OC Inventory

Solution d'inventaire de parc informatique

Dossier d'exploitation

Version 1.0

Auteur
David Bouzerar
Développeur

IT Consulting & Development

TABLE DES MATIERES

1 - VERSIONS

Auteur	Date	Description	Version
David Bouzerar	27/03/2023	Création du document	0.1
David Bouzerar	02/04/2023	Ajout des premiers chapitres	0.3
David Bouzerar	04/04/2023	Rédaction du document	0.6
David Bouzerar	20/04/2023	Intégration de CircleCI	0.8
David Bouzerar	20/04/2023	Ajout des prérequis	0.8
David Bouzerar	20/04/2023	Ajout de la procédure de déploiement	0.8
David Bouzerar	20/04/2023	Finalisation de la procédure de déploiement	0.8
David Bouzerar	28/04/2023	Finalisation du document	1.0

2 - Introduction

2.1 - Objet du document

Le présent document constitue le dossier d'exploitation de l'application OC Inventory.

L'objectif de ce document est d'apporter les informations nécessaires à la bonne exploitation du système et d'assurer un support si un problème technique intervient.

2.2 - Références

Pour de plus amples informations, se référer :

1. **DCF - 01** : Dossier de conception fonctionnelle
2. **DCT - 02** : Dossier de conception technique de l'application

3 - PREREQUIS

3.1 - Système

3.1.1 - Serveur de Base de données

La base de données est hébergée sur un serveur virtuel dédié (souscrit chez **DigitalOcean**). Le **SGBDR** installé est **PostgreSQL 13**.

3.1.1.1 - Caractéristiques techniques

Le serveur dispose d'un OS **Linux** dépourvu d'interface graphique. Dans un premier temps, nous utiliserons un seul CPU, 1 Go de Ram et 25 Go de stockage et ajusterons, si cela s'avère utile, en fonction des retours de l'interface de surveillance de **DigitalOcean**.

Ubuntu Server 22.04 (basée sur **Debian**) est la distribution installée. Il s'agit d'une version **LTS** dont la fin du support est programmée en Avril 2027.

3.1.2 - Serveur d'application

L'application est hébergée sur un serveur virtuel dédié (souscrit chez **DigitalOcean**). **Python version 3.10** et **Django version 3.2.5** ainsi que les dépendances y sont installées. Nous y retrouvons également un serveur WEB **NGINX 1.18** ainsi que les clients **Sentry** et **Newrelic** nécessaires à la surveillance du serveur et de l'application.

3.1.2.1 - Caractéristiques techniques

Le serveur dispose d'un OS **Linux** sans interface graphique. Dans un premier temps, nous utiliserons un seul CPU, 1 Go de Ram et 25 Go de stockage. S'il s'avère que les ressources soient trop limitées, nous pourrons les ajuster en fonction des retours obtenus sur **NewRelic** et l'interface de surveillance de **DigitalOcean**.

Ubuntu Server 22.04 (basée sur **Debian**) est la distribution installée. Il s'agit d'une version **LTS** dont la fin du support prendra fin en Avril 2027.

3.2 - Bases de données

Le **modèle physique de données** est visible dans le dossier de conception technique de l'application, ainsi que dans la partie « ressources » du **repository**.

Les modèles sont créés à partir des informations fournies par ce document. Si des modifications sont effectuées sur la base, il est recommandé de maintenir à jour ce document.

3.3 - API

3.3.1 - OCS Inventory

L'**API OCS Inventory** est utilisée régulièrement lors de l'installation de nouveaux périphériques sur le parc informatique. Elle fournit notamment des informations techniques et d'identification sur les périphériques.

3.3.2 - Immo

La base **Immo** fournit à l'application des informations « administratives » concernant les périphériques. Elle permet notamment d'indiquer la date d'achat, le numéro d'inventaire, le numéro de bon de commande, etc... Celle-ci permet également de faire le lien vers les utilisateurs des périphériques.

4 - PROCEDURE DE DEPLOIEMENT

4.1 - Déploiement

4.1.1 - Paramétrer l'accès aux serveurs

Comme expliqué précédemment, l'application est hébergée chez **DigitalOcean**. Si, pour une raison quelconque, il est nécessaire de changer d'hébergeur, il faudra donc souscrire à une des offres proposées puis choisir une distribution **Linux** qui sera installée sur la machine virtuelle. Concernant le choix de l'offre il faudra l'adapter selon les ressources nécessaires, à savoir le nombre de **CPU**, la quantité de **RAM** ainsi que le stockage.

Une fois cela fait, il est nécessaire de permettre la connexion **SSH** sur le serveur. Pour cela, ouvrir le port **TCP 22** nécessaire dans le cadre d'une connexion **SSH**. Il est également nécessaire d'autoriser le trafic sur les ports **TCP 80** (**HTTP**) et **TCP 443** (**HTTPS**).

