A ROBUST SET OF EMAILING APPLICATIONS THAT ALLOW AN EFFECTIVE SOLUTION FOR BULK EMAIL SENDING AND RECEIVING

Zixuan Chen[1], Garret Washburn[2]

[1]Trinity Preparatory School, 5700 Trinity Prep Ln, Winter Park, FL 32792
[2]Computer Science Department, California State Polytechnic University, Pomona, CA 91768

ABSTRACT

In the current digital age, the capability of a business or workforce team to send out or receive a mass amount of emails is increasingly becoming more important. However, with this becoming more and more of a necessity, the tools to do so are not developing as rapidly. The method this paper seeks to provide as a solution to this problem is the Email Sender and Receiver applications that are described within this paper [1]. The main technologies used to create these Python applications behind the scenes are the Mailgun API, the Gmail API, and the PySimpleGui Python library [2]. One of the challenges that we faced during the development of the Email Sender and Receiver applications was the scheduling of emails for recipients in different timezones, as it is important that the sending of emails be accurate to their time across the globe. Another challenge faced during development was the configuration and maintenance of the Google Project in order to use the Gmail API, as the backend for the Email Receiver application relies on the Gmail API to grab from the user account [3]. To ensure the functionality and efficiency of the Email Sender and Receiver applications, we conducted experiments on both applications to test their limits. For the Email Sender application, an experiment was conducted to see how the size of the email being sent would affect how the application runs, and we found the application to be quite stable. Additionally, for the Email Receiver application, another experiment was conducted to test and see how far into the past we could grab emails and not impact the application. As a result, we found the Gmail API limits how many emails you may grab in one request. To conclude, the Email Sender and Receiver applications are viable tools for companies and individuals alike, as both applications provide an easy-to-use bulk email tool that doesn't compromise on functionality.

KEYWORDS

Bulk Email, Email System, Software, Bulk Email System

1. INTRODUCTION

A major problem for companies today is getting mass amounts of emails out to each client or member of a team. Some companies see sending these copious amounts of emails as a necessity for business, as client acquisition is a major part of any business model. Additionally, reaching out to the maximum possible number of individuals for a job position is beneficial for a company, as it will give them a lot more options and potential employees. This problem affects not only the businesses who consider it a necessity to send mass amounts of emails to clients but also the individuals who need to receive emails in a short time. As a matter of fact, email marketing is

David C. Wyld et al. (Eds): SPTM, AIS, SOENG, AMLA, NLPA, IPPR, CSIT – 2024
pp. 127-137, 2024. CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.141111

such a necessity in businesses, as studies have shown that "for every dollar spent on email marketing, you can expect an estimated return of $46. If that didn't get your attention, maybe this will: Email marketing delivers the highest return on investment (ROI) of any other Internet marketing approach available [4]." Additionally, in a study conducted in an office setting, "58% of mass emails were reported being opened by staff and only 28% of them were read in detail," which doesn't sound like a lot, however, 28% is still very cost productive considering the low cost. If this problem was never solved with some software solution, and businesses kept doing things "the old fashioned way," businesses would have to hire hundreds of employees to complete tasks that a single program can do in way less time.

In the research paper, you are about to read, three differing methodologies of solving the same problem outlined in this paper are covered to convey the necessity of the methodology proposed in this paper. These three methodologies are the Mailgun API and the Mailchimp and SendPulse services [5].

The Mailgun API is a methodology that attempts to solve the problem of bulk email sending and receiving, however, it lacks simplicity and a formal fronted user interface. Our applications seek to streamline the front end but create a simple, easy-to-use interface.

Mailchimp is an online service that seeks to standardize mass email sending and receiving with a web interface. Mailchimp, as it is an online service, requires a lot of configuration and personably identifiable information to set up an account. The Email Sender and Receiver applications resolve this issue by not having any formal login, minus the configuration for the Gmail account setup.

The final methodology, SendPulse, is an entirely online web application that seeks to solve the problem of sending and receiving mass emails only with its online interface. This, to some, is a fine quality. However, there is no denying the simplicity of an entirely on-device application, which the Email Sender and Receiver applications seek to be.

