Updating Data Migration Scripts

Introduction

I am currently the newest member of a small team which is building a centralised Access Management service capable of meeting the access control requirements across all of our client's systems. An adjacent data migration service, also built and maintained by the team, provides a Jenkins pipeline for automated data migration to the Access Management database. A colleague had suggested it would be worth updating the migration script to remove any leading or trailing whitespaces from the values to ensure uniformity and avoid error, so a ticket was created and assigned to me.

The Development Process

The first step was to have a 'three amigos' meeting. This is an agile ceremony in which the Business Analyst, Developer and QA collaborate to flesh out the story, bringing together perspectives from different stages of the Software Development Lifecycle. I discussed with them what a comprehensive set of acceptance criteria should be, querying how whitespaces within values should be handled to determine the scope of the development work. When we ultimately arrived at a common understanding of the requirements motivating this story, our Business Analyst could then type this up and I could add a development plan(see figure 1) and, as I was going to be working on it right away, move the ticket from 'Discovery' to 'In Progress'.

As a DEVELOPER, I want to make the Data Migration Script White space (Leading & Trailing) free so that during the Data Ingestion process, the Target data table(s) receives a clean version of the data set

ACCEPTANCE CRITERIA

Scenario 1: Ensure leading or trailing white spaces are trimmed
GIVEN Data to migrate
WHEN the Data Migration script is triggered it is able to trim leading or trailing white spaces from the source data wherever either occur
THEN a clean data set is ingested to the target table

Scenario 2: Ensure leading and trailing white spaces are trimmed
GIVEN Data to migrate
WHEN the Data Migration script is triggered it is able to trim leading and trailing white spaces from the source data wherever both occur
THEN a clean data set is ingested to the target table

Scenario 3: Ensure the script is not removing any white space between the values
GIVEN Data to migrate
WHEN the Data Migration script is triggered it is not removing any white spaces between words from the source data
THEN a clean data set is ingested to the target table

Scenario 4: Handle missing value/record in the CSV file
GIVEN Data to migrate
WHEN the csv file is missing value/record
THEN the Data Migration script should be able to handle the exception.

Scenario 5: No change to data migration process otherwise
GIVEN data to migrate
WHEN there are no leading or trailing white spaces in the source data
THEN there should be no change to how the data migration script handles incoming data.

Dev Plan

- Update the current Data Script to ensure it takes care of the scenarios mentioned above
- Test the script in the local environment before pushing it to the data ingestion pipeline
- Pre/Post Data Validation

Figure 1 – User story with acceptance criteria following three amigos

The data migration service exists in a separate repository to that which holds the bulk of our team's work, so I needed to familiarise myself with it in the first instance. After cloning the repository, and the now rote process of setting up a branch according to naming convention, I set about analysing the existing migration scripts.

Four of the columns in the Access Management table- service name, resource type, resource name and relationship- are foreign keys. The relationship column is nullable, however the others are mandatory fields, and any new entry to the Access Management table would fail without them (I had earlier created a data model diagram for the Access Management team, see figure 2 below for an illustration of the relationship between tables). The 'init' script therefore sets up a temporary staging table, imports data from the CSV into it and populates the necessary fields in the resources, services and roles tables ahead of the main data migration to the access management table.

