MINT Capstone Project

Email-to-REST User-Request System

Supervisor: Paul Lu

Student Name: Sam El-Awour

Email:Elawour@Ualberta.ca

Acknowledgement 4

Introduction 4
- Propose 4
- Problem 4
- System Overview 4
- Goal & Motivation 5

Application’s Functionality 6
- Figure (1):Application’s Functionality Diagram 6

Use-Case Scenarios for the E2R Application 7

Processed Email Types : 8
- 1- Shopify Emails (Figure (3), (11)): 8
- 2- Customer Emails (Figure (8)): 8
- 3- Follow Up Emails (Figure (2)): 8
  - Figure (2):Follow Up Email Sample 8

Application Processing Steps: 9
- 1- Shopify Emails: 9
  - Step 1: 9
    - Figure (3): Shopify Email Sample 9
  - Step 2: 10
    - Figure (4):REST Call in JSON format 10
  - Step 3: 10
  - Step 4: 10
    - Figure (5):Ordering Server Response 10
    - Figure (6): Order Confirmation Sample 11
- 2- Customer Emails: 11
  - Figure (7): Item List Sample 11
  - Step 1: 11
    - Figure (8): Customer Email Sample 12
  - Step 2: 12
- 3- Follow Up Emails (Figure (32)): 12

Email Order Templates 13
- Figure (9): Simple Single Item Sample 13
- Figure (10): List of Simple Items Sample 13
- Figure (11): Complex Single Item Sample 14

Figure (12): List of Complex Items Sample 14

Requirements and Specifications 14

Functional Requirements 14

Figure (13): Application Functional Requirements 15

1- Source 15

Figure (14): Email Sample in HTML Format 15

Figure (15):Email Sample in Plain Text Format 16

2- Application 16

Table(1): Application Python Libraries 17

3- Destination 17

Hardware Requirements 17

Application’s Limitations and Assumptions 17

Email Data Error 18

Figure (16):Well-Formed Email Sample 19

Figure (17): Non-Well-Formed Email Sample 19

Application Errors: 20

Design 20

Application Code 23

Main application 23

Figure (19): Application Main Function (Main-ver7_.Py) 23

Check_Inbox() function: 24

Figure (20): Application Check_Mail Function (Main-ver7_.Py) 24

ProcessOrderFromShopify() function: 25

Figure (21): Application ProcessOrderFromShopify Function (Main-ver7_.Py) 25

Figure (22): Application ProcessOrderFromCustomeremail Function - Part 1 (Main-ver7_.Py) 26

Figure (23): Application ProcessOrderFromCustomeremail Function - Part 2 (Main-ver7_.Py) 27

SendDataToIncompleteStateServer() function: 28

Figure (24): Application SendDataToIncompleteStateServer Function (Main-ver7_.Py) 28

Figure (25): Application SendDataToOrderingServer Function (Main-ver7_.Py) 28

SendAckemail() function: 29

Figure (26): Application SendAckemail Function (Main-ver7_.Py) 29

Figure (27): Application ResendemailToCustomerFunction (Main-ver7_.Py) 30

Ordering Server Code: 31

Figure (28):Ordering Server (Ordering_Server.Py) 31

Incomplete State Server Code: 32

Figure (29): Incomplete State Server (IncompleteStateServer.py) 32

Testing **32**

Test Plan 32

Test Items 32

Functional Testing 33

Test Environment 33

Table (2): Testing Environment 33

Test Results 34

Figure (30): Application Test Results 34

Figure (31): Ordering Server Test Results 35

Figure (32): Incomplete State Server Test Results 35

Installation Instruction **36**

Conclusion **37**

Confirmation of Originality **37**

References **37**

Acknowledgement

At first, I would like to acknowledge the access to Wenting Zhang's capstone project proposal, which helped forming the structure and the main headings of this proposal. I would also like to express my great appreciation to Professor Paul Lu for the willingness to supervise this project and his valuable suggestions and directions.

Introduction

This introduction gives a brief description of the scope of this project including an explanation of the problem project aimed to resolve, an overview of the system developed and finally a goal and motivation intended from working on this project.

Purpose

The purpose of this project is to automate email orders processing with a well-structured application that monitors email inboxes for incoming email orders that has a particular structure. The application also parses order information and communicates them as a REST call to an electronic ordering service in a language that the service understands. This process is often referred to as email to REST (E2R) system which intends to listen to orders that come from e-commerce services such as Shopify and Pinnacle Cart in the form of an email.

Problem

E-commerce services enable merchants to post products they intend to sell and allow customers to purchase online. Merchants also receive order emails from customer directly. Once orders are made, they get sent to merchants in the form of emails that contain orders' information such as item, price, and quantity including items properties such as color, size, and materials, where merchants retrieve order information from the email and place the order manually.

As the number of orders increases, the number of email orders increases and at some point it may become very difficult to handle manually and keep up with responding to orders in a timely manner.

System Overview

The E2R Application is designed to automate this process by actively monitoring the merchant mailbox, capture information required, and direct it to an electronic ordering system through REST calls.

The E2R application is developed as an implementation for the client-server architecture using the Internet Message Access Protocol (IMAP) to retrieve emails. Simple Mail Transfer Protocol (SMTP) is used to send emails and implements a REST-protocol system to transfer order

information.

Most E2R email parsers that are currently available are exclusive services that are offered by profitable third parties with very particular and limited purposes which could raise some security concerns. An example of an email parser is https://mailparser.io/

Goal & Motivation

The goal of he E2R system is to provide an alternative to both closed source profitable software available and manual processing with a fully integrated, and customizable Python implementation that can be used not only for this purpose but can be customized for different other purposes.

My main motivation for developing this application is to get a full understanding of Python implementation, working on and interacting with different server platforms such as email, web, and database servers and thus allowing me to put programming languages for use in network automation.

Application's Functionality

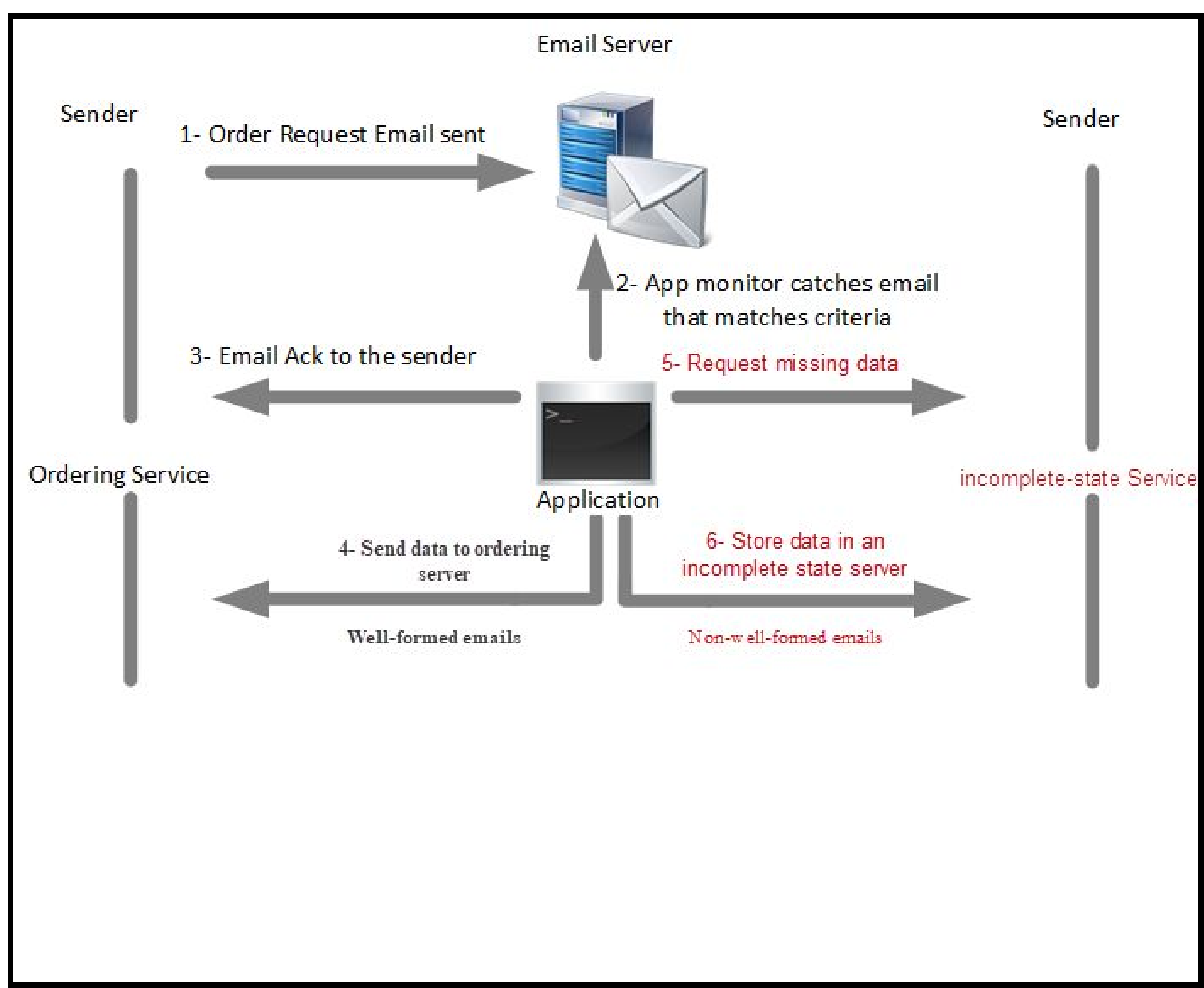

Figure (1):Application's Functionality Diagram

Figure (1) shows the functionality of the E2R application. As shown in step 1 from the figure, an order email is sent by either Shopify or a customer to the marchant's email inbox. This email is then received and stored on the email server mailbox. Step 2 shows that the application regularly monitors the email server for new emails - Figure (19) shows the application code that is responsible for this function - and matches a certain criteria to identify the source of the email. Figure (21) and Figure (22) show the application code responsible for processing emails based on the source.

