Day 2 Exercises

List of all assignments for the second day together. Please find all DDL and all other needed queries for the exercises in following file on GitHub: `Day2/day2_exercises.sql`

09 Monitoring the Data Pipeline

Exercise #1

In the first exercise in this session we are going to build an error notification for task. When task fails, an SNS message will be sent and we can react on that message somehow - send a slack/teams notifications for instance. As we do not have a Slack environment available we will build everything up to that part when it should be send to slack.

Let's start with error notifications setup in AWS:

1. Create an SNS Topic - select a standard type and name the topic `sf_error_notification`
2. Record the ARN of the topic
3. Create an IAM policy which will define following action:
   `sns:publish`

You can copy the following text to the JSON tab of the policy:

```
{
   "Version": "2012-10-17",
   "Statement": [
     {
       "Effect": "Allow",
       "Action": [
          "sns:Publish"
       ],
       "Resource": "<sns_topic_arn>"
     }
   ]
 }
```

You need to replace the `sns_topic_arn` with the ARN of the SNS topic you created in step 1.

Click to next and skip the tags definition, then click on review of the policy and name it, e. g. `snowflake_sns_topic.`

4. Now we need an IAM role. But we already have a role linked to our Snowflake account so we can try to reuse it and just attach another policy to that one.

- Open IAM roles, locate your snowflake role and click on **add permissions** and **attach policies.**
- Find the snowflake_sns_topic policy and add it to that role.

5. No we need to create a new object on Snowflake side. That object is called Notification integration. It is integration object that references the SNS topic we have created. Only ACCOUNTADMIN can create such object so do not forget to switch your role. You can find the DDL statement on GitHub in `day2/day2_exercises.sql` file.

6. Once we have the integration we need to grant Snowflake access to the SNS topic. Run the

```
desc notification integration my_error_notification;
```

And make a note of following values:

- SF_AWS_IAM_USER_ARN
- SF_AWS_EXTERNAL_ID

7. Now we have to modify the trust relationship in the IAM role. Let's open our role in IAM and go to **Trust relationship** tab. We already have there the configuration for our storage integration. The retrieved value of SF_AWS_IAM_USER_ARN should be same like we have already defined. So nothing should be changed.

But you should get different value for SF_AWS_EXTERNAL_ID. Here we need to change the configuration. We have to use an array for all the values in `sts:ExternalId` field. It should look like this:

```
"Condition": {
                "StringEquals": {
                    "sts:ExternalId": [
                        "<sf_aws_external_id_storage_integration>",
                        "<sf_aws_external_id_error_integration>"
                    ]
                }
            }
```

That's it. We have configured the error notification. Now we need two additional things:

- Enable the error integration in tasks
- Some process which will process sent SNS messages

8. Let's modify our task from previous exercise and add this error notification to it.

9. As a process which will be automatically processing the SNS messages we can use lambda function. Let's create one even though we will not finish the integration as we do not have the slack environment. But if you have any slack environment feel free to use it and test the whole integration.

Go to Lambda and create a new function called `sfErrorProcess`:

- Select author from scratch
- Select Python 3.8. as runtime
- Select x86_64 as architecture

Now we need to add a trigger to our function. It will be triggered automatically by new incoming SNS message. Let's click on add trigger:

- Search for SNS as a source
- And then paste ARN of our SNS topic or AWS console should offer it for you automatically.

Now we need to just write the processing code in Python. You can find such an example of such code in following file on github: `sfErrorProcess.py`

You can go and check the code or update it according your needs. The code is doing following:

- Format the message by using markdown
- Uses Slack API to send message into configured channel. It requires to have a slack integration configured.

Exercise #2

We are going to try to improve our pipeline little bit and built also email notification for monitoring the task state. We will build a stored procedure which will be checking the state of the task. In case of the suspended state, an email will be sent to defined email addresses.

This solution will use another notification integration but this time the type of it will be email. In this case we do not need to create any integration towards cloud provider as the email is sent directly by Snowflake. But this feature currently works only for Snowflake accounts hosted on AWS. If your account uses another cloud provider this will not work for you.

1. Create a new notification integration called my_email_int where the TYPE will be EMAIL and allowed recipients will be your verified email address. This needs to be done as accountadmin

You must have verified email address assigned to your Snowflake profile in order to receive e-mails from Snowflake. Go to your profile settings, enter your email address and click on send verification Email. If you have already done this, then it is not needed.