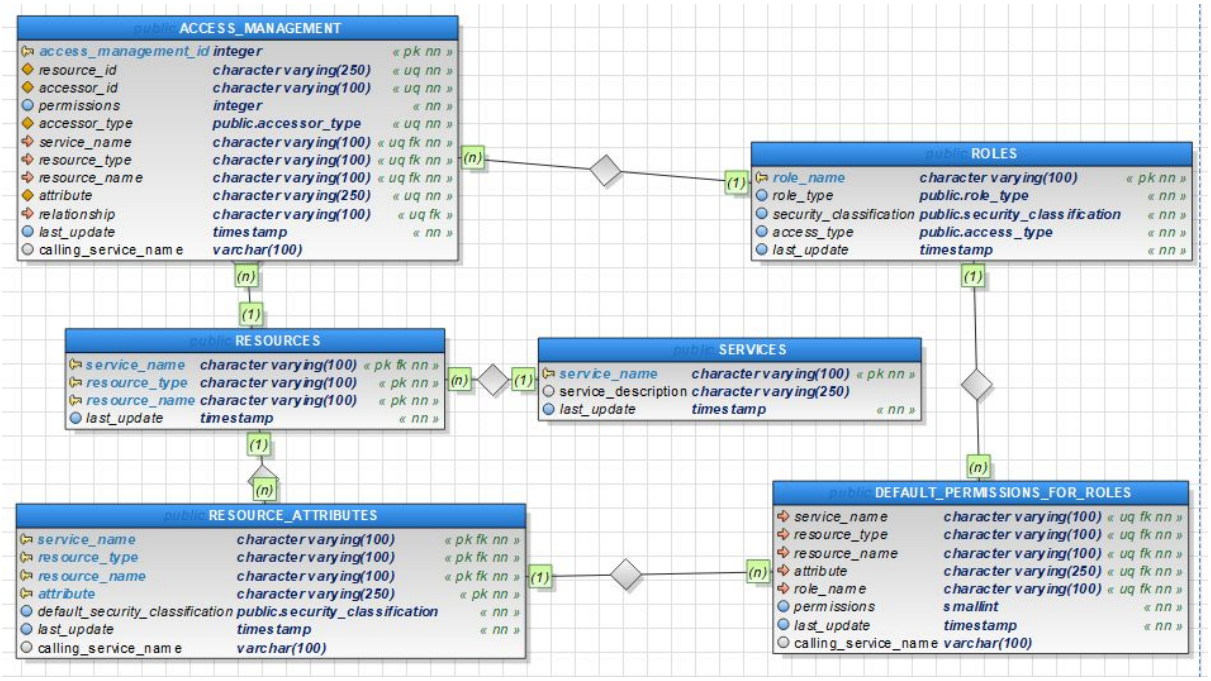

Figure 2 – Access Management data model diagram

The 'main' migration script which runs afterwards similarly starts by setting up a temporary staging table into which data will be imported, as well as a migration errors table that is dropped and created once per migration and is used to help produce the metrics file at the end of its run. The main script then drops the resources foreign keys constraints, the relationship foreign key constraint and the unique constraint to avoid unnecessarily slowing the data migration. Rather than every insert of a foreign key into the access management table requiring checks that the foreign keys exist in the corresponding table- which would slow performance considerably with large volumes of data being migrated- errors are handled by the section of the script in figure 3.

```
WITH migration_errors AS (
    DELETE FROM stage
    WHERE resource_id IS NULL
        OR accessor_type IS NULL
        OR accessor_id IS NULL
        OR "attribute" IS NULL
        OR permissions IS NULL
        OR service_name IS NULL
        OR resource_name IS NULL
        OR resource_type IS NULL
        OR NOT EXISTS (SELECT service_name, resource_name, resource_type FROM resources AS r
            WHERE stage.service_name = r.service_name AND stage.resource_name = r.resource_name AND
            stage.resource_type = r.resource_type)
        OR (relationship IS NOT NULL AND relationship NOT IN (SELECT role_name FROM roles))
    RETURNING resource_id, accessor_type, accessor_id, "attribute", permissions, service_name, resource_name,
        resource_type, relationship
)
INSERT INTO access_management_migration_errors SELECT * FROM migration_errors;
```

Figure 3 – Portion of the data migration script which handles errors.

This checks the not null constraints, the resources foreign keys constraints, and the relationship foreign key constraint. The unique constraint is handled implicitly as postgreSQL treats duplicate records as the same record.

After theses constraints are dropped, the script imports the data, sifting for duplicates and errors as it goes. Finally it recreates the dropped constraints, writes the metrics file and drops the temporary stage table.

Having gained a good general comprehension of how the existing scripts worked, and honed in on the appropriate point to add code for trimming whitespace from incoming data, I could now look into how to write the code. PostgreSQL provides a function- *btrim(string, trim_character)* - which removes specified characters from the beginning and end of a string, and if the second parameter is omitted it will assume whitespace as the character to trim. In the first instance I thought it best to trim the whitespace in the same transaction that reads the data from the CSV file, but after some trial and error discovered that the PostgreSQL doesn't support this. Ultimately I solved the problem by running the trimming once the data had already been read from the file (see figures 4 and 5 below).

```
17  17    \COPY stage FROM 'am-migration.csv' DELIMITER ',' CSV HEADER;
18  18
    19  + UPDATE stage
    20  + SET resource_id = BTRIM(resource_id),
    21  +     accessor_id = BTRIM(accessor_id),
    22  +     attribute = BTRIM(attribute),
    23  +     service_name = BTRIM(service_name),
    24  +     resource_name = BTRIM(resource_name),
    25  +     resource_type = BTRIM(resource_type),
    26  +     relationship = BTRIM(relationship);
    27  +
19  28    WITH ins_services AS (
20  29        INSERT INTO services
21  30            SELECT service_name, 'Service for annotations' AS service_description
```

Figure 4 – My change to the init script

```
34  34
35  35    SELECT COUNT(*) AS "rows to migrate" FROM stage;
36  36
    37  + UPDATE stage
    38  + SET resource_id = BTRIM(resource_id),
    39  +     accessor_id = BTRIM(accessor_id),
    40  +     attribute = BTRIM(attribute),
    41  +     service_name = BTRIM(service_name),
    42  +     resource_name = BTRIM(resource_name),
    43  +     resource_type = BTRIM(resource_type),
    44  +     relationship = BTRIM(relationship);
    45  +
37  46    WITH file_duplicates AS (
```

