Table of Contents

1. [Introduction 1](#_Toc204183032)
2. [Objective 1](#_Toc204183033)
3. [Key Logic Components 1](#_Toc204183034)
4. [Datasets Used 2](#_Toc204183035)
5. [Configuration Parameters 2](#_Toc204183036)
6. [Functions Overview 2](#_Toc204183037)
7. [Execution Instructions 13](#_Toc204183038)
8. [Logging Behavior 14](#_Toc204183039)
9. [Maintenance Notes 14](#_Toc204183040)
10. [Error Handling 15](#_Toc204183041)

Power BI Dataset Refresh Script

# Introduction

This document outlines the functionality and architecture of the Power BI Dataset Refresh Script. This script streamlines and secures the refresh of multiple Power BI dashboards through scheduled execution and automated error handling. It is particularly suited for environments requiring daily dashboard updates, consistent monitoring, and robust logging without manual intervention.

# Objective

The primary objective of the Power BI Refresh Automation Script is to ensure timely and reliable refreshes of datasets associated with various Power BI users. The script optimizes gateway usage, prevents duplicate operations, and minimizes manual oversight by introducing batching, controlled concurrency, and a retry mechanism. This automation ensures business reports remain updated without the need for direct human involvement.

# Key Logic Components

The script follows the following logic:

* **Batch Refreshing:** Refreshes datasets in batches of 12 (configurable).
* **Concurrency Control:** Limits concurrent refreshes to 4 per batch.
* **Retry Logic:** Retries failed refreshes up to 3 additional times (configurable).
* **Duplicate Prevention:** Eliminates duplicate refresh attempts by identifying dataset and user combinations.
* **Blocked Dataset Filtering:** Excludes datasets listed in **blocked\_datasets.csv**
* **Detailed Logging:** Logs user name, dataset id, refresh status, start/end times, and error messages into an Excel file.

# Datasets Used

Following are the datasets used:

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Name** | **Description** |
| 1. | dashboards.csv | Input file containing user credentials |
| 2. | blocked\_datasets.csv | List of datasets to be excluded from refresh logic |

# Configuration Parameters

| **S.No.** | **Parameter** | **Description** | **Default Value** |
| --- | --- | --- | --- |
| 1. | BATCH\_SIZE | Number of datasets per batch | 12 |
| 2. | CONCURRENT\_REFRESHES | Maximum number of parallel refreshes | 4 |
| 3. | RETRY\_BATCH\_SIZE | Batch size used during retries | 6 |
| 4. | MAX\_RETRIES | Number of retry attempts for failed datasets | 3 |

# Functions Overview

The script contains a number of functions which performs the tasks from loading the datasets to executing the process while handling errors. Every function is modularized to promote maintainability and clarity:

1. **load\_dashboard\_configs:**

def load\_dashboard\_configs(csv\_file):

    dashboards = []

    seen\_users = set()

    with open(csv\_file, mode='r', newline='', encoding='utf-8') as f:

        reader = csv.DictReader(f)

        for row in reader:

            user\_key = (row['User name'], row['Password'])

            if user\_key not in seen\_users:

                seen\_users.add(user\_key)

                dashboards.append(row)

    return dashboards

This function is made to read the **dashboards.csv** file, which contains user credentials required to access and refresh their respective datasets in Power BI Service. To prevent redundant authentication and dataset refresh attempts, the function filters out duplicate entries based on a unique (User name, Password) combination. It returns a clean list of distinct user configurations that are used later in the script for token generation and dashboard operations.

1. **load\_blocked\_datasets:**

def load\_blocked\_datasets(file\_path):

if not os.path.exists(file\_path):

return set()

df = pd.read\_csv(file\_path)

return set(zip(df['User Name'], df['DATASET\_ID']))

This function reads the **blocked\_datasets.csv** file, which contains a list of dataset IDs along with the associated Power BI user names. These datasets are considered "blocked" and should not be refreshed under any circumstances. The function converts the CSV into a set of (**User Name**, **Dataset ID**) tuples for efficient lookup. This set is later used to filter out and exclude blocked datasets from the refresh pipeline before any API calls are made.

1. **get\_token:**

def get\_token(config):

url = f"https://login.microsoftonline.com/{TENANT\_ID}/oauth2/token"

data = {

'grant\_type': 'password',

'resource': RESOURCE,

'client\_id': CLIENT\_ID,

'client\_secret': CLIENT\_SECRET,

'username': config['User name'],

'password': config['Password'],

'scope': SCOPE

}

try:

response = requests.post(url, data=data)

return response.json().get('access\_token')

except Exception as e:

print(f"Token fetch failed for {config['User name']}: {str(e)}")

return None

This function is responsible for acquiring an OAuth2 access token for a specified user by submitting client credentials, resource scope, and user login information to the Azure Active Directory (AAD) token endpoint. The token retrieved is essential for authenticating subsequent API requests to the Power BI REST service.