If an email was identified as a well-formed format, it gets extracted and formatted then steps 3 and 4 in Figure (1) are implemented. In step 3, an email acknowledgement is sent back to the email sender confirming that the order is received. Step 4 in the figure is the final step in this scenario where formatted email data is sent to ordering server to place the order as done by the code in Figure (25)

However, if email is not in a well-formed format, issues are recorded, steps 5 and 6 in Figure (1) are implemented. In step 5, a follow up email is sent back to the email sender stating issues found and requesting missing information. Step 6 in the figure is the final step in this case where obtained email data is sent to an incomplete state server to store information until it is complete as done by the code in Figure (24)

As shown in Figure 1, the application focuses on three main functionalities:

1- Monitoring email server for email orders

2- Verifying email format

3- Sending order data to the proper destination

Use-Case Scenarios for the E2R Application

Although this application is built and tested on a specific platform (Shopify email orders) and customer email orders, it can also be used to pull data fields from any auto-generated emails such as online service requests, emails from contact forms, or any machine sent type of emails. Application is designed to make an efficient, automated, and reliable tool for a variety of scenarios.

As a proof of concept, this application has been tested against a practical use case scenario using the Shopify business model email templates and customer email orders. As they define themselves, Shopify is an e-commerce platform that allows merchants to set up online stores to sell their products online. Customers can shop items from several online stores and once an order is placed and checked out Shopify sends an email to merchants pre-configured email address to deliver order details.

The E2R application monitors merchants mailboxes for a specific email format and templates to capture order values and process them.

Primarily, Shopify order emails come from Shopify's electronic system with a controlled pre-configured product list, which helps the E2R match certain criteria for incoming email orders and extract order information easily. However, when processing emails that come straight from customers, the application will have to verify the consistency of the emails ensuring all required information is present.

Processed Email Types :

1- Shopify Emails (Figure (3), (11)):

- Email is always received from the same source
- Email always has the store name in the subject line
- Email body is within a consistent format.

2- Customer Emails (Figure (8)):

- All other email orders
- Emails are always verified

3- Follow Up Emails (Figure (2)):

- Emails response for incomplete requests
- Email always has a reference number in the subject line

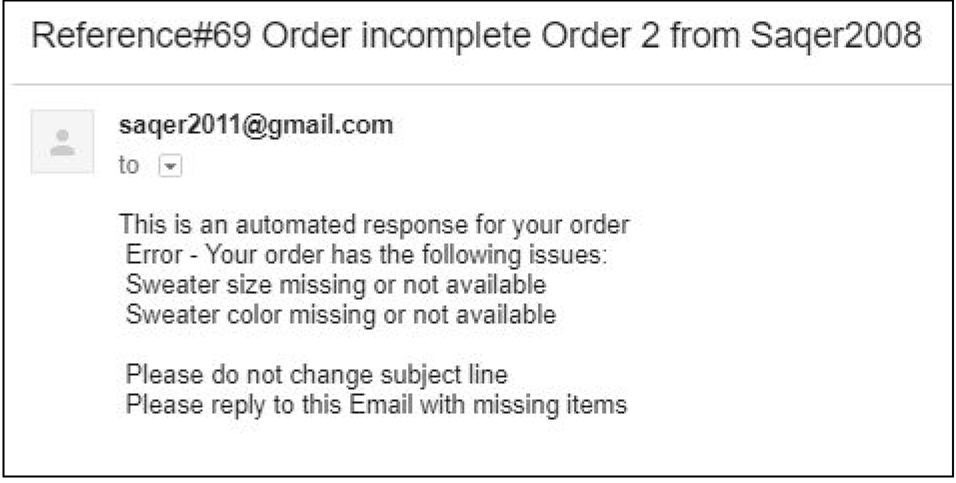

Figure (2):Follow Up Email Sample

Application Processing Steps:

1- Shopify Emails:

Step 1:

The application uses the IMAP protocol to perform an email client function to retrieve new email information from the email server. As in Figure (3), customer "Steve Shipper" places an

order to buy products from merchant “Danny”. Order details is sent to Danny with Steve’s shipping information and order details.

As shown in the figure, the email from address, email to, subject line are pretty consistent.

As for order details, this E2R application handels four email templates with four different order details - templates are discussed in a subsequent section.

From: **Shopify Orders**<orders@shopify.com>
Date: Wed, Jan 10, 2018 at 2:50 PM
Subject: [SamUofA] Order #9999 placed by Steve Shipper
To: danny@merchants.com

Hello Danny,

John Smith placed a new order with your store, Jan 10 02:50PM:

- 1x Aviator sunglasses (SKU: SKU2006-001) for $89.99 each
- 1x Mid-century lounger (SKU: SKU2006-020) for $154.99 each

View order #9999

Payment processing method:

visa, bogus

Delivery method:

Generic Shipping

Shipping address:

Steve Shipper

123 Shipping Street

Shippington, Kentucky 40003

United States

555-555-SHIP

Figure (3): Shopify Email Sample

Step 2:

As in Figure (4), order details are extracted from the email received and translated into a different format which the ordering system understands, then sent over to the ordering server.

REST call:
{ quantity:1,item:"Aviator sunglasses", SKU:SKU2006-001, unitprice:$89.99},{quantity:1,item:" Mid-century lounge", SKU:SKU2006-020, unitprice:$154.99}

Send REST call for ordering server

Figure (4):REST Call in JSON format

Step 3:

System waits until the order is acknowledged by the ordering server. Other responses may come back as (Sold out, Order rejected, or Order cancelled)

Step 4:

Finally and as Figure (5) shows, ordering system replies back with a confirmation email. System uses SMTP protocol to perform an email client function and send an email from marchant's account.

Response:
Order acknowledged

Sent from ordering server

Figure (5):Ordering Server Response

To: orders@shopify.com
From: danny@marchants.com
Subject: order#9999
confirmed

This is an automated response for:
 1x Aviator sunglasses (SKU: SKU2006-001) for $89.99 each
 1x Mid-century lounger (SKU: SKU2006-020) for $154.99 each

Figure (6): Order Confirmation Sample

2- Customer Emails:

When working with customer emails, unlike Shopify orders are not controlled with pre-configured listing, instead a database reference will have to be used to cross reference against items ordered in the email. For the purpose of this capstone, a static list is suggested in the form of a Python dictionary (Figure (7))

```
Items = {'SunGlasses':{'item': 'Aviator sunglasses', 'SKU': '0001', 'colors':
['Black','Brown'], 'price': '$46'}, 'Lounger':{'item': 'Century Lounger', 'SKU': '0002', 'price':
'$46'}, 'T-Shirt':{'item': 'Puma T-Shirt', 'SKU': '0003', 'sizes': [' L ',' M ',' S '], 'price':
'$40'},'Sweater':{'item': 'Adidas Sweater', 'SKU': '0004', 'sizes': [' L ',' M ',' S '], 'colors':
['Black','Brown'],'price': '$60'}}
```

Figure (7): Item List Sample

Step 1:

Figure (8) on the next page shows an email that was received from a customer. At first, email subject and body data is extracted.