2. Grant this new integration to our sysadmin role
3. Now we can write a stored procedure which will have one input parameter and it will be task_name. The procedure will do following:
    - Check the state of the task by running show task command
    - In case the state will be suspended it will call system stored procedure SYSTEM$SEND_EMAIL and send the email notification to verified email address.

10 Stored Procedures

We are going to improve the monitoring of our data pipeline based on snowpipe, stream and task. We will build another stored procedure which will be monitoring the stale state of our STR_LANDING stream.

Two input parameters:

- `stream_name` – name of the stream we want to check
- `staleness_limit` - how many days before stream become stale we want to be notified

Procedure will use a stale_after value from show stream command and calculate the stale_after value by doing date diff between this value and current timestamp

In case the calculated value of days is lower or equal to provided staleness limit then email should be sent with following message:

```
Please check stream. It might became stale in 5 and   your defined
notification limit is: 6
```

And procedure will return following message:

```
      Email has been sent. Stream will soon become stale.
```

In case the condition will not be met then procedure will just return this message:

```
      Stream stale state is ok. It will become stale in 6  days and your
notification limit is: 4
```

11 User defined functions

Please use the stored procedure from previous session as a skeleton. This one will be very similar. You need to find out how to write that error message which should be send by email and it combines text with input_parameters stored in variables.

We are going to practice scalar and table UDF in this example. Firstly we will try to extract a domain name from user emails in our `snowpipe_landing` table.

Then we will try to improve the solution and write table function which will extract 3 values from single email and return them as a table. We are going to extract the username, first and second domain. This time we will use Python as language for our UDF and UDTF.

1. Let’s write a UDF called `get_email_domain(email string)`:
    - It uses Python as a language
    - It extracts email domain part (everything after @ char) from given email

2. Let’s write another function called `extract_email(email string)`:
    - It returns table with following schema: `username string, domain1 string, domain2 string`
    - It uses Python as language
    - It extracts following values from given email:
        - Username - everything before ‘@‘ char
        - Domain1 - everything after ‘@‘ and before ‘.’
        - Domain2 - everything after ‘.’ In domain part

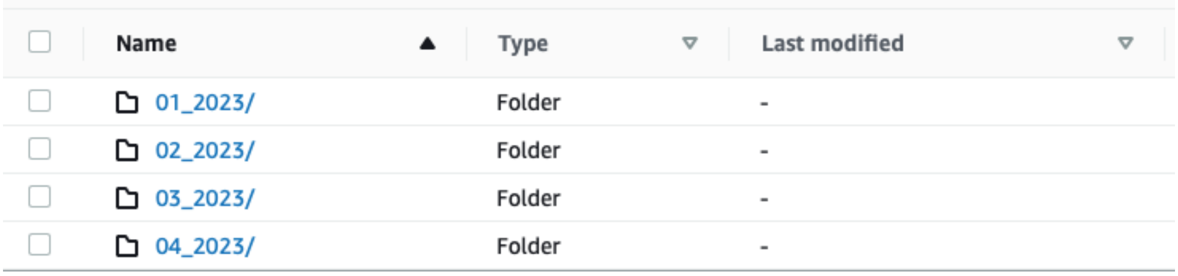
12 External tables

This exercise is dedicated to external tables. You can find the source data which we will need in following Github location:

- `Day2/source_files/external_tables`

Please download all 4 files to your local computer and then upload to your external stage location. Firstly we are going to upload only 3 files out of four.

1. Create a subdirectory external_tables in you external stage
2. Create four another directories inside the external_stage directory to simulate we are loading data each month and they are organised in separated directories.
    - 01_2023
    - 02_2023
    - 03_2023
    - 04_2023



□	Name ▲	Type ▽	Last modified ▽
□	01_2023/	Folder	-
□	02_2023/	Folder	-
□	03_2023/	Folder	-
□	04_2023/	Folder	-

3. Upload the file `ext_table1.csv` to folder 01_2023 and accordingly also files `ext_table2.csv` and `ext_table3.csv`

Now we are ready to go and we can create external tables in Snowflake. To validate we have our files available in the stage we start with listing all the files in external stage under the external_tables subdirectory.

