Cloning the library

```
git clone https://github.com/Gael-Lejeune/opencve-docker
```

Installation

Prerequisite

- Install Docker : https://www.docker.com/products/docker-desktop

Installation steps

Create a copy of the opencve.cfg.example file

```
$ cd opencve-docker && cp ./conf/opencve.cfg.example ./conf/opencve.cfg
```

Edit the opencve.cfg file

```
server_name = <your_listening_ip>:8000
secret_key = <your_secret_key>
```

listening_ip can be setup to 127.0.0.1
secret_ket should be between quotes and should not contain the character %

Update the SMTP Configuration

For the moment, I used the outlook smtp server

```
[mail]
; Choices are 'smtp' or 'sendmail'
email_adapter = smtp

; The 'From' field of the sent emails
email_from = examplemail@outlook.com

; Configuration to set up SMTP mails.
smtp_server = smtp.office365.com
smtp_port = 587
smtp_use_tls = True
```

```
smtp_username = examplemail@outlook.com
smtp_password = examplepassword
```

Note that the email_from and smtp_username should be the same.

Build the OpenCVE image

```
$ docker-compose build
```

Initialize the database

```
$ docker exec -it webserver opencve upgrade-db
```

Import the data

```
$ docker exec -it webserver opencve import-data
```

Create an admin

```
$ docker exec -it webserver opencve create-user zied zied@example.com --admin
Password:
Repeat for confirmation:
[*] User zied created.
```

Start the beat

```
$ docker-compose up -d celery_beat
```

Check that everything is working fine

You can execute

```
$ docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS
PORTS                      NAMES
97e3ef4af44f   opencve:1.2.3   "./run.sh celery-beat"   20 seconds ago   Up 58
minutes                            celery_beat
faf7f59fff38   opencve:1.2.3   "./run.sh celery-wor…"   16 hours ago     Up 58
minutes                            celery_worker
```

```
df0faac8526d   opencve:1.2.3   "./run.sh webserver …"   16 hours ago     Up 58
minutes   0.0.0.0:8000->8000/tcp     webserver
63b7e90d2cd7   redis:buster    "docker-entrypoint.s…"   46 hours ago     Up 58
minutes   127.0.0.1:6379->6379/tcp   redis
38af0f416957   postgres:11     "docker-entrypoint.s…"   46 hours ago     Up 58
minutes   127.0.0.1:5432->5432/tcp   postgres
```

If stauts is up, everything is working fine.

Understanding the code

Tree

```
├──conf
└──opencve
    ├──opencve
    │   ├──api
    │   ├──checks
    │   ├──commands
    │   ├──controllers
    │   ├──migrations
    │   ├──models
    │   ├──static
    │   ├──tasks
    │   ├──templates
    │   └──views
```

api

//TODO

Commands

This folder contains python scripts that are used to interact with the application from the command line.
An example is create_user.py that is used to create a user:

```
root@df0faac8526d:/app# opencve create-user doc doc@test.com
Password:
Repeat for confirmation:
[*] User doc created.
```

If you want to create a new script, it is as follows
example.py

```
import click
#Import what you need
@click.command()
@click.argument("arg1")
@click.argument("arg2")
@with_appcontext
def example(arg1,arg2):
    //DO SOMETHING
```

Then add it to the cli.py file

```
from opencve.commands.example import example
.
.
.
cli.add_command(example)
```

After building and running the project again, you can access the webserver using :

```
$ docker exec -it webserver bash
```

and run

```
$ opencve example arg1 arg2
```

Controllers

//TODO

Migrations

//TODO

Models

Models are using to create a table in the database. We can take the client.py as an example:

```
from opencve.extensions import db
from opencve.models import BaseModel,
clients_vendors,clients_products,users_clients
class Client(BaseModel):
    __tablename__ = "clients"

    name = db.Column(db.String(), nullable=False, unique=True)
```

```
    # Relationships
    vendors = db.relationship("Vendor", secondary=clients_vendors)
    products = db.relationship("Product", secondary=clients_products)
    users = db.relationship("User", secondary=users_clients)
```

In this example, the new table Client will inherit the BaseModel columns and will add the columns name, vendors, products and users.
As you can see, the last three columns are in a relationship with other table and we can assure that using:

```
from opencve.models import clients_vendors
vendors = db.relationship("Vendor", secondary=clients_vendors)
```

in **init**.py we have:

```
clients_vendors = db.Table(
    "clients_vendors",
    db.Column(
        "client_id", UUIDType(binary=False), db.ForeignKey("clients.id"),
primary_key=True
    ),
    db.Column(
        "vendor_id",
        UUIDType(binary=False),
        db.ForeignKey("vendors.id"),
        primary_key=True,
    ),
)
```

which is used to link client vendors column with the vendors table.

