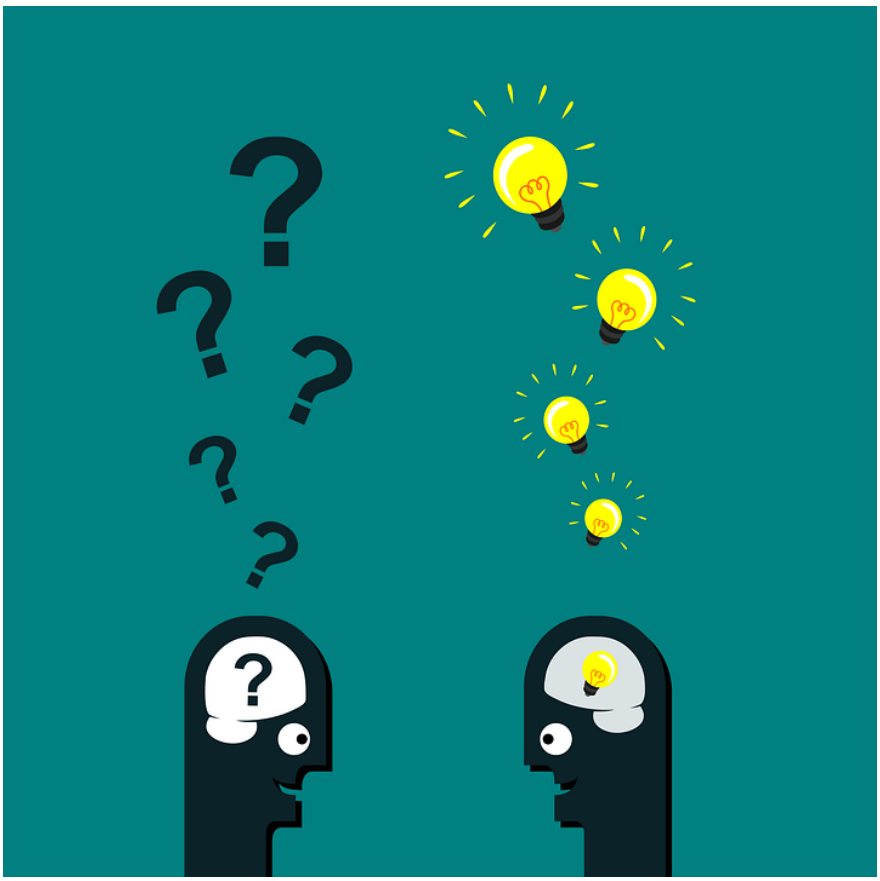
Advanced spreadsheet automation using Python leveraged by Azure container instances

This blog plans to demonstrate the automation of spreadsheet creation using Python, which would run on an Azure container instance. This will be followed by pushing it to Azure Blob Storage for future reference and data retrieval. I have used Pandas data frames, Azure Logic App, Azure Key Vault, Azure SQL database, Azure Container Instances, IAM, and Azure Blob Storage to fulfil this use case.

Problem statement

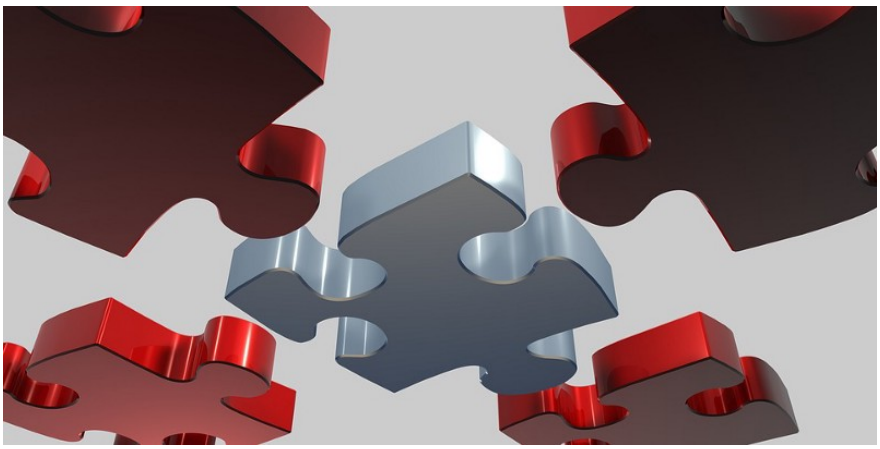
Exporting a SQL table to an excel spreadsheet is essential for many organizations. Non-technical teams like management, accounting, sales, etc. may not possess the skill to connect to a server, select the database, and query a table. I came across this requirement in multiple projects that I have delivered.