A partir de là, se connecter au serveur grâce à la commande :

```
ssh root@(IP_DU_SERVEUR)
...
```

Puis ajouter un nouvel utilisateur pour autant que nécessaire :

```
root@(SERVER_NAME):~# adduser (USERNAME)
Enter new UNIX password :
Retype new UNIX password :
...
```

Autoriser ce nouvel utilisateur à exécuter la commande « *sudo* » :

```
root@(SERVER_NAME):~# usermod -aG sudo (USERNAME)
...
```

Basculer sur l'utilisateur nouvellement créé :

```
root@(SERVER_NAME):~# su - (USERNAME)
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

(USERNAME)@(SERVER_NAME):~$
...
```

Créer un fichier « *authorized_keys* » dans le répertoire « *.ssh* » et y coller la clé publique de l'utilisateur de la session. Si le répertoire n'existe pas, le créer et modifier les permissions :

```
root@(SERVER_NAME):~# mkdir .ssh
root@(SERVER_NAME):~# chmod 700 .ssh
...
```

Pour des raisons de sécurité, il est recommandé, une fois les utilisateurs créés, d'empêcher la connexion **SSH** au serveur par l'intermédiaire du compte **Root**. Pour cela, éditer le fichier de configuration :

```
root@(SERVER_NAME):~# vi /etc/ssh/sshd_config
...
```

Puis modifier la valeur de « *PermitRootLogin* » sur « *no* » :

```
PermitRootLogin no
```

Vous pouvez également modifier la valeur de « *PasswordAuthentication* » sur « *no* » :

```
PasswordAuthentication no
```

➔ (Avant de modifier cette valeur, s'assurer que la connexion SSH est possible)

Et recharger le « *process* » **SSH** pour appliquer les modifications :

```
root@(SERVER_NAME):~# service ssh reload
...
```

Pour se reconnecter au serveur, il faudra maintenant saisir le **login** d'un utilisateur :

```
ssh (USERNAME)@(IP_DU_SERVEUR)
...
```

4.1.2 - Uploader l'application sur le serveur

4.1.2.1 - Installation des dépendances utiles

Avant toute chose, nous devons nous assurer que le serveur dispose des dernières mises à jour et le cas échéant, les installer :

```
(USERNAME)@(SERVER_NAME):~/$ sudo apt update && sudo apt full-upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
```

Puis procéder à l'installation des dépendances :

```
(USERNAME)@(SERVER_NAME):~/$ sudo apt install python3-pip python3-dev libpq-dev postgresql
postgresql-contrib
...
```

4.1.2.2 - Upload

Il suffit pour cela de cloner le **repository Git** de l'application :

```
(USERNAME)@(SERVER_NAME):~/$ git clone https://github.com/Eidocode/OC_Project13.git
Cloning into 'OC_Project13'...
remote: Enumerating objects: 1306, done.
remote: Counting objects: 100% (182/182), done.
remote: Compressing objects: 100% (118/118), done.
remote: Total 1306 (delta 93), reused 123 (delta 49), pack-reused 1124
Receiving objects: 100% (1306/1306), 939.82 KiB | 20.00 MiB/s, done.
Resolving deltas: 100% (767/767), done.
...
```

4.1.3 - Environnement virtuel

4.1.3.1 - Création de l'environnement

Mettre en place l'environnement virtuel, pour cela, installer « **pip** » et « **venv** »

```
(USERNAME)@(SERVER_NAME):~/$ sudo apt install python3-venv
...
```

Puis créer le nouvel environnement virtuel :

```
(USERNAME)@(SERVER_NAME):~/$ python3 -m venv project_env
...
```

Il ne reste plus qu'à l'activer :

```
(USERNAME)@(SERVER_NAME):~/$ source project_env/bin/activate
(project_env) (USERNAME)@(SERVER_NAME):~/$
...
```

4.1.3.2 - Installation des dépendances de l'application

```
(project_env) (USERNAME)@(SERVER_NAME):~/OC_Project13$ pip install -r requirements.txt
...
```

4.1.4 - Base de données

4.1.4.1 - Création de la base

Concernant la création de la base de données, il est nécessaire de préalablement faire quelques ajustements. Nous devons dans un premier temps créer la base, puis un utilisateur à qui l'on attribuera des droits sur la base.

Pour cela, il est nécessaire d'accéder à la console **PSQL** en procédant de la façon suivante :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo -u postgres psql
psql (14.7 (Ubuntu 14.7-0ubuntu0.22.04.1))
Type "help" for help.

postgres=#
...
```

Une fois connecté sur la console **PSQL**, nous pouvons créer la base de données, ainsi que le nouvel utilisateur :

```
postgres=# CREATE DATABASE oc_inventory;
CREATE DATABASE

postgres=# CREATE USER (username) WITH PASSWORD '(password)';
CREATE ROLE
...
```

La base et l'utilisateur sont maintenant créés. Il est toutefois nécessaire, d'après la documentation officielle de Django, de modifier certains paramètres de la base de données, dans le but d'améliorer les performances lors du traitement des requêtes :

