Institut National de Radioélectricité et de Cinématographie

Enseignement technique secondaire de qualification

Accès aux études supérieures

Avenue Jupiter, 188

1190 Forest

Surveillance de réseau : NetMonitor

Projet personnel de Victor Vereecke

Pour l’obtention du certificat de qualification

Technicien(ne) en informatique

2025-2026

I. Remerciements

Avant tout, j'aimerais d'abord remercier mes parents qui m'ont aidé, soutenu et encouragé à poursuivre ma passion de l'informatique depuis le début.

J'aimerais également remercier tous les professeurs qui m'ont aidé pendant mes 4 ans d'informatique à l'INRACI, notamment M. Ben Sellam qui m'a donné le sujet de mon TFE et M. Hadjit pour ses encouragements.

Je tiens aussi à remercier l'ensemble de mes camarades de classe pour l'entraide et la bonne ambiance qui ont rendu ces 4 années d'études beaucoup plus agréables.

II. Table des matières

III. Introduction

1. Contexte

De nos jours, de maintes infrastructures modernes allant de simples bureaux aux hôpitaux fonctionnent avec des dizaines voire centaines d'ordinateurs connectées entre eux, automatisant et simplifiant le travail de millions de personnes.

Cependant, entretenir un tel réseau peut s'avérer très compliqué vu le nombre d'appareils. En effet, une seule panne risque de paralyser une partie si pas tout le réseau, et donc ralentir voire empêcher tout travail. En plus de cela, plus on a d'appareils sur le réseau, plus cela sera couteux en temps et en diagnostics.

J'estime qu'il est donc important de connaître l'état de chacun des appareils tout le temps afin de remarquer et remédier à tout problème avant qu'il cause une panne ou des dégâts.

Ceci est le but de mon TFE, c'est-à-dire une application qui surveille et prévient l'utilisateur de problèmes présents sur ses appareils en temps réel pour qu'il y remède avant que cela ne cause plus de problèmes.

2. Qu'est ce concrètement ?

Mon TFE est un projet qui récupère les données telles que l'utilisation du processeur, de la mémoire, du stockage et les services actifs & arrêtés. Elles sont ensuite reprises par le serveur et affichées sous forme de graphiques, noms et listes sur un Dashboard accessible sur n'importe quel appareil avec un navigateur web dans le même réseau que le serveur.

Le projet est principalement constitué de deux applications, une qui agit comme « Agent » qui récupère les informations que mon projet surveille et les envoie à une autre application qui agit comme serveur qui va les traiter et les afficher sur un Dashboard avec des graphiques sur lequel on peut paramétrer des alertes, désactiver certaines vérifications, et bien plus…

IV. Analyse de besoins

1. Description du problème

Comme expliqué ci-dessus, le problème que cherche à résoudre mon projet est le coût & le temps que prend une analyse et réparation lors d'un problème technique au sein d'une infrastructure informatique.

Avant qu'un appareil tombe en panne, il y a des signes ou des choses qui mènent à cette panne. Par exemple, la surutilisation du processeur peut mener à sa surchauffe qui peut-elle, mener à la mort de ce dernier, ce qui peut couter plusieurs centaines, voire milliers d'euros s'il s'agit d'un processeur haute-gamme.

De plus, s'il s'agit d'un appareil important, il risque d'impacter d'autres parties de la topologie, causant d'autres pertes.

2. Public ciblé & Cahier de charges / Fonctionnalités attendues

Mon projet cherche donc à alléger le travail de l'administrateur système/réseau de n'importe quel environnement en lui fournissant un outil qui doit être :

- Accessible facilement et rapidement n'importe où
- Capable d'afficher l'état de tous les appareils concernés
- Notifier l'administrateur de potentiels problèmes dès qu'ils surgissent
- Garder une trace de tous ces problèmes
- Assurer une installation facile

3. Contraintes

J’ai été soumis à plusieurs contraintes lors de la conception, et la création de mon TFE.

