Assignment - Quality Assurance for URL shortener

Problem: Design a service like TinyURL, a URL shortening service, a web service that provides short aliases for redirection of long URLs.

Solution: Basically we need a one to one mapping to get shorten URL which can retrieve original URL later. This will involve saving such data into database.

We should check the following things:

- What's the traffic volume / length of the shortened URL?
- What's the mapping function?
- Single machine or multiple machines?

Traffic: Let's assume we want to serve more than 1000 billion URLs. If we can use 62 characters [A-Z, a-z, 0-9] for the short URLs having length n, then we can have total 62^n URLs. So, we should keep our URLs as short as possible given that it should fulfill the requirement. For our requirement, we should use n=7 i.e the length of short URLs will be 7 and we can serve 62^7 ~= 3500 billion URLs.

Basic solution:
To make things easier, we can assume the alias is something like http://tinyurl.com/<alias_hash> and alias_hash is a fixed length string.

To begin with, let's store all the mappings in a single database. A straightforward approach is using alias_hash as the ID of each mapping, which can be generated as a random string of length 7.

Therefore, we can first just store <ID, URL>. When a user inputs a long URL "http://www.google.com", the system creates a random 7-character string like "abcd123" as ID and inserts entry <"abcd123", "http://www.google.com"> into the database.

In the run time, when someone visits http://tinyurl.com/abcd123, we look up by ID "abcd123" and redirect to the corresponding URL "http://www.google.com".

Problem with this solution:
We can't generate unique hash values for the given long URL. In hashing, there may be collisions (2 long urls map to same short url) and we need a unique short url for every long url so that we can access long url back but hash is one way function.

Better Solution:

One of the most simple but also effective one, is to have a database table set up this way:

```
Table Tiny_Url(
ID : int PRIMARY_KEY AUTO_INC,
Original_url : varchar,
Short_url : varchar
)
```

Then the auto-incremental primary key ID is used to do the conversion: (ID, 10) <==> (short_url, BASE). Whenever you insert a new original_url, the query can return the new inserted ID, and use it to derive the short_url, save this short_url and send it to cilent.

Code for methods (that are used to convert ID to short_url and short_url to ID):

```
# Python3 code for above approach

def idToShortURL(id):
    map = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    shortURL = ""

    # for each digit find the base 62
    while(id > 0):
        shortURL += map[id % 62]
        id //= 62

    # reversing the shortURL
    return shortURL[len(shortURL): : -1]

def shortURLToId(shortURL):
    id = 0
    for i in shortURL:
        val_i = ord(i)
        if(val_i >= ord('a') and val_i <= ord('z')):
            id = id*62 + val_i - ord('a')
        elif(val_i >= ord('A') and val_i <= ord('Z')):
            id = id*62 + val_i - ord('A') + 26
        else:
            id = id*62 + val_i - ord('0') + 52
    return id

if (__name__ == "__main__"):
    id = 12345
    shortURL = idToShortURL(id)
    print("Short URL from 12345 is : ", shortURL)
    print("ID from", shortURL, "is : ", shortURLToId(shortURL))
```

Time complexity : O(n)
Auxiliary Space : O(1)

Multiple machines:

If we are dealing with massive data of our service, distributed storage can increase our capacity. The idea is simple, get a hash code from original URL and go to corresponding machine then use the same process as a single machine. For routing to the correct node in cluster, Consistent Hashing is commonly used.

Following is the pseudo code for example,

Get shortened URL

- hash original URL string to 2 digits as hashed value hash_val
- use hash_val to locate machine on the ring
- insert original URL into the database and use getShortURL function to get shortened URL short_url
- Combine hash_val and short_url as our final_short_url (length=8) and return to the user

Retrieve original from short URL

- get first two chars in final_short_url as hash_val
- use hash_val to locate the machine
- find the row in the table by rest of 6 chars in final_short_url as short_url
- return original_url to the user

Other factors:

One thing I'd like to further discuss here is that by using GUID (Globally Unique Identifier) as the entry ID, what would be pros/cons versus incremental ID in this problem?

If you dig into the insert/query process, you will notice that using random string as IDs may sacrifice performance a little bit. More specifically, when you already have millions of records, insertion can be costly. Since IDs are not sequential, so every time a new record is inserted, the database needs to go look at the correct page for this ID. However, when using incremental IDs, insertion can be much easier – just go to the last page.

Automation Test Cases for URL Shortening Service:

Tools/Frameworks: Selenium with Python, JUnit, TestNG

1. Positive Test Cases:

Test Case 1: Successful URL Shortening

- **Description:** Verify that a valid long URL can be successfully shortened.
- Steps:
    1. Enter a valid long URL in the input field in the code.
    2. Run the code.

