Assignment 1: A RESTful service

Group 33

Andrei Dragomir 15897966
Vivek A Bharadwaj 15927733
Arca Kutlu 14719789

February 10, 2025

1 Implementation

This implementation provides a URL shortener service using Flask. Users can submit long URLs via a REST API, receiving shortened URLs in return. The system ensures that only valid URLs are accepted and prevents duplicate storage. The implementation supports both automatically generated short IDs and user-defined custom IDs.

The way we interpreted the two endpoint specifications provided (namely "/" and "/:id") led us to the following understanding: The "/" endpoint offers functionality on the general data storage of the URL mappings, with the **GET** and **DELETE** methods returning or clearing all the URL mappings stored respectively and the "/:id" endpoint offers specific management over a single entry of the stored URL mappings. Table 1 has clear specifications on what cases we are handling for each method and endpoint.

Important design decisions that we made during the implementation was that we did not allow for the storage of duplicate long URLs even if they had different unique URLs. The reason for this was because we considered that this (single instance, single user) service would be designed to be as lightweight as possible. Although this check could introduce lookup overhead on a large number of entries, as opposed to just storing the respective duplicate, we managed to assess the complexity of the search method used and concluded that we are dealing with a O(n) complexity which is acceptable.

On the actual generation of the short ID for all URLs provided, we used a Base62 code generation of length 6. The base 62 stems from the fact that each character in the code to be generated is sourced from a set of characters containing 26 lowercase letters (a-z), 26 uppercase letters (A-Z) and 10 digits (0-9). This implies that there are around 56 billion possible combinations, therefore collisions with IDs already in use are highly unlikely. However, although that is the assumed case, we still check for the availability of the ID and re-generate if a collision is observed.

Lastly, in order to check the validity of the URLs provided, we used a regex which matches a string starting with *http://* or *https://* and then containing only characters other than spaces, double quotes, less than and greater than symbols. We are aware that it is a very limited check for a URL format but we thought it would be sufficient for the purpose of this Assignment. If it were for this URL validation to be used in production, we would have out-sourced a library for validation such as rfc3986.

2 Handling multiple users

For handling multiple users, we would implement two new functionalities to the service. The first is the user having the option to keep the URL-mapping private or mark it as global to let other users be able to use the same mapping as well. The second functionality is an authentication system which prevents users from modifying or deleting other users' private URL-mappings. With scalability and query efficiency in mind, we designed a database with three tables; `LongURLs(id, long_url)` to store unique long URLs, `ShortURLs(id, short_url, long_url_id, is_global` to link the short and long URLs as well as `is_global` variable to indicate whether the mapping is private or public, and `UserShortURLMapping(user_id, short_url_id)`, which keeps track of the users private URLs.
If a user would **POST** a global entry, the system would first search the `id` from `LongURLs` in `ShortURLs(long_url_id)`. If there is no match in either one or both tables, a new entry would

be created for the missing tables, otherwise the existing global URL would be returned. For private **POST**, the system would create the new entry in a similar fashion, but would also map it in `UserShortURLMapping`. By giving the user an already existing global short URL, we avoid unnecessary duplicate entries in the database, whilst still giving the option for creating private entries.
GET, **PUT** and **DELETE** check if the private URL belongs to the user in `UserShortURLMapping` before proceeding with the operation. **GET** returns the user-specific short URL, and if it fails, the system then searches global URL mappings. For **PUT** operations, the URL mapping is only changeable if the mapping belongs to the user. Lastly, **DELETE** operations delete the user's association with the mapping, meaning that users who have been assigned with the global URL will not suddenly lose it. Private URLs with no user ID's will be marked for deletion.
For the implementation of the user id itself, creating a login system would be excessive and inconvenient for users. For a more seamless and streamlined authentication, the user will receive their id, which is generated the same way as their short URLs, when sending a **POST** request. This id can be stored locally on the machine. However, these keys are susceptible to being tampered with. Using the `make_response` Flask library extension, we would be able to create HTTP-only cookies, with the added bonus the keys having expiration options, which could be used to implement extra features like the user deleting their "account", or inactive user ids and their private URLs being deleted automatically after a certain period of time.

3 Bonus Implementations

1. **Custom Short URLs:** One of the bonus functionalities added to our URL shortener is the ability to create custom short URLs. This feature allows users to specify a desired short ID instead of using a randomly generated Base62 encoded string. When a user submits a URL along with a 'custom_id', the system first checks if the provided ID is available in the memory dictionary. If the custom ID is unique, it is assigned to the URL. Otherwise, an error response is returned indicating that the ID is already taken. If no custom ID is provided, the system by default generates a random 6 character long Base62 short ID.

2. **Expiry Time for Shortened URLs:** Another functionality we added to the project is the ability to set an expiry time for shortened URLs. Users can define an expiration timestamp in the format **"YYYY-MM-DD HH:MM:SS"** which is then converted into a UNIX timestamp and stored along with the original URL. When a user attempts to access a shortened link, the system checks whether the expiry time has passed. If the link has expired, it is removed from the storage, and a "410 Gone" response is returned to indicate that the resource is no longer available. If no expiration time is provided, the short URL will be stored indefinitely.

3. **Automatic Cleanup of Expired Links:** We have also implemented an automatic cleanup mechanism that periodically removes expired URLs. This is done through a background thread that runs at regular 10 minute intervals, scanning the stored URLs for any that have surpassed their expiry time. If expired entries are found, they are deleted to free up space and prevent unnecessary checks.

4 Work distribution

Vivek: Bonus Implementation, Encoding Algorithm, URL Validation Regex
Arca: Handling Multiple Users Question
Andrei: Base Implementation

Method	Endpoint	Description (Response Codes & Reasons)
POST	`/`	• **201 (Created)**: A new short URL is successfully generated. • **400 (Bad Request)**: The provided URL is missing or has an invalid format. • **409 (Conflict)**: The provided URL already exists in storage.
GET	`/`	• **200 (OK)**: Returns a list of all stored short IDs. • **200 (OK, but empty)**: No short URLs exist in storage (returns `null`).
DELETE	`/`	• **404 (Not Found)**: All stored mappings are deleted (even if empty).
GET	`/<id>`	• **301 (Moved Permanently)**: Redirects to the original URL. • **404 (Not Found)**: The short ID does not exist.
DELETE	`/<id>`	• **204 (No Content)**: The short ID is deleted successfully. • **404 (Not Found)**: The short ID does not exist.
PUT	`/<id>`	• **200 (OK)**: The original URL is successfully updated. • **400 (Bad Request)**: The request is missing the new URL or contains an invalid URL format. • **404 (Not Found)**: The short ID does not exist.

Table 1: API Endpoints and Response Codes (Basic Implementation)

Method	Endpoint	Description (Response Codes & Reasons)
POST	`/`	• **400 (Bad Request)**: The expiration time has an invalid format. • **409 (Conflict)**: The custom ID is already in use.
PATCH	`/<id>`	• **200 (OK)**: The original URL is successfully updated. • **400 (Bad Request)**: The request is missing one of the two possible values for modification (the new custom id or the new expiry time). • **400 (Bad Request)**: The new expiry time is not a valid format. • **409 (Not Found)**: The custom ID provided is already in use.

Table 2: API Endpoints and Response Codes Extended (BONUS)