4. Let's start with the easiest option - create and external table without knowing the file structure. We are not going to define the columns. Just the location, file format (csv).
5. Query the table to see how the data are presented
6. Try to query the table again, now by querying the individual columns by $1:c1, $2.c2, etc.
7. And query it for the last time now with proper data type and name:
    - `$1:c1::timestamp_ntz created,`
    - `$1:c2::number id,`
    - `$1:c3::varchar first_name,`
    - `$1:c4::varchar last_name,`
    - `$1:c5::varchar email,`
    - `$1:c6::varchar gender`

8. Let's create a second version of the external table as now we know the structure and we can include it into external table definition.
9. Query the table to check it out
10. We can try to query it together with external table metadata: `METADATA$FILENAME, METADATA$FILE_ROW_NUMBER`
11. Create the last version of the external table - now with defined partitions. We are going to use the created column and extract the load year month from it. A partition column must evaluate as an expression that parses the path and/or filename information in the `METADATA$FILENAME` pseudocolumn.
12. We can extract that info with following command:

```
select distinct to_date(split_part(METADATA$FILENAME,'/',3), 'MM_YYYY')
from @my_s3_stage/external_tables;
```

13. Now we have a definition of the partition column so we can include it into table definition and create the third external table.
14. Query the table to verify it works
15. Query the distinct load year month to find out which partitions we have available in external table. We should get the partitions from 2023-01 up to 2023-03
16. Now, let's upload the last file into the external stage into the 04_2023 directory
17. If we query the external table now we can see that data are still not available
18. We have to refresh the external table so let's do it
19. We will get an output of the command which will be saying the the last file which we have uploaded has been registered for our external table.
20. If we query the table now, the data from the latest file will be available.

13 Data Unloading

This exercise will be dedicated to practising unloading the data into internal and external stages. Let's start with internal stage and csv/parquet data. We are going to use `GPT_TWEETS` table as a source.

We can check the content of the table stage with command: `list @%gpt_tweets;`

Let's prepare the data for offloading. We want to have a user activity overview data set containing only users with at least two tweets and following attributes:

- total number of sent tweets
- total replies received
- total likes received
- ordered from the most active users

Once we have a query for such data set we can include it into COPY unloading command and unload the data into table stage, under `csv` subdirectory. As a file format we are going to use our `my_csv_format.`

Let's also try to offload the data into json format and external stage. This could be the use case when you need to share the data via some external system via API. This is also related to practising the semi structured data processing functions like `object_construct()` and `array_agg().`

Let's do one example together and then you will try to do it yourself. We are going to turn our table into the json document and then unload it into our external stage under `json` subdirectory.

Another simple example is showing how to construct an array. And now you should know everything for creation the JSON file yourself.

And now tasks for you:

1. Unload the same user activity overview dataset into the parquet format:
    - Use `parquet` subdirectory in table stage
    - Use custom file name `tweet_activity_summary`

2. Please create a JSON file with following structure:

```
{
    username: "abc",
    tweetSummary: [
     {
        tweet: "Random tweet text 1",
        tweetStats: {
            likeCount: 4,
            replyCount: 2,
            retweetCount: 3,
            quoteCount: 0

        }

    ]
    }
}
```

You can see that JSON file contains an array of objects with all the tweets per user, along with statistics for that tweet.

Note:

- You will need functions `OBJECT_CONSTRUCT()` and `ARRAY_AGG()`
- To create an array of objects you will need either CTE or join the dataset with this structure but without array back to base table in order to create an array
- So you need to create this JSON structure without array first and then do the aggregation for each user

14 SQL API

This exercise is dedicated to using the SQL API. We will go through authorization and sending the API request to Snowflake. For sending the API requests we are going to use Postman.

Let's start with authorization. We will use key pair authentication and for that we need generate private & public key and then assign it to our user.

1. Let’s check if our user does not have assigned some key by performing `desc user <my_username>`

We will get a list of user’s parameters where one is called RSA_PUBLIC_KEY.

2. We can generate the encrypted private key with following command:

```
openssl genrsa 2048 | openssl pkcs8 -topk8 -v2 des3 -inform PEM -out
rsa_key.p8
```

We need to provide our passphrase.

