

Automate Updates to Catalog information

Introduction

For an online fruits store, you need to develop a system that updates the catalog information with data provided by your suppliers. The suppliers send the data as large images with an associated description of the products in two files (.TIF for the image and .txt for the description). The images needed to be converted to smaller jpeg images and the text needs to be turned into an HTML file that shows the image and the product description. The contents of the HTML file needed to be uploaded to a web service that is already running using Django. You also needed to gather the name and weight of all fruits from the .txt files and use a Python request to upload it to your Django server.

You created a Python script that processed the images and descriptions and then updated your company's online website to add the new products.

Once the task was complete, the supplier was notified with an email that indicated the total weight of fruit (in lbs) that were uploaded. The email had a PDF attached with the name of the fruit and its total weight (in lbs).

Finally, in parallel to the automation running, we wanted to check the health of the system and send an email if something goes wrong.

Fetching supplier data

Note: The original source files (download_bucket_file.sh and example_upload.py) referenced in this project were provided as part of a **Google-provided lab environment** and are **not publicly accessible**.

In the original lab, product data was stored in a **Google Cloud Storage bucket** and included:

- High-resolution product images in .TIF format.
- Corresponding product descriptions in .txt format.

While these source scripts are not available in this repository due to restricted access, the logic and automation for downloading, processing, and uploading the data have been **reimplemented or simulated** using Python scripts included in this project.

You'll first need to get the information from the supplier that is currently stored in a Cloud Storage Bucket. The supplier has sent data as large images with an associated description of the products in two files (.TIF for the image and .txt for the description).

Here, you'll find two script files **download_bucket_file.sh** and the **example_upload.py** files. You can view it by using the following command.

```
ls ~/
```

Output:

```
download_bucket_file.sh  example_upload.py
```

To download the file from the supplier onto 'terminal' we will first grant executable permission to the **download_bucket_file.sh** script.

```
sudo chmod +x ~/download_bucket_file.sh
```

Run the **download_bucket_file.sh** shell script.

```
./download_bucket_file.sh
```

Output:

```
--2024-07-29 15:45:07-- https://storage.googleapis.com/gwg-
content/gic223/supplier-data.tar.gz

Resolving storage.googleapis.com (storage.googleapis.com)... 7
4.125.197.207, 74.125.135.207, 74.125.142.207, ...

Connecting to storage.googleapis.com (storage.googleapis.com)|
74.125.197.207|:443... connected.

HTTP request sent, awaiting response... 200 OK

Length: 88653399 (85M) [application/x-gzip]

Saving to: 'supplier-data.tar.gz'

supplier-
data.tar.gz    100%[=====>]   84.55M   5
1.9MB/s      in 1.6s

2024-07-29 15:45:10 (51.9 MB/s) - 'supplier-
data.tar.gz' saved [88653399/88653399]
```

```
ls
```

Output:

```
download_bucket_file.sh  example_upload.py  supplier-data.tar.gz
```



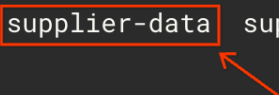
You have now downloaded a file named **supplier-data.tar.gz** containing the supplier's data. Let's extract the contents from this file using the following command:

```
tar xf ~/supplier-data.tar.gz
```

This creates a directory named **supplier-data**, that contains subdirectories named **images** and **descriptions**.

```
ls
```

```
download_bucket_file.sh  example_upload.py  supplier-data  supplier-  
data.tar.gz
```



List contents of the **supplier-data** directory using the following command:

```
ls ~/supplier-data
```

Output:

```
descriptions  images
```

The subdirectory **images** contain images of various fruits, while the **descriptions** subdirectory has text files containing the description of each fruit. You can have a look at any of these text files using **cat** command.

```
cat ~/supplier-data/descriptions/007.txt
```

Output:

```
Mango  
  
300 lbs  
  
Mango contains higher levels of vitamin C than ordinary fruit.  
Eating mango  
  
can also reduce cholesterol and triglycerides, and help preven  
t cardiovascular  
  
disease. Due to its high level of vitamins, regular consumptio  
n of mango play an  
  
important role in improving body function and moisturizing the  
skin.  
  
The first line contains the name of the fruit followed by the  
weight of the fruit and finally the description of the fruit.
```