```
postgres=# ALTER ROLE (username) SET client_encoding TO 'utf8';
ALTER ROLE

postgres=# ALTER ROLE (username) SET default_transaction_isolation TO 'read committed';
ALTER ROLE

postgres=# ALTER ROLE (username) SET timezone TO 'Europe/Paris';
ALTER ROLE
...
```

Enfin, il ne reste plus qu'à attribuer tous les droits sur la base à notre nouvel utilisateur. Nous pourrons ainsi l'utiliser pour lire, écrire et supprimer des données.

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE oc_inventory TO (username);
GRANT
...
```

Il ne reste plus qu'à quitter la console **PSQL** :

```
postgres=# \q

(project_env) (USERNAME)@(SERVER_NAME):~/
...
```

4.1.4.2 - Fichier de configuration

Il est nécessaire de renseigner le fichier de configuration de l'application afin qu'elle puisse accéder à la base de données nouvellement créée. Pour cela, nous devons éditer le fichier « *production.py* » situé dans le dossier « *settings* » de l'application :

```
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': const.MAIN_DB_NAME,
        'USER': const.USER_ID,
        'PASSWORD': const.USER_PWD,
        'HOST': const.SRV_IP,
        'PORT': 'const.PSQL_PORT,
    }
}
...
```

4.1.4.3 - Migrations

Comme expliqué précédemment, la configuration de l'application fonctionne avec l'utilisation de variables d'environnement. C'est également le cas pour la partie « *DATABASES* » des settings. Comment l'indique les valeurs, la récupération des variables d'environnement se fait dans le fichier « *Tools/const.py* ». Il est donc indispensable de déclarer ces variables de la manière suivante (à effectuer pour les variables nécessaires à l'utilisation de la base) :

```
(…) (USERNAME)@(SERVER_NAME):~/ export DJANGO_SETTINGS_MODULE=inventory.settings.production
(…) (USERNAME)@(SERVER_NAME):~/ export PSQL_SRV='localhost'
...
```

La base étant créée et les variables déclarées, il est maintenant nécessaire d'effectuer les migrations :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ python3 manage.py makemigrations
Migrations for 'auth':
  /home/adm-redn2k4/project_env/lib/python3.10/site-packages/(…)
    - Alter field email on user
Migrations for 'product':
...
```

```
(project_env) (USERNAME)@(SERVER_NAME):~/ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, ...
Running migrations:
...
```

Cela aura pour effet de créer les tables, contraintes etc… de notre base de données. Il est possible d'importer directement des données compatibles, pour cela, partons du principe que nous possédons un fichier nommé « *store.json* » :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ python3 manage.py loaddata ./dumps/store.json
Installed 476 object(s) from 1 fixture(s)
...
```

4.1.5 - Gestion des statiques

Pour générer les fichiers statiques, il est nécessaire d'indiquer à **Django** de le faire. Pour cela, le fichier de configuration contient une constante nommée « *STATIC_ROOT* » nécessaire à la collecte de ces fichiers statiques. Cette constante existe uniquement si la variable d'environnement « *ENV* » existe et qu'elle possède la valeur « *PRODUCTION* ».

Pour le moment cette variable n'existe pas, il est donc nécessaire de la créer :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ export ENV=PRODUCTION
...
```

Dans ce cas, la variable d'environnement n'existera que dans l'instance « *SSH* » en cours. Elle est volatile, mais cela suffira pour générer les statiques, nous verrons plus loin comment la rendre persistante.

Nginx doit avoir accès en lecture aux dossiers contenant les *staticfiles*. Il est donc préférable de les positionner dans un dossier qui soit accessible à l'utilisateur « *www-data* ». Le chemin */var/www/html* semble être la destination idéale. Il faut donc pour cela, modifier le chemin de **STATIC_ROOT** dans le fichier « *settings.py* » de l'application :

```
...
STATIC_ROOT = '/var/www/html/oc_inventory/static'
STATIC_URL = '/static/'
...
```

Il faut maintenant créer les répertoires dans */var/www/html* et donner les droits à l'utilisateur courant :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo mkdir -p /var/www/html/oc_inventory/static
...
(…) (USERNAME)@(SERVER_NAME):~/ sudo chown -R (user):(group) var/www/html/oc_inventory/static/
...
(…) (USERNAME)@(SERVER_NAME):~/ sudo chmod -R 755 /var/www/html/oc_inventory/static/
...
```

Reste maintenant à générer les fichiers statiques :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ python3 manage.py collectstatic
# ...
143 static files copied to '/var/www/html/oc_inventory/static'
...
```

Maintenant que les staticfiles sont collectés, il faut simplement redonner les droits sur le répertoire à l'utilisateur www-data. Pour cela, simplement lancer la commande :

```
(…) (USERNAME)@(SERVER_NAME):~/sudo chown -R www-data:www-data /var/www/html/oc_inventory/static/
...
```

L'application doit maintenant s'afficher correctement.

4.1.6 - Créez un administrateur

Il ne reste qu'à créer un utilisateur possédant les droits « *administrateur* » pour l'interface d'administration de l'application :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ python3 manage.py createsuperuser
...
```

