion

This project successfully implements a scalable, efficient URL shortening service using a Snowflake ID generator, Base62 encoding, and URL validation. The system supports high-concurrency access, ensures uniqueness, and provides insightful analytics through the /stats endpoint.
Future improvements could include user authentication, persistent database storage, and custom short links to expand functionality. By following RESTful design principles and best practices, the service is well-equipped for real-world deployment and large-scale use.