Working with supplier images

In this section, you will write a Python script named **changeImage.py** to process the supplier images. You will be using the PIL library to update all images within **~/supplier-data/images** directory to the following specifications:

- **Size:** Change image resolution from **3000x2000** to **600x400** pixel
- **Format:** Change image format from **.TIFF** to **.JPEG**

Create and open the file using **nano** editor.

```
nano ~/changeImage.py
```

Add a shebang line in the first line.

```
#!/usr/bin/env python3
```

This is the challenge section, where you will be writing a script that satisfies the above objectives.

Note: The raw **images** from images subdirectory contains alpha transparency layers. So, it is better to first convert **RGBA** 4-channel format to **RGB** 3-channel format before processing the images. Use the **convert("RGB")** method for converting RGBA to RGB image.

Python script:

```
#!/usr/bin/env python3

from PIL import Image
import os

path = "./supplier-data/images/"

for f in os.listdir("./supplier-data/images"):
    if f.endswith(".tiff"):
        split_f = f.split(".")
        name = split_f[0] + ".jpeg"
        im = Image.open(path + f).convert("RGB")
        im.resize((600, 400)).save("./supplier-
data/images/" + name, "JPEG")
```

After processing the images, save them in the same path **~/supplier-data/images**, with a JPEG extension.

Once you have completed editing the **changeImage.py** script, save the file by clicking **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Grant executable permissions to the **changeImage.py** script.

```
sudo chmod +x ~/changeImage.py
```

Now run the **changeImage.py** script:

```
./changeImage.py
```

Now, let's check the specifications of the images you just updated. Open any image using the following command:

```
file ~/supplier-data/images/003.jpeg
```

Output:

```
/home/student/supplier-data/images/003.jpeg: JPEG image data, JFIF
standard 1.01, aspect ratio, density 1x1, segment length 16, baseline,
precision 8, 600x400, components 3
```

Uploading images to web server

You have modified the fruit images through **changeImage.py** script. Now, you will have to upload these modified images to the web server that is handling the fruit catalog. To do that, you'll have to use the Python **requests** module to send the file contents to the **[external-IP-address]/upload** URL.

Copy the **external IP address** of your instance from the Connection Details Panel on the right side and enter the **IP address** in a new web browser tab. This opens a web page displaying the text **Fruit Catalog**.

In the home directory, you'll have a script named **example_upload.py** to upload images to the running fruit catalog web server. To view the **example_upload.py** script use the **cat** command.

```
cat ~/example_upload.py
```

Output:

```
#!/usr/bin/env python3

import requests

# This example shows how a file can be uploaded using
# The Python Requests module

url = "http://localhost/upload/"

with open('/usr/share/apache2/icons/icon.sheet.png', 'rb') as
opened:

    r = requests.post(url, files={'file': opened})
```

In this script, we are going to upload a sample image named **icon.sheet.png**.

Grant executable permission to the **example_upload.py** script.

```
sudo chmod +x ~/example_upload.py
```



Execute the **example_upload.py** script, which will upload the images.

```
./example_upload.py
```

Now check out that the file **icon.sheet.png** was uploaded to the web server by visiting the URL **[external-IP-address]/media/images**, followed by clicking on the file name.

Output:

Index of /media/images

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 icon.sheet.png	2024-07-29 13:21	8.8K	

Apache/2.4.59 (Debian) Server at 34.82.12.192 Port 80

In a similar way, you are going to write a script named **supplier_image_upload.py** that takes the **jpeg** images from the **supplier-data/images** directory that you've processed previously and uploads them to the web server fruit catalog.

Use the **nano** editor to create a file named **supplier_image_upload.py**:

```
nano ~/supplier_image_upload.py
```

Complete the script with the same technique as used in the file **example_upload.py**.