4.2 - Configuration du serveur HTTP

4.2.1 - Installation NGINX

L'installation du serveur **HTTP NGINX** se fait facilement mais nécessite une petite configuration. Débutons par le téléchargement / installation de l'application :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo apt-get install nginx
...
```

Si l'installation s'est bien déroulée, nous devrions tomber sur la page « **Welcome to nginx !** » en saisissant l'adresse IP publique du serveur sur un navigateur WEB.

4.2.2 - Configuration NGINX

4.2.2.1 - Création du fichier de configuration

Les liens vers les fichiers de configuration sont regroupés dans le dossier « *sites-enabled* ». Il est donc nécessaire d'y indiquer le chemin vers le fichier de configuration de l'application :

Il suffit donc de créer le fichier de configuration puis de créer un lien symbolique vers celui-ci :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo touch /etc/nginx/sites-available/oc_inventory
...
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo ln –s /etc/nginx/sites-available/oc_inventory
/etc/nginx/sites-enabled
...
```

Une entrée vers le fichier de configuration doit apparaître maintenant dans « *sites-enabled* » :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ vi /etc/nginx/sites-enabled



"==============================================================================
" Netrw Directory Listing                                        (netrw v156)
"   /etc/nginx/sites-enabled
"   Sorted by      name
"   Sort sequence:
[\/]$,\<core\%(\.\d\+\)\=\>,\.h$,\.c$,\.cpp$,\~\=\*$,*,\.o$,\.obj$,\.info$,\.swp$,\.bak$,\~$
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:special
"==============================================================================
../
./
default@                          --> /etc/nginx/sites-available/default
oc_inventory@                         --> /etc/nginx/sites-available/oc_inventory
```

4.2.2.2 - Configuration

Il reste maintenant à configurer **NGINX** pour l'application *OC_Inventory*. **NGINX** permet également la gestion des fichiers statiques, il est donc utile de lui donner ce rôle.

Si un fichier statique est demandé, **NGINX** doit rediriger vers celui-ci sans passer par l'application, sinon, il redirigera le trafic vers l'application.

Ouvrons dans un premier temps le fichier de configuration nouvellement créé :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo vi /etc/nginx/sites-available/oc_inventory
...
```

Ci-dessous, la configuration que doit contenir ce fichier :

```
server {
    listen 80;
    server_name (SERVER_DOMAIN_NAME);
    root (APPLICATION_ABSOLUTE_PATH);

    location /static {
        alias (APPLICATION_ABSOLUTE_PATH)/static/;
    }

    location / {
        proxy_set_header Host $http_host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_redirect off;
        if (!-f $request_filename) {
            proxy_pass http://127.0.0.1:8000;
            break;
        }
    }
}
```

Voici à quoi doit ressembler le fichier de configuration **NGINX** pour l'application **OC_INVENTORY**. La partie « *SERVER_DOMAIN_NAME* » doit être remplacée par le nom de domaine préalablement souscrit. Si nous en sommes à un stade où nous n'avons pas encore souscrit à un nom de domaine, il est possible de le remplacer par l'adresse IP publique du serveur.

La partie « *APPLICATION_ABSOLUTE_PATH* » doit être remplacée par le chemin absolu de l'application sur le serveur.

Pour le reste, nous pouvons voir que le bloc « *location /static* » contient le chemin vers les fichiers statiques de l'application. **NGINX** s'appuiera donc là-dessus et sur le chemin de l'application pour la redirection.

Il reste alors à charger la nouvelle configuration, il faut pour cela relancer le serveur **NGINX** :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo service nginx reload
...
```

Il est également nécessaire de mettre à jour la partie « *ALLOWED_HOSTS* » du fichier de settings « *production.py* » de l'application :

```
...
ALLOWED_HOSTS = ['(SERVER_DOMAIN_NAME)']
...
```

Il faut donc saisir le nom de domaine de l'application ou l'IP publique si nous n'en avons pas pour le moment.

4.2.3 - Configuration de Supervisor & Gunicorn

4.2.3.1 - Installation de Supervisor

Comme expliqué précédemment, **Supervisor** est un gestionnaire permettant de démarrer des services et les surveiller. Dans le cas présent, il nous permet d'exécuter « **Gunicorn** » au démarrage du serveur puis de surveiller son fonctionnement.

L'installation se fait simplement par le gestionnaire de package **Ubuntu** :

```
(project_env) (USERNAME)@(SERVER_NAME):~/ sudo apt-get install supervisor
...
```

4.2.3.2 - Configuration de Supervisor

Supervisor s'appuie sur un fichier de configuration situé dans « */etc/supervisor/conf.d* ». Chaque fichier « *.conf* » représente un processus dont **Supervisor** doit s'occuper.

Il faut donc créer un fichier de configuration pour l'application :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo touch /etc/supervisor/conf.d/oc_inventory_gunicorn.conf
...
```

Puis, l'éditer :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo vi /etc/supervisor/conf.d/oc_inventory_gunicorn.conf
...
```