D’abord, mon projet consiste à lancer un serveur Flask, un écran OLED, et d’autre programmes au même temps qui reçoivent et renvoient des informations constamment. Un microcontrôleur comme l’ESP32 ne convient donc pas à mon projet.

Il me faut donc obligatoirement un Raspberry Pi, qui est un microordinateur faisant tourner un système d’exploitation entier comme Linux (Contrairement à l’ESP32 qui lui, fait tourner qu’un seul programme à la fois).

Et à cause des prix de mémoire RAM qui ont beaucoup monté ces derniers mois, je n’ai pas pu acheter un Raspberry Pi neuf, j’ai donc dû en emprunter un à un ami, ce qui m’a limité à un Raspberry Pi 3 avec 1 Go de mémoire RAM.

Enfin, il y a la contrainte du temps, qui m’a particulièrement impactée car j’ai réalisé la majorité du travail sur mon projet plutôt tard dans l’année (A partir des Vacances de Printemps), ce qui m’a laissé environ 2 mois pour tout finaliser.

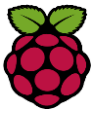
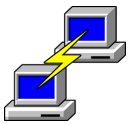
V. Etude préalable & comparaison

1. Solutions existantes

Bien évidemment, je n'ai pas inventé ce genre de solution, il en existe un paquet qui sont bien plus développés que mon projet comme Zabbix, Nagios et d'autres avec chacune leurs avantages et leurs inconvénients, cependant, il y a quand même certains domaines ou mon projet conviendrait mieux.

/	Avantages	Inconvénients
Zabbix, Nagios, …	• Prévu pour des infrastructures avec des milliers de serveurs • Surveillance via beaucoup de protocoles (SSH, SNMP, WMI, …) • Gestion de droits très poussée pour une liaison avec un annuaire d'entreprise (AD/LDAP) • Beaucoup de fonctionnalités (Carte avec l'emplacement des appareils, gestion à distance, résumé des détections mensuelle, support de plugins, …)	• Souvent très couteux • Nécessite un serveur performant • Configuration complexe et longue • Plus difficile à maintenir en termes de développement
Mon TFE	• Très léger & rapide (Tourne sur un Raspberry Pi 3) • Simple à configurer (2-3 fichiers de configuration) • Langage facile à retenir	• Concu pour une centaine d'appareils maximum • Aucun support pour la surveillance en agentless (routeurs, etc…) • Sécurité basique • Minimum de fonctionnalités

2. Logiciels utilisés

Image	Nom	Courte Description	Utilité dans le projet
	Windows Server 2016	Version de Windows utilisée sur les serveurs sur laquelle on peut paramétrer plein de services (AD, FTP, ISS, DHCP)	Sert comme serveur de test avec l'agent installé et plusieurs services activés & surveillés (AD, ISS & FTP)
	Windows 10	Système d'exploitation crée par Microsoft utilisé sur des ordinateurs prévus pour des simples utilisateurs permettant de lancer des applications basiques.	Sert comme client de test avec l'agent installé.
	VMware Workstation 25H2	Logiciel de virtualisation de type 2 permettant de simuler un réseau et des ordinateurs.	Permet de simuler la topologie entière sans devoir acheter des ordinateurs physiques.
	Raspbian OS	Version de Debian Linux optimisée pour tourner sur un micro-ordinateur Raspberry Pi	Système d'exploitation installé sur le Raspberry Pi, il sert à lancer le serveur.
	SolidWorks 2025	Logiciel par Dassault Systèmes permettant de réaliser des modèles CAD, des simulations de liquides et bien plus	Permet de créer le modèle 3D de la boîte du serveur
	PuTTY	Logiciel permettant de se connecter à distance sur des appareils en utilisant le protocole SSH et Telnet	Permet de se connecter à distance au Raspberry Pi pour lancer le serveur