Python script:

```
#!/usr/bin/env python3

import requests

import os

# This example shows how a file can be uploaded using
# The Python Requests module
```

```
url = "http://localhost/upload/"

for f in os.listdir("./supplier-data/images"):

    if f.endswith(".jpeg"):

        with open('./supplier-
data/images/' + f, 'rb') as opened:

            r = requests.post(url, files={'file': opened})
```

Once you have completed editing the **supplier_image_upload.py** script, save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Grant executable permission to the **supplier_image_upload.py** script.

```
sudo chmod +x ~/supplier_image_upload.py
```













Run the **supplier_image_upload.py** script.

```
./supplier_image_upload.py
```

Refresh the URL opened earlier, and now you should find all the images uploaded successfully.

Output:

Index of /media/images

Name	Last modified	Size	Description
<hr/>			
 Parent Directory		-	
 001.jpeg	2024-07-29 13:24	19K	
 002.jpeg	2024-07-29 13:24	17K	
 003.jpeg	2024-07-29 13:24	28K	
 004.jpeg	2024-07-29 13:24	18K	
 005.jpeg	2024-07-29 13:24	20K	
 006.jpeg	2024-07-29 13:24	14K	
 007.jpeg	2024-07-29 13:24	13K	
 008.jpeg	2024-07-29 13:24	14K	
 009.jpeg	2024-07-29 13:24	24K	
 010.jpeg	2024-07-29 13:24	21K	
 icon.sheet.png	2024-07-29 13:21	8.8K	

Apache/2.4.59 (Debian) Server at 34.82.12.192 Port 80

Uploading the descriptions

The Django server is already set up to show the fruit catalog for your company. You can visit the main website by entering **external-IP-address** in the URL bar or by removing **/media/images** from the existing URL opened earlier. The interface looks like this:

Fruit Catalog

Check out the Django REST framework, by navigating to **external-IP-address/fruits** in your browser.

Django REST framework

Fruits List

Options GET

List all fruit or create a new fruits

GET /fruits/

HTTP 200 OK
Allow: GET, POST, OPTIONS
Content-Type: application/json
Vary: Accept

Media type: application/json

Content:

POST

Currently, there are no products in the fruit catalog web-server. You can create a test fruit entry by entering the following into the **content** field:

```
{"name": "Test Fruit", "weight": 100, "description": "This is the description of my test fruit", "image_name": "icon.sheet.png"}
```

After entering the above data into the content field click on the POST button. Now visit the main page of your website (by going to **http://[external-IP-address]/**), and the new test fruit you uploaded appears.

Fruit Catalog

Test Fruit

Weight: 100 lbs



This is the description of my test fruit

To add fruit images and their descriptions from the supplier on the fruit catalog web-server, create a new Python script that will automatically POST the **fruit images** and their respective **description** in JSON format.

Write a Python script named **run.py** to process the text files (**001.txt**, **003.txt** ...) from the **supplier-data/descriptions** directory. The script should turn the data into a JSON dictionary by adding all the required fields, including the image associated with the fruit (**image_name**), and uploading it to **http://[external-IP-address]/fruits** using the Python **requests** library.

Create **run.py** using the **nano** editor:

```
nano ~/run.py
```

Add the shebang line and import necessary libraries.

```
#!/usr/bin/env python3

import os

import requests
```

Now, you'll have to process the .txt files (named **001.txt**, **002.txt**, ...) in the **supplier-data/descriptions/** directory and save them in a data structure so that you can then upload them via JSON. Note that all files are written in the following format, with each piece of information on its own line:

- name
- weight (in lbs)
- description

The data model in the Django application **fruit** has the following fields: **name**, **weight**, **description** and **image_name**. The **weight** field is defined as an **integer** field. So when you process the weight information of the fruit from the .txt file, you need to convert it into an integer. For example if the weight is "**500 lbs**", you need to **drop "lbs"** and **convert "500" to an integer**.