The parameter **config** is a dictionary-like object that would be derived from each row of the **dashboards.csv** file. It contains user-specific credentials such as User name and Password, which are used to authenticate the request.

It also includes robust error handling. if the authentication request fails for any reason (such as invalid credentials, network issues, or token service unavailability), the function logs a descriptive error message and gracefully returns **None** without halting script execution.

1. **get\_user\_datasets:**

def get\_user\_datasets(user\_config, token):

headers = {'Authorization': f'Bearer {token}'}

response = requests.get("https://api.powerbi.com/v1.0/myorg/datasets", headers=headers)

if response.status\_code == 200:

return response.json().get("value", [])

print(f"Failed to fetch datasets for {user\_config['User name']}: {response.text}")

return []

This function is responsible for retrieving all datasets associated with a given Power BI user account. It uses the access token generated by the **get\_token()** function to make an authenticated GET request to the Power BI REST API endpoint.

This function accepts two parameters:

* **user\_config:** A dictionary-like object derived from a row in the **dashboards.csv** file. It contains user-specific credentials such as User name and Password. These values are used to associate error messages with the correct user if the request fails.
* **token:** A bearer token string obtained from Azure Active Directory via the **get\_token()** function. This token is passed in the Authorization header of the API request to verify identity and permissions.

If the API call succeeds (**status\_code == 200**), the function returns a list of dictionaries representing the metadata of each dataset accessible to that user. If the request fails, the function prints an error message including the user name and the response error text. In such cases, it returns an empty list to allow the script to continue processing other users without interruption.

1. **start\_refresh:**

def start\_refresh(config, token):

headers = {"Authorization": f"Bearer {token}", "Content-Type": "application/json"}

url = f"https://api.powerbi.com/v1.0/myorg/datasets/{config['DATASET\_ID']}/refreshes"

try:

check\_response = requests.get(url, headers=headers)

if check\_response.status\_code == 200:

refreshes = check\_response.json().get("value", [])

if refreshes and refreshes[0]["status"] in ["Unknown", "InProgress"]:

return "InProgress", None

response = requests.post(url, headers=headers)

if response.status\_code == 202:

return "Started", headers

else:

return f"Failed: {response.text}", None

except Exception as e:

return f"Exception: {str(e)}", None

The start\_refresh() function is responsible for safely initiating a Power BI dataset refresh using the Power BI REST API. Its primary goal is to prevent redundant refresh attempts and ensure that only one refresh process runs per dataset at a time. It handles both status-checking and refresh-triggering in a controlled and fault-tolerant manner.

It accepts the following parameters:

* **config**: This is a dictionary containing dataset-related metadata and user credetentials. It includes the DATASET\_ID, User name, and other user credentials which are required for authenticating and identifying the dataset to be refreshed.
* **token**: A valid Bearer token retrieved through the **get\_token()** function. It is used to authenticate API requests on behalf of the user and authorize access to Power BI REST endpoints.

The function begins by constructing the dataset refresh URL using the dataset ID provided in config, and sets up the necessary request headers with the token.

It then sends a **GET** request to the Power BI REST API to check the current refresh history of the dataset. If the API responds with a status code 200, it proceeds to parse the response. Specifically, it looks at the "**value**" list (which contains refresh events) and examines the latest refresh record. If the most recent status is "**Unknown**" or "**InProgress**", this indicates that a refresh is already in progress or pending. In such a case, the function does not attempt a new refresh. Instead, it returns the string "**InProgress**" along with **None** to indicate that a new refresh should not be initiated at this time.

If no active refresh is detected or if the status list is empty, the function sends a **POST** request to the same URL to trigger a new dataset refresh. If this request is successful and the API returns a status code 202, it confirms that the refresh was accepted by the server. The function then returns the string "**Started**" and the request headers for use in further status checks.

If the API responds with any other code (e.g., 400, 403, 404), it implies a failure to start the refresh. In this case, the function returns "Failed: <error message>" and **None** to signal that the request did not succeed.

Finally, if any unexpected error occurs such as network timeouts, connectivity issues, or malformed responses the function catches the exception using a try-except block. It returns "Exception: <error>" to ensure the script continues to run gracefully without interruption.