The proposed solution to the problem of the necessity of bulk email sending in this paper is a set of GUI applications designed to send and receive mass amounts of emails from a database of email recipients [6]. This designed solution would solve this problem as it would enable companies and individuals to easily send bulk emails in a short amount of time, as well as to receive and sort email responses from the clients. This solution would not only effectively solve this problem, but it would also make the process of sending and receiving bulk emails much simpler and more streamlined through the use of GUIs. The GUIs designed for the applications are purposefully crafted to be simple, and enable the user to use them with very little confusion. Additionally, the bulk email sender application, unlike other solutions, enables sender to utilize a csv file that includes email addresses as well as other identifiable information to compose personalized emails for each user. Also, the bulk email sender also has a scheduling feature, allowing the user to schedule all emails for a specific time corresponding to the timezone. Conversely, the bulk email receiver enables the user to set a specific time up until they would like to receive their emails, allowing easy tracking of the received response emails from the clients [7]. Both applications are designed to maintain simplicity while upholding functionality, enabling the user a smooth experience sending and receiving bulk emails as opposed to other solutions that are needlessly complicated or don't work well.

The experiments performed in this paper were designed to test and ensure the proper functionality of the proposed Email Sender and Receiver applications. The first experiment delved into the time complexity of how long it would take to load emails into the application given a set time into the past. Throughout the experiment process, we noticed that after a certain number of emails, the gmail api tended to cap off and not return any more emails for viewing,

which ensured the stability of the Email Receiver. The second experiment carried out and displayed in this paper was designed to test the Email Sender application, specifically, to see how the size of the HTML would affect the time it takes for the user to receive the email. This experiment revealed that the relationship between html size and send time was linear, however, the difference was quite minute and was essentially the same. After performing these experiments, we were quite convinced in the stability and efficacy of the Email Sender and Receiver Applications.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Time Conversion

One major component of our toolset applications was the sendEmails function which includes the use of the official Mailgun API. One of the biggest things we had to consider when developing this function was the time conversion for clients potentially in different time zones. To solve this important problem, we created and employed the use of various time zone calculation functions. Within these timezone calculation functions, the most prominent Python module we utilized was the datetime library, which enables the creation of datetime objects that can be manipulated and converted to different formats. Another issue we had was planning out how we could use the Mailgun API for every email in a CSV file [8]. To solve this problem, we utilized the pandas python library, which makes the process of iterating through CSV files much easier.

2.2. Gmail API

Another key functionality of our toolset applications was the getEmails function that utilizes the official Gmail API. The official Gmail API employs the use of Google Projects, in that the Gmail API is enabled to be used inside of a Google Project and all Gmails associated with the Google project are what the Gmail API gets emails from. The nature of this part of the project was very complicated because there were now two steps of verification just to grab emails from a Gmail account, the credentials for the Google Project and the tokens for each user, proving they are a part of the Google Project.

2.3. Integrate The Functions

The last component of the bulk email toolset applications that made up a very crucial part of the overall project was the GUI's for each application. To create these GUI's, we employed the use of the python module PySimpleGUI, which is a python library that prides itself on being a simple way to make GUI's. A very prominent challenge we faced when designing the GUI's, was how we could integrate the sendEmails and getEmails functions, as they required a lot of different inputs that all had to be precise. To ensure that all inputs were filled out, we created global variables that were checked every time the user tried submitting their request with the GUI's.

3. SOLUTION

The overall structure of the bulk email toolset applications are made up of three major components, the sendEmails and getEmails functions, as well as the GUI's. Due to the nature of this project, the applications are actually split up, making two different paths that the structure flows through. To send emails, the flow of the application starts with the GUI. Within the GUI, the user inputs all of the required information for when they would like their email to be sent,

after which, the program sends all of the information to the MailGun API so that the MailGun API may schedule and eventually send the email at the inputted time. On the other hand, to receive emails the flow is very similar, however, different in that it only receives emails. The first step for the user to receive emails is to fill out the getEmails GUI with all of the required information and credentials. After these have been filled out, the user can submit the request and contact the Gmail API, which will return any available emails that fit within the parameters described by the user in the GUI. To build these flows in our application, we employed the use of two different GUI files, which were called sendEmailGui and receiveEmailGui [9]. Each of these GUI files used the PySimpleGui python library to create two GUI structures, in which we custom fit the inputs required for both the sendEmail and getEmail functions. This structure for our applications enabled us to maintain a functional GUI set up for what our applications required.