The **image_name** field will allow the system to find the image associated with the fruit. Don't forget to add all fields, including the **image_name**! The final JSON object should be similar to:

```
{ "name": "Watermelon", "weight": 500, "description": "Watermelon is good for relieving heat, eliminating annoyance and quenching thirst. It contains a lot of water, which is good for relieving the symptoms of acute fever immediately. The sugar and salt contained in watermelon can diuretic and eliminate kidney inflammation. Watermelon also contains substances that can lower blood pressure.", "image_name": "010.jpeg" }
```

Iterate over all the fruits and use **post** method from Python requests library to upload all the data to the URL **http://[external-IP-address]/fruits**.

Python script:

```
#!/usr/bin/env python3

import os
import requests

fruits = {}
keys = ["name", "weight", "description", "image_name"]
index = 0
path = "./supplier-data/descriptions/"
img_path = "./supplier-data/images/"
for file in os.listdir("./supplier-data/descriptions"):
    with open(path + file) as f:
        for ln in f:
            line = ln.strip()
            if "lbs" in line:
                nline = line.split()
                wght = int(nline[0])
```

```

        fruits["weight"] = wght

        index += 1

    else:

        try:

            fruits[keys[index]] = line

            index += 1

        except:

            fruits[keys[2]] = line

    index = 0

    split_f = file.split(".")

    name = split_f[0] + ".jpeg"

    for file in os.listdir("./supplier-data/images"):

        if file == name:

            fruits["image_name"] = name

            response = requests.post("http://<External_IP>/fruits/
", json=fruits)

            fruits.clear()

```

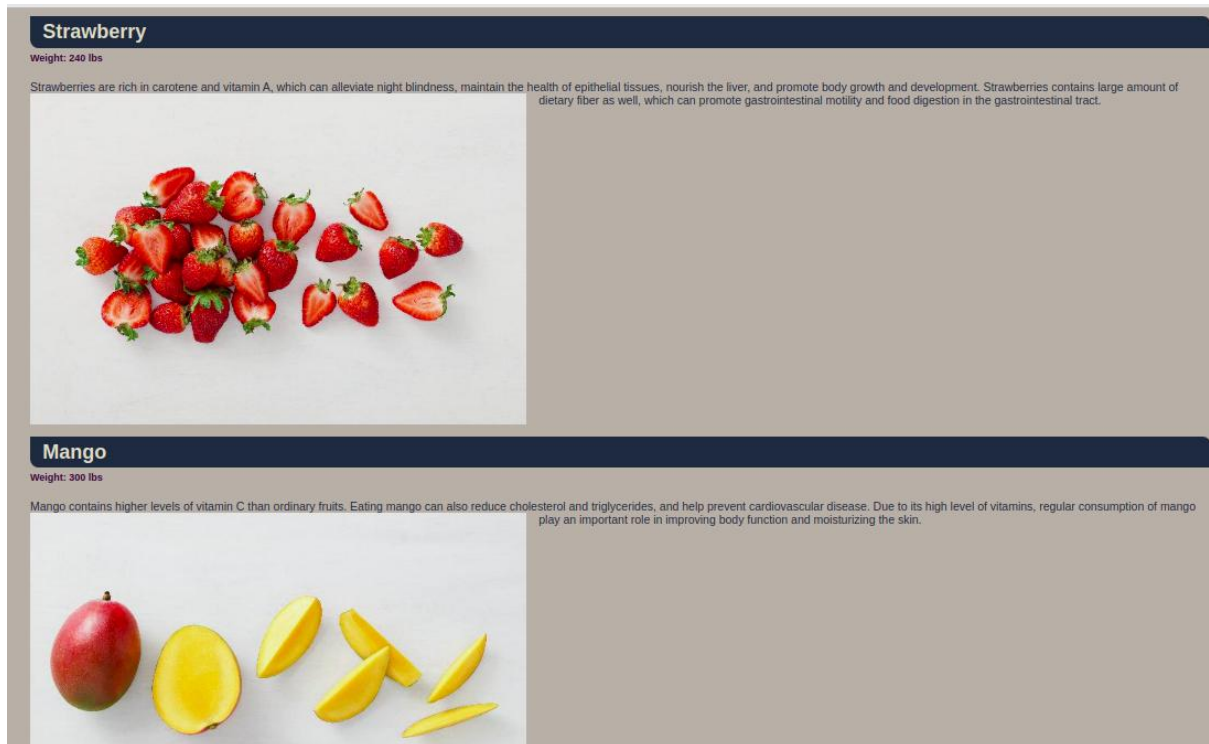
Once you complete editing **run.py** script, save the file by clicking **Ctrl+o**, **Enter** key, and **Ctrl+x**.
Grant executable permission to the **run.py** script.

```
sudo chmod +x ~/run.py
```

Run the **run.py** script:

```
./run.py
```

Now go to the main page of your website (by going to **http://[external-IP-address]/**) and check out how the new fruits appear.