3. Once we have a private key, we can generate the public key:

```
openssl rsa -in rsa_key.p8 -pubout -out rsa_key.pub
```

4. Now we have to assign the public key to our user - remove the public key delimiters

```
 Alter user <username> set rsa_public_key = ‘key’
```

5. We can verify that connection with the key pair authentication works in SnowSQL

```
snowsql -a <accountIdentifier> -u <username> --private-key-path <path to
private key>
```

6. We have working key pair authentication so we can proceed and generate the JWT token which we also need for authorisation. Again we can use SnowSQL for that:

```
snowsql -a <account> -u <user> --private-key-path <path to private key> —
generate-jwt
```

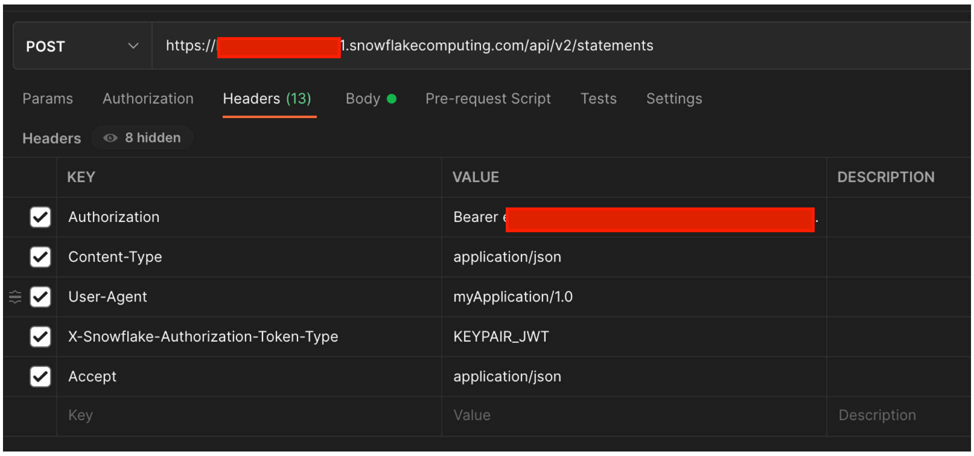
7. Now we have everything what we need for sending an API request. Our requests will have following headers:

```
authorization: Bearer <jwt_token>
Content-Type: application/json
Accept: application/json
User-Agent: myApplication/1.0
X-Snowflake-Authorization-Token-Type: KEYPAIR_JWT
```

We also need to provide a request body. Let’s try with simple SQL query selecting data from one of our tables:

```
{
"statement": "select * from gpt_tweets limit 20",
"timeout": 60,
"database": "DATA_ENGINEERING",
"schema": "PUBLIC",
"warehouse": "COMPUTE_WH",
"role": "SYSADMIN"
}
```

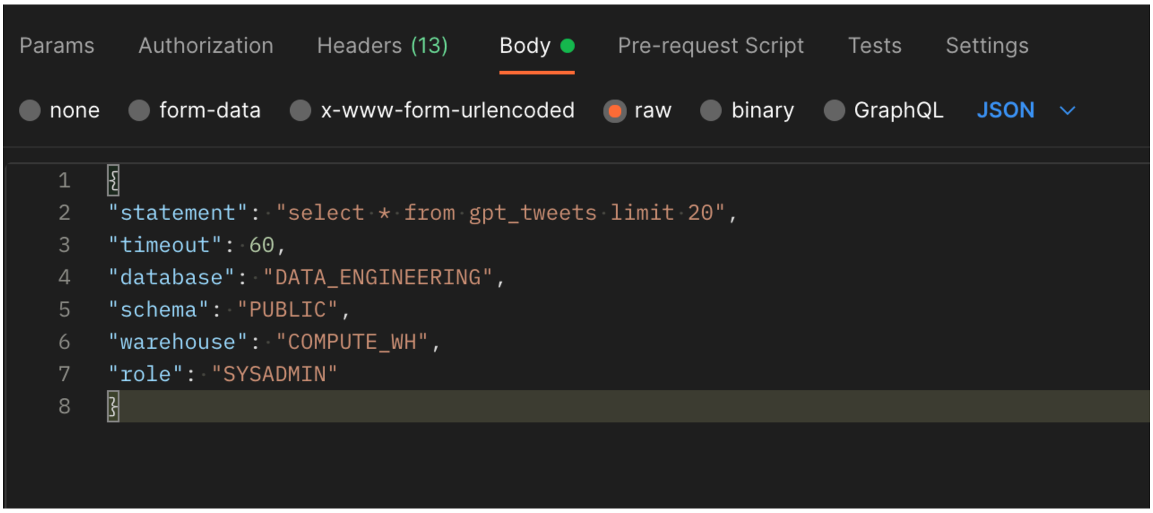
8. Let’s use postman for sending the request:



https://[redacted].snowflakecomputing.com/api/v2/statements

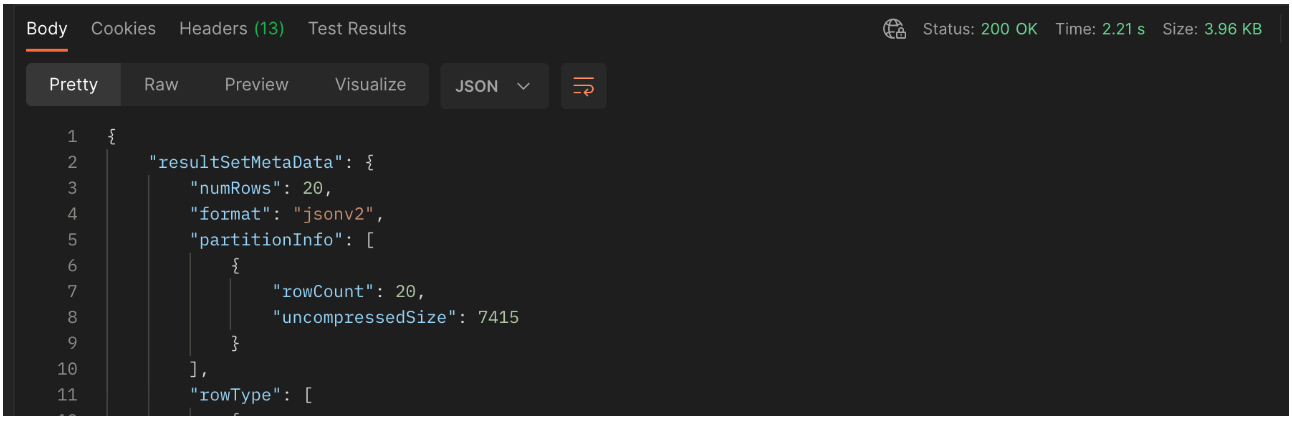
	KEY	VALUE	DESCRIPTION
☑	Authorization	Bearer [redacted]	
☑	Content-Type	application/json	
☑	User-Agent	myApplication/1.0	
☑	X-Snowflake-Authorization-Token-Type	KEYPAIR_JWT	
☑	Accept	application/json	
	Key	Value	Description

Request body specification:



```
{
"statement": "select * from gpt_tweets limit 20",
"timeout": 60,
"database": "DATA_ENGINEERING",
"schema": "PUBLIC",
"warehouse": "COMPUTE_WH",
"role": "SYSADMIN"
}
```

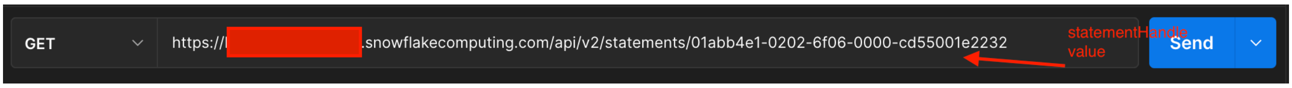
And the response:



Status: 200 OK Time: 2.21 s Size: 3.96 KB

```
{
    "resultSetMetaData": {
        "numRows": 20,
        "format": "jsonv2",
        "partitionInfo": [
            {
                "rowCount": 20,
                "uncompressedSize": 7415
            }
        ],
        "rowType": [
```

As a part of the response we also get the statementHandle value which we can use with another API endpoint to verify the status of the request - useful especially for long running queries.



https://[redacted].snowflakecomputing.com/api/v2/statements/01abb4e1-0202-6f06-0000-cd55001e2232

16 External function

We are going to focus on creation of the external function in this exercise. We will try to leverage Amazon Translate API provided by Boto AWS SDK and translate and return German translation for input strings in English.