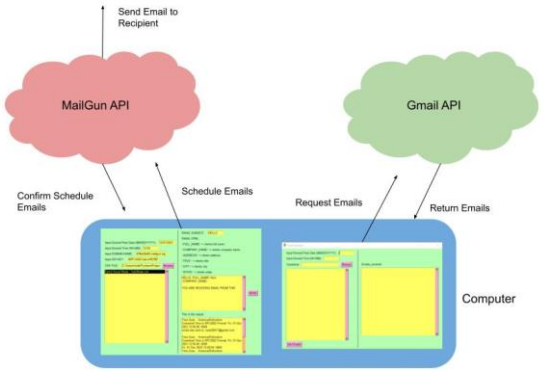

Figure 1. Overview of the solution

The scheduleEmail function included in this project is what actually sends the post request to the Mailgun API. This actually is what takes in all of the user inputted information, such as desired time, API key, email html, etc., and performs the http request containing all of the data to the MailGun API server.

Figure 2. Screenshot of the server

```
### METHOD TO CREATE EMAIL ###
def scheduleEmail(DOMAIN_NAME, API_KEY, email_subject, email_html, recipient,
                  year, month, day, hour, minute):
    ### GET LOCAL TIME ###
    timezone = get_timezone(recipient['City'], recipient['State'],
                            recipient['Zip Code'], recipient['Country'])
    local_rfc2822_time = get_rfc2822_time(year, month, day, hour, minute,
                                          timezone)

    email_html = email_html.replace("~FULL_NAME~", recipient['Full Name'])
    email_html = email_html.replace("~COMPANY_NAME~", recipient['Company Name'])
    email_html = email_html.replace("~TITLE~", recipient['Title'])
    email_html = email_html.replace("~ADDRESS~", recipient['Address'])
    email_html = email_html.replace("~CITY~", recipient['City'])
    email_html = email_html.replace("~STATE~", recipient['State'])

    ### SEND POST REQUEST TO API ###
    requests.post("https://api.mailgun.net/v3/" + DOMAIN_NAME + "/messages",
                  auth=("api", API_KEY),
                  data={
                      "from": "Excited User <postmaster@" + DOMAIN_NAME + ">",
                      "to": recipient['Email'],
                      "subject": email_subject,
                      "html": email_html,
                      "o:deliverytime": local_rfc2822_time,
                      'o:tracking-opens': True
                  })

    print("email was sent to: " + recipient['Email'] + "\n")
```

Figure 3. Screenshot of code 1

The first thing that is done in the scheduleEmail function is the grabbing of the proper timezone based on the information from the recipient, that is passed in the recipient parameter of the function. Upon grabbing the correct timezone info for the recipient, the timezone information is then converted into the rfc2822 format, as the MailGun API requires this format. After the proper timezone has been created, the email_html is sorted for the 'key words' that are to be replaced from information from the recipient that is defined in the user inputted csv file. After the email_html has been processed, the POST request is sent containing all of the information in the proper format that is required by MailGun. All of the parameters that are given for this function are inputted by the user through the GUI, and the DOMAIN_NAME and API_KEY are created from MailGun by signing up for an account and creating a domain and api key pair.

Another major component within the project is the emailReceiver function, which does all of the email grabbing in the background for the Email Receiver GUI. Specifically, the emailReceiver function utilizes the Gmail api as a component of a Google Project that the email is a part of, and grabs emails from the Gmail accounts associated with the Google Project.