```
[program:oc_inventory]
command = (VENV_ABSOLUTE_PATH)/bin/newrelic-admin run-program (VENV_ABSOLUTE_PATH)/bin/gunicorn
oc_inventory_app.wsgi:application
user = (USERNAME)
directory = (APPLICATION_ABSOLUTE_PATH)
autostart = true
autorestart = true
environment = DJANGO_SETTINGS_MODULE='inventory.settings.production',SECRET_KEY="(SECRET_KEY)
",NEW_RELIC_CONFIG_FILE=(APPLICATION_ABSOLUTE_PATH)/newrelic.ini,PSQL_SRV=(IP_SRV),DB_
USER=(DB_USER),DB_PWD=(DB_PWD),EMAIL_HOST=(SMTP),EMAIL_HOST_USER=(EMAIL_USER),EMAIL_HOST_PWD=(EMA
IL_PWD)
```

La configuration proposée contient également les informations concernant la configuration de **New Relic**. Se reporter à la rubrique concernée pour le mettre en place.

Pour le reste, voici les champs à modifier :

- *APPLICATION_ABSOLUTE_PATH* : Chemin absolu de l'application sur le serveur
- *VENV_ABSOLUTE_PATH* : Chemin absolu de l'environnement virtuel
- *USERNAME* : Le nom de l'utilisateur chargé du lancement de l'application
- *SECRET_KEY* : Clé secrète de l'application utilisée par **Django**

A noter que nos variables d'environnements sont renseignées dans ce fichier de configuration.

NGINX étant absolument primordial, ajoutons également une surveillance sur celui-ci :

```
[program:nginx]
command = /usr/sbin/nginx –g "daemon off;"
autostart = true
autorestart = true
numprocs = 1
startsecs = 0
process_name=%(program_name)s_%(process_num)02d
stderr_logfile=/var/log/supervisor/%(program_name)s_stderr.log
stderr_logfile_maxbytes=10MB
stdout_logfile=/var/log/supervisor/%(program_name)s_stdout.log
stdout_logfile_maxbytes=10MB
```

Il ne reste plus qu'à demander à **Supervisor** de prendre en compte les nouveaux processus à surveiller :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl reread
nginx: available
oc_inventory: available
...
```

Puis lui demander de démarrer la supervision des nouveaux processus :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl update
nginx: added process group
oc_inventory: added process group
...
```

On s'assure enfin que **Supervisor** est actif et surveille bien les processus :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl status
nginx                            RUNNING   pid 18429, uptime 0:01:47
oc_inventory                     RUNNING   pid 16309, uptime 0:01:47
...
```

5 - SUPERVISION/MONITORING

5.1 - Sentry

L'installation de **Sentry**, qui pour rappel va nous permettre d'avoir un retour sur les erreurs et les exceptions rencontrées éventuellement rencontrées par l'application. Il nous servira également de gestionnaire des **logs** sur certains éléments suivis de l'application.

5.1.1 - Installation

SDK Sentry est un package **Python**. Il est donc normalement installé en même temps que les « *requirements* » de l'application. Si ce n'est pas le cas, l'installation se fait simplement par la commande suivante :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ pip install --upgrade sentry-sdk
...
```

5.1.2 - Configuration

La configuration de **Sentry** se fait par l'intermédiaire du fichier « *production.py* » dans les « *settings* » de l'application :

```
import logging
import sentry_sdk

from sentry_sdk.integrations.django import DjangoIntegration
from sentry_sdk.integrations.logging import LoggingIntegration

...
sentry_logging = LoggingIntegration(
    level=logging.INFO,
    event_level=logging.INFO
)

sentry_sdk.init(
    dsn="(LINK_WITH_SENTRY_ACCOUNT)",
    integrations=[DjangoIntegration(), sentry_logging],
    traces_sample_rate=1.0,
    send_default_pii=True
)
...
```

Les erreurs et exceptions seront maintenant automatiquement remontées vers le « *Dashboard* » **Sentry** de l'application. A noter que les **logs** affichés par la librairie « *logging* » seront également remontés et porteront le « *tag* » **INFO**.

5.2 - New Relic

Comme indiqué précédemment, **New Relic** permet de surveiller, entre autres, les performances de l'application.

5.2.1 - Installation

A la manière de **Sentry**, **New Relic** possède également un package **Python** pouvant être installé par l'intermédiaire de **pip** :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ pip install newrelic
...
```

5.2.2 - Configuration

Une fois installé, il est nécessaire de générer un fichier de configuration à la racine du projet. **New Relic** permet de le générer automatiquement, il suffit de saisir la commande suivante :

```
(…)(USERNAME)@(SERVER_NAME):~/ newrelic-admin generate-config (LICENCE_KEY) newrelic.ini
...
```

Il est indispensable de saisir la « *LICENCE_KEY* » donnée lors de la création de l'entrée dans le *Dashboard* de **New Relic**. La configuration est alors terminée et les données remonteront automatiquement. Le service est exécuté automatiquement puisque nous l'avons déjà intégré dans notre fichier de configuration **NGINX**, lors du lancement de **GUNICORN**.

