Pakistan Civil Aviation Authority Summer Internship 2023, IT Department.

Automated Email Attachment Download and Sorting System

Submitted By:
Muhammad Owais.
Senior Year student Ghulam Ishaq Khan Institute.
Intern HQCAA IT Department.

Supervisor : Nadeem'ullah Baig

Catalog

Email Attachment Download and Sorting System

Introduction

The Automated Email Attachment Download and Sorting System is a Python-based solution designed to automate the process of downloading email attachments from Microsoft Outlook and organizing them into separate folders based on their file types.
This report provides a detailed overview of the system, including its limitations, adaptations for other email clients, storing data logs in a database, and considerations for running the system on a server.
Additionally, it highlights the specific application of the system in the context of private email servers, focusing on the case of the Pakistan Civil Aviation Authority.

Limitations and Recent Updates

Limitations Working on Microsoft Outlook:

The system is primarily developed for use with Microsoft Outlook and utilizes the **win32com.client** library, which is specifically designed for Outlook integration. Consequently, it may not be directly compatible with other email clients. It is important to note that the system assumes a Windows environment due to its reliance on the Win32 API. Furthermore, changes in the Outlook API or email client behavior may require code adjustments to maintain compatibility.

Update-Induced Changes and the Importance of Message ID:

A recent update in the Outlook application prompted us to modify the code and introduce new features, such as the inclusion of the Message ID. This unique identifier allows for accurate tracking and prevention of duplicate entries in the email log. By comparing the Message ID with existing entries, the system ensures that previously downloaded attachments are not reprocessed, thereby optimizing efficiency and preventing data redundancy.

Permissions and Access Challenges

Permission Popups:

When accessing Outlook programmatically, user may encounter permission popups that require manual interaction for authentication or authorization. These popups are a security measures recently implemented by Outlook to ensure secure access to user accounts. Although they can interrupt the automated process, *they can be handled by granting the necessary permissions or utilizing techniques such as running the code with elevated privileges.*

Making of an Executable File

Converting the Jupyter Notebook to Python3 Script:

The Jupyter Notebook code for the Email Attachment Download and Sorting System has been transformed into a modular Python script (Provided at the end with analysis). This modularization approach involves breaking down the code into separate functions, each with a specific task.
Functions such as **`load_logs`**, **`retrieve_inbox_messages`**, **`download_attachments`**, **`sort_attachments_by_extension`**, and **`execute_email_attachment_download_sorting`** were created to handle different aspects of the system.
The new modular structure improves code organization, readability, and maintainability. The conversion to a Python script enhances the system's usability and allows for easier future modifications and integration with other projects or applications.

Run as a Background Process:

To convert the code into an executable file that runs in the system tray and monitors new emails for attachments, we can use Python library called `pystray`. `pystray` provides a simple interface for creating system tray applications in Python.

Using ‘pystray’:

Install the **`pystray`** library and any other dependencies required by your code. Use `pystray` to create the system tray icon. We can define an icon image and a menu for our new tray icon. The menu can include options such as "Start", "Stop", and "Exit" to control the execution of the email attachment download and sorting process.

Then we would have to monitor for new emails periodically. We can use a loop with a delay or schedule the code to run at regular intervals using a scheduler library like `schedule`. Within the monitoring mechanism, call the function to check for new emails and perform the necessary operations.

Executable file :

Build the executable file: Use a tool like **`pyinstaller`** to build the executable file from your Python code. `pyinstaller` packages your Python script and its dependencies into a standalone executable that can be run on Windows.

Adaptation for Other Email Clients

Identifying the Target Email Client:

The Email Attachment Download and Sorting System, originally designed for Microsoft Outlook, can be adapted to work with other email clients as well. Popular email clients such

as Gmail, Yahoo Mail, and Exchange are among the potential candidates. To make the system compatible with a different email client, some modifications to the code are necessary.
The first step in adapting the system is to identify the target email client. Whether it's Gmail, Yahoo Mail, or Exchange, understanding the specific requirements and functionalities of the chosen client is crucial. Each email client has its own set of protocols, APIs, and libraries that facilitate interaction with its services.

Utilizing Libraries and APIs:

Once the target email client is determined, the next step is to explore the available Python libraries or APIs designed for that particular client. These libraries offer functionality to connect to the email client's server, retrieve emails, access attachments, and perform other necessary operations.
For example, when working with Gmail, the Gmail API can be utilized, which provides a comprehensive set of methods for interacting with Gmail accounts programmatically. Similarly, the Yahoo Mail API can be employed for Yahoo Mail integration, and the Exchange Web Services (EWS) API can be leveraged for Exchange email integration.