Generate a PDF report and send it through email

Once the **images** and **descriptions** have been uploaded to the fruit store web-server, you will have to generate a PDF file to send to the supplier, indicating that the data was correctly processed. To generate PDF reports, you can use the **ReportLab** library. The content of the report should look like this:

Processed Update on <Today's date>

[blank line]

name: Apple

weight: 500 lbs

[blank line]

name: Avocado

weight: 200 lbs

[blank line]

...

Script to generate a PDF report

Create a script **reports.py** to generate PDF report to supplier using the **nano** editor:

```
nano ~/reports.py
```

Add a shebang line in the first line.

```
#!/usr/bin/env python3
```

Using the **reportlab** Python library, define the method **generate_report** to build the PDF reports. We have already covered how to generate PDF reports in an earlier lesson; you will want to use similar concepts to create a PDF report named **processed.pdf**.

Python script:

```
#!/usr/bin/env python3

from reportlab.platypus import SimpleDocTemplate
from reportlab.platypus import Paragraph, Spacer, Table, Image
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib import colors

def generate_report(attachment, title, paragraph):
    styles = getSampleStyleSheet()
    report = SimpleDocTemplate(attachment)
    report_title = Paragraph(title, styles["h1"])
    report_info = Paragraph(paragraph, styles["BodyText"])

    table_style = [('GRID', (0, 0), (-1, -1), 1, 1, colors.black),
                   ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
                   ('ALIGN', (0, 0), (-1, -1), 'CENTER')]

    empty_line = Spacer(1, 20)

    report.build([report_title, empty_line, report_info])
```

Once you have finished editing the script **reports.py**, save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Create another script named **report_email.py** to process supplier fruit description data from **supplier-data/descriptions** directory. Use the following command to create **report_email.py**.

```
nano ~/report_email.py
```

Add a shebang line.

```
#!/usr/bin/env python3
```

Import all the necessary libraries(**os**, **datetime** and **reports**) that will be used to process the text data from the **supplier-data/descriptions** directory into the format below:

name: Apple

weight: 500 lbs

[blank line]

name: Avocado

weight: 200 lbs

[blank line]

...

Once you have completed this, call the main method which will process the data and call the **generate_report** method from the **reports** module:

```
if __name__ == "__main__":
```

You will need to pass the following arguments to the **reports.generate_report** method: the text description processed from the text files as the **paragraph** argument, the report title as the **title** argument, and the file path of the PDF to be generated as the **attachment** argument (use **'/tmp/processed.pdf'**)

```
reports.generate_report(attachment, title, paragraph)
```

Python script:

```
#!/usr/bin/env python3

import os

import datetime

import reports

import emails


dt = datetime.date.today().strftime("%B %d, %Y")

date = "Processed Update on " + dt

names = []

weights = []
```

```
path = "./supplier-data/descriptions/"

for file in os.listdir("./supplier-data/descriptions"):
    with open(path + file) as f:
        for ln in f:
            line = ln.strip()
            if len(line) <= 10 and len(line) > 0 and "lb"not in line:
                fruit_name = "name: " + line
                names.append(fruit_name)
            if "lbs" in line:
                fruit_weight = "weight: " + line
                weights.append(fruit_weight)

summary = ""
for name, weight in zip(names, weights):
    summary += name + '<br />' + weight + '<br />' + '<br />'

if __name__ == "__main__":
    reports.generate_report("/tmp/processed.pdf", date, summary)

    sender = "automation@example.com"

    receiver = "<USERNAME>@example.com".format(os.environ.get('USER'))

    subject = "Upload Completed - Online Fruit Store"

    body = "All fruits are uploaded to our website successfully. A detailed list is attached to this email."

    message = emails.generate_email(sender, receiver, subject, body, "/tmp/processed.pdf")

    emails.send_email(message)
```


Once you have completed the **report_email.py** script. Save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Send report through email

Once the PDF is generated, you need to send the email using the **emails.generate_email()** and **emails.send_email()** methods.

