UNIVERSITY OF AMSTERDAM

Assignment 1 Report - Web Service and Cloud Based Software

Group 13

February 2026

1 Implementation

1.1 In memory mapping URL - Short ID

The Mapping of URL and its associated ID is stored in-memory as a hash table (Key - Short ID, Value - URL).

1.2 Base 64 design with an example

The URL shortener has an internal counter that is used to keep track of which ID. The counter starts at 0 and increments before creating a short URL. The value of the counter is base 64 encoded and the result is used as short URL. Our base 64 implementation uses capital and lowercase alphabets, numbers, dash and underscore. Here, `A` has value 0, `B` has value 1, `a` has value 26, `-` has value 62 and _ has value 63. Each iteration, the algorithm calculates the counter modulo 64, looks up the corresponding character, appends it to the result and divides the counter by 64 for next iteration.

Using base 64 is more efficient than decimal for short URLs. The number 100,000 is 6 characters long in decimal. That number encoded as base 64 is `Yag`, which is only 3 characters.

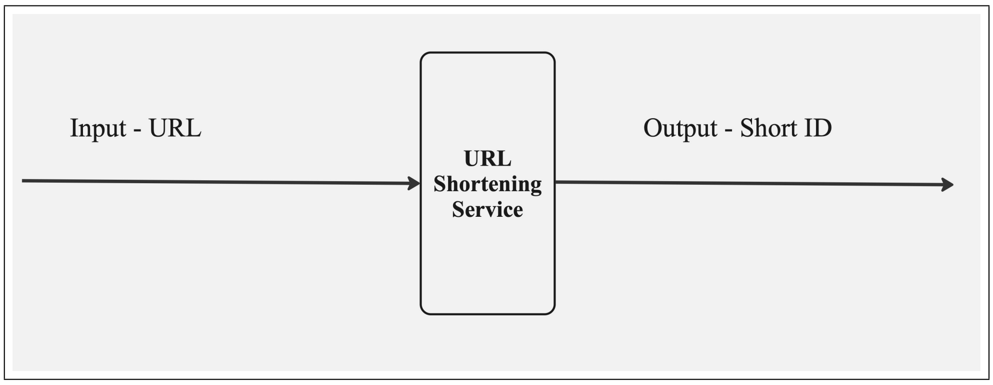
1.3 URL Regex validation

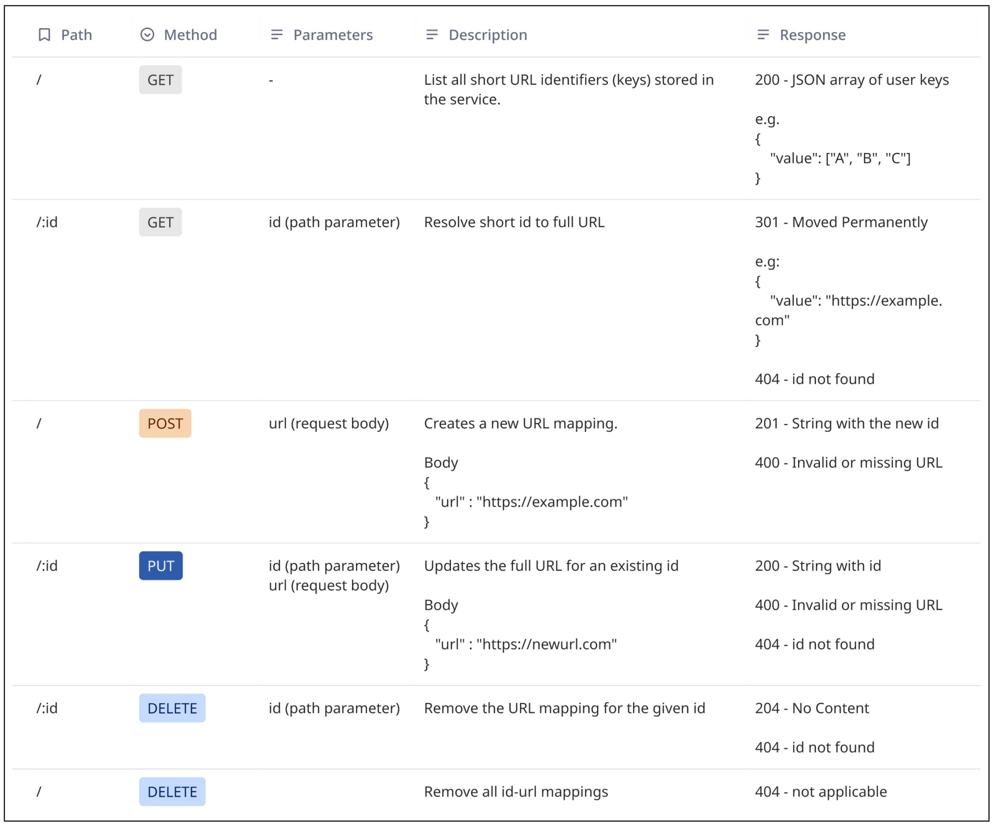
We validate the incoming url with a matching regex filter, which looks like this:

```
("((http|https)://)(www.)?"
+ "[a-zA-Z0-9@:%._\\+~#?&//=]"
+ "{2,256}\\.[a-z]"
+ "{2,6}\\b([-a-zA-Z0-9@:%"
+ "._\\+~#?&//=]*)")
```