Adapting the Code:

To adapt the system for a different email client, modifications to the code are required to align it with the functionalities and methods provided by the chosen library or API. This involves updating the code logic for email retrieval, message parsing, attachment handling, and other relevant operations.
For instance, if using the Gmail API, the code needs to be adjusted to use the API's authentication mechanism, endpoint URLs, and methods for fetching emails and attachments. Similarly, for Yahoo Mail or Exchange, the code must be modified to utilize the specific APIs and their respective authentication protocols.
By making these necessary changes, the Email Attachment Download and Sorting System can be seamlessly integrated with other email clients, extending its capabilities beyond Microsoft Outlook.

To adapt the system for different email clients, you can consider using the following libraries and APIs:

1. **Gmail API:** Enables programmatic access to Gmail accounts, providing methods for authentication, email retrieval, message parsing, and attachment handling.
2. **Yahoo Mail API:** Allows integration with Yahoo Mail accounts, offering APIs for authentication, email retrieval, message management, and attachment handling specific to Yahoo Mail.

3. **Exchange Web Services (EWS) API:** Designed for integration with Microsoft Exchange email servers, offering functionalities for authentication, email retrieval, message manipulation, and attachment handling within an Exchange environment.

4. **IMAP Libraries:** General-purpose libraries like imaplib provide methods for connecting to email servers, retrieving emails, and handling attachments using the IMAP protocol.

5. **POP3 Libraries:** General-purpose libraries like poplib provide methods for connecting to email servers, retrieving emails, and handling attachments using the POP3 protocol.

6. **SMTP Libraries:** SMTP libraries like smtplib facilitate the integration of email-sending capabilities into the system, allowing automated email responses or notifications.

Please note that while the system can be adapted for other email clients, this report primarily focuses on its implementation and limitations when working with Microsoft Outlook.

Storing Data Logs in a Database

A Choosing a Database System:

To store data logs in a database, users need to select a suitable database system based on their requirements. Options include SQLite, MySQL, PostgreSQL, or Microsoft SQL Server.

Establishing a Database Connection:

Python database libraries such as sqlite3, MySQLdb, psycopg2, or pyodbc can be used to establish a connection to the selected database. These libraries enable the insertion of log data into the corresponding database tables using SQL statements or object-relational mapping (ORM) techniques provided by the libraries.

Running the System on a Server

Deploying the System on a Server:

To run the system on a server, users must set up a server environment with the necessary dependencies and libraries installed. It is crucial to ensure that the server has access to the email accounts and the required network connectivity.

Adapting File Paths and Permissions:

Code modifications are necessary to specify appropriate file paths on the server for saving downloaded attachments. Adjustments to file permissions should be made to enable the server to write and organize the attachments in the desired folders.

Handling Email Retrieval:

When deploying the system on a server, the code needs to be adjusted to connect directly to the email server using IMAP or POP3 protocols. Emails can be retrieved from the server environment, and the code logic should be updated to process emails on the server rather than relying on the local Outlook application.

Maintaining Data Logs:

Data logs, whether stored in a database or other means, should be accessible to the server environment to record relevant information about the downloaded attachments. The server environment should have appropriate access privileges to read and write data logs as required.

Timeline of Progress:

1. Initial Research and Understanding of Requirements
   a) Searching and looking for ways to access emails programmatically.
2. Setting Up the Development Environment
   a) Finding the right python libraries.
3. Code Development for Email Retrieval and Attachment Handling **
   a) Created multiple notebook scripts till got the one working well.
4. Testing and Debugging of the System *
   a) Created 5-6 prototypes, each having different approach :
      https://github.com/MuhammadOwais02/Pakistan-Civil-Aviation-Authority-Internship/tree/main/prototypes
5. Integration of Message ID and Permission Handling **
   a) After the outlook update had to replace some code due to new firewalls.
6. Making a separate working modular ".py" file that runs in background and completes the tasks.

Purposed in this report:

1. *Adaptation for Other Email Clients and Database Integration*
2. *Server Deployment Considerations and Code Modifications*
3. *Refinement and Optimization of the System*

Conclusion:

The Automated Email Attachment Download and Sorting System provides a practical and efficient solution for automating the process of managing email attachments in Microsoft Outlook. While it has certain limitations, particularly in its dependence on Outlook and Windows, the system can be adapted for other email clients with the appropriate modifications.