Figure 4. Screenshot of the boxes

```
def getEmails(timestamp, credentials_name):
  # If modifying these scopes, delete the file token.json.
  SCOPES = ["https://www.googleapis.com/auth/gmail.readonly"]
  creds = None
### CONNECT TO THE GMAIL API ###
service = build("gmail", "v1", credentials=creds)
results = service.users().messages().list(userId="me",
                                          q=f"in: after:{timestamp}",
                                          labelIds=['INBOX']).execute()
messages = results.get("messages")

# create array of emails sent to us
info = []

# check if messages is empty
if messages != None:
  for msg in messages:
    txt = service.users().messages().get(userId="me",
                                         id=msg["id"],
                                         format="full").execute()

    try:
      # getting the value of 'payload' and the headers inside of it
      payload = txt["payload"]
      headers = payload["headers"]

      for header in headers:
        if header['name'] == 'Subject':
          Subject = header['value']
        if header['name'] == 'From':
          Sender = header['value']

      # grabbing the parts of the email
      parts = payload.get("parts")[0]
      data = parts['body']['data']
      data = data.replace("-", "+").replace("_", "/")

      msg = base64.urlsafe_b64decode(data.encode("UTF8"))
      body = str(email.message_from_bytes(msg))

      #print(body)
      info.append((Sender, Subject, body))

    except Exception as e:
      #print(e)
      continue

return info
```

Figure 5. Screenshot of code 2

The code depicted above is the major part of the emailReceiver function, as it displays all of the major functionalities of contacting the Google project and using the Gmail API to grab emails. Firstly, the code builds the ‘service’ connection to the Google Project with our credentials, and specifies the intention to utilize the “gmail” api. Upon building the service connection, the code then uses the service to access and grab an unsorted list of email information by accessing the users messages and grabbing all of the most current emails from a specific timestamp. The timestamp is an inputted parameter, and varies on what the user inputs. After creating the unsorted list ‘results’, the function then accesses the messages inside of the list, and for each message it will go through and sort all items inside the message and display them. The contents of each message are sorted as items Sender, Subject, and body, and are appended to the ‘info’ array so that all three contents for each message may be returned to the GUI screen.

The Graphic User Interface is also an essential component of the project as it ensures easy and efficient interaction with the user [10]. The backbone library for the GUI in the project is PySimpleGui, as it is the python module used to create the windows and designs for both GUIs, and it does so in a very simple manner.