Tasks

The task folder contains 3 main scripts , events.py, alerts.py and reports.py. I will explain one by one:

events.py

The script will check the nist database to see if there are any new CVEs. If yes, they will be downloaded and added to that database and an event will be created.

alerts.py

The script will check if there are any new events, if yes it will create alerts for the concerned users.

alerts.py

The script will check if there are any new alerts, if yes it will send reports to the concerned users (via the platforme and emails if smtp is configured)

Templates

Contains the HTML pages, it uses jinja to interact with the views controllers.

views

Used to control the templates and provide them with data This is an example:

```
from flask import request, render_template
from opencve.controllers.main import main
from opencve.controllers.clients import ClientController #We import the client
controller
from opencve.utils import get_clients_letters
@main.route("/clients") #The url of the page
def clients():
    clients, _, pagination = ClientController.list(request.args)
    return render_template(
        "clients.html", #The path to the template
        clients=clients, # Arguments we can send to the HTML page
        letters=get_clients_letters(),
        pagination=pagination,
    )
```

In the clients.html, we will be able to access the clients variable using jinja:

```
{% for client in clients.items %}
.
.
.
.
{% endfor %}
```

New added features

Client model

```
from opencve.context import _humanize_filter
from opencve.extensions import db
from opencve.models import BaseModel,
clients_vendors,clients_products,users_clients


class Client(BaseModel):
    __tablename__ = "clients"

    name = db.Column(db.String(), nullable=False, unique=True)
```

```
    # Relationships
    vendors = db.relationship("Vendor", secondary=clients_vendors)
    products = db.relationship("Product", secondary=clients_products)
    users = db.relationship("User", secondary=users_clients)
```

In addition, i needed to create clients_vendors, clients_products and users_clients in order to assure the relationship between the client table and vendors, products and users table respectively. We discussed those relationships before and how to create them in the **init**.py file.

In admin.py, I needed to add the ClientModelView as below :

```
from opencve.models.clients import Client
class ClientModelView(AuthModelView):
    page_size = 20
    create_modal = False
    edit_modal = False
    can_view_details = True
    column_list = ["name", "created_at"]
```

Then build and run the docker containers and run:

```
docker exec -it webserver opencve upgrade-db
```

Command to create a client

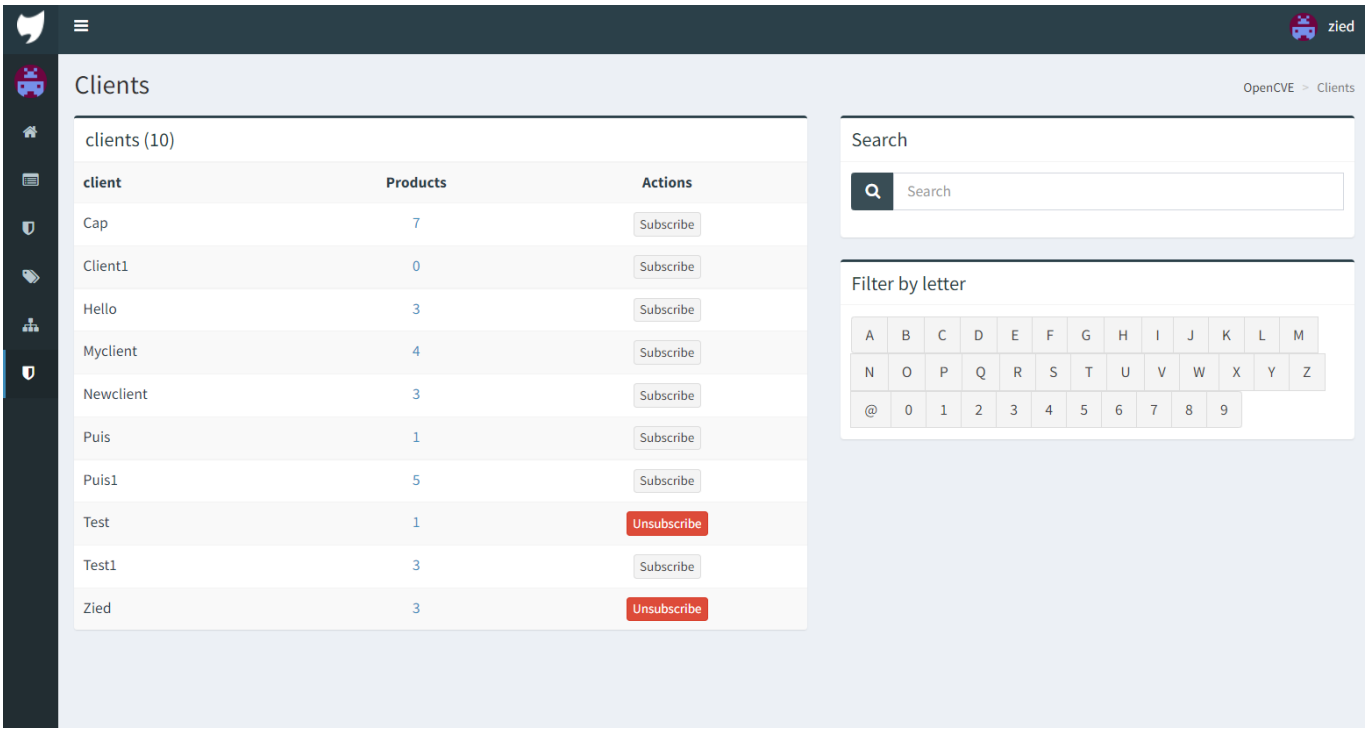
The create_client.py script is used to create a client, it can be used as follows:

```
$ docker exec -it webserver opencve create-client <client-name> <existing-user-
username>
```