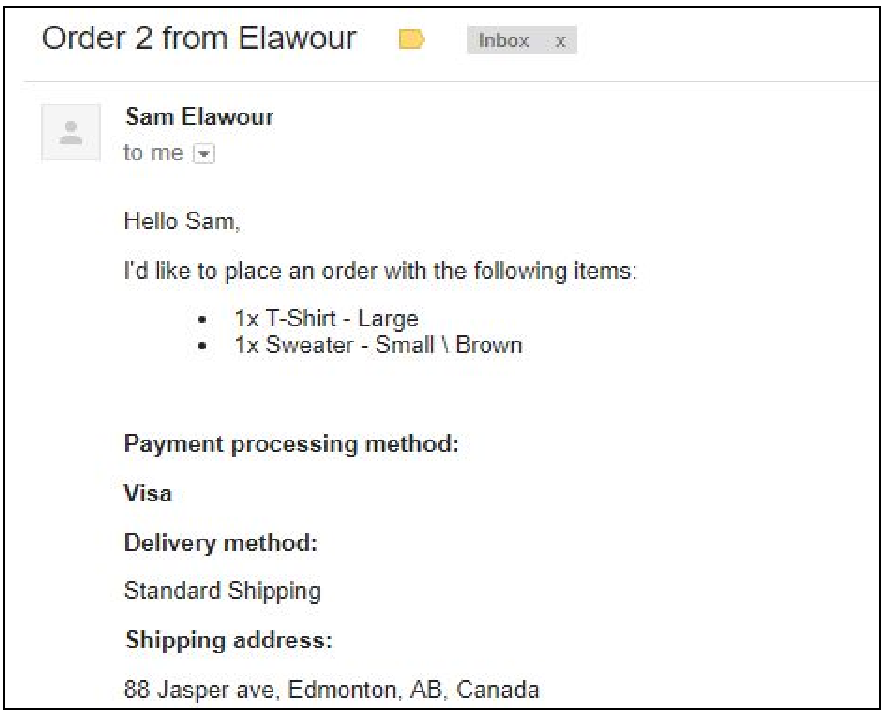

Figure (8): Customer Email Sample

Step 2:

Order details are extracted from the email received and extracted data is verified:

A. If the received email has all order information required (well-formed email) then email data is translated into the appropriate format which the ordering system understands, the email is then sent over to the ordering server.
B. If the received email is missing some order information (non-well-formed emails), email is sent to a store location (incomplete-state server) and a request is sent to the sender including a reference number to reply with missing information.

Steps 3 and 4 are similar to the above.

3- Follow Up Emails (Figure (32)):

Although this email type is listed last, it actually is the first check on the application process. Application always checks to see if there is a reference number in the email subject to identify if this is a follow up email from a previous non-well-formed email. If it is the case, it sends the email directly to the incomplete state server along with the reference number and a time stamp. If not, it will move on to check whether it is one of the two other types above.

Email Order Templates

The application is designed to process orders with four different email order templates and various properties such as item, price and quantity including items properties such as color, size, and materials. The four templates are as follow:

1- **Simple single item order:** As in Figure (9), the application will be able to process email orders that have a single item with no properties (simple item includes: item name, price, and quantity)

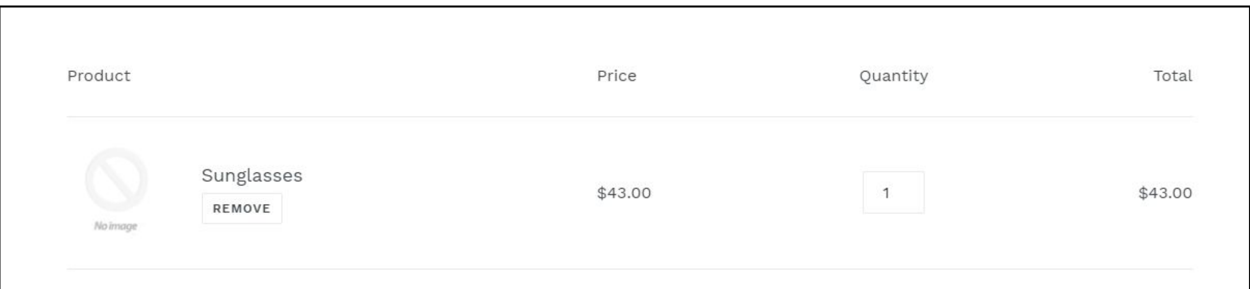

Figure (9): Simple Single Item Sample

2- **Simple item list orders:** As in Figure (10), the application will also be able to process email orders that have a list of simple items

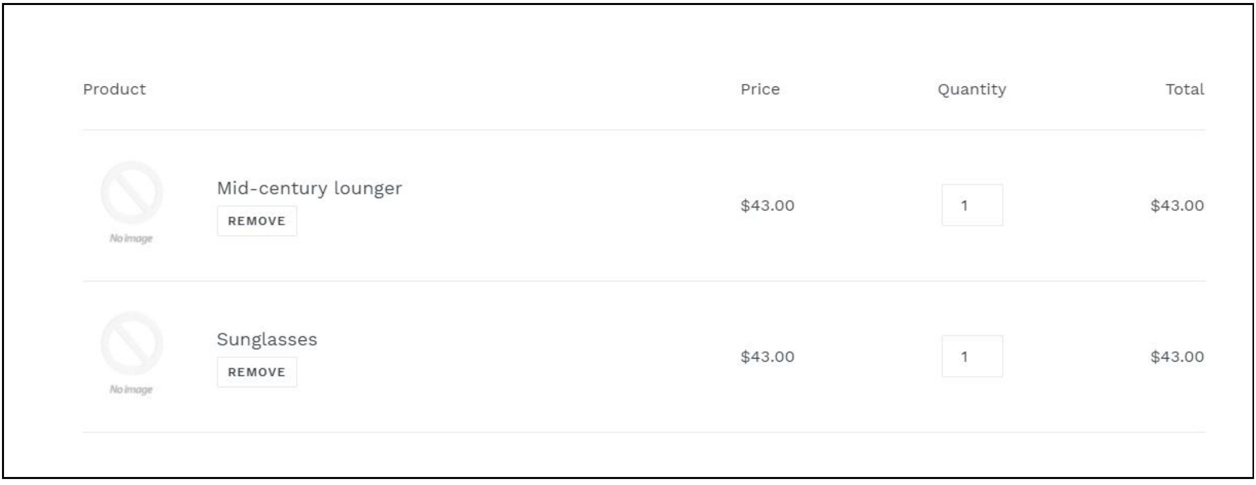

Figure (10): List of Simple Items Sample

3- **Complex Single Item Order:** Figure (11), shows email orders that have a single item with properties (Complex Item Includes: Item Name, Size, Color, Material, Price, and Quantity)

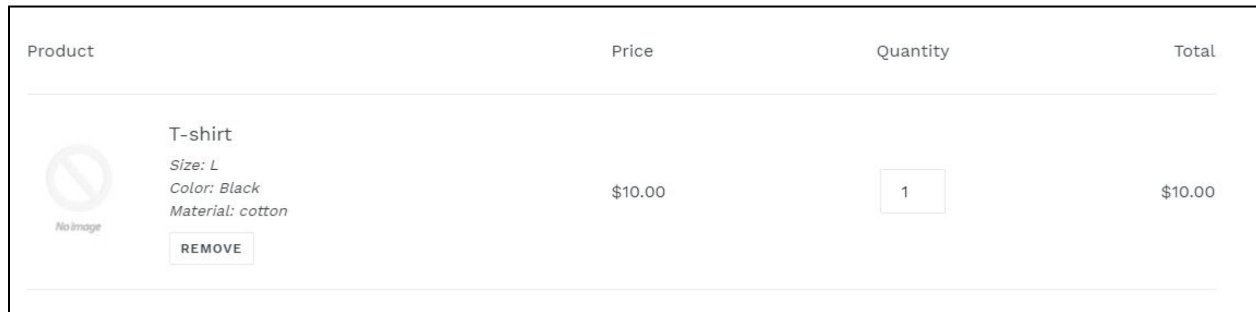

Figure (11): Complex Single Item Sample

4- **Complex Item List Orders:** and finally Figure (12), shows email orders that have a list of complex items

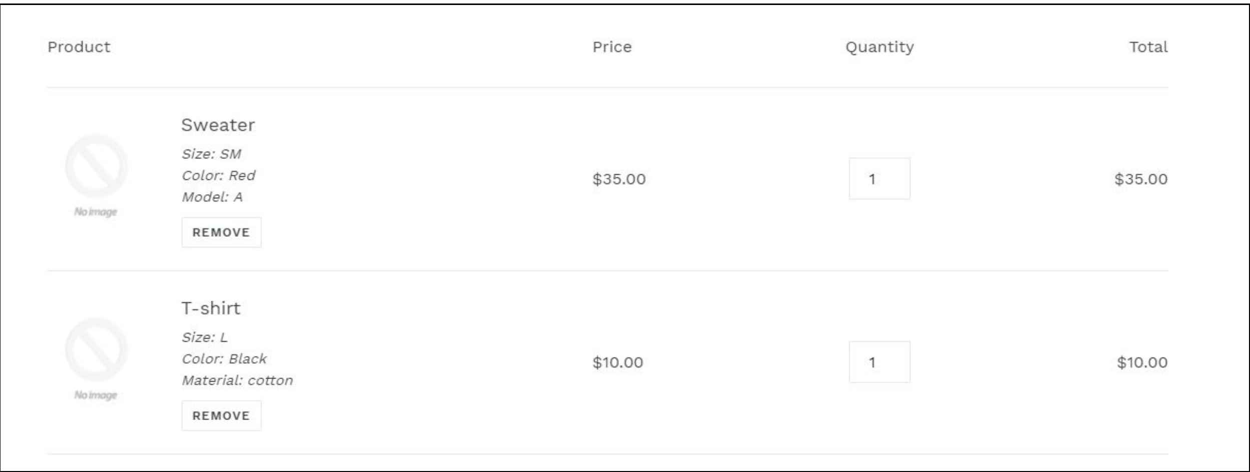

Figure (12): List of Complex Items Sample

Requirements and Specifications

Functional Requirements

As shown in Figure (13) below, the E2R application functions in between the source (order Email data) and the Designation (receiving system). After email order is received and verified, data output is formed in a universal modern format that is understood by the receiving system.