Create **emails.py** using the **nano** editor using the following command:

```
nano ~/emails.py
```

Define **generate_email** and **send_email** methods by importing necessary libraries.

Python script:

```
#!/usr/bin/env python3

import email.message
import mimetypes
import os.path
import smtplib

def generate_email(sender, recipient, subject, body, attachment_path):

    """Creates an email with an attachment."""

    # Basic Email formatting

    message = email.message.EmailMessage()

    message["From"] = sender
    message["To"] = recipient
    message["Subject"] = subject
    message.set_content(body)

    # Process the attachment and add it to the email

    attachment_filename = os.path.basename(attachment_path)
```

```

mime_type, _ = mimetypes.guess_type(attachment_path)
mime_type, mime_subtype = mime_type.split('/', 1)

with open(attachment_path, 'rb') as ap:
    message.add_attachment(ap.read(),
                           maintype=mime_type,
                           subtype=mime_subtype,
                           filename=attachment_filename)

return message

def generate_error_email(sender, recipient, subject, body):
    """Creates an email without an attachment."""
    # Basic Email formatting
    message = email.message.EmailMessage()
    message["From"] = sender
    message["To"] = recipient
    message["Subject"] = subject
    message.set_content(body)

    return message

```

Once you have finished editing the **emails.py** script, save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Now, open the **report_email.py** script using the **nano** editor:

```
nano ~/report_email.py
```

Once you define the **generate_email** and **send_email** methods, call the methods under the main method after creating the PDF report:

```
if __name__ == "__main__":
```

Use the following details to pass the parameters to **emails.generate_email()**:

- **From:** automation@example.com
- **To:** student@example.com
- **Subject line:** Upload Completed - Online Fruit Store
- **E-mail Body:** All fruits are uploaded to our website successfully. A detailed list is attached to this email.
- **Attachment:** Attach the path to the file **processed.pdf**

Once you have finished editing the **report_email.py** script, save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Grant executable permissions to the script **report_email.py**.

```
sudo chmod +x ~/report_email.py
```