Figure 5 – My change to the main script

While development was in progress I tested my changes locally via the command line. This is the quickest, least involved way to test the script works as expected. I just needed to ensure the csv file I was using for the test was in the am-data-migration folder and then in Bash run a series of commands (see figure 6) that would spin up a Docker container, copy my files across to the container and then run the migration to see how my script would fare.

```
764108@LTGB06897 MINGW64 ~/Documents/AM/am-lib (master)
$ docker-compose up -d am-accessmgmt-api-db
The S2S_SECRET variable is not set. Defaulting to a blank string.
Creating network "am-lib_am-accessmgmt-api-network" with driver "bridge"
Creating am-accessmgmt-api-db ... done

764108@LTGB06897 MINGW64 ~/Documents/AM/am-lib (master)
$ docker cp ../../am-data-migration/am-data-migration am-accessmgmt-api-db:/

764108@LTGB06897 MINGW64 ~/Documents/AM/am-lib (master)
$ ./gradlew migratePostgresDatabase
Starting a Gradle Daemon, 10 busy Daemons could not be reused, use --status for details

> Task :migratePostgresDatabase
outOfOrder mode is active. Migration of schema "public" may not be reproducible.

BUILD SUCCESSFUL in 16s
1 actionable task: 1 executed

764108@LTGB06897 MINGW64 ~/Documents/AM/am-lib (master)
$ winpty docker exec -it am-accessmgmt-api-db bash
bash-5.0# cd am-data-migration/
bash-5.0# ./am-migration-runner.sh localhost 5432 am amuser annotation-am-migration-init-v2.sql

* Starting AM migration...

* AM DB hostname: localhost
* AM DB port: 5432
* AM DB name: am
* AM DB username: amuser
```

Figure 6 – Testing locally in git bash

Once I was happy with it the next step was to run a migration using the Jenkins pipeline, to satisfy myself that it all works as expected. It did, so a pull request could then be made in the repository and peer reviews requested, and the story was moved from 'In Progress' to 'In Review'.

Finally I needed to run the migration once more to demonstrate the functionality to our product owner so as to receive their sign off and bring the story to a close. The repository's readme explains that the pipeline:

- reads the migration data file from Azure Blob Storage
- reads the migration script from this repository
- carries out the data migration to the Access Management database in the specified environment

For the first part of that process I needed to prepare a CSV file that would demonstrate the new functionality and how it meets the various acceptance criteria in figure 1. Figure 7 below shows the CSV file I prepared for the demonstration.

am-migration-whitespace-demo.csv

```
1 resource_id,accessor_type,accessor_id,attribute,permissions,service_name,resource_name,resource_type,relationship
2    1d1c7869-b3e8-4339-ac2d-8ce8b9a2b989,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/0e39b927-b659-4094-a2d8-b75
3 1d1c7869-b3e8-4339-ac2d-8ce8b9a2b989   ,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/6569081f-dc58-4109-ae6b-c92
4    1d1c7869-b3e8-4339-ac2d-8ce8b9a2b989   ,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/a6bc5ae5-67bb-4bba-a07c-
5 1d1c7869-b3e8-4339   -ac2d-8ce8b9a2b989,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/5ed9844f-abe3-44a5-9b3b-6e7
6 ,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/69935093-02df-48de-9c88-fb47c8539e32,6,Annotations,documentAnnotat
7 1d1c7869-b3e8-4339-ac2r-8ce8b9a2b989,USER,510003,/b1071139-efd4-4ed2-85ed-d087fcd2a1af/66907bc4-860f-4384-9b96-0ca75a
8
```

Figure 7 – Demo data for migration

During the demonstration to the Product Owner, I explained the contents of the CSV file and the expectations relating to each line before running the migration:

- Line 1 shows the headers.

- Line 2 shows a resource id with leading whitespaces. The expectation here is that the whitespaces will be trimmed as per the acceptance criteria's Scenario 1, then – as this sample data is taken from a previously migrated file- recognised as a duplicate.
- Line 3 shows a resource id with trailing whitespaces. The expectation here is the same as line 2.
- Line 4 shows a resource id with both leading and trailing whitespaces. The expectation here is the same again as per the acceptance criteria's Scenario 2.
- Line 5 shows a resource id with whitespaces in the middle of the value. These whitespaces should not be trimmed as per Scenario 3, and the migration process is expected to therefore treat this as new data.
- Line 6 shows an entirely missing resource id. This should be treated as a migration error as per scenario 4.
- Line 7 provides normal data. Only the resource id is slightly altered so that this row would be treated as new data and migrate, meeting the expectation of Scenario 5.