1. **check\_status()**

def check\_status(dataset\_id, headers):

try:

url = f"https://api.powerbi.com/v1.0/myorg/datasets/{dataset\_id}/refreshes"

response = requests.get(url, headers=headers)

if response.status\_code == 200:

return response.json()["value"][0]["status"]

else:

return f"Check Failed: {response.text}"

except Exception as e:

return f"Check Exception: {str(e)}"

This function is responsible for retrieving the latest refresh status of a Power BI dataset. It accepts the following parameters:

* **dataset\_id**: This is the unique identifier of the Power BI dataset being monitored. It is used to construct the API endpoint URL that queries the refresh history of the corresponding dataset.
* **headers**: This is a dictionary containing the HTTP headers required for authorization. It includes the Bearer token obtained during the authentication process and any content-related headers necessary to interact with the Power BI REST API securely.

The function starts by forming the API endpoint URL using the provided **dataset\_id**. It then performs a **GET** request to the Power BI REST API to fetch the refresh history of the dataset. If the API call is successful and returns a **status code of 200**, it extracts and returns the "**status**" value from the most recent entry in the "**value**" list of the response. This status can be one of "Completed", "Failed", "Unknown", or "InProgress" , each indicating the current state of the dataset's refresh cycle.

If the API responds with any status code other than **200**, it means that the call was not successful. In such cases, the function returns a string message beginning with "**Check Failed:"** followed by the raw response text. This helps in capturing useful error information for debugging and logging.

Furthermore, if an unexpected error or exception occurs such as a network issue or JSON parsing failure, the function catches it using a try-except block and returns a string that starts with "**Check Exception:**" followed by the exception message. This approach ensures the script remains fault-tolerant and continues processing other datasets even if one status check fails.

1. **batch\_process()**

def batch\_process(dashboards, blocked, retry\_mode=False):

refresh\_log = []

failed\_datasets = []

for batch\_start in range(0, len(dashboards), BATCH\_SIZE if not retry\_mode else RETRY\_BATCH\_SIZE):

batch = dashboards[batch\_start:batch\_start + (BATCH\_SIZE if not retry\_mode else RETRY\_BATCH\_SIZE)]

print(f"\nStarting {'Retry' if retry\_mode else 'Main'} Batch: {batch\_start // (BATCH\_SIZE if not retry\_mode else RETRY\_BATCH\_SIZE) + 1}")

refreshing = []

batch\_index = 0

while batch\_index < len(batch) or refreshing:

while len(refreshing) < CONCURRENT\_REFRESHES and batch\_index < len(batch):

config = batch[batch\_index]

dataset\_id = config['DATASET\_ID']

user = config['User name']

batch\_index += 1

print(f"Attempting refresh: Dataset={dataset\_id}, User={user}")

token = get\_token(config)

if not token:

refresh\_log.append(log\_entry(config, "Failed", "Token Error"))

failed\_datasets.append(config)

continue

status, headers = start\_refresh(config, token)

start\_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

if status == "Started":

refreshing.append({"config": config, "headers": headers, "start\_time": start\_time})

elif status == "InProgress":

print(f"Already refreshing: {dataset\_id} for {user}")

else:

print(f"{status}: {dataset\_id} for {user}")

refresh\_log.append(log\_entry(config, "Failed", status, start\_time))

failed\_datasets.append(config)

if refreshing:

print("Waiting 60s to check status...")

time.sleep(60)

still\_refreshing = []

for r in refreshing:

status = check\_status(r['config']['DATASET\_ID'], r['headers'])

end\_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

if status == "Completed":

print(f"Success: {r['config']['DATASET\_ID']} ({r['config']['User name']})")

refresh\_log.append(log\_entry(r['config'], "Success", "", r['start\_time'], end\_time))

elif status == "Failed":

print(f"Failed: {r['config']['DATASET\_ID']} ({r['config']['User name']})")

refresh\_log.append(log\_entry(r['config'], "Failed", "Refresh Failed", r['start\_time'], end\_time))

failed\_datasets.append(r['config'])

else:

still\_refreshing.append(r)

refreshing = still\_refreshing

return refresh\_log, failed\_datasets

This function serves as the execution engine of the script. It manages the end-to-end logic of refreshing datasets in controlled batches, enforcing concurrency limits, handling failures through retries, and generating structured logs. This function is responsible for orchestrating the dataset refresh process across both main execution and retry attempts, ensuring the process is scalable, fault-tolerant, and traceable.

