Azure Meal- CD
(Code Documentation)

Overview:

The **Meal Project** is a cloud-based ETL solution hosted on Microsoft Azure. It processes raw data stored in Azure Blob Storage, performs transformations using Azure Function App, and ingests the results into an Azure SQL Database. The processed data is visualized in Power BI to generate actionable insights.

Azure Blob Etl:

Azure Function TimerTrigger ETL Process:

This Python script implements an ETL (Extract, Transform, Load) pipeline using Azure Function. It processes blob files from Azure Blob Storage, extracts data, transforms it, and loads the results into a database.

The script uses the following libraries and modules:

1- logging: For logging messages.
2- azure.functions: To define Azure Functions.
3- azure.storage.blob: For interaction with Azure Blob Storage.
4- pandas: For data manipulation.
5- pymssql and pyodbc: For database connections.
6- dotenv: To load environment variables from config.py file.
7- re, os, traceback: For regex operations, file manipulation, and error handling.

Key Constants and Variables

Config.py file having Name of the Azure Blob Storage container, Connection string for Azure Blob Storage., Folder paths for input, output, and faulty files.
Below the code screenshot is provided, **Config.py**

Config.py

config.py > ...

```python
# config.py
import os
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Database connection details
server_name = os.environ.get('SERVER')
user_name = os.environ.get('USER_NAME')
pwd = os.environ.get('PASSWORD')
db_name = os.environ.get('DATABASE')

# Other configurations, such as storage connection string
STORAGE_CONNECTION_STRING = os.environ.get('AzureWebJobsStorage')
CONTAINER_NAME = 'csv-files'
INPUT_FOLDER = 'InputFiles/'
OUTPUT_FOLDER = 'Archive/'
FAULTY_FOLDER = 'Faulty/'
```

Databases:

database.py file having insert_metdata and insert_weights function.
In insert_metdata we are inserting data group_id, file_code, filename, variety, number_of_samples and formulations.
In insert_weights we are inserting data group_id, file_code, wld, weight and datedifference.

Below the code screenshot is provided, **Database.py**

Database.py

```python
import os
import logging
import traceback
import pymssql

def insert_metadata(file_metadata_records, conn):
    # Ensure connection is valid
    if conn is None:
        logging.error("Database connection is None. Aborting insert.")
        return

    # Test the connection by running a lightweight query
    try:
        cursor = conn.cursor()
        cursor.execute("SELECT 1")  # Lightweight query to validate the connection
    except pymssql.OperationalError as e:
        logging.error(f"Database connection is not active. Error: {e}")
        return
    try:
        for _, row in file_metadata_records.iterrows():
            try:
                 # Check if the record exists using the condition
                cursor.execute("""
                    SELECT COUNT(1) FROM dbo.File_Metadata
                    WHERE group_id = %s
                    AND file_code = %s
                """, (row['group_id'], row['file_code']))

                record_exists = cursor.fetchone()[0]

                if record_exists == 0:  # No record found, perform the insert
                    cursor.execute("""
                        INSERT INTO dbo.File_Metadata (Filename, Variety, Formulation, Number_of_Samples, UploadDate, group_id, file_code)
                        VALUES (%s, %s, %s, %s, %s, %s, %s)
                    """, (row['Filename'], row['Variety'], row['Formulation'], row['Number_of_Samples'], row['UploadDate'], row['group_id'], row['file_code']))
                    logging.info(f"Inserted File_Metadata record: {row.to_dict()}")
                else:
                    logging.info(f"Skipped File_Metadata record (duplicate): {row.to_dict()}")

            except Exception as e:
                logging.error(f"Error during database insertion: {traceback.format_exc()}")
    except pymssql.Error as e:
        logging.error(f"Error during insertion: {e}")
    except pymssql.Error as db_error:
        logging.error(f"Database error: {db_error}")
    finally:
        conn.commit()
        cursor.close()

def insert_weights(weight_records, conn):
    # Check if the connection is None
    if conn is None:
        logging.error("Database connection is None. Aborting insert.")
        return

    # Test the connection by running a lightweight query
    try:
        cursor = conn.cursor()
        cursor.execute("SELECT 1")  # Lightweight query to validate the connection
    except pymssql.OperationalError as e:
        logging.error(f"Database connection is not active. Error: {e}")
        return

    try:
        for _, row in weight_records.iterrows():
            try:
                # Check if the record exists using the condition
                cursor.execute("""
                    SELECT COUNT(1) FROM dbo.WeightLoss
                    WHERE group_id = %s
                    AND file_code = %s
                    AND WLD = %s
                    AND Weight = %s
                    AND DateDifference = %s
                """, (row['group_id'], row['file_code'], row['WLD'], row['Weight'], row['DateDifference']))

                record_exists = cursor.fetchone()[0]

                if record_exists == 0:  # No record found, perform the insert
                    cursor.execute("""
                        INSERT INTO dbo.WeightLoss (group_id, file_code, WLD, Weight, DateDifference)
                        VALUES (%s, %s, %s, %s, %s)
                    """, (row['group_id'], row['file_code'], row['WLD'], row['Weight'], row['DateDifference']))
                    logging.info(f"Inserted weight record: {row.to_dict()}")
                else:
                    logging.info(f"Skipped weight record (duplicate): {row.to_dict()}")

            except Exception as e:
                logging.error(f"Error processing weight record {row.to_dict()}: {e}")
    except pymssql.Error as db_error:
        logging.error(f"Database error: {db_error}")
    finally:
        conn.commit()
        cursor.close()