Code (google colab):

https://colab.research.google.com/drive/1BoBqdZUhkBJfhjR1Fo2TuTI1UkncLRty?usp=sharing

Code Analysis (Jupyter Notebook '.ipynb'):

1. The code starts by importing necessary libraries and modules, including win32com.client for Outlook integration, os for file and folder operations, and pandas for data handling.

2. It creates a DataFrame to store the email log and sets the file path for the email log file. If the file already exists, the code loads the existing log data into the DataFrames. Otherwise, it creates new empty logs.

3. The code establishes a connection with Outlook and retrieves the inbox folder. It creates a folder to save the attachments, ensuring that the folder exists or is created if it doesn't.

4. The code iterates over the unread emails in the inbox, checks if the emails contain attachments, and retrieves relevant information such as sender, receiver, time, subject, and message ID.

5. It checks if the message ID is already present in the email log to avoid duplicate entries. If the email is not in the log, it adds the email information to the email log DataFrame and proceeds to process the attachments. For each attachment, the code saves the attachment to the specified file path, generates a new filename with a unique identifier, and records the attachment information in the attachments log DataFrame. **(Both the DataFrames are created in a manner that will be helpful to manage them in a database.)**

6. After processing all the attachments, the code updates the serial counters and saves the updated email log and attachments log to the specified Excel file.

7. The code includes a function sort_attachments_by_extension that creates separate folders for each unique file extension encountered within the attachments folder. It moves the files with corresponding extensions to their respective folders.

Code Analysis Summary:

The provided code effectively handles the email attachment downloading and log updating process in Microsoft Outlook. It demonstrates a clear workflow by connecting to Outlook, retrieving emails, saving attachments, and maintaining data logs. The inclusion of the sort_attachments_by_extension function further enhances the organization of attachments by grouping them based on file extensions. However, to achieve the practical goals mentioned in the report about the this service's prototype, some adaptations may be required. Additionally, the code can be optimized and modularized to improve code readability and maintainability.