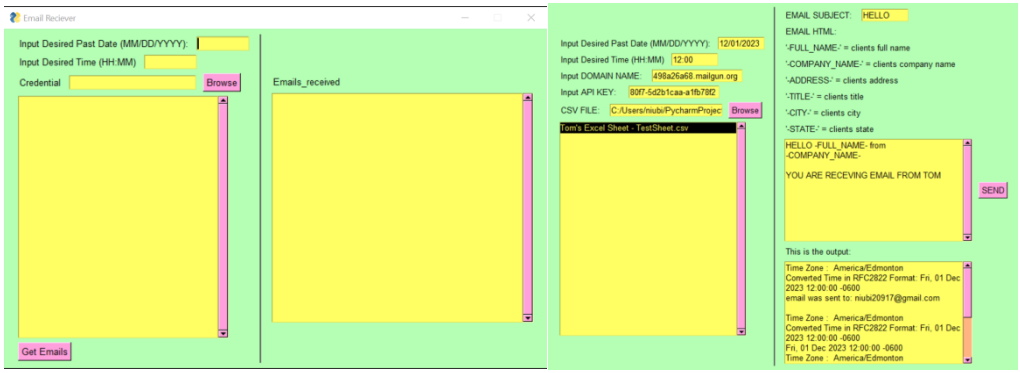

Figure 6. Screenshot of the system

```
right_column = [
                [sg.Text("EMAIL SUBJECT: "), sg.In(size=(10, 2), enable_events=True, key="-SUBJECT-")],
                [sg.Text("EMAIL HTML: ")], [sg.Text("'-FULL_NAME-' = clients full name")],
                [sg.Text("'-COMPANY_NAME-' = clients company name")],
                [sg.Text("'-ADDRESS-' = clients address")],
                [sg.Text("'-TITLE-' = clients title")],
                [sg.Text("'-CITY-' = clients city")],
                [sg.Text("'-STATE-' = clients state")],
                [
                    sg.Multiline(size=(40, 10),
                                 enable_events=True,
                                 key="-HTML-",
                                 autoscroll=True),
                    sg.Button("SEND", key="-SEND-")
                ],
                [
                    sg.Text("This is the output:"),
                    sg.Multiline(size=(40, 10),
                                 enable_events=True,
                                 key="-OUTPUT-",
                                 autoscroll=True),
                ]]

layout = [[
    sg.Column(left_column),
    sg.VSeperator(),
    sg.Column(right_column),
]]
```

```
while True:
  event, values = window.read()
  if event == "Exit" or event == sg.WIN_CLOSED:
    break

  if event == "-DATE-":
    scheduled_timestamp = receiveEmails.calctimestamp(values['-DATE-'])
    print(scheduled_timestamp)

  if event == "-TIME-":
    time_of_day = receiveEmails.calcseconds(values["-TIME-"])
    print(time_of_day)

  if event == "-DOMAIN-":
    domain_name = values["-DOMAIN-"]

  if event == "-API-":
    api = values["-API-"]

  ### FOLDER SELECTION EVENT ###
  if event == "-FOLDER-":
    folder = values["-FOLDER-"]
    try:
      file_list = os.listdir(folder)
    except:
      file_list = []

    fnames = [
        f for f in file_list
        if os.path.isfile(os.path.join(folder, f)) and f.lower().endswith((
            ".csv"))
    ]

    window["-FILE LIST-"].update(fnames)
    check = True
```

Figure 7. Screenshot of code 3

The format of the GUI and its underlying function are crucial parts of the functionality of the GUI. The two major components of each of the GUIs are the layout structure and the while loop. Within the layout structure, all of the components that make up the design of the GUI are kept track of in an array, which can be turned into columns as displayed in the final 'layout' structure. The while loop, on the other hand, is the key to the running of the GUI. It is constantly performing checks on each of the key events, and upon detecting a key event, it will run the code that corresponds to each key. For example, upon hearing the "-TIME-" event key, which is triggered by a user inputting the desired time into the GUI, the code will take the user input and assign it to the 'time_of_day' variable after running the input through the calcseconds() function. Overall, the GUI is the backbone for each of the applications, as it provides a structure for ease of use.

4. EXPERIMENT

4.1. Experiment 1

One potential weak point specifically in our Receive Emails application, is the user has complete freedom to input a time that is really far into the past, which would result in a lot of emails being returned which could crash the application.

The design for this experiment is going to involve grabbing emails from certain distances in the past and measuring how long it takes to be returned and displayed in the application. We will start by grabbing emails from around a month ago and measure the time it takes to be returned to the screen, and then by increments of 3 months we will continue to measure the time difference until we reach around 2 years into the past. This experiment is set up in this manner as it will

provide tangible data to answer the question of how the amount of emails will impact the running of the application.

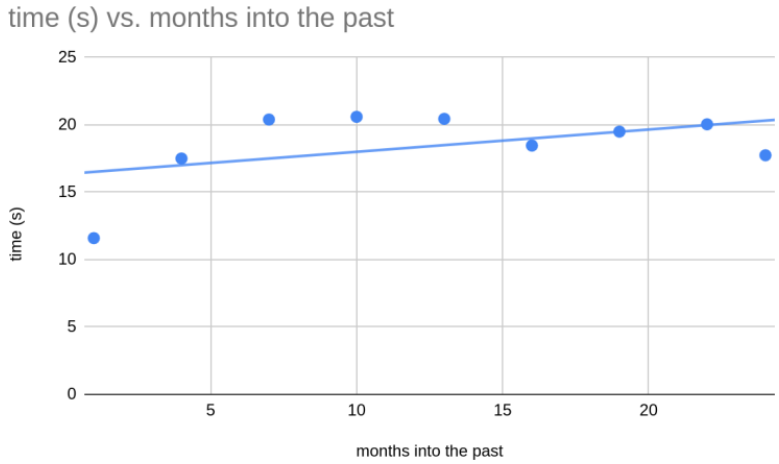

Figure 8. Figure of experiment 1

After conducting the experiment, we noticed a very interesting thing about the Gmail API, which is the fact that the Gmail API actually will only send a set amount of emails from back into the past. The shortest amount of time getting emails from the Gmail API was 11.6 seconds when we were trying to grab emails from 1 month into the past. Then we noticed that around 7 months is when the time started to cap at around 20 seconds. We believe this is because the Gmail API is limited to returning a certain max amount of emails, and around 7 months into the past is when the Gmail API found that the max amount of emails to return. So no matter how far back into the past the Gmail API tried to go, it couldn't grab any more emails than the max. This feature can prevent the application from crashing when receiving too many emails.