Let's start with creation of the lambda function in AWS which will be doing the invocation of the Translate API. But first we need to make a note of our AWS Account ID. We can find this value in My Account menu.

We can go and create a Lambda function:

- Default option:start from scratch
- Give it name `translate`
- Choose Python 3.8 as a language
- Expand "Choose or create an execution role" and select "Create a new role with basic Lambda permissions."

Open the function and deploy the code which you can find on Github in `week3/external_fnc_lambda.py` and deploy the code.

Now we need to add a IAM policy for AWS Translate service. Go to configuration of the lambda, and click on Role name in Execution role menu. This will redirect you to IAM console. Here under permission tab, click on Add permissions and attach Policies and search for **TranslateFullAccess** policy, select it and click attach.

Now we need to do an integration between Snowflake and AWS. We have to create an IAM role for that.

Let's create a new IAM role:

- Choose "AWS account" when asked to select the type of trusted entity.
- Choose another AWS account and paste your Account ID for now.
- Click next to skip the permission policies
- Name the role `SFLambdaRole and click` create role.
- Make a note of the Role ARN value. We will need it later.

Let's move on. Now we need an API gateway for the integration. Go to API gateway console and select REST API by clicking the build button:

- Select NEW API
- Name it `TranslatePI`
- Make an Endpoint Type Regional

Click create API and we will be in the /Methods window now. From the "Actions" drop-down, select "Create Resource." Let's use `sf-proxy` as a name for this resource and click create.

Go again into the "Actions" menu and click "Create Method". Select POST from the drop down menu and click on the check button. Now you have the stup window available:

- Choose Lambda function as integration type.
- Check the Use Lambda Proxy integration
- In the Lambda function name paste the name of the function we have created before.

Click save and ok. Now we have to deploy the API. From "Actions" menu select "Deploy API":

- Select [New Stage] in deployment stage
- Use some stage name e.g. test and click deploy

Now we are redirected to stage editor page. Under stages, expand your stage until you see “POST” under your resource name. Click on “POST” and make a note of the “INVOKE URL” field for this POST request.

Now we have to secure our API so that only your Snowflake account can access it.

In the API Gateway console, go to your API POST method and click on "Method Request.”

Inside the Method Request, click on the pencil symbol next to "Authorization" and choose "AWS_IAM." Click the checkmark just to the right to save. Then, click on "Method execution." Find the ARN under "Method Request." And make a note of that value.

Next, go to "Resource Policy" from the left panel. You'll be setting the resource policy for the API Gateway to specify who is authorized to invoke the gateway endpoint. Once there, paste in this resource policy:

Paste there this resource policy:

```
{
    "Version": "2012-10-17",
    "Statement":
    [
        {
        "Effect": "Allow",
        "Principal":
            {
            "AWS": "arn:aws:sts::<12-digit-number>:assumed-role/
<external_function_role>/snowflake"
            },
        "Action": "execute-api:Invoke",
        "Resource": “<your method request arn>/*/POST/MyResourceName”
        }
    ]
}
```

- Replace the <12-digit-number > with your Account ID
- Replace the <external_function_role> with the Role name which you have created
- Replace the “Resource” value with your Method Request ARN value.

Click save. And we need to redeploy the API as it was secured now. Go back to resources and click on Actions and Deploy API where you will select the stage you have created earlier.

Now we have lambda function, IAM role and policy + API gateway. Let’s continue in Snowflake where we have to create an API integration.

You can find the SQL statements in the `day2/exercises.sql` on GitHub. You have to replace two values in the DDL script. You have to use your IAM Role which we have created - `SFLambdaRole` - please use the ARN of the role, not a name. Then you need API invocation URL. this will go into the `api_allowed_prefixes` field.

Once done, run the desc integration command. We have to make a note of following property values:

- API_AWS_IAM_USER_ARN
- API_AWS_EXTERNAL_ID

Like we did for storage integration. Now we have to setup the trust relationship between Snowflake and AWS for our api integration. Go to AWS console, search for IAM and our `SFLambdaRole` role.

Go to “Trust relationship” tab and click on edit trust policy button there. Find the value of “Statement”.”Principal”.”AWS”. There you have to replace the arm value with arn value from the previous step. So use there the value of API_AWS_IAM_USER_ARN from the describe integration command.

