MICROSERVICES MAINTENANCE ISSUE WORKAROUND - BACKGROUND

Microservices architecture may be great in scalability but problems may occur especially in handling maintenance where services can be related to one another causing a chain reaction to the maintenance and testing process.

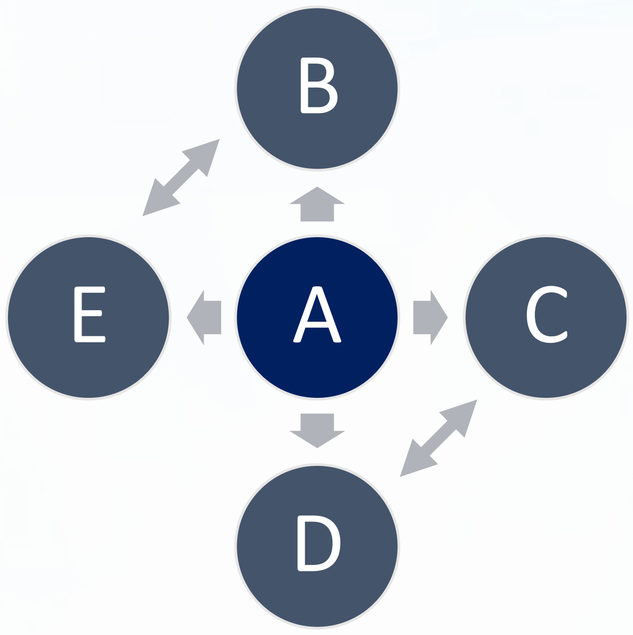

Figure 1.0 – Example of Microservices System, where each Letter denote a Service

In ***Figure 1.0*** above, let's take an example where **service A** will undergo some changes – thus will affect other services that depends on A. In this case, **service B, C, D and E will also require for testing** since there might be affected modules from the changes of service A. This will develop a higher maintenance cost every time there are changes even a small one as testing would be needed for other services related.

THE SOLUTIONS

Since testing is inevitable as preventing it could lead to potential error, thus a proper testing mechanism is required to ensure no repeating process occur that will make the maintenance cost to be higher. Below are possible solution to be considered.

a) Solution 1 – Automated Testing

The utilization of automated testing will be very suitable in this scenario. Using third-party testing tools such as ***Selenium*** or it's mobile built ***Appium*** will greatly reduce the maintenance cost. This is due to the fact that with *Selenium*, we can develop test script specifically for each of the modules in the service as long as the changes does not affect the related service's logic.

Hence, every services will have their very unique testing script that **can be executed repeatedly** especially when there are changes that theoretically may affect the service is conducted. Let's dive into an example of *python* script in ***Snippet 1.0***.

```
1. from selenium import webdriver
2. from selenium.webdriver.common.keys import Keys
3. from selenium.webdriver.common.by import By
4. from selenium.webdriver.support.ui import Select
5. import time

6. # Create a new instance of the Chrome driver
7. driver = webdriver.Chrome()

8. # Navigate to the webpage with the membership subscription form
9. driver.get("https://musicplaylist.com/membership-form")

10.# Locate the input fields and submit button
11.membership_type_dropdown = Select(driver.find_element(By.ID, "membership_type"))
12.duration_input = driver.find_element(By.ID, "duration")
13.submit_button = driver.find_element(By.ID, "submit_button")

14.# Select subscription type from dropdown
15.membership_type_dropdown.select_by_visible_text("Gold Membership")

16.# Enter duration
17.duration_input.send_keys("12")

18.# Click the submit button
19.submit_button.click()

20.# Wait for a few seconds to see the result
21.time.sleep(5)

22.# Close the browser window
23.driver.quit()
```

Snippet 1.0 – Example of Python Testing Script for Subscription Service

Line	Explanation
1-5	• Importing *Selenium* library
6-9	• Getting the driver of the internet browser used, this depends on the production environment. For mobile, it will be different.
10-13	• Locate the components in the user interface.
14-17	• Assign pre-defined value(s) in the components located.
18-23	• Execution of event by clicking button and observing the results before auto-closing the browser.

Snippet 1.0 shows a python script that utilize *Selenium* library to test **Membership Subscription Service**. ***Snippet 1.1*** shows the execution of the python script using operating system terminal.

```
python membership_test.py
```

Snippet 1.1 – Executing Python the Testing Script with Terminal

Now that as in the case study provided, we have service A, service B and Service C – each of them will have their own testing scripts for every modules within in and they all can be automated for us to detect any issues of errors.