Figure (13): Application Functional Requirements

1- Source

The data source for the E2R application is email data retrieved from Merchandiser's email account. This data is expected to be consistent and in a certain format that contains all required information for the order to be completed as highlighted in Shopify email example in Figure (14).

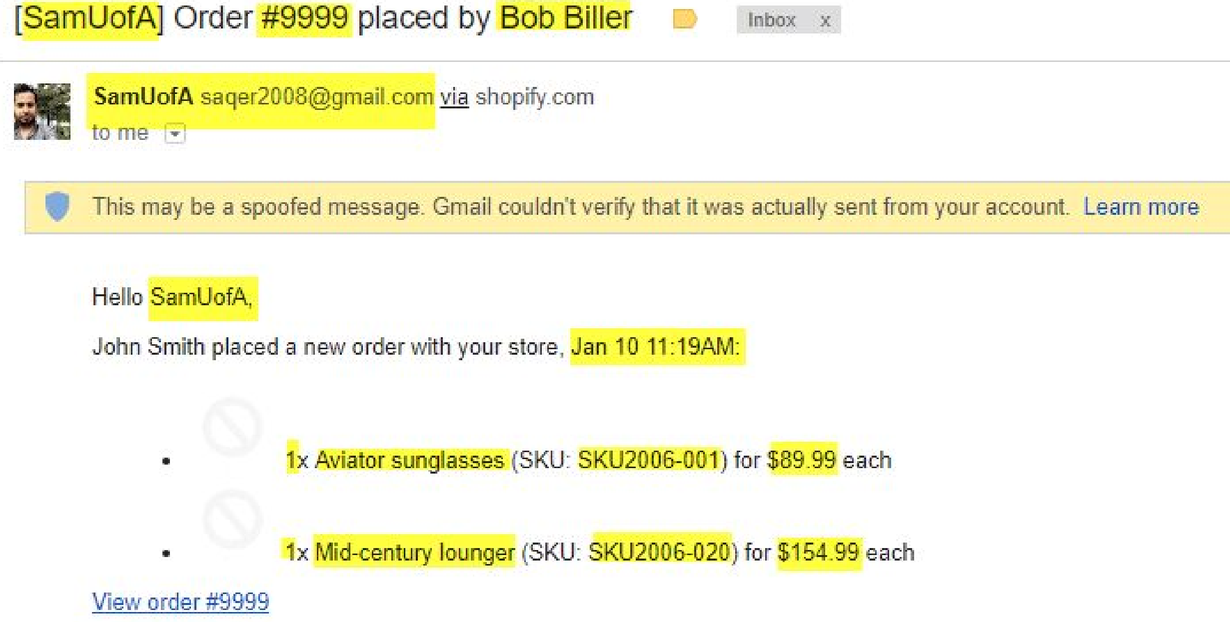

Figure (14): Email Sample in HTML Format

As highlighted above, order Information and order items are the values extracted by E2R application to be processed. Figure shows data in the HTML format which includes pictures, attachments, and information that are unnecessary to be captured. Therefore, data at first is captured in plain text format so that all unnecessary information are ignored and email data are easier to process as shown in Figure (15) in the following page.

from: SamUofA <saqer2008@gmail.com> via shopify.com
to: saqer2008@gmail.com
date: Wed, Jan 10, 2018 at 2:50 PM
subject: [SamUofA] Order #9999 placed by Bob Biller
body: Bob Biller placed a new order with your store, Jan 10 02:50PM:

- 1x Aviator sunglasses (SKU: SKU2006-001) for $89.99 each
- 1x Mid-century lounger (SKU: SKU2006-020) for $154.99 each

Figure (15):Email Sample in Plain Text Format

2- Application

The E2R application is run through the command line interface for user interaction.

Application uses many Python libraries to perform various tasks. Table (1) below lists Python libraries and the functionality of every library used in the application:

Python Library	Usage
import sys	This library is used by the Python interpreter in the application to access variables and functions
import imaplib	This library is used in the application to establish a connection with the email server and retrieve emails
import getpass	This library is used in the application only when email account password is not in the program's code and entered manually upon application run - for security reasons.
import datetime	This library is used in the application to convert date and time information in proper format
import smtplib	Most common Python modules to send emails using SMTP. This library is used in the application when sending order resend requests and order confirmations

Table(1): Application Python Libraries

3- Destination

Application uses modern frameworks for developing both the front end and the backend interfaces:
JSON: Javascript framework used to format order information
FLASK: Python framework

Hardware Requirements

Although the E2R system is a very light application that may not require a lot of processing power, RAM or storage. As the number of emails required to be processed at a time increases, the requirement for processor power and RAM increases with the increased I/O between storage and RAM. While coding, testing, and prototyping this application, it was best done with a regular desktop (4Gb of RAM, Dual Core 2.0GHz).

The recommended hardware requirements and system specification as determined from using the Python interpreter and Python libraries are as follows:

- 2 GHz Processor
- 2 GB RAM
- Minimum HDD 10GB free space

Application's Limitations and Assumptions

The E2R Applications assumes that the input data coming from order email is in consistent format as it processes information from both email subject line and body. Application depends on the text strings before and after the values required to be captured. That is why if this email text is not properly formatted, values captured may result in inaccurate values and can not be trusted.

If the email is not consistent or it has missing values or improperly-formated extracted values (e.g. quantity is not an integer), the application will be able to detect that and request a resend. In this case, application does not re-format email contents but rather it requests the missing information.

As a result of the application's process, data is returned as a JSON format and it is expected that the receiving system is able to read them and process them.

Below is the two main error handling the system performs:

1. Email Data Error

The main purpose for email data error handling in the application is to capture non-well formed emails from customers with missing order information. email data errors are handled by checking for any missing or wrong format data as follows:

- First application checks if the following order information exists (See highlights in Figure (16) and application code lines 56 through 59 in Figure (21)).:
    a. Store Name
    b. Order Number
    c. Customer Name
- Second, it validates mandatory order items (See highlights in Figure (16) and application code lines 124 through 154 in Figure (22) and lines 155 through 181 in Figure (23))
    a. Item Qty
    b. Item Description
    c. Item SKU
    d. Item Price
- Possibilities and actions (See application code lines 184 through 201 in Figure (23))
    1. Unknown source and/or missing ot wrong store name - email is dealt with as customer email
    2. Missing/improper format order # - try to get it from the body - generate an order number
    3. Missing customer name - try to get it from body - Ignored
    4. Missing/improper format item qty - assumed as 1
    5. Missing/improper format item description - Request Resend
    6. Missing/improper format item SKU - retrieved from Order Information
    7. Missing/improper format item price - retrieved from Order Information

Figure (16) shows a well-formed email order where mandatory order information highlighted in Figure (16), elements 1-7. As shown in the figure, some data is duplicated in both the email subject line and email body (Figure (16), elements 1-7) and as listed above, the action performed if application fails to extract data from email subject line it will attempt to get them from email body instead before it fails to request resend.

Other order data as in Figure (16), elements 4-7. Also as listed above, the action performed if application fails to extract data from email body line it will fail to request resend:

