Tong Wu, Zhaolin Fang, Leonardo Kuffo

Group 9: Assignment 1

Shorty URL: A simple URL Shortener Web Service

I. DESIGN | Our URL shortener service follows the REST (REpresentational State Transfer) design pattern architectural style. Hence, it is a RESTful Web Service. The client (i.e. service consumer) accesses the resources in the WS using a unique URL. The web interactions are done using the HTTP protocol following the main HTTP methods for CRUD operations: Create resources (POST), retrieve resources (GET), update resources (PUT) and delete resources (DELETE). The design of the Web Service methods and their contract is described in Table 1.

PATH	METHOD	PARAMETERS	METHOD DESCRIPTION	RESPONSES
/<id>	GET	**id**: shortened url id [int] *(Query Parameter)*	On success, redirects user to the URL associated to the shortened id	302: Redirecting to original url 404: id not found in the registry 500: Unknown error
/<id>	PUT	**id**: shortened url id [int] *(Query Parameter)* **url**: New url [string] *(Body Parameter)*	On success, updates the URL associated to the shortened id	404: id not found in the registry 400: Bad format url 200: Successfully updated 500: Unknown error
/<id>	DELETE	**id**: shortened url id [int] *(Query Parameter)*	On success, deletes the URL associated to the shortened id from the registry	404: id not found in the registry 204: Successfully deleted 500: Unknown error
/	POST	**url**: New url [string] *(Body Parameter)*	On success, add the URL to the registry and associate it to a shortened id. Returns the shortened id.	201: `id` 400: Bad format url 500: Unknown error
/	GET	*None*	On success, returns a JSON object with the shortened ids as keys and the original URLs as values.	200: `{ id: "value" }` 500: Unknown error
/	DELETE	*None*	On success, empty the URLs registry.	200: Registry emptied 404: Registry already empty 500: Unknown error
/home	GET	*None*	Return HTML template of web client.	200: Returns HTML template

Table 1: Web Service Specifications

II. IMPLEMENTATION | We implemented our URL shortener service using Flask. A light-weight Python-based web development framework. As Table 1 shows, we mainly implemented 6 methods for the service consumers to allow them to create, retrieve, update and delete their shortened URLs. We do so by **associating** the original URLs to **unique IDs** and storing them inside the server. Unique IDs are managed in the server as a **global integer counter** that starts from 0 and increments when the clients send new URLs to shorten. These unique IDs and the original URLs are associated inside a global **python dictionary** stored in memory. In this dictionary, keys are the unique shortened URLs IDs and the values are the original URLs (e.g. `{0: "https://en.wikipedia.org/", 1: "https://twitter.com/home"}`). Hence, when a user accesses the method `GET /1`, they will be redirected to `https://twitter.com/home`. Furthermore, in the POST and PUT methods we checked the correctness of the URL in the user's request before saving it into the registry, to avoid the procedural errors. We do so using a third party package named *validators*[1]. Finally, we implemented a web client with a user interface to facilitate access to our service. We deployed our service using a third party cloud provider: Heroku[2]. Which provides us with a public URL in which our service and UI tool is hosted: https://shortyurl-ws.herokuapp.com/home.

III. LIMITATIONS | The simplicity of our Web Service comes with limitations. First, the storage of the resources is done in memory. Hence, the resources are volatile and they will be lost on server restart. Second, the unique IDs are sequential. Hence, one can navigate easily through other users' shortened URLs. However, since this Web Service is intended for one-user use only, we can afford this shortcoming.

[1] https://validators.readthedocs.io/en/latest/
[2] https://www.heroku.com/

IV. QUESTION 3 | In order to implement a URL-shortener for multiple users we need to maintain the state of the system in a non-volatile storage. In other words, we need a database. It is important to note that by doing so we are not making our RESTful service stateful. We are persisting the state of the resources that the service manages. Each method call is still totally independent from each other.

Under our database we must store the **users** registered in the system and the **URLs** they have shortened. For each user we should store the username, their password and any other extra information at the discretion of the stakeholders. For each URL shortened we should store the identification of the user who created the shortened URL, the shortened URL id, the original URL, a timestamp of creation, a timestamp of update, and any other extra information at the discretion of the stakeholders. Since our structure schema is simple (i.e. only two structures: users, urls), we could use a NoSQL database based on object- or document-storage such as Firestore or MongoDB. NoSQL databases scale and perform better for non-complex structures such as ours.

To implement this feature in our current RESTful Service we would have to refactor the methods (i.e. endpoints) to take into account the users in the implementation. One efficient way to perform this refactor would be to add a **middleware** of authentication that is executed before calling any of the service methods. This middleware would add the user information on the request for the main method to use. Next, the methods must query and manage in the database only the URLs of the authenticated user coming from the authentication middleware. The main logic and flow of the methods will remain the same. Changes are only going to happen in the way the information is retrieved and managed.

V. MEMBERS CONTRIBUTION

Tong Wu	*Three methods, Section I of report*
Zhaolin Fang	*Three methods, Section II of report*
Leonardo Kuffo	*Answer to Question 3, UI tool development and README file*