The user will be able to manage the client subscribtions.: The user should be created beforehand. A user can only manage one client.

A view to see all the existing clients

The page templates/clients.html alongside views/client.py and controllers/subscriptions.py are used to view the client table and manage subscribtion for users.

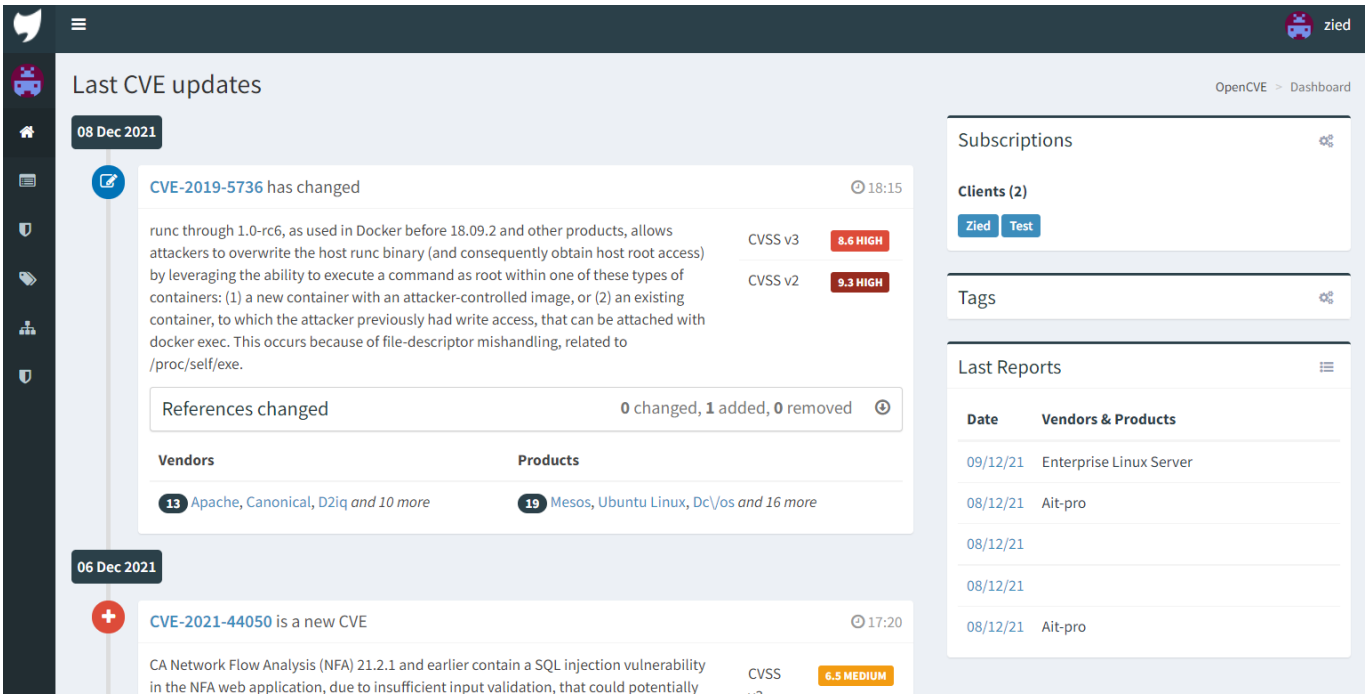


It is used exactly like the vendors page but to manage clients.