A business requirement was to query different tables by executing a sequence of complex queries in a non-simultaneous fashion, creating an output table with the required fields and dumping the data into a CSV.

Expected Solution

1. The entire process should take place with minimal human intervention.
2. The application must be dockerized and deployed to a container instance to minimise costs.
3. The process should be reliable and efficient.

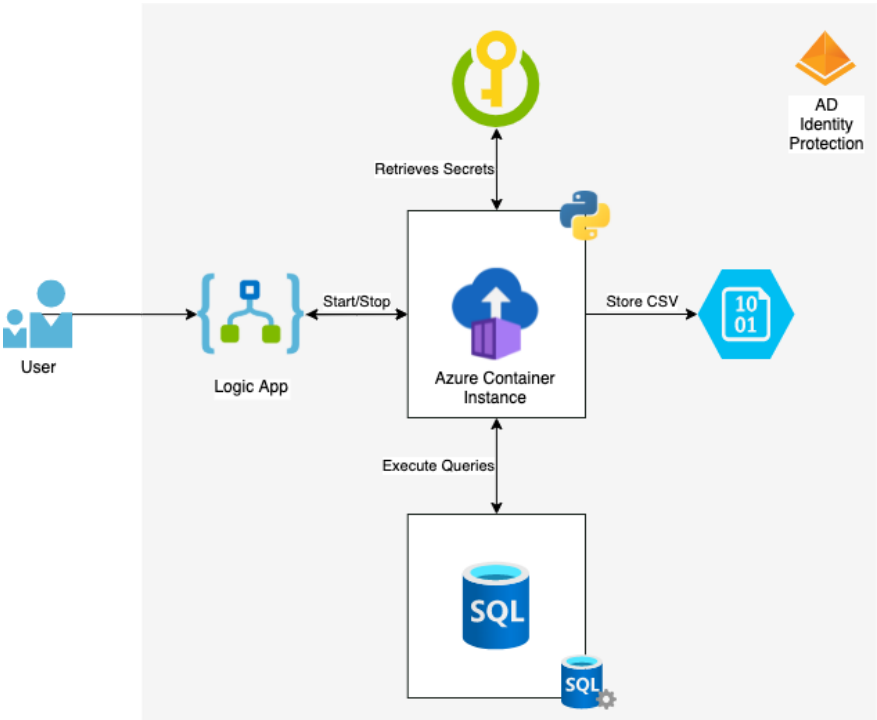


Other basic and advanced use cases

1. Your sales team needs to analyse the record of a recent campaign.
2. Your account team needs access to employee data.
3. Employee attendance-related details are to be retrieved and stored as spreadsheets for regulatory purposes.
4. The expenditure report needs to be pulled by the finance team.
5. Daily retrieval of business-related data for a specific application/service followed by dumping it into a CSV.

Solution

Architecture Diagram:



I developed a python application to streamline the entire automation process which would query tables in an Azure SQL database, create a structure with necessary fields, and finally dump it to a CSV. This CSV would then be stored in Azure blob storage for future reference/retrieval purposes.

Post development, I dockerized the application, created an image, pushed the image to DockerHub and deployed it to an Azure container instance. Azure states the benefits of running a container instance as follows:

> *Develop apps fast without managing virtual machines or having to learn new tools—it's just your application, in a container, running in the cloud.*

And that was exactly what I wanted to achieve! For more details, you can check out this link...