6 - DONNEES DE TEST (OPTIONNEL)

6.1 - Intégration de Docker

Durant la phase de développement, des données de test générées aléatoirement ont été mises en place pour simuler une utilisation courante de l'application sans porter préjudice aux données réelles de l'application. Pour cela, il est recommandé d'utiliser une base de données de test (qui peut être créée comme nous l'avons fait précédemment), mais nous devons également simuler les données fournies par les **API OCS** et **IMMO**.

Nous utilisons deux bases de données différentes sur des containers « **Docker** » basés sur des images **PostgreSQL**. Commençons par installer **Docker** sur notre distribution :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo apt install apt-transport-https ca-certificates
curl software-properties-common
...
(project_env)(USERNAME)@(SERVER_NAME):~/ curl -fsSL https://download.docker.com/linux/ubuntu/gpg
| sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
...
(project_env)(USERNAME)@(SERVER_NAME):~/ echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
...
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo apt update
...
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo apt install docker-ce
...
```

Une fois **Docker** et ses dépendances installées, on s'assure qu'il soit bien en cours d'exécution :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo systemctl status docker
...
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2023-04-27 13:30:41 UTC; 32s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
...
```

Nous disposons d'une image **Docker** avec **PostgreSQL** déjà paramétré, nous pouvons l'utiliser de la manière suivante :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker run --name psql_srv -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=**** -p 5433:5432 -v /data:/var/lib/postgresql/data -d backup-ocs_immo_psql-
260423
...
```

Il ne reste ensuite qu'à se connecter au container :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker exec -it psql_srv bash
...
```

Nous pouvons créer les bases de données *ocs_db* et *immo_db* comme nous le ferions sur une distribution **Ubuntu server** classique.

Suite à cela, nous pouvons créer les données qui seront importées ensuite dans l'application **OC-Inventory**. Si nous disposons d'un backup des données contenues dans ces bases, nous pouvons les importer de la façon suivante. Dans un premier temps, nous devons les copier dans le container :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker cp ocs_backup.sql immo_backup.sql psql_srv:/tmp/
...
```

Suite à cela, il ne nous reste plus qu'à importer les données dans les bases nouvellement créées :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker exec -t psql_srv sh -c 'psql -U postgres -d ocs_db < /tmp/ocs_db_dump.sql'
...
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker exec -t psql_srv sh -c 'psql -U postgres -d immo_db < /tmp/immo_db_dump.sql'

...
```

Nous pouvons faire en sorte que le container soit exécuté au démarrage du serveur :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker update --restart unless-stopped psql_srv
...
```

Dorénavant, si le serveur doit redémarrer, le container sera exécuté en même temps que les autres services précédemment configurés.

6.2 - Script & Crontab

Dans notre cas, nous intégrons, tous les jours, une partie de ces données de test. Pour cela, nous avons créé un script exécutant la commande « manage.py fill_main_db » dont l'exécution est gérée par une tâche créée sur le « crontab » du serveur.
Pour commencer, nous créons un répertoire « scripts » contenant deux fichiers, un premier script Python nommé « log-datetime.py » et un autre bash nommé « fill_db.sh ». Le premier permet de récupérer la date et l'heure actuelle que nous intégrerons dans les logs du script lors de chaque exécution. Le second contient les informations nécessaires à l'exécution de la commande précédemment citée :

fill_db.sh :

```
#!/bin/sh

export SECRET_KEY="**********************"
export DJANGO_SETTINGS_MODULE=inventory.settings
export PSQL_SRV='localhost'
export DB_USER='admdjango'
export DB_PWD='*********'
export PSQL_PORT='localhost'

. /(Project_venv_PATH)/bin/activate

python3 /(SCRIPTS_PATH)/log-datetime.py Devices
python3 /(PROJECT_PATH)/manage.py fill_main_db
```

Log-datetime.py :

```
import sys
import datetime

arg1 = sys.argv[1]
current_dt = datetime.datetime.now()
current_dt = current_dt.strftime("%d-%m-%Y %H:%M:%S")

print("======================================")
print(f"  {arg1} : {current_dt}")
print("======================================")
```

Puis on rend le script bash exécutable :

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo chmod +x fill_db.sh
...
```

Il ne reste ainsi plus qu'à ajouter une tâche exécutant le script dans le crontab :

```
(…)(USERNAME)@(SERVER_NAME):~/ crontab -e
...
```

```
...
0 4 * * * /(SCRIPTS_PATH)/fill_db.sh >> /(LOG_PATH)/fill_db.log 2>&1
...
```

De cette façon, le script sera exécuté tous les jours à 4h du matin et retournera la sortie dans un fichier nommé « fill_db.log » (y compris les erreurs s'il y en a).