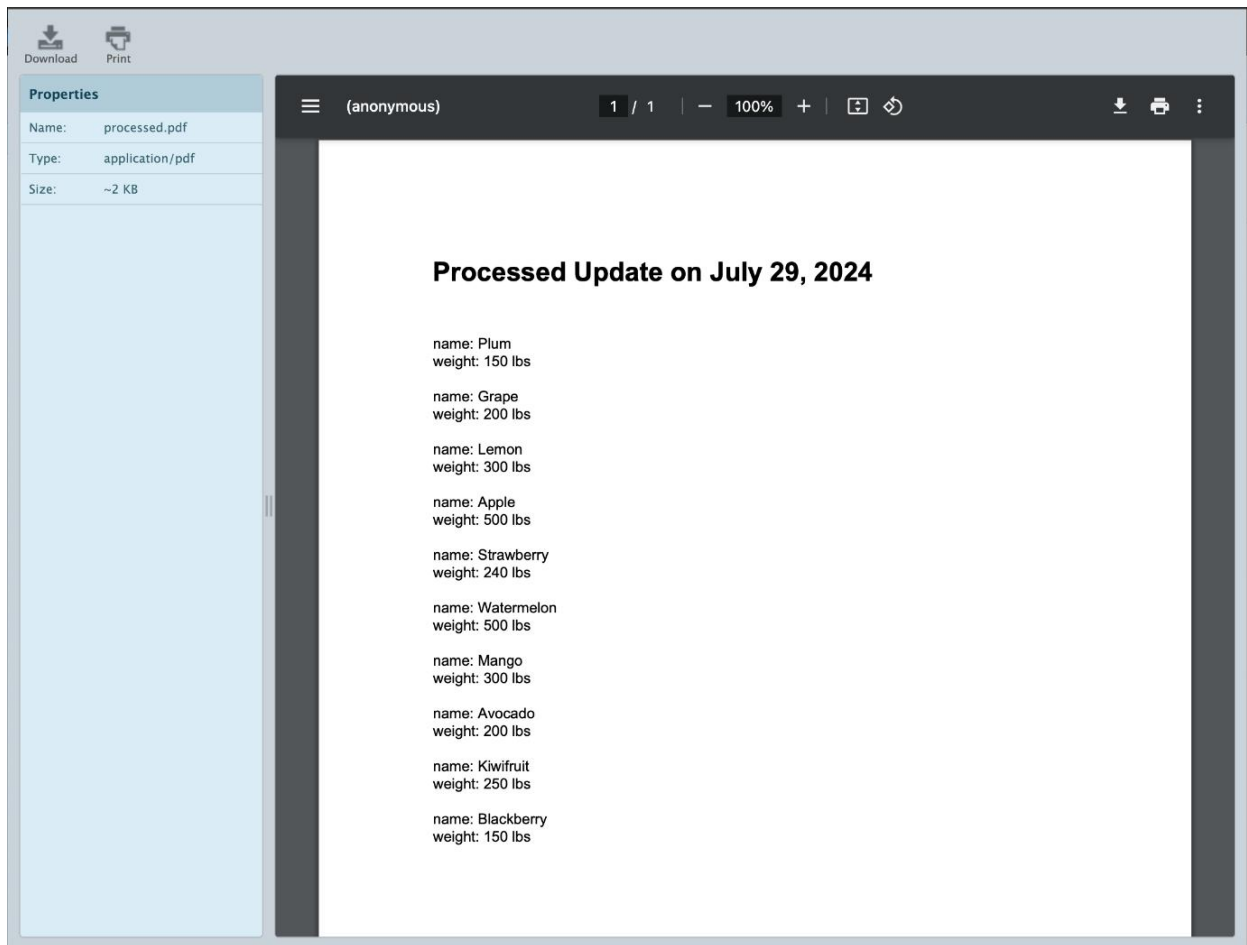
Run the **report_email.py** script.

```
./report_email.py
```

Now, check the webmail by visiting **[external-IP-address]/webmail**. Here, you'll need a login to **roundcube** using the *student* as **username** and *password* for the **password**, followed by clicking **Login**.

Now you should be able to see your inbox, with one unread email. Open the mail by double clicking on it. There should be a report in PDF format attached to the mail. View the report by opening it.

Output:



Health check

This is the last part of the lab, where you will have to write a Python script named **health_check.py** that will run in the background monitoring some of your system statistics: CPU usage, disk space, available memory and name resolution. Moreover, this Python script should send an email if there are problems, such as:

- Report an error if CPU usage is over 80%
- Report an error if available disk space is lower than 20%
- Report an error if available memory is less than 100MB
- Report an error if the hostname "localhost" cannot be resolved to "127.0.0.1"

Create a python script named **health_check.py** using the **nano** editor:

```
nano ~/health_check.py
```

Add a shebang line.

```
#!/usr/bin/env python3
```

Import the necessary Python libraries (eg. `shutil`, `psutil`) to write this script.

Complete the script to check the system statistics every 60 seconds, and in event of any issues detected among the ones mentioned above, an email should be sent with the following content:

- **From:** automation@example.com
- **To:** student@example.com
- **Subject line:**

Case	Subject line
CPU usage is over 80%	Error - CPU usage is over 80%
Available disk space is lower than 20%	Error - Available disk space is less than 20%
available memory is less than 100MB	Error - Available memory is less than 100MB
hostname "localhost" cannot be resolved to "127.0.0.1"	Error - localhost cannot be resolved to 127.0.0.1

- **E-mail Body:** Please check your system and resolve the issue as soon as possible.