I am going to provide the steps to replicate this use case as well as some tips and tricks for customizing it as per your business requirements:

Pre-requisites

Some pre-requisites would be necessary for you to follow along or replicate the steps:

1. You will need an Azure subscription. If you do not have one, you can get one for free. Click here to create a free Azure account.
2. You will need an IDE to write your Docker file and the Python code. You can download Visual Studio Code for free.
3. Make sure you have Python installed and the VS Code Python extension.

Solution Implementation

Step 1:

> *Note: All code snippets are available as a public gist in the reference section.*

First I will write the Python code to automate the entire workflow.

```
import os
import base64
import pymysql
import csv
import json
import datetime
import sqlalchemy
import pytz
import pandas as pd
from flask import Flask, request
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobClient
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential

app = Flask(__name__)
credential = DefaultAzureCredential()
client = SecretClient(vault_url=KVUri, credential=cr
db_user = client.get_secret(username)
db_pass = client.get_secret(dbpass)
db_name = client.get_secret(dbname)
db_hostname = client.get_secret(hostname)
db_port = client.get_secret(portnumber)
blob_account = client.get_secret(blobacc)
blob_name = client.get_secret(blobname)
account_key = client.get_secret(clikey)
blob_container = client.get_secret(contname)

@app.route("/", methods=["GET"])
def hello():
    print("Function execution started")
    pool = sqlalchemy.create_engine(
        sqlalchemy.engine.url.URL(
            drivername="mysql+pymysql",
            username=db_user,
            password=db_pass,
            host=db_hostname,
            port=db_port,
            database=db_name,
        ),
        pool_size=5,
        max_overflow=2,
        pool_timeout=30,
        pool_recycle=1800
```

```
45      )
46      print('Connection established')
47      ist_zone=pytz.timezone('Asia/Kolkata')
48      print(ist_zone)
49      a=(datetime.datetime.now(ist_zone))
50      month=(a.month)
51      year=(a.year)
52      day=(a.day)
53      dmy=(str(a.day)+str(a.month)+str(a.year))
54      print(dmy)
55      datetime_object = datetime.datetime.strptime(str
56      month_name = (datetime_object.strftime("%b")).up
57      query = ['SELECT * FROM finrep WHERE retrievalTi
58      'SELECT * FROM tbl_fintask WHERE taskTime BETWEE
59      'SELECT * FROM tbl_finpredict WHERE predictTime
60      'SELECT * FROM tbl_fintime WHERE finTime BETWEEN
61      'SELECT * FROM tbl_maninfo WHERE manTime BETWEEN
62      'SELECT * FROM tbl_griminfo WHERE grimTime BETWE
63      'SELECT * FROM tbl_reapinfo WHERE reapTime  BETW
64      'SELECT * FROM tbl_tulninfo WHERE tulnTime BETWE
65      'SELECT * FROM tbl_fulnInfo WHERE fullTime BETWE
66      'SELECT * FROM tbl_cripjam WHERE cripTime BETWEE
67      'SELECT * FROM tbl_leasedin WHERE leasestamp BET
68      'SELECT * FROM tbl_joemam WHERE joemamstamp BETW
69      'SELECT * FROM tbl_classy WHERE classystamp BETW
70
71      blob_url=["dumfolder/cribdata1/func/"+str(year)+
72      "dumfolder/cribdata2/func/"+str(year)+"/"+str(mo
```

I will break down individual components of the code for a better understanding

These are the packages that I have imported

```
import os
import base64
import pymysql
import csv
import json
import datetime
import sqlalchemy
import pytz
import pandas as pd
from flask import Flask, request
```

Here, I am retrieving the secrets from Azure Key Vault

```
credential = DefaultAzureCredential()
client = SecretClient(vault_url=KVUri, credential=cre
db_user = client.get_secret(username)
db_pass = client.get_secret(dbpass)
db_name = client.get_secret(dbname)
db_hostname = client.get_secret(hostname)
db_port = client.get_secret(portnumber)
blob_account = client.get_secret(blobacc)
```