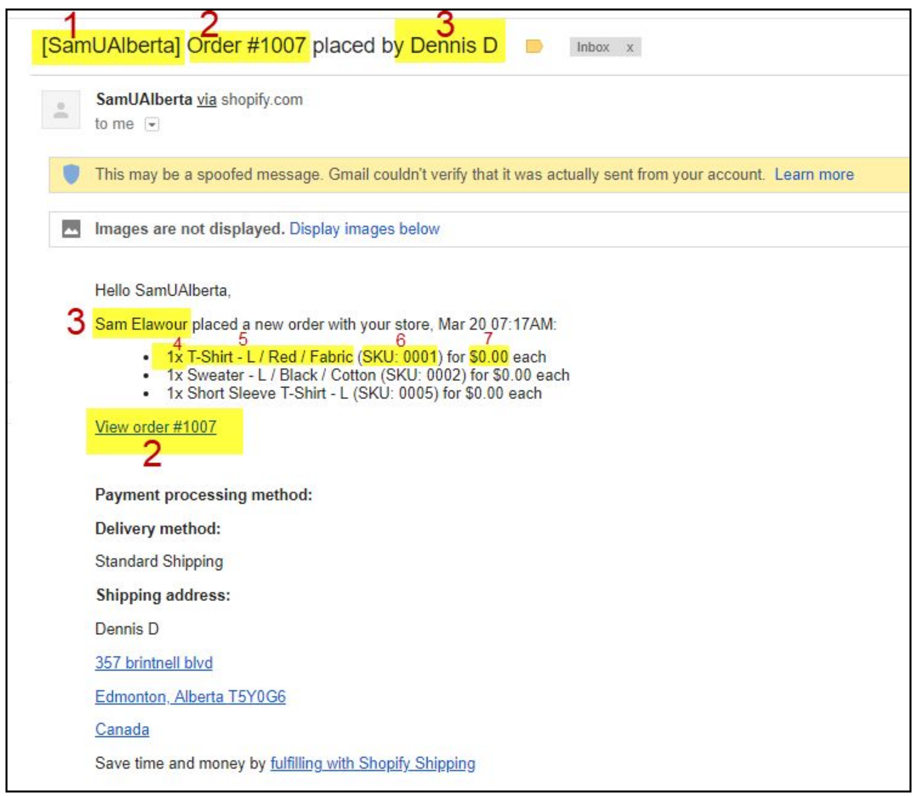

Figure (16):Well-Formed Email Sample

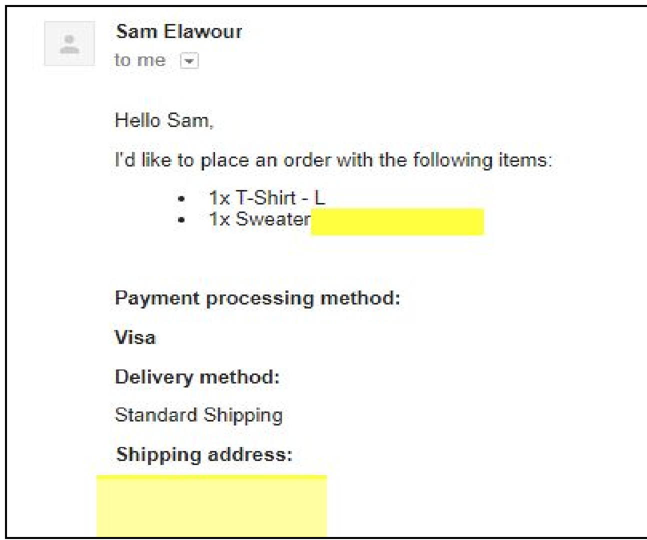

Figure (17): Non-Well-Formed Email Sample

2. **Application Errors:**

Main purpose for application error handling is to control and prevent system crashes and data losses. Application errors are handled through Python error handling (Try and Except).

Action when application errors occurs is to generate an error message that explains the error occured or send the code prompt to a different function where the process is carried on differently.

Design

To explain the logic flow of the application, flowchart in Figure (18) demonstrates the application process step by step.

- At the beginning of the flow, Application tries to login to email account

- If login is successful, emails are checked against the Shopify matching criteria discussed earlier in this report, if a match is found, email body is handed over to a separate function called “ProcessOrderFromShopify()” which extracts order information from the email, sends it to ordering server and sends an ack email to sender.

- Otherwise, email is handled through the “ProcessOrderFromCustomeremail” function which generates an order number and verifies the consistency of the email:
    - Order information is checked (such as customer name, delivery method,...etc)
    - Order items and item properties are checked against available item list.
    - If order information or items are missing any information order status is marked as “incomplete” and order issues are recorded. Otherwise order information from the email, sends it to ordering server and sends an ack email to sender.
    - If order status is marked as incomplete, it will be sent to incomplete state server and a follow up email is sent to sender

- Application will continue to the next email which guarantees that the application flow and processing other email orders is not affected by inconsistent email orders.

- Once all unread orders have been processed application exits after reformating captured information and sending them to the ordering system.

This design guarantees isolation and reliability where every email order is processed separately. If a flaw exists either due to an error caused by the application itself or the email format, is is identified and processed separately with complete isolation from the next email order.

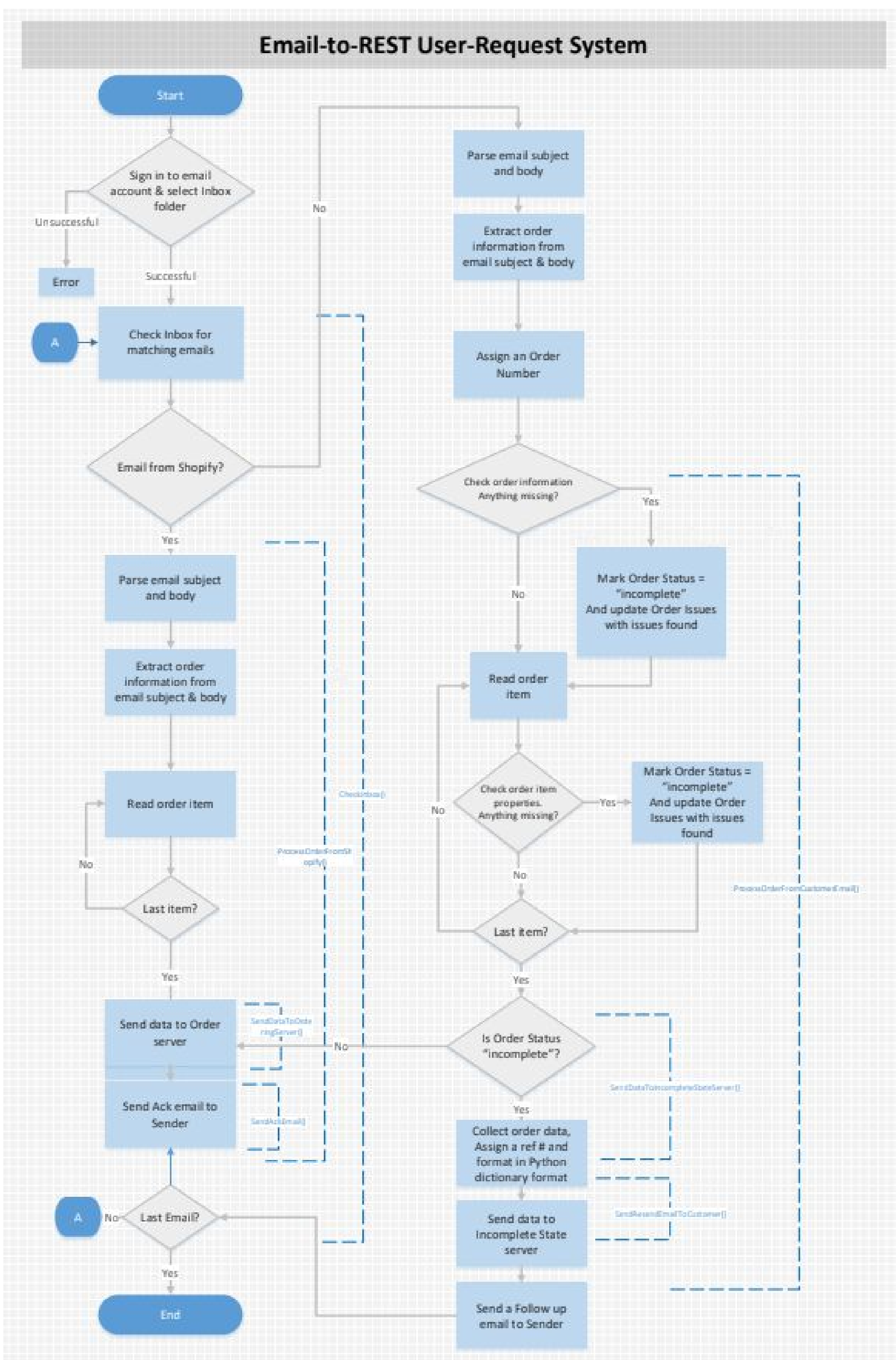

Figure (18): Application Flow Chart

Application Code

Main application

```
308   M = imaplib.IMAP4_SSL('imap.gmail.com')
309
310   try:
311     gmail_user = 'saqer2011@gmail.com'
312     gmail_password = 'S@qer123'
313     M.login(gmail_user, gmail_password)
314     rv, data = M.select("INBOX")  # to select the inbox folder mailbox
315     if rv == 'OK':
316       print ("Processing mailbox...\n")
317       Check_Inbox(M)
318       M.close()
319   except imaplib.IMAP4.error:
320     print ("Error logging in to Gmail server")
321
322
323   M.logout()
324
```

Figure (19): Application Main Function (Main-ver7_.Py)

As in Figure (19) lines 308 through line 323, using the IMAP protocol, application attempts to log in to email account and once it is logged in, the inbox folder is selected to start checking emails from (line 193). Finally, the application calls the CheckMail() function to start looking for order email match.

Check_Inbox() function:

```
15  def Check_Inbox(M):
16    rv, data = M.search(None, "ALL") #ALL for read and unread emails. UNSEEN for unread emails only
17    if rv != 'OK':
18      #print ("No messages found!")
19      return
20    OrderDetails_JSON = ""
21    for num in data[0].split():
22      rv, data = M.fetch(num, '(RFC822)')
23      if rv != 'OK':
24        print ("Error getting message", num)
25        return
26      msg = email.message_from_bytes(data[0][1])
27      msgFrom = (msg['from'])
28      msgSubject = (msg['Subject'])
29      msgID = (msg['Message-ID'])
30      date_tuple = email.utils.parsedate_tz(msg['Date'])
31      if date_tuple:
32        msg_received_date = datetime.datetime.fromtimestamp(email.utils.mktime_tz(date_tuple))
33
34      else:
35        msg_received_date = "unknown"
36
37      for part in msg.walk():
38       if part.get_content_type() == "text/plain": # ignore attachments/html
39        body = part.get_payload(decode=True)
40
41      if "Reference#" in msgSubject:
42        Ref = (msgSubject[msgSubject.find("Reference#")+11:msgSubject.find(" Order incomplete")])
43        data = "{Reference: "+ str(Ref)+" | Date:"+msg_received_date +" Update:"+body
44        senddatatoIncompletestateserver(Ref, data)
45
46
47      else:
48        if ("[SamUAlberta]" in msgSubject) & ("saqer2011@gmail.com" in msgFrom):
49          orderdetails =ProcessOrderFromShopify(body, msgSubject, msgFrom, msgID)
50        else:
51          orderdetails =ProcessOrderFromCustomerEmail(body, msg_received_date, msgSubject, msgFrom, msgID)
52
```

Figure (20): Application Check_Mail Function (Main-ver7_.Py)

Once connection is established, as shown in Figure (20) line code 16 through 51 application loops through unread emails, captures email data (such as email from field, subject line field, email body) looking for a match criteria to identify if email was originated from Shopify system or otherwise.