Note: There is no attachment file here, so you must be careful while defining the **generate_email()** method in the **emails.py** script or you can create a separate **generate_error_report()** method for handling non-attachment email.

Python script:

```
#!/usr/bin/env python3

import shutil
import psutil
import socket
import emails
import os

sender = "automation@example.com"

receiver = "<USERNAME>@example.com".format(os.environ.get('USER'))

body = "Please check your system and resolve the issue as soon as possible."

# Checks disk usage and sends email if available space < 20%
du = shutil.disk_usage("/")
du_prsnt = du.free/du.total * 100
```

```
if du_prsnt < 20:

    subject = "Error - Available disk space is less than 20%"

    message = emails.generate_error_email(sender, receiver, subject, body)

    emails.send_email(message)

# Checks CPU usage and sends email if usage >80%
cpu_prsnt = psutil.cpu_percent(1)
if cpu_prsnt > 80:

    subject = "Error - CPU usage is over 80%"

    message = emails.generate_error_email(sender, receiver, subject, body)

    emails.send_email(message)

# Checks for available memory, if < 100mb sends an email
mem = psutil.virtual_memory()
trs = 100 * 1024 * 1024 # 100MB
if mem.available < trs:

    subject = "Error - Available memory is less than 100MB"

    message = emails.generate_error_email(sender, receiver, subject, body)

    emails.send_email(message)

# Checks hostname and if cannot be resolved to "127.0.0.1" sends an email
hostname = socket.gethostbyname('localhost')
if hostname != '127.0.0.1':

    subject = "Error - localhost cannot be resolved to 127.0.0.1"
```

```
message = emails.generate_error_email(sender, receiver, subject, body)

emails.send_email(message)
```

Once you have completed the **health_check.py** script. Save the file by typing **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Grant executable permissions to the script **health_check.py**.

```
sudo chmod +x ~/health_check.py
```

Run the file.

```
./health_check.py
```

Next, go to the webmail inbox and refresh it. There should only be an email something goes wrong, so hopefully you don't see a new email.

Output:

Subject	From	Date	Size		
• Upload Completed - Online Fruit Store	automation@example.com	Today 0015	4 KB		

To test out your script, you can install the **stress** tool.

```
sudo apt install stress
```

Next, call the tool using a good number of CPUs to fully load our CPU resources:

```
stress --cpu 8 &
```

To move ahead, hit **Enter** and allow the stress test to run in the background, as it will maximize our CPU utilization.

Now run **health_check.py** python script.

```
./health_check.py
```

Check your inbox for any new email.

Output:

Subject	From	Date	Size		
• Error - CPU usage is over 80%	automation@example.com	Today 0024	831 B		
• Upload Completed - Online Fruit Store	automation@example.com	Today 0015	4 KB		

Open the email with the subject **Error - CPU usage is over 80%** by double clicking it.



To list all running jobs, use the following command:

```
jobs -l
```

Output:

```
[1]+ 18120 Running                  stress --cpu 8 &
You have mail in /var/mail/student

Note down the process ID that is stress testing the CPU
i.e., stress --cpu 8 &, in your local test editor.

Then, terminate the stress testing process, using the following
command.

kill [process-id]
```

Replace **[process-id]** with the process ID you noted down earlier.

Now, you will be setting a cron job that executes the script **health_check.py** every 60 seconds and sends health status to the respective user.

To set a user cron job use the following command:

```
crontab -e
```

Output:


```
GNU nano 3.2 /tmp/crontab.sKSfsV/crontab

# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
```

^G Get Help	^O Write Out	^W Where Is	^K Cut Text	F4 Justify	^C Cur Pos
^X Exit	^R Read File	^_ Replace	^U Uncut Text	^T To Spell	^_ Go To Line

Now, set the complete path for **health_check.py** script, and save by clicking **Ctrl+o**, **Enter** key, and **Ctrl+x**.

Congratulations!

Congrats! You've successfully created a python script that processes images and descriptions and then updates your company's online website to add the new products. You have also generated a PDF report and sent it by email. Finally, you have also set up monitoring of the system's health.