Update Dashboard

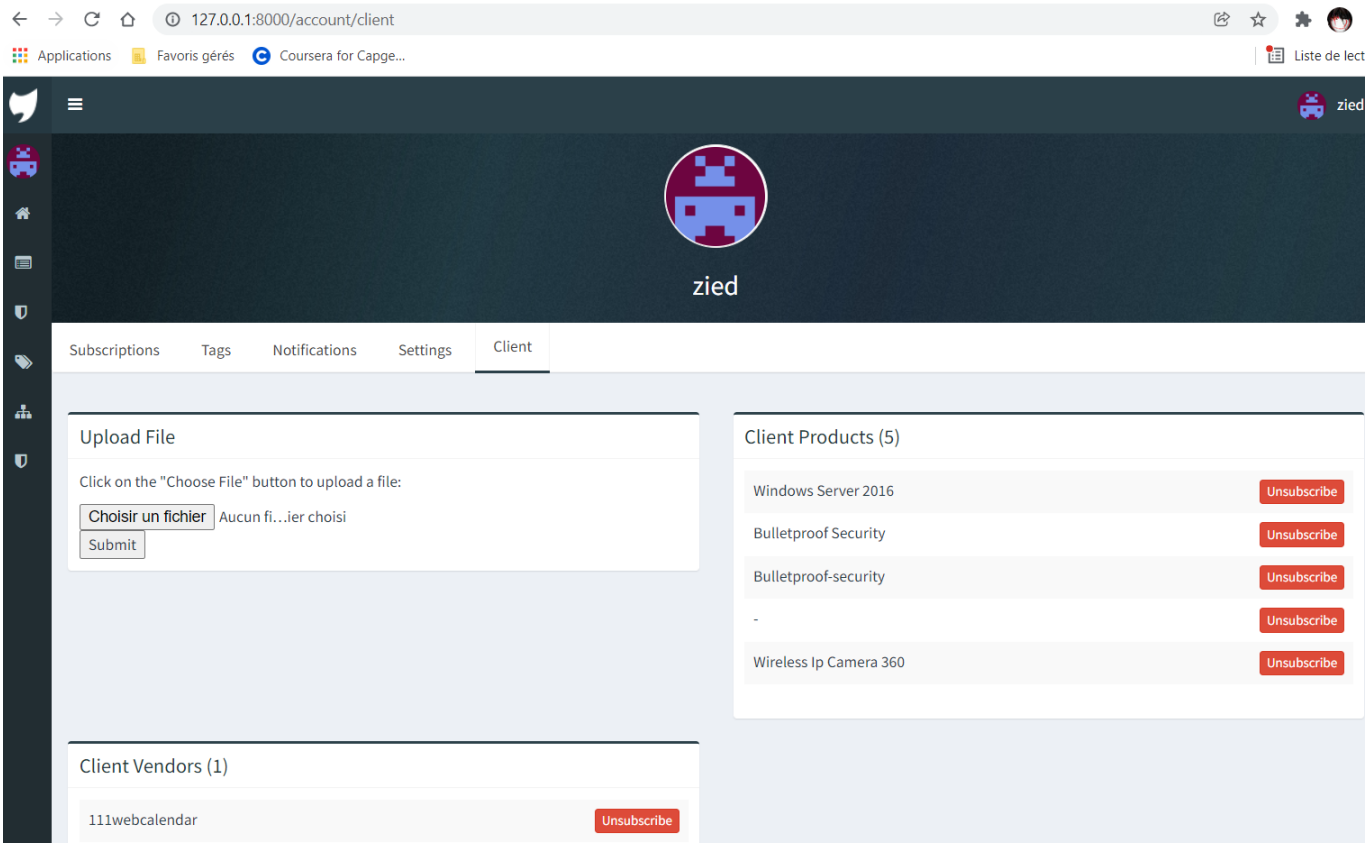
Updated dashboard to be able to see the CVEs related to a client.
This was done by updating the template/home.html page alongside controllers/home.py.
The main code was to check the current user clients and add their respective vendors and products to the vendors list.

```
    for i in current_user.clients:
        req=Client.query.filter_by(name=i.name).first().products
        vendors.extend([f"{p.vendor.name}{PRODUCT_SEPARATOR}{p.name}" for p in
req])
```



Add client section in the user profile

If a user is a client manager, he is now able to see and update the client subscribtions manually (we will see that next) or using an excel file.
We can do that by navigating to /account/client.