`((http|https)://)(www.)?` enforces the URL scheme. It requires the URL to begin with either `http://` or `https://`, and optionally allows a `www.` prefix. This rules out URLs without an explicit HTTP(S) scheme (e.g., `example.com`) and non-HTTP schemes.
`[a-zA-Z0-9@:%._+~#?&//=]{2,256}` matches the main "host" portion with a length constraint of 2 to 256 characters. The character class permits alphanumeric characters and a set of commonly observed URL symbols. In practice, this aims to accept typical domain strings such as `example`, `my-site123`, and similar variants.
`.[a-z]{2,6}\b` requires a dot followed by a top-level domain consisting of 2 to 6 lowercase letters (e.g., `.com`, `.org`, `.nl`). The word boundary `\b` ensures that the TLD ends cleanly before any subsequent path/query part.
`([-a-zA-Z0-9@:%._+~#?&//=]*)` matches the optional remainder of the URL after the domain, i.e., the path and/or query string. It allows zero or more characters from a set that includes typical path

separators (`/`) and query delimiters (`?`, `&`, `=`), enabling URLs such as `/wiki/Dijkstra_algorithm` or `?id=1&sort=asc`.





Path	Method	Parameters	Description	Response

/	GET	-	List all short URL identifiers (keys) stored in the service.	200 - JSON array of user keys
				
				e.g.
				{
				"value": ["A", "B", "C"]
				}

/:id	GET	id (path parameter)	Resolve short id to full URL	301 - Moved Permanently
				
				e.g:
				{
				"value": "https://example.com"
				}
				
				404 - id not found

/	POST	url (request body)	Creates a new URL mapping.
			
			Body
			{
			"url" : "https://example.com"
			}
				201 - String with the new id
				
				400 - Invalid or missing URL

/:id	PUT	id (path parameter)
		url (request body)
			Updates the full URL for an existing id
			
			Body
			{
			"url" : "https://newurl.com"
			}
				200 - String with id
				
				400 - Invalid or missing URL
				
				404 - id not found

/:id	DELETE	id (path parameter)	Remove the URL mapping for the given id	204 - No Content 404 - id not found
/	DELETE		Remove all id-url mappings	404 - not applicable

2 Multiple users

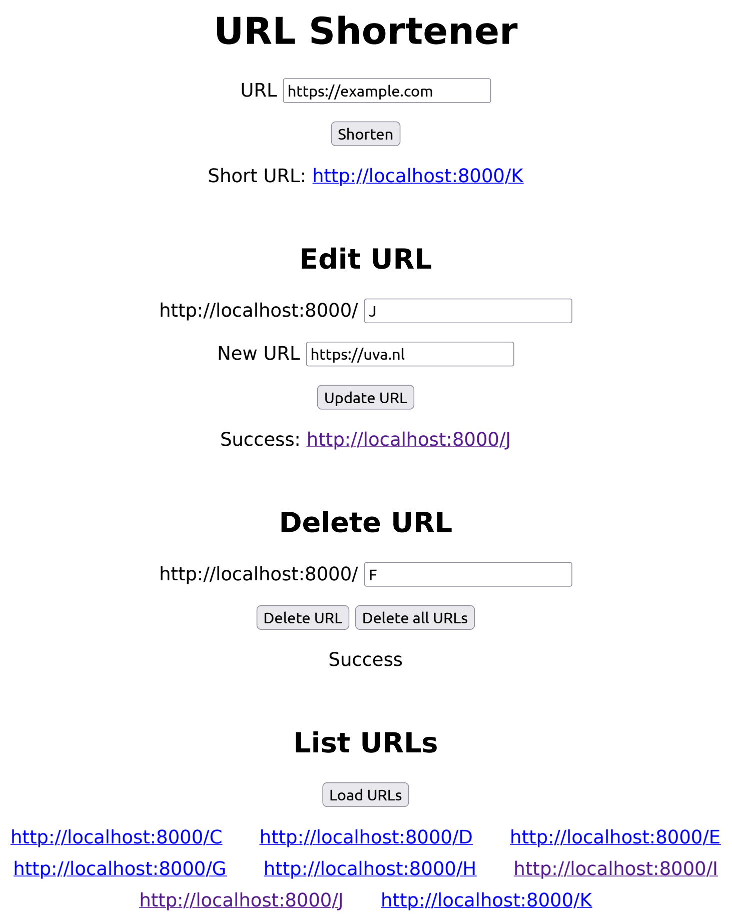
To ensure correct behavior under concurrent requests, the service employs a single thread lock together with a monotonic counter for ID generation. All operations that access or modify the shared in-memory mapping are executed within the critical section protected by the lock, preventing race conditions when multiple requests attempt to create, update, or delete entries simultaneously. The short identifier for each new URL is derived from an ever-increasing counter, which guarantees that every generated ID is unique without requiring collision detection. By combining serialized access to the storage with deterministic counter-based ID generation, the service can safely handle multiple users issuing requests at the same time while maintaining a consistent and correct mapping between identifiers and URLs.