This file was then uploaded to Azure Blob Storage. I'd not worked with Microsoft's cloud computing services before, but this turned out to be a fairly intuitive process once logged into the portal of navigating to the right page to uploading the file. The pipeline was then triggered through the Jenkins dashboard by navigating to the master branch, selecting Build with parameters setting those parameters- which are the migration data file name to be picked up from Azure blob storage and the migration script name- and finally clicking Build (figure 8).

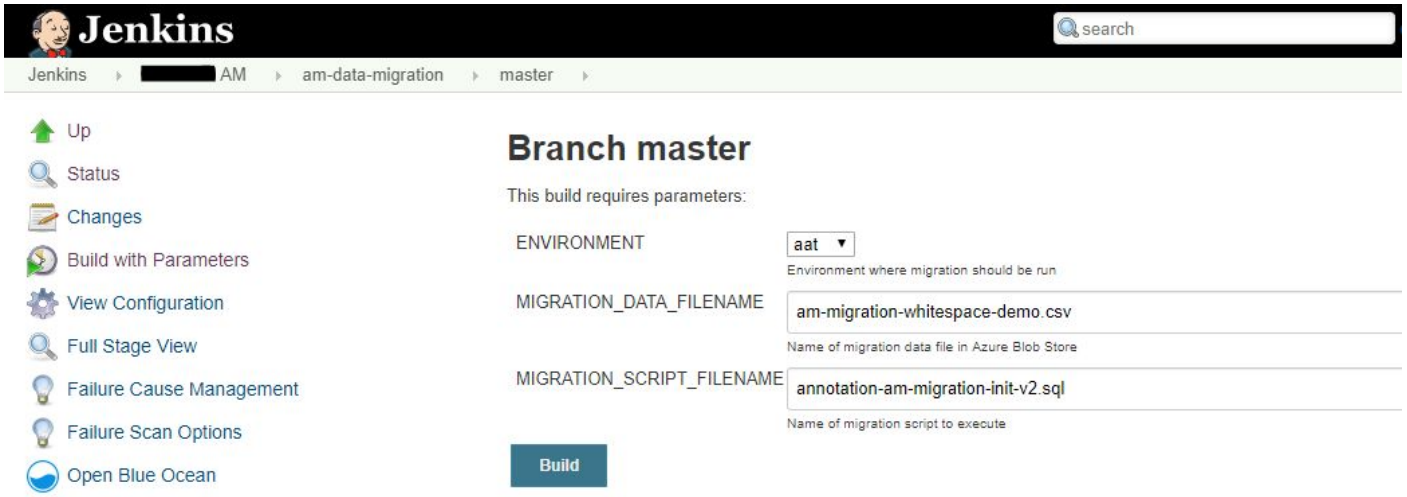

Figure 8 – Jenkins build with parameters.

Once the build had completed successfully, I could click through to the metrics file the build had produced to confirm that our expectations had been met (see figure 9): three rows recognised as duplicates, two rows migrated and one migration error.

```
 rows to migrate
-----------------
               6
(1 row)

 duplicate rows in file (skipping)
-----------------------------------
                                 0
(1 row)

 duplicate rows in access_management table (skipping)
------------------------------------------------------
                                                   3
(1 row)

 pre-migration access_management count
---------------------------------------
                              500183
(1 row)

 post-migration access_management count
----------------------------------------
                              500185
(1 row)

 migration errors
------------------
                1
(1 row)

 resource_id | accessor_type | accessor_id |                                attribute                                | permissions | service_name |   resource_name    | resource_type |
relationship
-------------+---------------+-------------+-------------------------------------------------------------------------+-------------+--------------+--------------------+---------------+-
-------------
             | USER          | 510003      | /b1071139-efd4-4ed2-85ed-d087fcd2a1af/69935093-02df-48de-9c88-fb47c8539e32 |           6 | Annotations  | documentAnnotation | annotation    |
(1 row)
```

Figure 9 – am migration metrics file

Following my demonstration, the Product Owner accepted the story and I was able to move the ticket to ‘Done’.

Reflective Statement

Since joining the Access Management team the initial challenge was to familiarise myself with the new codebase and understanding the functionality, which I did through manual testing and supporting QA for a while. This project was in fact the first code change I accomplished for the team. And graduating from manual testing to contributing code proved an invaluable way to develop my understanding of the Access Management project further. It was challenging, but having already worked with PostgreSQL in my previous team, I found this to be fairly accessible. Perhaps the most challenging aspect of the entire process was demonstrating the result of my work to the Product Owner, as this was an entirely new experience for me. It required some planning and preparation, both in terms of having a demo data migration ready to run and having an idea in mind of how to explain my work to non-technical peers. And presenting can be a stymying experience at the best of times. Ultimately the Product Owner was happy with my demonstration and to give their sign off, and the new and improved data migration scripts were now better prepared for future migrations.