7 - PROCEDURE DE DEMARRAGE

7.1 - Base de données

Si pour une raison ou une autre il est nécessaire de démarrer, manuellement, le service lié à **PostgreSQL**, saisir la commande suivante :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service postgresql start
...
```

7.2 - Application

Le démarrage de l'application se fait par l'intermédiaire du fichier de configuration **Supervisor** de celle-ci. Il n'est donc pas nécessaire de la démarrer manuellement. A savoir que l'application de **Monitoring NewRelic** est également gérée par **Supervisor**. Les variables d'environnements sont également déclarées dans le fichier de configuration **Supervisor**.

7.2.1 - Installation des dépendances

L'installation des dépendances de l'application se fait par l'intermédiaire du gestionnaire de paquets **pip**. Elles sont contenues dans le fichier « *requirements.txt* » à la racine du projet, pour les installer, simplement saisir la commande :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ pip install -r requirements.txt
...
```

7.2.2 - Génération des fichiers statiques

Nous en avons déjà parlé dans un chapitre précédent, mais pour rappel, voici la commande à saisir pour indiquer à **Django** de générer les statiques :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ python manage.py collectstatic
...
```

7.2.3 - Migrations

```
(project_env)(USERNAME)@(SERVER_NAME):~/ python manage.py makemigrations
...
(project_env)(USERNAME)@(SERVER_NAME):~/ python manage.py migrate
...
```

7.2.4 - Supervisor

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service supervisor start
...
```

7.2.5 - NGINX

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service nginx start
...
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service nginx status
● nginx.service - A high performance web server and a reverse proxy server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2023-04-28 14:44:43 UTC; 2h 23min ago
       Docs: man:nginx(8)
...
```

7.2.6 - Docker

```
(…)(USERNAME)@(SERVER_NAME):~/ sudo docker run --name (CONTAINER_NAME) -e POSTGRES_USER=postgres
-e POSTGRES_PASSWORD=**** -p 5433:5432 -v /data:/var/lib/postgresql/data -d (IMAGE_NAME)
...
```

8 - PROCEDURE DE MISE A JOUR

8.1 - Base de données

S'il s'avère nécessaire de redémarrer manuellement **PostgreSQL** (dans le cas d'une modification de la configuration par exemple), saisir la commande suivante :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service postgresql restart
...
```

8.2 - Application

Concernant l'application, si une mise à jour est effectuée et dans la mesure ou son lancement est géré par **Supervisor**, il faudra préalablement stopper **Supervisor** avant de le redémarrer une fois la mise à jour effectuée.

Par contre, si une mise à jour est effectuée côté **Supervisor**, il sera nécessaire d'exécuter les commandes suivantes :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl reread
...

(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl update
...

(project_env)(USERNAME)@(SERVER_NAME):~/ sudo supervisorctl restart
...
```

Cela aura pour effet de prendre en charge les modifications apportées puis de redémarrer **Supervisor**.

9 - PROCEDURE D'ARRET

9.1 - Base de données

Pour stopper l'activité du serveur **PostgreSQL**, saisir la commande :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service postgresql stop
...
```

9.2 - Application

Pour stopper l'application, il est nécessaire d'arrêter le service **Supervisor** :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo service supervisor stop
...
```

9.3 - Redémarrage du serveur Linux

Pour redémarrer le système d'exploitation :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo /sbin/reboot
...
```

Pour arrêter le serveur immédiatement :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo shutdown -h now
...
```

Ou de façon différée :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ sudo shutdown -h 30
...
```

Pour éteindre le serveur 30 minutes plus tard

10 - SAUVEGARDE ET RESTAURATION

Concernant la sauvegarde de la base de données, une tâche **CRON** tourne toutes les 4h sur le serveur. Nous pouvons l'éditer de la façon suivante :

```
(…)(USERNAME)@(SERVER_NAME):~/ crontab -e
...
```

```
...
*/4 * * * * backup_db.sh >> /var/log/oc_inventory/backup_db.log 2>&1
...
```

Il est possible de consulter un fichier **log** de la sauvegarde dans « */var/log/oc_inventory* »

Concernant la restauration de la base en cas de problème quelconque, il faudra saisir la commande suivante :

```
(project_env)(USERNAME)@(SERVER_NAME):~/ python manage.py loaddate (DATABASE_BACKUP)
...
```

11 - INTEGRATION CONTINUE

Comme expliqué dans les autres documents, nous utilisons un service d'intégration continue nommée **Circle CI**. Ce chapitre se trouve à la fin du document car il n'est pas lié au serveur de production lui-même mais plutôt au développement de l'application, ou dans le cadre du développement d'une nouvelle fonctionnalité par exemple.

Il est lié au « *repository* » **Github** et se charge de surveiller les **pull-requests** vers la branche « ***dev*** » de l'application. Lors d'un *push* sur la branche, l'outil met en place un environnement *docker*, contenant toutes les dépendances nécessaires à l'exécution des différents tests unitaires et fonctionnels écrits pour l'application.