4.2. Experiment 2

Another potential weak point in our application that could cause the Send Email Application to crash is how long the html the user inputs is. Depending on how long the html is, it could take longer to make the request to mailgun and for mailgun to send the emails.

To test how much the HTML size affects the time for the application to make the request to the MailGun API, we asked ChatGPT to design us a generic email template and intend to send that email template as a test. Upon sending that first email, we will continue to copy and paste the HTML over, and measure the time until we have copied the HTML 5 times, and measure how long it takes for the emails to be sent. The experiment is conducted in this manner so that we may see how the size of the HTML will impact the time for the application to send emails.

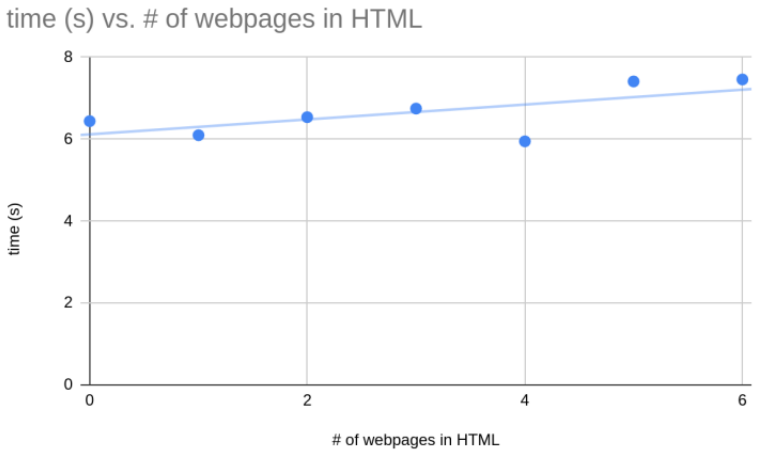

Figure 9. Figure of experiment 2

After performing the experiment, the data we collected suggested that while the size of the HTML does impact the time it takes for the request to be made, the increase in time in which it is affected is almost negligible. Making the first generic request which included no HTML, only the word "hello," took a short amount of time, the smallest being the outlier of 5.95 seconds, or around 6.44 seconds. The final request made in the experiment was the 6th request, which included the HTML of 5 of the ChatGPT-generated email formats, which was quite long [15]. It is to our pleasant surprise that the size of the HTML did not really impact the performance time of the Send Email Application except for around 1 second between a request with no HTML and a request with the HTML of 5 email templates. The result might be because of the way that HTML is designed to be efficient and take a small amount of time to send through the internet. Another potential reason is that the test case only contains two recipients, as such, it does not require a long time for the email to reach each recipient.

5. RELATED WORK

The Mailgun API is a bulk email scheduling tool that is used by applications and companies alike for scheduling emails to recipients [11]. The Mailgun API is more so a backend service, as it enables the user to send and receive mass amounts of emails. As Mailgun is typically a backend service, the user interface for Mailgun is not exactly user-friendly or straightforward, the user must create an interface or some tool to communicate with it. When compared to the Email Sender and Receiver applications, Mailgun severely lacks an easy-to-use interface. The Email Sender and Receiver applications both have clear and easy-to-use interfaces for all users. Additionally, the Email Sender and Receiver applications are entirely free, as opposed to Mailgun which requires a premium membership to operate.

Mailchimp is a marketing tool used for sending bulk emails, as well as tracking how emails are received and displaying the data in terms of graphs [12]. When signing up for Mailchimp, the user will notice that Mailchimp requires a lot of setup as well as a lot of personally identifiable information just to create an account. Additionally, Mailchimp fits in its own trackers and scripts, not enabling the user to completely control the email that is being sent to the recipients. The Email Sender and Receiver applications require no setup of personally identifiable information, and the Email Sender allows the user to completely customize the email that is sent to the recipients.