It accepts the following parameters:

* **dashboards:** A list of configuration dictionaries. Each dictionary represents a user-dataset pair and contains the dataset ID, user credentials, and any additional metadata necessary for refresh execution.
* **blocked:** A set of tuples representing the dataset-user combinations that are to be excluded from the refresh process. These combinations are typically loaded from the blocked\_datasets.csv file and ensure that restricted datasets are never sent to the gateway.
* **retry\_mode (default = False):** A boolean flag that determines whether the current execution cycle is part of the initial run (False) or a retry attempt (True). This affects the batch size and logging semantics.

The function begins by determining the appropriate batch size based on whether it is in retry mode or not. It then iterates through the input list in segments (**BATCH\_SIZE** or **RETRY\_BATCH\_SIZE**) and assigns each to a batch variable for processing.

Each batch is processed using a nested while-loop mechanism. Inside the loop, the function refreshes up to a limited number of datasets concurrently, as defined by the **CONCURRENT\_REFRESHES** parameter. It does this by initiating dataset refreshes for each eligible configuration using the **start\_refresh()** function. If a token cannot be acquired for a user, the function logs this failure immediately and skips the current dataset.

For datasets where refresh has been successfully initiated (i.e., **status == "Started"**), the function tracks them in a refreshing list, along with headers and a timestamp. Datasets already in an "**InProgress**" state are acknowledged but not retried immediately, while those that return failure responses are logged and appended to the **failed\_datasets** list.

Once the concurrent refresh queue is full or all datasets in the batch have been attempted, the function pauses execution for 60 seconds using **time.sleep(60)** to allow refresh processes to progress. After the wait, it checks the refresh status of each dataset using the **check\_status()** function. If a dataset has completed successfully, its outcome is logged accordingly. Failures are also logged and retained for potential retry. Datasets still in progress are carried over into the next polling cycle using the **still\_refreshing list**.

This logic continues until all datasets in the batch have either completed, failed, or been retried. The function then proceeds to the next batch (if any) and repeats the process.

Upon completion, the function returns two outputs:

* **refresh\_log:** A list of structured log entries for every attempted refresh, capturing user, dataset ID, timestamps, status, and error messages (if applicable).
* **failed\_datasets:** A list of configurations that failed to refresh even after being attempted during this cycle. This list is used for retry logic in the calling function (main()).

1. **log\_entry()**

def log\_entry(config, status, message, start\_time="", end\_time=""):

return {

"User Name": config["User name"],

"Dataset ID": config["DATASET\_ID"],

"Start Time": start\_time,

"End Time": end\_time,

"Status": status,

"Error Message": message

}

This function is responsible for creating a structured dictionary object that captures the outcome of a dataset refresh attempt. This dictionary is appended to a cumulative list, which is later used to generate a detailed Excel log file summarizing the refresh process for all datasets. This function ensures that each refresh operation—whether successful, failed, or retried—is recorded with sufficient context for auditing, debugging, or historical analysis.

The function accepts the following parameters:

* **config**: This is a dictionary containing user and dataset metadata. It includes the User name and DATASET\_ID, which are extracted from the dataset configuration and stored as part of the log entry. These fields help identify which dataset the log record belongs to and who initiated the refresh.
* **status**: This is a string value indicating the final state of the refresh attempt. It can have values such as "Success", "Failed", "Retry Success (1)", or "Retry Failed (2)", depending on the outcome and retry count. It provides a clear status marker for each entry in the log file.
* **message**: This parameter captures any error message or contextual note associated with the refresh attempt. It is particularly useful for logging API failures, exceptions, or validation errors returned from Power BI.
* **start\_time** (optional): This string captures the timestamp when the refresh was initiated. Including this value helps in calculating refresh duration and understanding execution timelines. If not provided, it defaults to an empty string.
* **end\_time** (optional): This string records the timestamp when the refresh completed (whether successful or not).

The function returns a dictionary with all the above fields mapped to their corresponding values.

1. **main()**

def main():

all\_users = load\_dashboard\_configs(CSV\_FILE)

blocked = load\_blocked\_datasets(BLOCKED\_FILE)

dashboards = []

for user in all\_users:

token = get\_token(user)

if not token:

continue

datasets = get\_user\_datasets(user, token)

for ds in datasets:

key = (user['User name'], ds['id'])

if key in blocked:

continue

config = user.copy()

config['DATASET\_ID'] = ds['id']

dashboards.append(config)

logs, failed = batch\_process(dashboards, blocked, retry\_mode=False)

for attempt in range(1, MAX\_RETRIES + 1):

if not failed:

break

print(f"\nRetry Attempt {attempt} for {len(failed)} failed datasets")

retry\_logs, failed = batch\_process(failed, blocked, retry\_mode=True)

for log in retry\_logs:

if log["Status"] == "Success":

log["Status"] = f"Retry Success ({attempt})"

elif log["Status"] == "Failed":

log["Status"] = f"Retry Failed ({attempt})"

logs.extend(retry\_logs)

df\_log = pd.DataFrame(logs)

today\_str = time.strftime("%d-%m-%Y")

filename = f"refresh\_log\_{today\_str}.xlsx"

df\_log.to\_excel(filename, index=False)

print(f"\nRefresh log saved to {filename} in {os.getcwd()}")

print("All refresh operations completed.")