Code (modular python '.py' script):

```
import win32com.client
import os
import pandas as pd
import shutil

# Function to load the email log and attachments log
def load_logs(email_log_filepath):
    if os.path.exists(email_log_filepath):
        email_log = pd.read_excel(email_log_filepath, sheet_name='EmailLog')
        attachments_log = pd.read_excel(email_log_filepath, sheet_name='AttachmentsLog')
        last_email_serial = email_log['EmailSerial'].max()                  #getting the last one
        last_attachment_serial = attachments_log['AttachmentSerial'].max()  #getting the last ones
        email_serial_counter = last_email_serial + 1
        attachment_serial_counter = last_attachment_serial + 1
    else:
        email_log = pd.DataFrame(columns=['EmailSerial', 'Sender', 'Receiver', 'Time', 'Subject', 'MessageID'])
        attachments_log = pd.DataFrame(columns=['EmailSerial', 'AttachmentSerial', 'FileType',
'RenamedAttachment'])
        email_serial_counter = 1
        attachment_serial_counter = 1
    return email_log, attachments_log, email_serial_counter, attachment_serial_counter

# Function to access Outlook and retrieve inbox messages
def retrieve_inbox_messages():
    outlook = win32com.client.Dispatch("Outlook.Application")
    mapi = outlook.GetNamespace("MAPI")
    inbox = mapi.GetDefaultFolder(6)  # Inbox folder
    messages = inbox.Items
    return messages

# Function to download attachments and update the log
def download_attachments(messages, email_log, attachments_log, email_serial_counter, attachment_serial_counter,
attachments_folder):
    for message in messages:
        if message.Unread and message.Attachments.Count > 0:
            sender = message.Sender
            receiver = message.ReceivedByName
            time = message.ReceivedTime
            subject = message.Subject
            if (email_log['MessageID'] == message.EntryID).any():
                continue
            email_log = email_log.append({
                'EmailSerial': email_serial_counter,
                'Sender': sender,
                'Receiver': receiver,
                'Time': str(time),
                'Subject': subject,
                'MessageID': message.EntryID
            }, ignore_index=True)
            attachments_info = []

            for attachment in message.Attachments:
                filename = attachment.FileName
                file_extension = os.path.splitext(filename)[1]
                new_filename = f"{email_serial_counter}_{attachment_serial_counter}_{filename}"
                filepath = os.path.abspath(os.path.join(attachments_folder, new_filename))
```

```
                if (attachments_log['RenamedAttachment'] == new_filename).any():
                    continue
                attachment.SaveAsFile(filepath)
                attachments_info.append({
                    'EmailSerial': email_serial_counter,
                    'AttachmentSerial': attachment_serial_counter,
                    'FileType': file_extension,
                    'RenamedAttachment': new_filename
                })
                attachment_serial_counter += 1
            for attachment_info in attachments_info:
                #attachments_log = attachments_log.append(attachment_info, ignore_index=True)
                attachments_log = pd.concat([attachments_log, pd.DataFrame([attachment_info])],
ignore_index=True)
            email_serial_counter += 1
    return email_log, attachments_log, email_serial_counter, attachment_serial_counter

# Function to sort attachments by extension
def sort_attachments_by_extension(attachments_folder):
    files = os.listdir(attachments_folder)
    extension_folders = {}
    for file in files:
        file_path = os.path.join(attachments_folder, file)
        if not os.path.isfile(file_path) or os.path.dirname(file_path) != attachments_folder:
            continue
        file_extension = os.path.splitext(file)[1]
        if file_extension not in extension_folders:
            extension_folder = os.path.join(attachments_folder, f"{file_extension}_folder")
            os.makedirs(extension_folder, exist_ok=True)
            extension_folders[file_extension] = extension_folder
        destination_folder = extension_folders[file_extension]
        destination_path = os.path.join(destination_folder, file)
        shutil.move(file_path, destination_path)

# Main function to execute the email attachment download and sorting process
def execute_email_attachment_download_sorting(email_log_filepath, attachments_folder):
    email_log, attachments_log, email_serial_counter, attachment_serial_counter = load_logs(email_log_filepath)
    messages = retrieve_inbox_messages()
    email_log, attachments_log, email_serial_counter, attachment_serial_counter = download_attachments(
        messages, email_log, attachments_log, email_serial_counter, attachment_serial_counter,
attachments_folder
    )
    with pd.ExcelWriter(email_log_filepath) as writer:
        email_log.to_excel(writer, sheet_name='EmailLog', index=False)
        attachments_log.to_excel(writer, sheet_name='AttachmentsLog', index=False)
    sort_attachments_by_extension(attachments_folder)

# Execute the main function
execute_email_attachment_download_sorting(
    email_log_filepath=r"D:\GIKI\CAA_intern\email_log_with_attachments.xlsx",
    attachments_folder=r"D:\GIKI\CAA_intern\attachments"
)
```

Code Analysis (python '.py' script):

The provided code is a modular implementation of an email attachment download and sorting process. It performs the following tasks:

1. **Load Logs** (`load_logs` function):
   This function reads the existing email log and attachments log from an Excel file if it exists. It retrieves the last serial numbers from the logs to increment the serial counters for new entries. If the file doesn't exist, it initializes empty logs with default serial counters.

2. **Retrieve Inbox Messages** (`retrieve_inbox_messages` function):
   This function accesses Microsoft Outlook using the `win32com.client` library and retrieves the messages from the inbox folder. It returns the collection of messages for further processing.

3. **Download Attachments and Update Log** (`download_attachments` function):
   This function iterates over the messages and checks for unread messages with attachments. It extracts relevant information such as sender, receiver, time, subject, and message ID. It saves the attachments to a specified folder, renames them with unique identifiers, and updates the email log and attachments log accordingly. It avoids processing duplicate messages or attachments that already exist in the logs.

4. **Sort Attachments by Extension** (`sort_attachments_by_extension` function):
   This function sorts the downloaded attachments into separate folders based on their file extensions. It creates a folder for each unique extension encountered and moves the files with that extension to their respective folders within the attachments folder.

5. **Execute the Main Function** (`execute_email_attachment_download_sorting` function):
   This function serves as the entry point of the program. It calls the `load_logs` function to initialize the logs, retrieves the inbox messages using `retrieve_inbox_messages`, and performs attachment download, log update, and sorting using `download_attachments` and `sort_attachments_by_extension`. It saves the updated logs to an Excel file.

By combining these functions and executing the main function, the code automates the process of downloading email attachments, updating logs, and sorting attachments based on their file extensions.

Note that the code assumes the use of Microsoft Outlook and the presence of an Excel file for storing the logs. It relies on the `win32com.client` library for Outlook access and the `pandas` library for reading and writing Excel files. Additionally, the code uses the `os` and `shutil` libraries for file operations.