SendPulse is an online bulk email sending tool, in which users are able to log in to the web interface with an account and send bulk emails to an inputted mailing list they create [13]. As opposed to the Send Email and Receiver applications proposed in this paper, SendPulse is a completely online web interface that requires a user to log in with their account info to send and receive bulk emails, as well as check interaction statistics. Having the applications natively enables the users to have all of their emails and mailing lists stored on their computer which is generally more secure and easier to access.

6. CONCLUSIONS

After comparing the Email Sender and Receiver applications to the previously mentioned methodologies, it is apparent that the Email Receiver application is lacking in how clearly it displays received emails as well as how streamlined the process of creating and finding the credentials.json file is [14]. A possible fix for how the application displays the email currently is to show the entire email rather than just in text form, this could be done by modifying the structure of the GUI to display the email as an object of the window. To solve the issue of the credentials input not being straightforward, a possible fix could be to have the credentials.json file be in the same directory as the application. Additionally, it would make sense for the Email Receiver application to be able to walk the user through the process of generating the credentials.json and token and utilizing that as the credentials for sending bulk emails.

The experience of producing this application has inspired and encouraged me to develop my personal interest in programming applications as a product. During the development process, I had the opportunity to think from a different perspective to solve problems and create a final product I am proud of. Thank you for reading.

REFERENCES

[1] Chen, Jianjun, Vern Paxson, and Jian Jiang. "Composition kills: A case study of email sender authentication." 29th USENIX Security Symposium (USENIX Security 20). 2020.
[2] Stančin, Igor, and Alan Jović. "An overview and comparison of free Python libraries for data mining and big data analysis." 2019 42nd International convention on information and communication technology, electronics and microelectronics (MIPRO). IEEE, 2019.
[3] Chen, Mia Xu, et al. "Gmail smart compose: Real-time assisted writing." Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019.
[4] Phillips, Patti P., and Jack J. Phillips. "Return on investment." Handbook of Improving Performance in the Workplace: Volumes 1 - 3 (2009): 823-846.
[5] Aini, Qurotul, Indri Handayani, and Nuril Huda. "Implementasi Fgr (First Generation Resources) dengan Menggunakan Mailchimp sebagai Sarana Informasi di Perguruan Tinggi." Technomedia Journal 1.1 Agustus (2016): 11-19.
[6] Memon, Atif M., and Myra B. Cohen. "Automated testing of GUI applications: models, tools, and controlling flakiness." 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.
[7] Kong, Ruoyan, Haiyi Zhu, and Joseph A. Konstan. "Learning to ignore: A case study of organization-wide bulk email effectiveness." Proceedings of the ACM on Human-Computer Interaction 5.CSCW1 (2021): 1-23.
[8] Mitlöhner, Johann, et al. "Characteristics of open data CSV files." 2016 2nd International Conference on Open and Big Data (OBD). IEEE, 2016.
[9] Hart, Steven N., et al. "VCF-Miner: GUI-based application for mining variants and annotations stored in VCF files." Briefings in bioinformatics 17.2 (2016): 346-351.
[10] Miranda, M. Gutierrez. "The importance of graphic users interface, analysis of graphical user interface design in the context of human-computer interaction." Edulearn11 Proceedings. IATED, 2011.

[11] Ferchichi, Lotfi. How to Write a Bulk Emails Application in Vb. Net and Mysql: Step by Step Fully Working Program. Xlibris Corporation, 2014.
[12] Kong, Ruoyan, Haiyi Zhu, and Joseph A. Konstan. "Learning to ignore: A case study of organization-wide bulk email effectiveness." Proceedings of the ACM on Human-Computer Interaction 5.CSCW1 (2021): 1-23.
[13] Sterne, Jim, and Anthony Priore. Email marketing: using email to reach your target audience and build customer relationships. John Wiley & Sons, Inc., 2000.
[14] Chiu, Hung-Chang, et al. "The determinants of email receivers' disseminating behaviors on the Internet." Journal of advertising research 47.4 (2007): 524-534.
[15] Bhuvaneswari, R., and P. R. Tharaniesh. "Exploring ChatGPT for Email Content Compression and Summarization." 2023 4th International Conference on Communication, Computing and Industry 6.0 (C216). IEEE, 2023.