Using an excel file (.xlsx), the client manager can update the client products. The excel file should be as follow:

A	B	C	D	E	F	G
		OS			STATUS	
Name	Domain	Vendor	Product	OS Version	Comment	OpenCVE tag
AZRSEA1LZBXPRXP1	PRD	RedHat	REHL (64 bit)	Red Hat Enterprise Linux Server release 7.6 (Maipo)	Obsolete	enterprise_linux_server
AZRSEAWDC01	PRD	Microsoft	Windows Server (64 bit)	Windows Server 2016 Datacenter	Maintained	windows_server_2016

where the vendor and opencve tag should be existing in the opencve database. We used this function to upload the file and read its content:

```
from opencve.controllers.clients import read_excel
@main.route("/account/client", methods=['GET', 'POST'])
@login_required
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        # If the user does not select a file, the browser submits an
        # empty file without a filename.
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if not allowed_file(file.filename):
            flash('File Format Not Allowed')
            return redirect(request.url)
        if file:
            extensions = secure_filename(file.filename).split(".")[1]
            filename = str(uuid.uuid4())+"."+str(extensions)
            app.config['UPLOAD_FOLDER']=UPLOAD_FOLDER
            path_to_file=os.path.join(app.config['UPLOAD_FOLDER'], filename)
            file.save(path_to_file)
            read_excel(current_user.isclient,path_to_file)
            flash(f'File Uploaded and products added to client
{current_user.isclient}')
            return redirect(request.url)
    return redirect(request.url)
```

The read_excel is the function that will read the excel input and populate the database, it can be found at controllers/clients.py

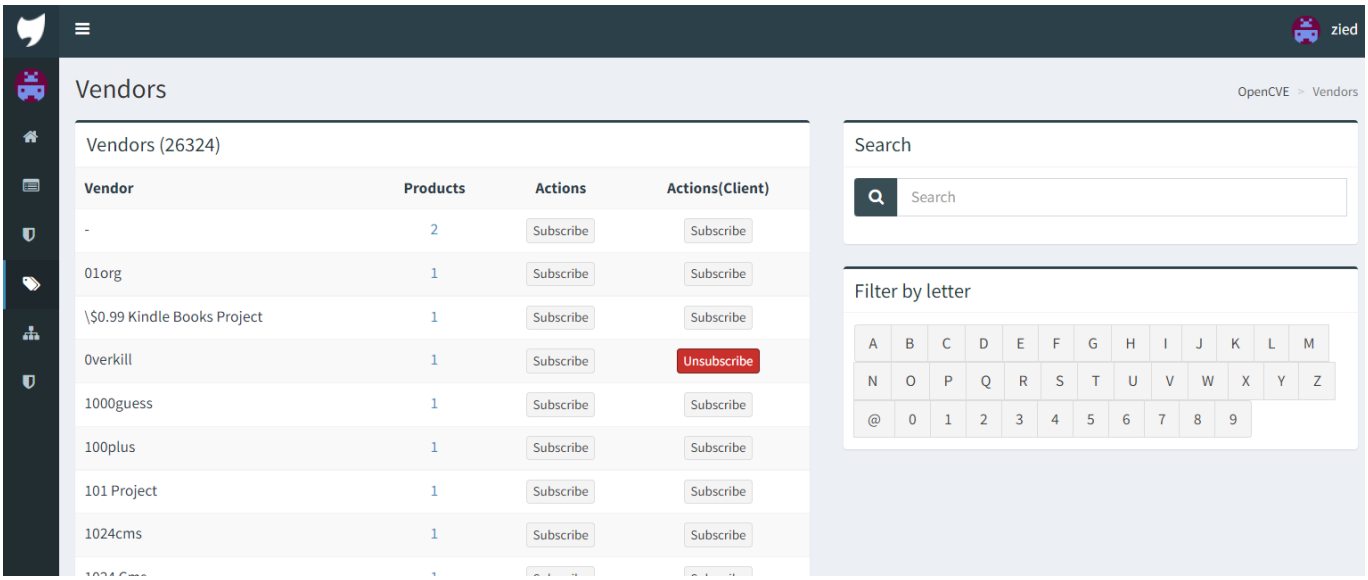
```
def read_excel(client,file_path):
    xlsx_file = Path(file_path)
    wb_obj = openpyxl.load_workbook(xlsx_file)
    sheet = wb_obj.active
    l = sheet.max_row
    data=[]
    for i in range(3,l):
        if sheet["G"+str(i)].value!=None:

d=str(sheet["C"+str(i)].value).lower()+":"+str(sheet["G"+str(i)].value).lower()
            if d not in data:
                data.append(d)
    for i in data:
        add_product(client,i.split(":")[0],i.split(":")[1])
```