Concernant sa mise en place, il est nécessaire de créer un dossier nommé « ***.circleci/*** » à la racine du projet ainsi qu'un fichier nommé « **config.yml** » à l'intérieur de ce dernier. Ce fichier doit contenir les éléments nécessaires à l'outil pour mettre en place l'environnement et exécuter les tests. Un fichier de *settings* **Django** minimaliste est également créé concernant la configuration de la base de données. Voici à quoi cela ressemble :

11.1 - Configuration de CircleCI

.circleci.config.yml

```yaml
version: 2.1

jobs:
  build:
    docker:
      - image: circleci/python:3.9
      - image: circleci/postgres:13
        environment:
          POSTGRES_USER: circleci
          POSTGRES_PASSWORD: circleci_pwd
          POSTGRES_DB: circle_test
    working_directory: ~/repo
    steps:
      - checkout
      - run: pip install -r requirements.txt
      - run:
          name: Set Secret Key
          command: |
            echo 'export SECRET_KEY=${SECRET_KEY}' >> $BASH_ENV
            source $BASH_ENV
          when: always
```

```
      - run:
          name: Set Email Host
          command: |
            echo 'export EMAIL_HOST=${EMAIL_HOST}' >> $BASH_ENV
            source $BASH_ENV
          when: always
      - run:
          name: Set Email User and Password
          command: |
            echo 'export EMAIL_HOST_USER=${EMAIL_HOST_USER}' >> $BASH_ENV
            echo 'export EMAIL_HOST_PWD=${EMAIL_HOST_PWD}' >> $BASH_ENV
            source $BASH_ENV
          when: always
      - run:
          name: Install Dockerize
          command: |
            sudo apt-get update
            sudo apt-get install wget
            wget https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-linux-
amd64-v0.6.1.tar.gz
            sudo tar -C /usr/local/bin -xzvf dockerize-linux-amd64-v0.6.1.tar.gz
            rm dockerize-linux-amd64-v0.6.1.tar.gz
      - run:
          name: Install Google Chrome
          command: |
            sudo apt-get update
            sudo apt-get install -y curl unzip xvfb
            curl -sS -o - https://dl-ssl.google.com/linux/linux_signing_key.pub | sudo apt-key
add -
            sudo sh -c 'echo "deb [arch=amd64] http://dl.google.com/linux/chrome/deb/ stable
main" >> /etc/apt/sources.list.d/google-chrome.list'
            sudo apt-get -y update
            sudo apt-get -y install google-chrome-stable
      - run:
          name: Install ChromeDriver
          command: |
            CHROME_VERSION=$(google-chrome --version | awk '{print $3}' | awk -F '.' '{print
$1}')
            CHROMEDRIVER_VERSION=$(curl --location --request GET
'https://chromedriver.storage.googleapis.com/LATEST_RELEASE_'$CHROME_VERSION)
            curl --silent --show-error --location --fail --retry 3 --output
/tmp/chromedriver_linux64.zip
"https://chromedriver.storage.googleapis.com/$CHROMEDRIVER_VERSION/chromedriver_linux64.zip"
            cd /tmp
            unzip chromedriver_linux64.zip
            sudo mv chromedriver /usr/local/bin/chromedriver
            sudo chmod +x /usr/local/bin/chromedriver
            echo 'export PATH=$PATH:/usr/local/bin' >> $BASH_ENV
            chromedriver --version
      - run:
          name: Wait for database
          command: dockerize -wait tcp://localhost:5432 -timeout 1m
      - run:
          name: Run tests
          command: |
            python manage.py test --noinput --settings=inventory.settings.test
```

Dans ce fichier de configuration, nous indiquons à l'outil d'utiliser les images ***Python 3.9*** ainsi que ***PostgreSQL 13*** lors de la création du *docker*. Nous passons également plusieurs variables d'environnement nécessaires à l'utilisation de l'application. Il est important de souligner le fait que les valeurs des variables d'environnement doivent être déclarées dans l'interface web de *CircleCI*. Pour cela, il est nécessaire de se rendre dans les settings du projet puis de déclarer les variables dans la partie « *Environment Variables* ». Bien évidemment, il ne faut surtout pas déclarer ces valeurs directement dans le fichier de configuration, celui-ci étant exposé.

Ensuite, nous lui demandons d'installer les dépendances python contenues dans le fichier « *requirements.txt* ».

Les étapes suivantes, nous indiquons des commandes *bash* pour télécharger et installer des applications tierces comme **Dockerize**, **Google Chrome** et **Chromedriver** nécessaires lors de l'exécution des tests fonctionnels.

Enfin, nous terminons par demander à *CircleCI* d'exécuter tous les tests (unitaires et fonctionnels) de l'application.

11.2 - Settings Django pour CircleCI

settings.test.py

```
from . import *

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'circle_test',
        'USER': 'circleci',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Pour finir, nous créons un fichier nommé « test.py » dans le répertoire settings de l'application. Celui-ci hérite des paramètres principaux de l'application (contenus dans le fichier *__init__.py*) en dehors de ce qui concerne la base de données. Pour celle-ci nous indiquons des informations différentes de celles indiquées habituellement pour l'application.

CircleCI est maintenant paramétré et devrait passer les différents tests avec succès.