This is a list of queries to be executed

```
query = ['SELECT * FROM finrep WHERE retrievalTime BE
    'SELECT * FROM tbl_fintask WHERE taskTime BETWEEN
    'SELECT * FROM tbl_finpredict WHERE predictTime B
    'SELECT * FROM tbl_fintime WHERE finTime BETWEEN
    'SELECT * FROM tbl_maninfo WHERE manTime BETWEEN
    'SELECT * FROM tbl_griminfo WHERE grimTime BETWEE
    'SELECT * FROM tbl_reapinfo WHERE reapTime  BETWE
    'SELECT * FROM tbl_tulninfo WHERE tulnTime BETWEE
    'SELECT * FROM tbl_fulnInfo WHERE fullTime BETWEE
    'SELECT * FROM tbl_cripjam WHERE cripTime BETWEEN
```

I plan to structure my blob like this

```
blob_url=["dumfolder/cribdata1/func/"+str(year)+"/"+s
    "dumfolder/cribdata2/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata3/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata4/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata5/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata6/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata7/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata8/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata9/func/"+str(year)+"/"+str(mon
    "dumfolder/cribdata10/func/"+str(year)+"/"+str(mo
```

Below is the most important block of my code i.e., the logic

```
def query_exec(query,blob_url,pool):
    print('inside function')
    try:
        print('connecting to azure sql db')
        with pool.connect() as conn:
            print('connected successfully')
            print('executing query')
            stmt = sqlalchemy.text(query)
            output = conn.execute(stmt)
            print('query executed successfully')
            rows = output.fetchall()
            rows=list(rows)
            columns = output.keys()
            columns1=list(columns)
            print(columns)
            print(rows)
            print("output runs")
            blob_conn_str = f'DefaultEndpointsProtoco
```

Step 2:

In this step, I will dockerize the above code and push it to Dockerhub

```
FROM python:3.8-alpine

COPY ./requirements.txt .

WORKDIR /app
```

```
RUN pip install -r requirements.txt

COPY . /app

ENTRYPOINT [ "python" ]

CMD ["view.py" ]
```

Note: You will have a different requirements.txt file based on your use case.

I will now build a docker image, tag it, and push it to DockerHub

Build the Docker image:

```
docker image build -t spreadsheet-automation .
```

Tag the Docker image:

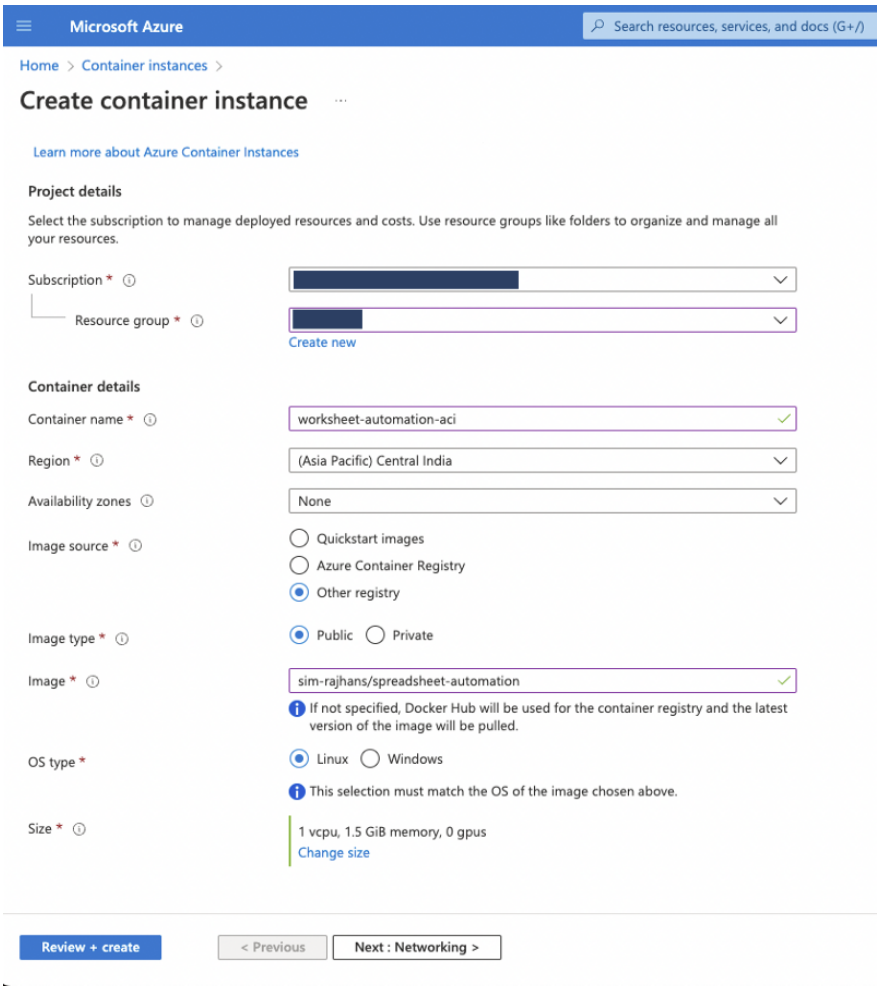
```
docker tag spreadsheet-automation sim-rajhans/automations
```

Push the Docker image to DockerHub:

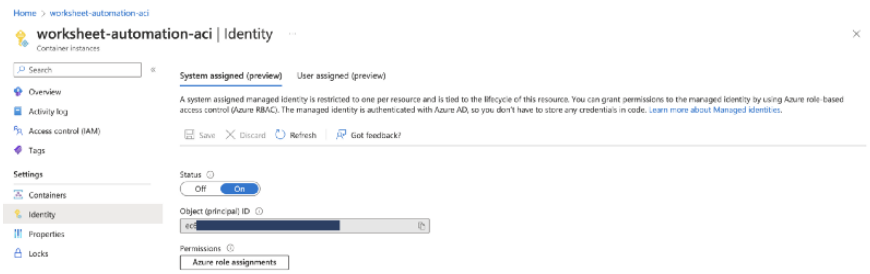
```
docker push sim-rajhans/spreadsheet-automation
```

Step 3:

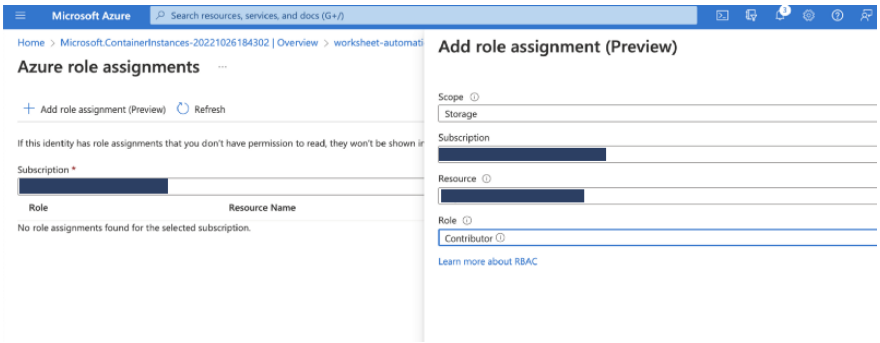
Now, I will create an Azure Container Instance from the Azure Portal. Make sure that you enter your image name.