For the moment, the function is not really optimized and can only use .xlsx files.

Add client Action in vendor and products

If the current user is a client manager, he will be able to add new vendors and products to the client subscribtions directly from the vendors or products pages:



We check the current_user.isclient to see if he is a client manager or not

```
{% if current_user.isclient %}
<th class="text-center">Actions(Client)</th>
{% endif %}
```

else the column Actions(Client) won't be visible to the user.

The subscriptions are managed using the controllers/subscriptions.py script, the script below is how we add a vendor to a client

```
    #client+vendor
    if request.form["obj"] == "vendorclient":
        current_client=Client.query.filter_by(name=current_user.isclient).first()
        if not is_valid_uuid(request.form["id"]):
            return _bad_request(request.form["obj"], request.form["id"])
        vendor = Vendor.query.get(request.form["id"])
        if not vendor:
            return _bad_request(request.form["obj"], request.form["id"])

        # Subscribe
        if request.form["action"] == "subscribe":
            if vendor not in current_client.vendors:
                current_client.vendors.append(vendor)
                db.session.commit()
```

```
            return json.dumps({"status": "ok", "message": "vendor added"})

        # Unsubscribe
        if request.form["action"] == "unsubscribe":
            if vendor in current_client.vendors:
                current_client.vendors.remove(vendor)
                db.session.commit()

            return json.dumps({"status": "ok", "message": "vendor removed"})
```

and we use this function from the templates/vendors.html in this way:

```
{% if vendor in current_client.vendors %}
 <button class="btn btn-danger btn-xs subscribe"
    id="unsubscribe_vendorclient_{{ vendor.id }}" type="button">Unsubscribe
</button>
{% else %}
<button class="btn btn-default btn-xs subscribe"
    id="subscribe_vendorclient_{{ vendor.id }}" type="button">Subscribe
</button>
{% endif %}
```

Managing reports and emails

In order to create alerts for users subscribed to a client i edited the tasks/alerts.py script

```
 # Product contains the separator
            if PRODUCT_SEPARATOR in v:
                vendor = Vendor.query.filter_by(
                    name=v.split(PRODUCT_SEPARATOR)[0]
                ).first()
                product = Product.query.filter_by(
                    name=v.split(PRODUCT_SEPARATOR)[1], vendor_id=vendor.id
                ).first()
                clients = Client.query.filter(
                    Client.products.contains(product)
                    ).all()
                for user in product.users:
                    if user not in users.keys():
                        users[user] = {"products": [], "vendors": []}
                    users[user]["products"].append(product.name)
                for client in clients:
                    for user in client.users:
                        if user not in users.keys():
                            users[user] = {"products": [], "vendors": []}
                        users[user]["products"].append(product.name)

            # Vendor
            else:
```

```
                    vendor = Vendor.query.filter_by(name=v).first()
                    clients = Client.query.filter(
                        Client.vendors.contains(vendor)
                        ).all()
                    for user in vendor.users:
                        if user not in users.keys():
                            users[user] = {"products": [], "vendors": []}
                        users[user]["vendors"].append(vendor.name)
                    for client in clients:
                        for user in client.users:
                            if user not in users.keys():
                                users[user] = {"products": [], "vendors": []}
                            users[user]["vendors"].append(vendor.name)
```

We loop through the user clients and select the vendors and products from them.

This will create a list of users that will receive emails.

Encountred problems

- Understaing the code was challenging as I had no prior experience in web dev.
- When I create a new model or edit an existing one, the changes are not applied immediatly. We need to run the opencve upgrade-db command to do that... Sometimes this doesn't work either and I had to add the tables manually using SQL commands.
- Setting up the SMTP server was tricky, the problem was is that the "email_from" and "smtp_username" should be the same else it won't work. The logs didn't explain this and i had to do alot of testing to figure it out.
- Testing the emails functionality to send reports was tricky aswell.