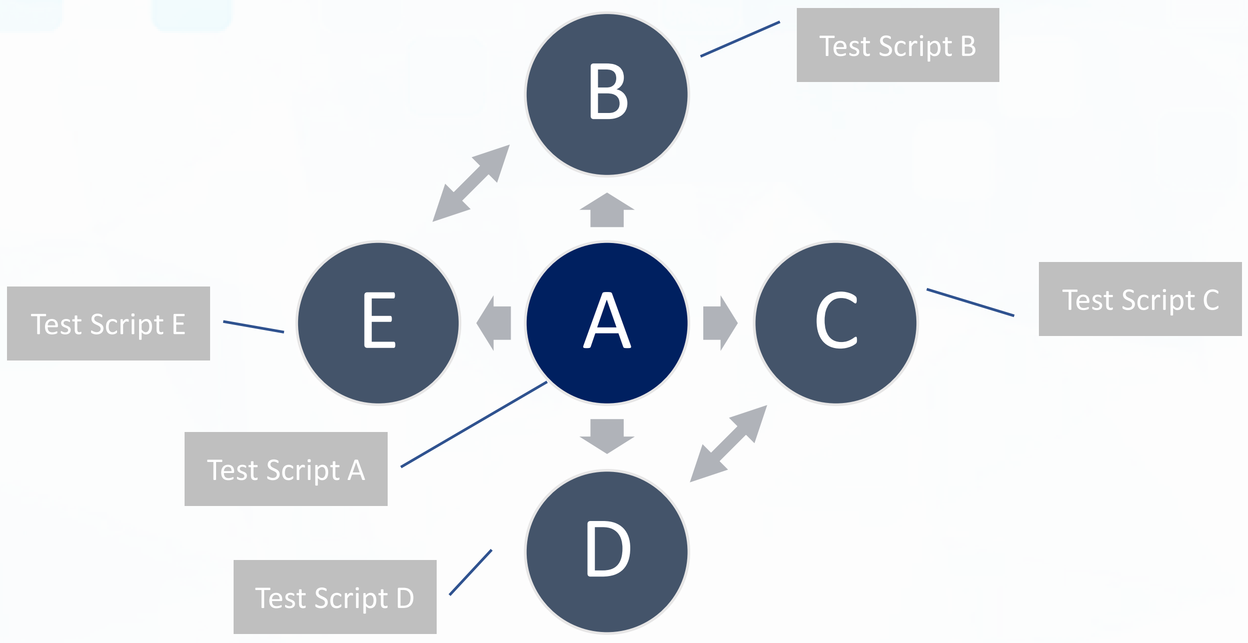

Figure 1.1 – Microservices with each Service assigned Unique Test Scripts

For example when there is an API standardization changes to Service A, theoretically service B and service C might be as well affected – in this scenario, using automated testing script will simply cut-off the maintenance time and changes only need to be done when there are actually errors affected by the changes in service A.

b) Solution 2 – APIs End Points Library

Since back-end errors usually occur due to the data differences or anomalies, thus the main sources of data would be from APIs of each services. By using a structure where the end points are stored in a single file, let's take for example – an encrypted JSON file, if there are any changes to the APIs' end points then the only thing we need to reconfigure is the JSON file itself.

In ***Snippet 1.3***, we can see how the JSON file would be and depending on what framework the system is using, we can fetch the APIs end points string. This method will enable if any changes are required to the end points of any services' API, then developer will only need to modified the JSON file without touching other services.

This however apply to such case only, if the end points involve different data output or result that is significant, thus thorough testing may be required for any services that depends on it.

```
{
  "ServiceA": {
    "endpoints": {
      "getProfile": "/user/profile",
      "update": "/user/update",
      "preferences": "/user/preferences",
      "history": "/user/history"
    }
  },
  "ServiceB": {
    "endpoints": {
      "playlist": "/recommendations/playlist",
      "trending": "/recommendations/trending",
      "genre": "/recommendations/genre",
      "personalized": "/recommendations/personalized"
    }
  },
  "ServiceC": {
    "endpoints": {
      "status": "/subscription/status",
      "upgrade": "/subscription/upgrade",
      "cancel": "/subscription/cancel",
      "billing": "/subscription/billing"
    }
  }
}
```

Snippet 1.3 – Example of JSON (JavaScript Object Notation) File containing End Points Library

Encrypting the JSON file would be a necessary step taken to add a security layer the end points from being exploited by exploiter who manage to get in.

In order to get the end point desired, we can do as below in ***Snippet 1.4***. In this snippet, we use the reference "getProfile" to get the end point string in getting user profile data.

```
const apiUrl = base_url + apiEndpoints.ServiceA.endpoints.getProfile;
fetch(apiUrl, {
     method: 'GET', … rest of the code
});
```

Snippet 1.4 – Example of Obtaining End Point String using Reference

If there are any addition, removal or modification to the end points – then developer would only need to modify the JSON file.

CONCLUSION

By combining these two solution, maintenance time and cost can be reduced as only minimal testing would be required to any of the services affected. Applying the methods discussed are also applicable before or even after development phase.