Then find the “Statement”.”Condition” field which is currently empty. Paste the following string between the curly braces:

```
"StringEquals": { "sts:ExternalId": "xxx" }
```

You have to replace the xxx with the second value from desc integration command. Put there value `from API_AWS_EXTERNAL_ID field and` click on update policy button.

Uff, we are almost ready. Now it is time to create the external function in Snowflake. Please run the DDL statement for the external function. You just need to replace two values there again:

- Replace the `< api_integration_name >` with the name of your API integration
- Replace the `< invocation_url >` value with your resource invocation URL.

And we are finally done! Let’s try our external function with passing some English string and get the translation as a response.

17 Snowpark transformations

We are going to practise basic data frame transformations related to Snowpark for Python. We will test out the new Python worksheets which are in public preview to find out its limitations and then we will continue with local Python environment and doing the development in Jupyter notebook.

Please find following files with source code on Github:

- `Day2/python_worksheet.txt` - Python code for the Python worksheet in Snowsight.
- `Day2/snowpark_transformations.ipynb` - Jupyter notebook with Snowpark code which you should run from your local

Please create a new Python worksheet in Snowsight and paste the content of `python_worksheet.txt` file into that. Then you can experiment with the code and run it.

Please download the Jupyter Notebook and run it inside your local Python environment. You can go through it statement by statement to find out how the transformations are built.

In the end, there is a task to create one more Snowpark transformation by using the Snowpark API. **You should try to find how long is the shortest and longest tweets.**

18 Deploying Python UDFs

This exercise is dedicated to deploying Python UDFs which are being developed locally to Snowflake. We are going to try deploy the function manually first and the we will introduce a new community based tool SnowCLI which can make the work easier.

1. Let’s start with creation of UDF via Snowpark session

We need to create an internal stage which will be holding our UDF Python code. You can use following DDL command to create a stage:

```
            create or replace stage udf_stage;
```

Please activate your local Python environment. You can download the from `Day2/18_udf_deploy_man.py` from GitHub with source code for this exercise. If you prefer the Jupiter notebooks, you can download this one: `Day2/18_udf_deploy_man.ipynb`

You can go through code and try to deploy the UDF into Snowflake via Snowpark and then call it.

2. In the second part of the exercise we will try to use SnowCLI for creation and deployment of the UDFs.

1. Let’s install the snowcli into our python environment:

```
      pip install snowflake-cli-labs
```

2. Then you should be able to get a help for the cli by using command: `snow —help`
You can see what everything is possible to do with snowcli.

3. Let’s create a new folder where we will place our new udf:

```
      mkdir my_udf
      cd my_udf
```

4. Now we can call snowcli command `snow function init`

It will populate our directory with bunch of files where one of it is `app.py` which will be the handler for our function.

5. You can try to run the function locally by running python `app.py`

6. Let’s try to install also some packages. Open the requirements.txt file and put there `pandas.`

7. Then install it by running `pip install -r requirements.txt`

8. Now we can import pandas into `app.py`

9. Let’s update our `app.py` to create a simple data frame and return it back

```
      import pandas as pd

      def hello() -> dict:
            df = pd.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
            return df.to_dict()
```

10. We have to change the return type of the function in `app.toml` file from `string` to `variant`

```
      [default]
      input_parameters = "()"
      return_type = "variant"
      name = "helloFunction"
      handler = "app.hello"
      file = “app.zip"
```

11. We have tested it out locally and now we want to publish it in Snowflake. So first we have to authenticate ourselves. Snowcli takes the connection details from SnowSQL which we have already configured. The config file should be in `~/.snowsql/config`

12. Run the `snow login` command
    - Confirm the path for config file
    - Put the connection name you want to use

13. Now we are going to do a deployment. Run the command:

```
snow function create
```

`SnowCLI` will create an app.zip file, upload it to the stage and create the UDF in Snowflake.

14. Once the functions is deployed we can test it via SnowCLI with command:

```
snow function execute -f "helloFunction()"
```

15. You can again check the stages in snowflake. You will find out that there is a new stage called Deployment. It holds the zip file with the function code. As a last command you can also try to run the UDF in Snowsight:
    - `Show stages;`
    - `Ls @DEPLOYMENTS;`
    - `Select helloFunction();`