	Visual Studio Code	Logiciel IDE servant à simplifier le codage en gérant les bibliothèques, la colorisation syntaxique et le débogage du code.	Programme utilisé pour coder l'intégralité du projet.
	Gemini	IA crée par Google servant comme assistant pour la génération de code, rédaction de document, recherche d'informations et bien plus.	Utilisée pour générer une grande partie du code du Dashboard et du site web et aide pour le débogage du backend du serveur et de l'agent.
	Claude AI	IA crée par Anthropic ayant la même utilité que Gemini mais étant meilleure pour le raisonnement et le code.	Même utilité que Gemini, mais aussi utilisé pour aider à déterminer quelles bibliothèques python et logiciels utiliser lors de la conception du projet.
	Qwen 3-Max	IA comme Gemini et Claude, mais étant entièrement gratuite au détriment de ses capacités	Utilisé au début du développement pour avoir une idée de quoi utiliser pour le projet.
	Firefox	Navigateur web crée par Mozilla permettant d'ouvrir des pages HTML et d'accéder à internet.	Utilisé pour visualiser le Dashboard et le site web, et pour faire des recherches pendant le développement du projet.
	WinSCP	Logiciel permettant de transférer des ficher avec plusieurs protocoles, notamment grâce à SFTP, FTP, SCP, et d'autres	Utilisé pour transférer les fichiers entre le Raspberry Pi et mon pc pour tester.

	PrusaSlicer	Logiciel permettant de générer du G-Code en partant d'un modèle en « .stl » pour les imprimantes 3D	Utilisé pour générer le G-Code pour imprimer le modèle 3D de la boîte du serveur.
	InstallForge	Logiciel permettant de créer des installeurs interactifs pour Windows	Utilisé pour créer un installeur pour l'agent sur les hôtes Windows.

3. Languages de code

Image	Nom	Description	Pourquoi l'utiliser ?
	Python	Langage de programmation simple utilisant de nombreuses de bibliothèques pour développer rapidement des programmes	C'est le langage vu en classe et celui qu'on est censé utiliser pour notre Projet. Utilisé dans le projet pour réaliser le programme agent, le programme de configuration et le backend du serveur.
	HTML	Langage de balisage utilisé pour structurer et organiser les éléments textuels et visuels d'une page web	Utiliser pour former le squelette de l'interface du Dashboard
	CSS	Langage servant à mettre en forme l'interface web (couleurs, polices, disposition, …), garantissant un design moderne et adapté à tous les types d'écrans	Utilisé pour donner une apparence moderne au Dashboard.
	JavaScript	Langage de programmation dynamique exécuté par le navigateur permettant de créer des animations, des fenêtres, etc…	Utiliser pour faire communiquer le Dashboard avec l'API Flask, c'est-à-dire afficher & mettre à jour les données sous forme de graphique et faire fonctionner les champs et paramètres, c'est ce qui lie le Dashboard au backend
	JSON	Pas vraiment un langage de programmation mais plutôt un format de fichier pour enregistrer des données pour une utilisation globale à travers beaucoup d'applications	Il est parfaitement compatible avec JS et fonctionne aussi avec python via la bibliothèque Jsonify, cela me permet de garder tous les données envoyées et reçues entre l'agent et le serveur dans le même format.