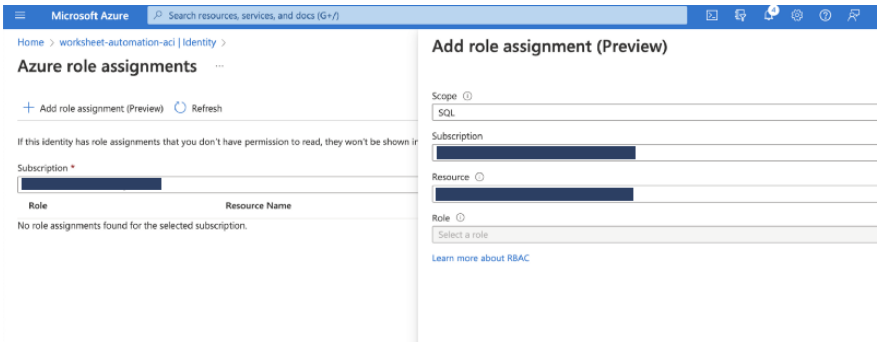
Once created, I will navigate to the Identity section and enable system-managed identity



I will add a role assignment to allow ACI to access Azure Storage Account

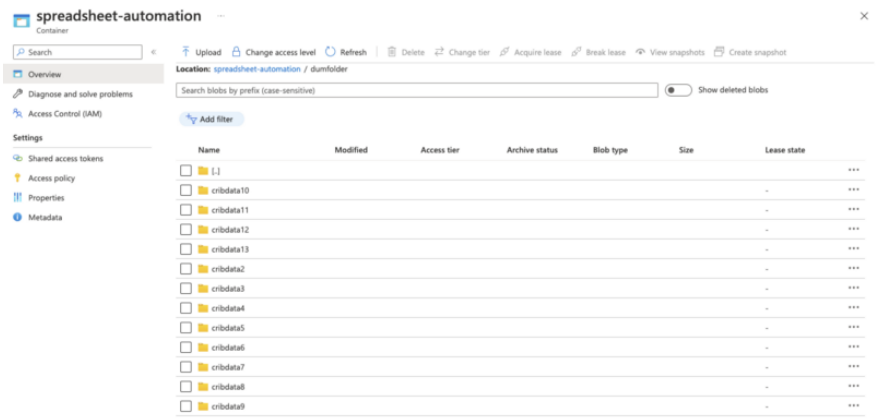


Then I will add role assignment to allow ACI to access Azure SQL Database



Step 4:

Once, the permissions are in place, I will proceed to check the execution of my application. The blob storage would have populated with the file structure.



As expected, the folder structure and the CSV files were successfully populated with the results of subsequent queries!

Best practice guide:

1. Whenever you are dealing with access, always try to assign fine-grained access.
2. Never leave keys and connection strings or any other confidential details open in the code.
3. Use Azure key vault to store secrets.

Challenges:

1. I implemented the entire solution on a virtual machine before moving it to an Azure Container Instance.
2. On a virtual machine, the application executed and performed perfectly. However, the cost of running a VM was too much.
3. To solve this I moved the entire application to an Azure container instance by dockerizing it which was slightly challenging due to the service level access issues.
4. Writing the code involved a lot of complexities and dependency on multiple services.

Business benefits:

1. The cost was reduced by 3 times
2. The entire solution was automated and could be replicated for any other use case internally
3. Manual work was minimized and the automation worked without manual intervention.
4. The overhead of maintaining the VM was reduced as the container instance was managed by Azure now.

Further implementation:

In addition to the above solution, I created a logic app with a trigger to stop the container whenever the execution was completed. This prevented the container from indefinitely consuming resources and avoided unintended costs.

Conclusion:

I have successfully created the Python code to automate advanced spreadsheet creation based on custom queries. I have leveraged Azure Services like Azure SQL Database, Azure Container Instances, IAM, Azure Storage, and Azure key vault to achieve the use-case.

References:

- https://azure.microsoft.com/en-us/products/container-instances/
- https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction
- https://docs.docker.com/engine/reference/commandline/push/
- https://gist.github.com/simran-rajhans/a119579765c19ba0ba7dba0ee89a0510

Published By

Name: Simran Rajhans

LinkedIn: https://www.linkedin.com/in/simran-rajhans