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

Configuration

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, the outlook smtp server is used

```
[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.

Check files & Line endigs

Make sure that `./conf/opencve.cfg` and `./run.sh` are LF line terminated (and not CRLF, it could cause errors while building or running the container)

Building & Cleaning

```
make
```

This will **not** import the data in the database. Check Import data for more informations on that part.

```
make clean
```

Please note that this command will prune your docker volumes, thus deleting any that is not currently used.

More details about the build

Build the OpenCVE image

```
make build
```

Alternatively, you could use :

```
docker compose build
```

Create the container and start them

```
make up
```

Alternatively, you could use :

```
docker-compose up -d postgres redis webserver celery_worker celery_beat
```

N.B. : This will run `make build`

Initialize & upgrade the database

```
make upgrade
```

Alternatively, you could use :

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

N.B. : This will run `make build` *and* `make up`

Import data

```
make import
```

Alternatively, you could use :

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

You can also use the following command to import less CVEs, making it a bit faster and more suitable for testing.

```
make import-light
```

Alternatively, you could use :

```
docker exec -it webserver opencve import-light
```

Create an admin

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

Create and start the beat

```
make beat
```

Alternatively, you could use :

```
docker-compose up -d postgres redis webserver celery_worker 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 status is up everywhere, 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
```

Alternatively, you could use :

```
docker exec -it webserver opencve example
```

Controllers

Contains some backend logic and models controllers.

Migrations

Contains the migrations files generated when building the project. Those can mostly be ignored as they are automatically generated.

Models

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

```
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)


    @property
    def human_name(self):
        return _humanize_filter(self.name)

    def __repr__(self):
        return "<Client {}>".format(self.name)
```

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.

reports.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 to be rendered. 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 %}
```

Features that are specific to this fork

Please not that this section is about features that are currently in progress or likely to evolve. This section may not be up to date.

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)


    @property
    def human_name(self):
        return _humanize_filter(self.name)

    def __repr__(self):
        return "<Client {}>".format(self.name)
```

In addition, we 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"]
```

Client tables are now added automatically when building the project.

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>
```

The user is able to manage the client subscribtions : The user should be created beforehand.

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.

A view to see the details of a client and modify it

The user 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 "/client/\".

Using an excel file (.xlsx), the user can update the client products. The excel file format is described on the web page. We've implemented a searching algorithm that will look for the product in the database if the tag is not precised. If a tag is precised, it has to be in the database.

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

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

Updated Dashboard

Updated dashboard to be able to see the CVEs related to a followed 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.

Added client Action in vendor and products

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

Managing reports and emails

In order to create alerts for users subscribed to a client we 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

- When you want to upgrade the DB, you will sadly have to make clean and thus re-import the data. The "import-light" function may be optimized to make it even faster and thus help with this problem.
- Setting up the SMTP server is tricky, the problem 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.
- Running the default present tests is still a challenge
- Please use info() or logger.info() to debug or display logs as print() may not display correctly