```

Function_App

TimerTrigger Function

The function runs on a schedule (every 10 minutes) using the TimerTrigger decorator. It triggers the ETL process by processing blobs in a specified container.

The core ETL process:

1. Lists blobs from the input folder.
2. Downloads and processes each blob.
3. Transforms the data using `pandas`.
4. Inserts transformed data into the database.
5. Archives successful blobs or moves faulty blobs.

Extract:

- List blobs from the input folder using `container_client.list_blobs()`.
- Download blob content and read it into a pandas DataFrame.

Transform:

- Validate required columns (`Sample Number`, `Variety`, `Formulation`, `Number of Samples`).
- Reshape data using `pandas.melt()`.
- Calculate additional metrics (e.g., average weight loss).

Load:

- Insert metadata and weight data into the database using `insert_metadata` and `insert_weights` functions.
- Handle errors with rollback and proper logging.

Archive and Faulty Handling:

- Successful blobs are moved to the archive folder.
- Faulty blobs are moved to the faulty folder.

Below the code link is provided

https://drive.google.com/drive/folders/1FIv4TglDPFCA9nickwoiumRogaEMfbPU?usp=sharing

Commands to Run App:

To run your Azure Function commands in the project environment using **CMD** (Command Prompt) or PowerShell, you need to ensure you're working in the correct environment and using the Azure Functions Core Tools.

Here's a step-by-step guide to run:

1- Install Azure Functions Core Tools (if not already installed):

```
npm install -g azure-functions-core-tools@4 --unsafe-perm true
```

2- Navigate to the project directory, open the cmd in the project directory and run the command.Use the `cd` command to navigate to the directory where your Azure Functions project is located. Example:

```
cd:path-to-your-project\MyAzureFunctionApp\.venv\Scripts\activate
```

3- Set Up Your Environment (If applicable):

- Ensure any virtual environments or dependencies (like Python environments) are activated. If you're using Python, you can activate the environment like this:

```
cd:path-to-your-project\MyAzureFunctionApp
```

4- Run the Azure Function Commands:

- Once you're in the correct environment, you can use the following commands:
- **Publish (Deploy) Changes**: If you made any changes to your function code and need to deploy it to Azure, use the command mention below :

```
func azure functionapp publish FunctionApp12144
```

5- Start the Function Locally or using portal: using portal you can simply click onIf you want to test the function locally (instead of using Azure), use the command mention below :

```
func start --verbose
```