Once identified, application calls the appropriate function depending on the email type (Figure (20) line code 48 through 51.

ProcessOrderFromShopify() function:

```
53 def ProcessOrderFromShopify(body, msgSubject, msgFrom, msgID):
54   body = body.decode('utf-8') +"END"
55 
56   orderID = (msgSubject[msgSubject.find(" #")+2:msgSubject.find(" placed")])
57   msgSubject = msgSubject + "|"
58   customeremail = msgFrom
59   customer = (msgSubject[msgSubject.find("by ")+2:msgSubject.find("|")])
60   orderdetails =  (body[body.find("store, ")+6:body.find("END")])
61   #order_received_date = (orderdetails[orderdetails.find(" ")+1:orderdetails.find("\n")])
62   Paymentprocessingmethod = (orderdetails[orderdetails.find("Payment processing method:")+26:orderdetails.find("Delivery method:")])
63   Paymentprocessingmethod = Paymentprocessingmethod.replace('\r', ' ').replace('\n', ' ')
64   Deliverymethod = (orderdetails[orderdetails.find("Delivery method:")+16:orderdetails.find("Shipping address")])
65   Deliverymethod = Deliverymethod.replace('\r', ' ').replace('\n', ' ')
66   Shippingaddress = (orderdetails[orderdetails.find("Shipping address:")+17:orderdetails.find("Canada")])
67   Shippingaddress = Shippingaddress.replace('\r', ' ').replace('\n', ' ')
68   orderdetails = (orderdetails[orderdetails.find("*"):orderdetails.find("View order #")])
69   items_JSON = ""
70   print ("--------------")
71 
72 
73 
74   if orderdetails != "":
75     for line in orderdetails.splitlines():
76     #{orderID:1007, customer:  Dennis D ,qty:  1 , desc: T-Shirt - L / Red / Fabric  , SKU: U: 0001 ,price:  $0.00, qty:  1 ,
77     #desc: Sweater - L / Black / Cotton  , SKU: U: 0002 ,price:  $0.00, qty:  1 , desc: Short Sleeve T-Shirt - L  , SKU: U: 0005 ,price:  $
78       if line != "":
79         item = (line[line.find("*"):line.find("each")])
80         if ("*" in item):
81           itemqty = int(item[item.find("*")+2:item.find("x")])
82         if ("-" in item):
83           itemqty = (item[item.find("-")+2:item.find("x")])
84         itemdesc = (item[item.find("x ")+2:item.find("(SKU")])
85         itemSKU = (item[item.find("SKU: ")+2:item.find(")")])
86         itemprice = (item[item.find("for ")+3:item.find(" each")])
87         if items_JSON != "":
88           items_JSON = items_JSON + ", qty: "+str(itemqty)+" , desc: "+itemdesc+" , SKU: "+itemSKU+" ,price: "+itemprice
89         else:
90           items_JSON = items_JSON + "qty: "+str(itemqty)+" , desc: "+itemdesc+" , SKU: "+itemSKU+" ,price: "+itemprice
91 
92   orderdetails_Dict = orderdetails_JSON = "{orderID:"+orderID+", customer: "+ customer+", customer Email: "+customeremail+", Payment proces
93   print (orderdetails_JSON)
94   senddatatoorderingserver(orderdetails_JSON)
95   print ("--------------")
96 
97   #SendAckEmail(gmail_user, gmail_password, msgFrom, orderID, orderdetails)
```

Figure (21): Application ProcessOrderFromShopify Function (Main-ver7_.Py)

As shown in Figure (21) line code 53 through 95, if email was identified as a Shopify email, email contents are processed using a consistent email text and values are captured based on text string before and after the values.
Once values are captured and verified, they are formatted into JSON, sent to ordering server, and returned to the main function.

ProcessOrderFromCustomeremail() function:

```
101 def ProcessOrderFromCustomerEmail(body, msg_received_date, msgSubject, msgFrom, msgID):
102   body = body.decode('utf-8') +"END"
103 
104   orderID = randint(1, 100)
105   msgSubject = msgSubject
106   customeremail = msgFrom
107   orderdetails =  (body[body.find("store, ")+6:body.find("END")])
108 
109   #order_received_date = (orderdetails[orderdetails.find(" ")+1:orderdetails.find("\n")])
110   Paymentprocessingmethod = (orderdetails[orderdetails.find("*Payment processing method: *")+29:orderdetails.find("*Delivery")])
111   Paymentprocessingmethod = Paymentprocessingmethod.replace('\r', ' ').replace('\n', ' ')
112   Deliverymethod = (orderdetails[orderdetails.find("*Delivery method:*")+18:orderdetails.find("*Shipping")])
113   Deliverymethod = Deliverymethod.replace('\r', ' ').replace('\n', ' ')
114   Shippingaddress = (orderdetails[orderdetails.find("*Shipping address:*")+19:orderdetails.find("Canada")])
115   Shippingaddress = Shippingaddress.replace('\r', ' ').replace('\n', ' ')
116   #orderdetails = (orderdetails[orderdetails.find("*"):orderdetails.find("View order #")])
117   items_JSON = ""
118 
119   print ("-------------")
120 
121   OrderStatus="Complete"
122   OrderIssue=""
123 
124   if Paymentprocessingmethod == " ":
125     OrderStatus="incomplete"
126     OrderIssue=OrderIssue+"Payment processing method missing \n"
127   if Deliverymethod == " ":
128     OrderStatus="incomplete"
129     OrderIssue=OrderIssue+"Delivery method missing \n"
130   if Shippingaddress == " ":
131     OrderStatus="incomplete"
132     OrderIssue=OrderIssue+"Shipping address missing \n"
133 
134 
135   Items = {'Sun Glasses':{'item': 'Aviator sunglasses', 'SKU': '0001', 'color': ['black','brown'], 'price': '$46'}, 'Lounger':{'item': 'Cen
136   ItemList = {'Sun Glasses', 'Lounger', 'T-Shirt', 'Sweater'}
137   item =""
138   itemqty = ""
139   items_JSON = ""
140   #for item in ItemList:
141   if orderdetails != "":
142 
143     for line in orderdetails.splitlines():
144       for eachKey in Items.keys():
145         if eachKey in line:
146           itemproperties= list(Items[eachKey].items())
147           item = eachKey
148           if ("x" in line):
149               itemqty = line.find("x")
150               itemqty = line[0:itemqty]
151               itemqty = itemqty.replace('-', '').replace('-', '')
152           else:
153             itemqty = "1"
154           LineSubItem = ""
```

Figure (22): Application ProcessOrderFromCustomeremail Function - Part 1 (Main-ver7_.Py)

```
154            LineSubItem = ""
155            for property_, SubItems in itemproperties:
156              if type(SubItems) is list:
157
158                for SubItem in SubItems:
159                  if SubItem in line:
160
161                    item = item +" - " +SubItem
162                    LineSubItem = SubItem
163                    break
164                if LineSubItem ==  "":
165                  OrderStatus="incomplete"
166                  OrderIssue=OrderIssue+" " + eachKey +" " + str(property_) + " missing or not available \n"
167
168
169            if items_JSON != "":
170              if (str(itemqty) !="") & (item !=""):
171                items_JSON = items_JSON + ", qty: "+str(itemqty)+" , desc: "+item
172            else:
173              if (str(itemqty) !="") & (item !=""):
174                items_JSON = "qty: "+str(itemqty)+" , desc: "+item
175
176      if items_JSON == "":
177        OrderStatus="incomplete"
178        OrderIssue=OrderIssue+" Item missing or  not available \n"
179    else:
180      OrderStatus="incomplete"
181      OrderIssue=OrderIssue+"Order items missing \n"
182
183
184    if OrderStatus=="incomplete":
185      print (OrderStatus)
186      print (OrderIssue)
187      Ref = randint(1, 100)
188      ReSendEmailToCustomer(Ref,gmail_user, gmail_password, msgFrom, msgSubject, OrderIssue)
189      orderdetails_JSON = orderdetails_Dict ="{Reference: "+ str(Ref)+" | Date:"+str(msg_received_date) +" Update incomplete order. OrderID: '
190      senddatatoIncompletestateserver(Ref, orderdetails_Dict)
191      #print (orderdetails_Dict)
192      print ("-------------")
193    else:
194      OrderIssue = "None"
195      orderdetails_JSON = orderdetails_Dict = "{orderID:"+str(orderID)+", customer Email: "+customeremail+", Payment processing method: "+Pay
196      print (orderdetails_Dict)
197      senddatatoorderingserver(orderdetails_JSON)
198      SendAckEmail(gmail_user, gmail_password, msgFrom, str(orderID), orderdetails)
199      print ("-------------")
200
201    return (orderdetails_JSON)
```

Figure (23): Application ProcessOrderFromCustomeremail Function - Part 2 (Main-ver7_.Py)

However, As shown in Figure (22) and Figure (23) line code 101 through 201, if email was identified as a customer email, an order ID is generated, and email contents are checked line by line searching for order information then items being orders text and values are captured based on text string before and after the values. Every Item is checked against available items' list shown in line 135 in Figure (22). Order status and order issues are tracked and recorded as every item is being checked such as in lines 176 through 178 in Figure (23).