The main() function serves as the central coordinator for the entire Power BI dataset refresh automation workflow. It is the primary entry point for execution and oversees the complete lifecycle of dataset processing: from configuration loading and user authentication to refresh initiation, retry handling, and final logging.

The function does not accept any parameters directly but internally depends on the following global constants and configuration files:

* **CSV\_FILE**: Refers to the **dashboards.csv** file that contains user credentials and category metadata.
* **BLOCKED\_FILE**: Refers to the **blocked\_datasets.csv** file, which lists dataset and user combinations that should be excluded from the refresh process.
* **MAX\_RETRIES**: Specifies how many times the script will retry failed dataset refreshes. This is applied in the retry loop.

The execution begins by calling **load\_dashboard\_configs()** to read user information from the CSV file and eliminate duplicate credentials. Simultaneously, **load\_blocked\_datasets()** is invoked to create a set of blocked dataset/user combinations for later exclusion.

Next, the script enters a loop over all users. For each user, it retrieves an authentication token using **get\_token()**. If the token cannot be obtained (e.g., due to invalid credentials or API failures), that user is skipped. Otherwise, the script calls **get\_user\_datasets()** to fetch all datasets accessible by that user.

Each dataset is evaluated against the blocked set. If it is not blocked, a new configuration dictionary is constructed by copying the user's credentials and appending the **DATASET\_ID**. This dictionary is added to a master list called dashboards, which aggregates all eligible dataset refresh jobs across users.

Once the dashboards list is populated, it is passed to the batch\_process() function to initiate the first round of dataset refreshes. This function returns two outputs:

* **logs**: A list of refresh attempt results, including success, failure, and associated metadata.
* **failed**: A list of datasets that failed to refresh during the first round and are eligible for retries.

The script then enters a retry loop that runs up to **MAX\_RETRIES** times. On each iteration, it prints a retry message and re-submits the list of failed datasets to **batch\_process()** with **retry\_mode=True**, allowing it to apply a smaller retry batch size. After each retry round, the retry results are tagged with either "Retry Success (n)" or "Retry Failed (n)", where n represents the attempt number. All retry logs are appended to the master logs list for comprehensive tracking.

Finally, the script compiles the full log data into a pandas DataFrame, writes it to an Excel file named **refresh\_log\_<date>.xlsx**, and saves it to the working directory. A confirmation message is printed with the saved filename and the current path, marking the successful completion of the refresh cycle.

# Execution Instructions

* 1. Ensure Python 3.9+ and the following packages are installed:
* pandas
* requests
* openpyxl
  1. Place the following files in the same directory:
* Dashboards Refresh\_v2.4.py
* dashboards.csv
* blocked\_datasets.csv

# Logging Behavior

The script provides real-time visibility into execution progress and records detailed refresh logs. Logging behavior includes:

* **Terminal Output**:
  + Batch initiation messages (Main and Retry batches)
  + Dataset ID and associated user for each refresh attempt
  + Refresh status updates: start, in progress, success, and failure
  + Retry attempt announcements and retry outcome
* **Excel Log Output**:
  + One log file is generated per day
  + Each dataset processed is tracked with start/end times, status, and errors

This dual logging approach helps with both live monitoring and post-run audits.

# Maintenance Notes

To keep the automation efficient and sustainable, the following maintenance practices are recommended:

* **Parameter Tuning**: Adjust MAX\_RETRIES, BATCH\_SIZE, and CONCURRENT\_REFRESHES based on dataset size, frequency, and available gateway capacity.
* **Blocked Dataset Review**: Keep the blocked\_datasets.csv file updated to reflect datasets that should not be refreshed, especially deprecated or archived datasets.
* **Log Management**: Implement a log archival or rotation strategy to prevent clutter and ensure long-term traceability. Old log files can be zipped, timestamped, or moved to a dedicated archival folder for reference.

# Error Handling

Token fetch failures, dataset access issues, and API errors (e.g., "Invalid dataset. This API can only be called on a Model-based dataset") are logged and do not break the script.