3 Bonus

3.1 Frontend UI

We implemented a frontend UI to make URL shortening more user-friendly. The UI is accessible `http://localhost:8000/` when the page is requested with header `Accept: text/html`. As shown in Figure 1, the UI has options for creating new URLs, editing existing ones, deleting URLs and showing all URLs. All UI options use the REST API that the server provides.

The server parses the `Accept` header to determine wether to return a JSON object containing all URLs or the UI. This header is sent in HTTP requests by clients to tell servers what type of response they expect. Browsers set the value of this header to `text/html`, among a few other types. The provided test suite doesn't send this header. The server serves the UI when `text/html` or `text/*` have a higher *quality value* than `application/json` and `application/*`. When the header is absent or when `application/json` or `application/*` have a higher quality value, a JSON with all URLs is served instead.



URL Shortener

URL https://example.com

Shorten

Short URL: http://localhost:8000/K

Edit URL

http://localhost:8000/ J

New URL https://uva.nl

Update URL

Success: http://localhost:8000/J

Delete URL

http://localhost:8000/ F

Delete URL Delete all URLs

Success

List URLs

Load URLs

http://localhost:8000/C http://localhost:8000/D http://localhost:8000/E
http://localhost:8000/G http://localhost:8000/H http://localhost:8000/I
http://localhost:8000/J http://localhost:8000/K

Figure 1: Screenshot of the frontend UI.

3.2 MongoDB Database

We enabled the service to persist all URL mappings in MongoDB to ensure durability across application restarts. Two collections are used. The mappings collection stores the actual data, where each document contains the shortened identifier as the primary key (_id) and the corresponding original URL (url). This design allows constant-time lookup by identifier and leverages MongoDB's built-in index on _id for efficient retrieval.

A second collection, counters, maintains a single document that tracks a monotonic sequence number used for ID generation. When a new URL is created, the service performs an atomic $inc operation on this document to obtain the next sequence value, encodes it using Base64 to produce a short slug, and inserts the mapping into mappings. Storing the counter in MongoDB guarantees that ID generation remains consistent and unique even after server restarts and under concurrent requests, without requiring application-level locks.

CRUD operations directly interact with MongoDB: reads use find_one(), updates use update_one(), deletions use delete_one(), and listing identifiers queries only the _id field. This approach replaces the previous in-memory dictionary while preserving the same API behavior and ensuring persistent, concurrent-safe storage.

Note that the storage layer is abstracted through an abstract_storage interface, allowing the service to remain independent of any specific backend. All core operations interact only with this abstraction, so the HTTP API behavior is unaffected by the underlying persistence mechanism. If the MongoDB database becomes unavailable, the application falls back to an in-memory implementation, ensuring that the service continues to operate with RAM-based storage without requiring changes to the route logic.

4 Division of work

Members	**Contribution**
Quincy Koper	Base 64 encoding/decoding, frontend UI and initial URL validation using `urllib.parse`
Albert Bao	The implementation and deployment of MongoDB database, add/delete/list/create/... functionalities
Harikrishna	API Route handlers, Implemented support for reading settings (like ports, database info and bonus flag) from external config files.

Table 1: Members and their contributions