As in lines 184 through 198 in Figure (23), once values are captured and verified, the order status is checked. If incomplete, a reference number is generated along with a timestamp, a follow up email is sent to customer with the reference number, and data is sent to incomplete state server to store by calling the "sendtoincompletestateserver" function with the reference number and the timestamp. Otherwise, they are formatted into JSON, sent to ordering server, and returned to the main function.

SendDataToIncompleteStateServer() function:

```
203
204  def senddatatoIncompletestateserver(Ref, data):
205
206    try:
207      requests.post('http://127.0.0.1:5001/post', headers={'Content-Type': 'application/json'}, data=json.dumps(data))
208      print ("DATA SENT TO INCOMPLETE STATE SERVER WITH REFERENCE #" + str(Ref))
209    except Exception as e: print (e)
```

Figure (24): Application SendDataToIncompleteStateServer Function (Main-ver7_.Py)

As shown in Figure (24) line code 206 through 209, this function's main purpose is capture data sent along with reference number and sent over to incomplete state server to store.

SendDataToIncompleteStateServer() function:

```
214  def senddatatoorderingserver(data):
215    try:
216      requests.post('http://127.0.0.1:5000/post', headers={'Content-Type': 'application/json'}, data=json.dumps(data))
217      print ("DATA SENT TO ORDERING SERVER SUCCESSFULLY")
218    except Exception as e: print (e)
219
```

Figure (25): Application SendDataToOrderingServer Function (Main-ver7_.Py)

As shown in Figure (25) line code 214 through 218, this function's main purpose is capture data sent and forward to ordering server to process.

SendAckemail() function:

```
276  def SendAckEmail(gmail_user, gmail_password, EmailTo, OrderID, orderdetails):
277    try:
278      subject = ' Confirmation of Order #'+OrderID
279    except:
280      print ("Error - Cannot retrieve order number for confirmation message")
281      subject = 'Confirmation of your order'
282
283    try:
284      body = "This is an automated response for your order: \n %s" % (orderdetails)
285    except:
286      print ("Error - Cannot retrieve order details for confirmation message")
287      body = 'Confirmation of your order details'
288
289
290    message = 'Subject: {}\n\n{}'.format(subject, body)
291
292    try:
293      server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
294      server.ehlo()
295      server.login(gmail_user, gmail_password)
296      server.sendmail(gmail_user, EmailTo, message)
297      server.close()
298      print ('Order Confirmed')
299    except:
300      print ('Error confirming order')
```

Figure (26): Application SendAckemail Function (Main-ver7_.Py)

As shown in Figure (26) line code 276 through 300, this function’s main purpose is login to the email account provided and send an acknowledgement email to sender's email address provided with order details.

```
250 def ReSendEmailToCustomer(Ref,gmail_user, gmail_password, EmailTo, msgSubject, OrderIssue):
251
252  subject = 'Reference#' +str(Ref)+ ' Order incomplete ' + msgSubject
253  body = "This is an automated response for your order \n Error - Your order has the following issues:\n"+OrderIssue+"\n Please do not chang
254
255  msg = email.mime.multipart.MIMEMultipart()
256  message = 'Subject: {}\n\n{}'.format(subject, body)
257  msg['Subject'] = subject
258  msg['body'] = body
259  msg['From'] = gmail_user
260  msg['To'] = EmailTo
261
262
263  send_to = [EmailTo]
264  try:
265    server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
266    server.ehlo()
267    server.login(gmail_user, gmail_password)
268    server.sendmail(gmail_user, EmailTo, message)
269    server.close()
270    print ('Order resend requested')
271  except:
272    print ('Error requesting order resend')
273
274
```

Figure (27): Application ResendemailToCustomerFunction (Main-ver7_.Py)

As shown in Figure (27) line code 250 through 272, this function's main purpose is login to the email account provided and send a request email to sender's email address provided with a reference number and order issues found in customer's email order.

Ordering Server Code:

```
1   import json
2   from flask import Flask, jsonify, request
3   #Used for starting the flask server
4   from flask import Flask #Import flask mains
5   from flask import request #Import flask requests
6   from flask import render_template #Import flask render templates
7   import json, urllib #import api modules
8   import time #Imporing time in darlek voice
9
10  app = Flask(__name__)
11
12  @app.route('/post', methods = ['POST'])
13  def Get_Order_Details():
14      Order_Details = request.get_json()
15      Order_Details = str(Order_Details)
16      print (Order_Details)
17      return Order_Details
18  if __name__ == '__main__':
19      app.run(debug=True)
```

Figure (28):Ordering Server (Ordering_Server.Py)

As shown in Figure (28) line code 1 through 19, this is a demonstration of the ordering system and as shown in the code, a socket is open, listening and waiting for data to be received then it is simply printed to the screen.

Incomplete State Server Code:

```
import json
from flask import Flask, jsonify, request
#Used for starting the flask server
from flask import Flask #Import flask mains
from flask import request #Import flask requests
from flask import render_template #Import flask render templates
import json, urllib #import api modules
import time #Imporing time in darlek voice

IncompleteOrders = {}
app = Flask(__name__)

@app.route('/post', methods = ['POST'])
def Get_Order_Details():
    Order_Details = request.get_json()
    Order_Details = str(Order_Details)
    Ref = (Order_Details[Order_Details.find("{Reference: ")+12:Order_Details.find(", Update")])
    Details = (Order_Details[Order_Details.find("Update"):Order_Details.find(" }")])
    IncompleteOrders.update({Ref:Details})
    print (IncompleteOrders)
    return Order_Details
if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5001, debug=True)
```

Figure (29): Incomplete State Server (IncompleteStateServer.py)

As shown in Figure (29) line code 1 through 23, this is a demonstration of the incomplete state server and as shown in the code, a socket is open, listening and waiting for data to be received and stored in a dictionary format.

Testing

Test Plan

The purpose of the test plan is to identify and correct defects and errors of the application. By running repetitive tests on different order emails, application is evaluated for expected results and the number of order emails it can handle.

Test plan aims to run the application with many test order emails and observe results to see if application behaves as expected. If application fails, errors are noted for debugging and corrections

Test Items

1. Test application with Shopify email orders
2. Test application with well-formed customer email orders
3. Test application with non-well-formed customer email orders

Functional Testing

- Does the application logic run as per the theory demonstrated in the flow chart?
- Does the application successcds to login to email account at all time?
- Does the application manage to distinguish Shopify emails from customer emails?
- Is the application able to detect non-well-formatted emails at all times?
- Does the application manage to handle non-well-formatted emails and track issues?
- Is the application able to send data extracted with the proper format?

Test Environment

Tests were run against test items listed above and application was modified as defects and errors were discovered. Table (2) shows a list of the components used in the test environment :

Hardware	PC:HP 8300 - CPU:Intel i5 - RAM:8GB - Drive: 256GB SSD
Operating System	Windows 10 Pro - Application runs on windows CMD
Python 3.7	Running in virtualenv & requests module
Receiving Server	Single-threaded Flask server accepting http POSTs
IDE	Notepad++
email	Gmail account
E-commerce	Shopify with basic paid plan

