1 Design & Implementation

In this exercise, a URL shortening web service is to be designed and implemented. The requirements on the URL-Shortener service are as follows:

- Map multiple URL's to unique ID's, which should be as short as possible and only be extended when necessary
- The service should validate that the URL to be shortened is well-formed
- Maintain the mappings in a local in-memory state or a persistent datastore that allows to maintain state between restarts

The ID generation algorithm that is used to map (arbitrarily long) URL's to ID's that are as short as possible is the first design decision that shall be discussed. A naive approach to generate such ID's is to use a generator function that yields the next permutation of n characters from an alphabet A, using a deterministic mapping algorithm that produces a countable sequence and keeps track of all the ID's that have been mapped already. Whenever $|A|^n$ ID's of length n have been assigned, the length of the ID's in increased to $n+1$. While this generates the shortest possible ID's when no ID's are deleted, it violates the requirement when shorter ID's are deleted and become free again. Also, such an approach requires synchronized access to the generator function, which makes scaling to threads, processes, and different instances impossible. Ideally, web services are autonomous, stateless, and do not have dependencies on other services, which is why we decided to use a one way hashing function $f : U \mapsto I$ that gives an injective mapping from a URL $url \in U$ as the input to an ID $id \in I$ with a very high probability. Since we want ID's to be as short as possible, we increment $j \in \mathbb{N}, 0 < j < |id|$ and check if the sub string $id_0...id_j$ can be used, which results in a insert complexity of $O(k)$ when using a hash table data structure, where k is the maximum length of any hash the hashing algorithm produces. Our implementation uses the MD5 hashing algorithm, which uses 128 bits or equivalently 32 digit hex codes. We further assume that every well formed URL can be mapped to one of the 2^{128} possible hashes, so that more than 32 digit hashes are never required. Neglecting very rare hashing collisions, this still does not give a strong invariant for the shortest possible ID but but gives reasonably short ID's in the average case while allowing high scalability, since only the shared data store has to be stateful. However, it should be noted that our implementation might benefit from not checking every $j < j_{min}$ on insert if a minimum ID length j_{min} is acceptable.

Secondly, the design decision to use an abstract storage backend interface that can be used to plug in different storage systems shall be discussed. To accomplish greater flexibility, the REST API call handlers use an abstract interface for managing the ID to URL mappings:

- `set(key, value, user_id, exists_ok=False)`
- `get(key)`
- `get_all(user_id)`
- `update(key, value, user_id)`
- `delete(key, user_id)`
- `delete_all(user_id)`

Our implementation includes two different specializations of this abstract interface, namely an in memory store based on Python dictionaries and a persistent implementation that uses Redis[1], a popular key value database, for storing ID and URL mappings.

For the implementation, we chose to use the popular Flask microframework [2] written in Python, which allows a convenient implementation of the REST HTTP handlers. We assumed that the `PUT` call is supposed to change the URL a given ID maps to, so that the creator of a link can change the redirection later to avoid dead links, which is why we added a *url* parameter for this call that was not originally mentioned in the table given in the assignment description.

Considering the validation of URL's, we decided to use the *validators* package that allows for convenient validation of a URL's correctness.

2 Multi-user URL-shortener

First of all, the URLShortener service does handle the publishing and sharing of short aliases for long URL's (`GET /:id`) out of the box, since the generated ID's are supposed to be available to all users. The challenge in implementing multi user support is maintaining state for the `PUT /:id`, `POST`, and `DELETE` requests, because two users that want to shorten the same URL must generate two distinct ID's that can be changed and deleted independently of each other. To implement this, we incorporated a unique identifier for a user with the URL that is hashed to create a short ID that is unique with very high probability. Our implementation uses the callers IP address if no explicit user id was supplied as a parameter to uniquely identify a user. However, it must be noted that the IP address can be easily forged and users that share the same public IP as it is common with NAT based local networks will be able to manage each others ID's, which is the reason why a production deployment should rather use a proper user id parameter to identify users. Subsequently, the backend has to keep track of who owns which shortened URL so that it can check if a user is allowed to perform a `DELETE` or `PUT` operation on a specific shortened URL. Also, the `DELETE` call that deletes all keys and the `GET` call that lists all keys uses this second mapping to show or delete only the shortened links owned by the respective user. In conclusion, an adequate implementation would use a combination of server-side and client-side state, where the user sends his user id with every request and the web service uses an external state to keep track of who owns which ID's and generates unique ID's for different users.

References

[1] Redis key value database. `https://redis.io/`. Accessed: 2021-04-11.

[2] Flask microframework. `https://flask.palletsprojects.com/en/1.1.x/`. Accessed: 2021-04-11.