Assignment 1: RESTFUL service

Lab Group 32

Ivy Rui Wang (iwa211), Delong Yuan (dyu208)

I. Design

We developed a RESTful service to implement a URL shortening service using the Flask framework. We chose to implement the URL shortening service using Flask, a lightweight and flexible web framework for Python. Flask provides easy-to-use tools for creating RESTful APIs, making it an ideal choice for this task. The objective of the service is to act as a repository of URLs, where each URL is mapped to a unique and short identifier. This allows other services or programs to call the REST API to resolve the short identifier to the full URL it refers to, as well as manage the URLs by adding new mappings, deleting old mappings, updating current mappings, and viewing all mappings.

Table I summaries the specifications of each endpoint in the URL shortening service.

II. Implementation details

A. Generate ID

When generating IDs using a combination of randomness and incrementing, we first create a random part for the ID. This random part can be generated using random characters or digits to ensure uniqueness and unpredictability. However, there is a possibility of collisions when generating random IDs, especially as the number of IDs increases. To mitigate this risk, we append an incrementing part to the random part. This incrementing part ensures uniqueness by incrementing the ID each time a new one is generated, avoiding collisions even if the random part is the same. The incremental portion consists of a number encoded in base62. The advantage of encoding the incremental portion in base62 is that it allows us to represent a large range of numbers using a compact set of characters. Base62 encoding uses a combination of alphanumeric characters (A-Z, a-z, 0-9), which maximizes the efficiency of representing numbers in a shorter format compared to decimal or binary encoding.

B. Check URL Validity

```python
def is_valid_url(url):
    """Check if the provided URL is valid."""
    regex = re.compile(
        r'^(?:http|http)s?://'  # http:// or https://
        r'(?:(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+(?:[A-Z]{2,6}\.?|[A-Z0-9-]{2,}\.?)|'  # domain...
        r'localhost|'  # localhost...
        r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})'  # ...or ip
        r'(?::\d+)?'  # optional port
        r'(?:/?|[/?]\S+)$', re.IGNORECASE)
    return re.match(regex, url) is not None
```

Fig. 1: URL Validation. Check if the provided URL is valid according to a regular expression pattern.

^(?:http|http)s?:// matches the protocol part of the URL, such as "http://" or "https://". **(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+** matches the domain name, allowing for alphanumeric characters and hyphens in the subdomains. **(?:[A-Z]{2,6}\.?|[A-Z0-9-]{2,}\.?)|** matches top-level domains such as ".com", ".org", ".net", etc. **localhost|** matches "localhost" as a valid domain. **\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})** matches IP addresses. **(?::\d+)?** matches an optional port number. **(?:\?|[/?]\S+)$** matches optional path segments in the URL.

C. Threading Locks

In our implementation, we used the *threading* package to add locks to the dictionary storing the mapping relationships. This allowed us to ensure thread safety and prevent data corruption when multiple users simultaneously create or update mappings in the dictionary. By adding locks, we enforce

Endpoint	Path	Method	Body of Request	Return Value
create_short_url	/	POST	{'value': $long\_url$}	{'id': id}, 201 {'error': 'Long URL is required in the request body'}, 400 {'error': 'Invalid URL format'}, 400
list_urls	/	GET	-	id/url pairs, 200
get_long_url	/⟨id⟩	GET	-	{value: long_url}, 301 {'error': 'Short URL not found'}, 404
update_long_url	/⟨id⟩	PUT	{'value': new_long_url}	{'value': new_long_url}, 200 {'error': 'New URL is required in the request body'}, 400 {'error': 'Invalid URL format'}, 400 {'error': 'Given ID is not existed'}, 404
delete_url	/⟨id⟩	DELETE	-	{'message': id has been deleted'}, 204 {'error': 'id not found'}, 404
delete_all_url	/	DELETE	-	{'message': 'All id/url pairs have been deleted'}, 404

TABLE I: Specifications of each endpoint in the URL shortening service.

mutual exclusion, meaning that only one thread can access and modify the dictionary at any given time. When a thread acquires the lock, it has exclusive access to the dictionary, ensuring that other threads are blocked from accessing it until the lock is released.

D. HTTP status code

1) HTTP 200: When a PUT request is made to update a mapping or when a GET request is made to list all mappings, a successful operation should return an HTTP status code of 200.

2) HTTP 201: When creating a new mapping successfully, we return an HTTP status code 201. The HTTP 201 status code signifies that the request has been fulfilled and resulted in the creation of a new resource on the server.

3) HTTP 204: When successfully deleting a long URL associated with an ID, the HTTP status code 204 is returned. The HTTP status code 204 indicates that the server has successfully processed the request and that there is no additional content to send in the response body.

4) HTTP 301: In our design, returning HTTP 301 along with the long URL indicates to the client that the original URL corresponding to the provided identifier has been permanently moved to the returned URL. This ensures proper redirection behavior in client applications and browsers.

5) HTTP 400: When sending a request with a missing or invalid parameter in the request body, an HTTP 400 status code is returned along with an error message indicating the issue.

6) HTTP 404: When querying for a specific ID that does not exist in the system, we return an HTTP 404 response.

III. RESPONSIBILITIES

Task	Ivy Wang	Delong Yuan
Specification Design	Collaborator	Collaborator
Code Implementation	Collaborator	Collaborator
Writing Report	Collaborator	Collaborator

TABLE II: Division of responsibilities between team members