Table (2): Testing Environment

Test Results

```
Python 3.6.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
rta  T5Y0G6     ,qty: 1 , desc: Aviator sunglasses  , SKU: U: 0003 ,price:  $0.0
0 }
DATA SENT TO ORDERING SERVER SUCCESSFULLY
--------------
--------------
{orderID:1003, customer:  Will  Smith, customer Email: SamUAlberta <saqer2011@gm
ail.com>, Payment processing method:     , Delivery method:     Standard Shippin
g    , Shipping address:     Will  Smith    123 qwe    Edmonton, Alberta  T6M0J2
     ,qty: 1 , desc: Aviator sunglasses  , SKU: U: 0003 ,price:  $0.00, qty:  ,
desc: Mid-century lounger  , SKU: U: 0003 ,price:  $0.00 }
DATA SENT TO ORDERING SERVER SUCCESSFULLY
--------------
--------------
{orderID:1006, customer:  Zak El, customer Email: SamUAlberta <saqer2011@gmail.c
om>, Payment processing method:     , Delivery method:     Standard Shipping
, Shipping address:     Zak El    357 brintnell blvd    Edmonton, Alberta  T5Y0G
6     ,qty:  , desc: Short Sleeve T-Shirt - L  , SKU: U: 0005 ,price:  $0.00 }
DATA SENT TO ORDERING SERVER SUCCESSFULLY
--------------
--------------
{orderID:1007, customer:  Dennis D, customer Email: SamUAlberta <saqer2011@gmail
.com>, Payment processing method:     , Delivery method:     Standard Shipping
  , Shipping address:     Dennis D    357 brintnell blvd    Edmonton, Alberta  T
5Y0G6     ,qty:  , desc: T-Shirt - L / Red / Fabric  , SKU: U: 0001 ,price:  $0.
00, qty:  , desc: Sweater - L / Black / Cotton  , SKU: U: 0002 ,price:  $0.00, q
ty:  , desc: Short Sleeve T-Shirt - L  , SKU: U: 0005 ,price:  $0.00 }
DATA SENT TO ORDERING SERVER SUCCESSFULLY
--------------
--------------
{orderID:17, customer Email: Saqer El awour <saqer2008@gmail.com>, Payment proce
ssing method:     *Visa*    , Delivery method:     Standard Shipping    , Shippi
ng address:     Sam El    000 Main blvd    Edmonton, Alberta T5Y0G6     ,qty:
   1 , desc: Sweater - Medium - Black,Order Status: Complete, Order Issues: None
 }
DATA SENT TO ORDERING SERVER SUCCESSFULLY
Order Confirmed
--------------
--------------
incomplete
Shipping address missing
Ln: 78  Col: 55
```

Figure (30): Application Test Results

Test results shown in Figure (30) are the outcome of a test done against 6 order email:

- 4 emails were identified as Shopify well-formed orders. Orders were accepted and sent to ordering server

- 1 email was identified as a customer well-formed order - Order was accepted and sent to ordering server
- 1 emails was identified as a customer non-well-form format (missing shipping address). Order was not accepted and was sent to incomplete state server

Test result show that the details of well-formed emails where non-well-formed emails were marked as incomplete and a resend request was sent to sender requesting another copy of the order email.

Command Prompt - FLASK_GET.py

```
Status: }
127.0.0.1 - - [01/Apr/2018 08:27:36] "POST /post HTTP/1.1" 200 -
{orderID:1003, customer:  Will  Smith ,qty:  1 , desc: Aviator sunglasses  , SKU:  0003 ,price:  $0.00, qty:
 , desc: Mid-century lounger  , SKU:  0003 ,price:  $0.00,Order Status: }
127.0.0.1 - - [01/Apr/2018 08:27:36] "POST /post HTTP/1.1" 200 -
{orderID:1006, customer:  Zak El ,qty:  , desc: Short Sleeve T-Shirt - L  , SKU:  0005 ,price:  $0.00,Order S
tatus: }
127.0.0.1 - - [01/Apr/2018 08:27:37] "POST /post HTTP/1.1" 200 -
{orderID:1007, customer:  Dennis D ,qty:  , desc: T-Shirt - L / Red / Fabric  , SKU:  0001 ,price:  $0.00, qt
y:  , desc: Sweater - L / Black / Cotton  , SKU:  0002 ,price:  $0.00, qty:  , desc: Short Sleeve T-Shirt - L
  , SKU:  0005 ,price:  $0.00,Order Status: }
127.0.0.1 - - [01/Apr/2018 08:27:37] "POST /post HTTP/1.1" 200 -
```

Figure (31): Ordering Server Test Results

```
Select Command Prompt - IncompleteStateServer.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 110-460-474
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
{'72 | Date:2018-04-05 13:09:57 Update incomplete order. OrderID: 10, customer E
mail: Sam Elawour <elawour@ualberta.ca>, Payment processing method:      *Visa*
  , Delivery method:     Standard Shipping    , Shipping address:  ,qty:      1
, desc: T-Shirt - Large, qty:      1 , desc: Sweater - Small - Black, Order Issu
es: Shipping address missing \n ': 'Update incomplete order. OrderID: 10, custom
er Email: Sam Elawour <elawour@ualberta.ca>, Payment processing method:     *Vis
a*    , Delivery method:     Standard Shipping    , Shipping address:  ,qty:
  1 , desc: T-Shirt - Large, qty:      1 , desc: Sweater - Small - Black, Order
Issues: Shipping address missing \n'}
127.0.0.1 - - [05/Apr/2018 16:15:46] "POST /post HTTP/1.1" 200 -
{'72 | Date:2018-04-05 13:09:57 Update incomplete order. OrderID: 10, customer E
mail: Sam Elawour <elawour@ualberta.ca>, Payment processing method:      *Visa*
  , Delivery method:     Standard Shipping    , Shipping address:  ,qty:      1
, desc: T-Shirt - Large, qty:      1 , desc: Sweater - Small - Black, Order Issu
es: Shipping address missing \n ': 'Update incomplete order. OrderID: 10, custom
er Email: Sam Elawour <elawour@ualberta.ca>, Payment processing method:     *Vis
a*    , Delivery method:     Standard Shipping    , Shipping address:  ,qty:
  1 , desc: T-Shirt - Large, qty:      1 , desc: Sweater - Small - Black, Order
Issues: Shipping address missing \n', '69 | Date:2018-04-05 13:37:11 Update inco
mplete order. OrderID: 71, customer Email: Saqer El awour <saqer2008@gmail.com>,
 Payment processing method:     *Visa*    , Delivery method:     Standard Shippi
```

Figure (32): Incomplete State Server Test Results

Figure (31) and Figure (32) show test results on the receiving servers side. Orders that came from well-formed emails made it to the ordering server and orders that came from non-well-formed emails made it to the incomplete state server.

Installation Instruction

All coding, testing, and prototyping were done on Python3.7 on Windows 10 OS. Here is a set of instructions on how to set up a working on environment for the application to run:

1. **Python 3**
    a. Download the latest version of Python 3 for Windows from the Python website
    b. Run Installation file and accept all defaults - Note installation location to reference when running the application

2. **Requests Python module:**
    a. Once Python is installed successfully, open windows command prompt
    b. Navigate to location noted above in step 1-b
    c. Run the following command: python -m pip install requests

3. **Flask:**
    a. Next, on the same windows command prompt location from Step 2-b
    b. Run the following command: pip install flask

4. **Setting up virtual Environment for:**
    a. Once installed, Flask requires a virtual environment which is easier to setup on Python3.6
    b. C:\Users\<User>\AppData\Local\Programs\Python\Python36-32\python.exe -m venv venv
    c. C:\Users\<User>\AppData\Local\Programs\Python\Python36-32\venv\Scripts\activate
    d. (venv) C:\Users\<User>\AppData\Local\Programs\Python\Python36-32\venv\Scripts>

Conclusion

The E2R application gives a simple and reliable way to process and deliver order information. It provides custom functionalities to help merchandiser with the ordering process and programmers with a concise syntax to build applications for other purposes. The experience of developing the application components in the system also helped me getting exposed to Python framework and its interaction with other services. Within the Python framework, I have successfully accomplished the requirements of the system. Once this system code is posted and made available, it can be used to help students and substitute manual systems currently in service. In summary, this system intends to bring great user experience to both merchandisers and ordering systems.

Confirmation of Originality

I affirm that the work done on this project and this report is my original work. I have acknowledged the access to Wenting Zhang's capstone project proposal in the acknowledgment section and identified all used sources in the reference list. I declare that none of my application code and this report has been developed or submitted by Wenting or any other student before.

References

Designing a RESTful API with Python and Flask. Retrieved from https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

Flask Quickstart. Retrieved from http://flask-restful.readthedocs.io/en/0.3.5/quickstart.html

How to use dictionaries in Python.Retrieved from http://www.pythonforbeginners.com/dictionary/how-to-use-dictionaries-in-python

IMAP email retrieve. Retrieved from https://github.com/rae/scripts/blob/master/clear-unread

Installing Python Modules. Retrieved from https://docs.python.org/2/installing/index.html

Lu, P. (2017) Possible MINT Capstone Projects

Python imaplib — IMAP4 protocol client. Retrieved from https://docs.python.org/2/library/imaplib.html?highlight=import%20imaplib

Python smtplib. Retrieved from https://docs.python.org/2/library/smtplib.html

Python String Find(). Retrieved from https://www.tutorialspoint.com/python/string_find.htm

Python String Splitlines(). Retrieved from https://www.tutorialspoint.com/python/string_splitlines.htm

Reid E. (2016) Python and IMAP. Retrieved from https://github.com/rae/scripts/blob/master/clear-unread

Satyanarayana, S. (2011) SMTP Example. Retrieved from https://github.com/tonyito/Polobot/blob/master/polobot_1.0.py

SMTP email send. Retrieved from https://github.com/tonyito/Polobot/blob/master/polobot_1.0.py

The Python Standard Library. Retrieved from https://docs.python.org/2/library/index.html

Zhang, W. (2017) email-to-REST User-Request Translation System Project Proposal