    3. Capture the generated short URL.
- **Expected Outcome:** Short URL is generated.

Test Case 2: Redirecting to Original URL

- **Description:** Confirm that the short URL redirects to the original long URL.
- Steps:
    1. Shorten a valid long URL.
    2. Open the generated short URL.
- **Expected Outcome:** Redirected to the original long URL.

Test Case 3: Custom Short URL Creation

- **Description:** Ensure that a user can create a custom short URL.
- Steps:
    1. Navigate to the URL shortening service.
    2. Enter a valid long URL and a custom alias.
    3. Click on the submit button.
    4. Capture the generated custom short URL.
- **Expected Outcome:** Custom short URL is generated and displayed.

2. Negative Test Cases:

Test Case 4: Shortening an Invalid URL

- **Description:** Validate that the system rejects an attempt to shorten an invalid or malformed URL.
- Steps:
    1. Navigate to the URL shortening service.
    2. Enter an invalid or malformed URL.
    3. Run the code.
- **Expected Outcome:** Proper error message is displayed.

Test Case 5: Custom Short URL Conflict

- **Description:** Confirm that the system handles a conflict when attempting to create a custom short URL with an existing alias.
- Steps:
    1. Shorten a valid long URL with a custom alias.
    2. Attempt to create another custom short URL with the same alias.
- **Expected Outcome:** Proper error message indicating alias conflict is displayed.

Test Case 6: Accessing Expired URL

- **Description:** Ensure that the system handles attempts to access a short URL that has expired.
- Steps:
    1. Shorten a valid long URL with a short expiration time.
    2. Attempt to access the short URL after it has expired.
- **Expected Outcome:** Proper error message or redirection to a predefined page.

3. Edge Cases:

Test Case 7: Maximum URL Length

- **Description:** Test the system's capability to handle the maximum length of a URL.
- Steps:
    1. Attempt to shorten a URL with the maximum allowed length.
- **Expected Outcome:** URL is successfully shortened and functional.

Test Case 8: Custom Short URL with Maximum Length

- **Description:** Verify that the system can handle a custom alias with the maximum allowed length.
- Steps:
    1. Attempt to create a custom short URL with an alias of maximum length.
- **Expected Outcome:** Custom short URL is successfully generated and functional.

4. Performance Test Case:

Test Case 9: Load Testing

- **Description:** Evaluate the system's performance under heavy traffic.
- Steps:
    1. Simulate a large number of concurrent shortening requests.
    2. Measure the response time for URL shortening.
- **Expected Outcome:** The service should handle the load efficiently without significant degradation in response time.

Reference: https://www.code-recipe.com/post/url-shortener

How to run?

1. Backend (Python using Flask):

Create a file named `app.py` with the following content:

```
pythonCopy codefrom flask import Flask, render_template, request, jsonify
from werkzeug.utils import redirect

app = Flask(__name__)

url_mapping = {}
counter = 1  # To simulate a unique identifier

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/shorten', methods=['POST'])
def shorten_url():
    global counter
    long_url = request.form['long_url']
    short_url = id_to_short_url(counter)
    url_mapping[short_url] = long_url
    counter += 1
    return jsonify({'short_url': short_url})

@app.route('/<short_url>')
def redirect_to_original(short_url):
    long_url = url_mapping.get(short_url, '/')
    return redirect(long_url)

def id_to_short_url(n):
    char_map = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    short_url = ""

    while n:
        short_url += char_map[n % 62]
        n //= 62

    return short_url[::-1]

if __name__ == '__main__':
    app.run(debug=True)
```

2. Frontend (HTML + JS):

Create a folder named `templates` in the same directory as `app.py`, and inside it, create a file named `index.html` with the following content:

```
htmlCopy code<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>URL Shortener</title>
</head>
<body>
    <h1>URL Shortener</h1>
    <form id="urlForm">
        <label for="long_url">Enter Long URL:</label>
        <input type="text" id="long_url" name="long_url" required>
        <button type="button" onclick="shortenUrl()">Shorten URL</button>
    </form>
    <p id="short_url"></p>

    <script>
        function shortenUrl() {
            var longUrl = document.getElementById('long_url').value;

            fetch('/shorten', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded',
                },
                body: 'long_url=' + encodeURIComponent(longUrl),
            })
            .then(response => response.json())
            .then(data => {
                document.getElementById('short_url').innerText = 'Shortened URL:
/' + data.short_url;
            })
            .catch(error => console.error('Error:', error));
        }
    </script>
</body>
</html>
```

Running the Application:

1. Open a terminal in the directory containing `app.py`.
2. Run the command: `python app.py` (Make sure you have Python installed).
3. Open your browser and go to `http://localhost:5000/`.