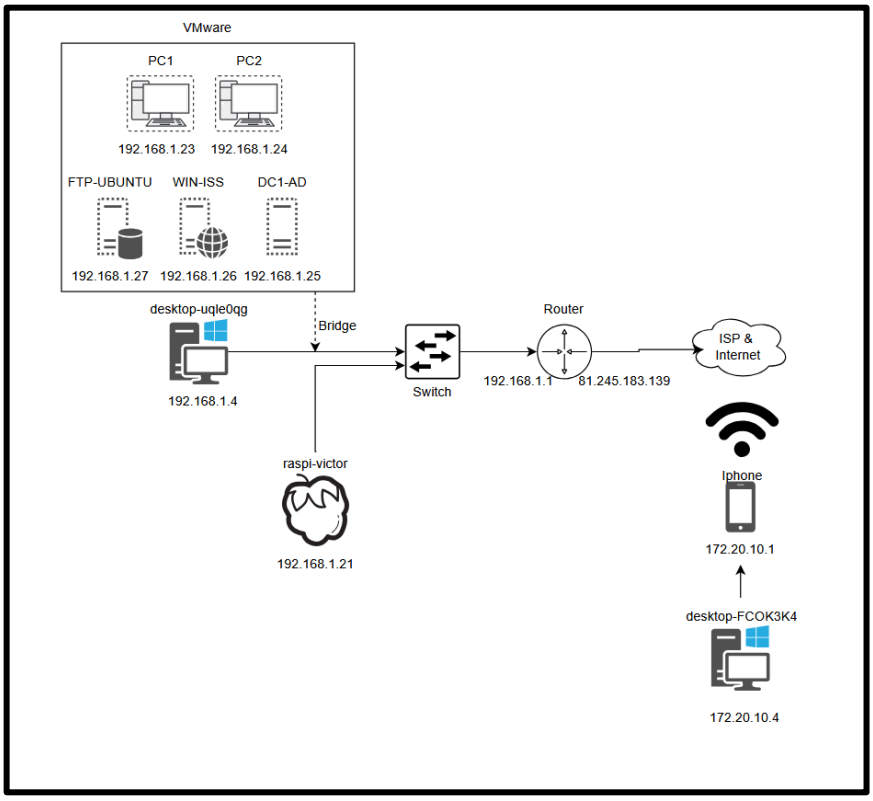
4. Bibliothéques python externes utilisées

Image	Nom	Description	Utilité dans mon projet
	Flask	Framework de développement web en Python léger et modulaire, conçu pour créer des applications web et des API de manière simple et rapide.	Utilisé pour développer le serveur. Il reçoit les données envoyées par l'agent et génère l'interface web (le Dashboard) consultable par l'administrateur.
	Werkzeug Security	Bibliothèque de sécurité qui fournit des outils de hachage de données hautement sécurisés (comme les algorithmes PBKDF2 ou SHA-256).	Permet de sécuriser l'accès à l'interface en chiffrant les mots de passe des administrateurs dans la base de données, évitant ainsi de les stocker en texte brut (plaintext).
	SQLite / MySQL Connector	Bibliothèques permettant à un programme Python de se connecter, et de communiquer avec des bases de données en SQL.	Utilisé pour lier le backend à la base de données afin d'enregistrer l'historique des pannes, l'état des machines et les logs du système.
	PSUtil	Bibliothèque Python permettant de récupérer des informations sur le matériel et l'utilisation du système.	Utilisée par l'agent pour récupérer les informations qu'on recherche (CPU, RAM, Stockage, Services & Hostname)
	Jsonify	Bibliothèque Python permettant de lire et utiliser des fichiers JSON	Utilisé principalement pour lire les configurations réseau pour le serveur & le client.

	Tkinter	Bibliothèque graphique intégrée nativement à Python, servant à créer des fenêtres et des interfaces logicielles locales.	Utilisée pour créer l'interface graphique du programme de configuration, permettant à l'utilisateur de modifier les paramètres réseau de l'agent avec une interface.
	Requests	Bibliothèque Python populaire qui permet d'envoyer des requêtes HTTP (GET, POST, etc.) vers un serveur web de façon très simple.	Utilisée par l'agent pour transmettre (avec une requête POST) toutes les métriques matérielles collectées directement vers l'API.
	Plyer	Bibliothèque Python externe permettant d'accéder de manière universelle aux fonctionnalités matérielles et logicielles des différents systèmes d'exploitation.	Utilisée pour déclencher des notifications push natives sur le bureau de l'administrateur dès qu'un problème critique survient.

VI. Conception physique du Projet

1. Topologie

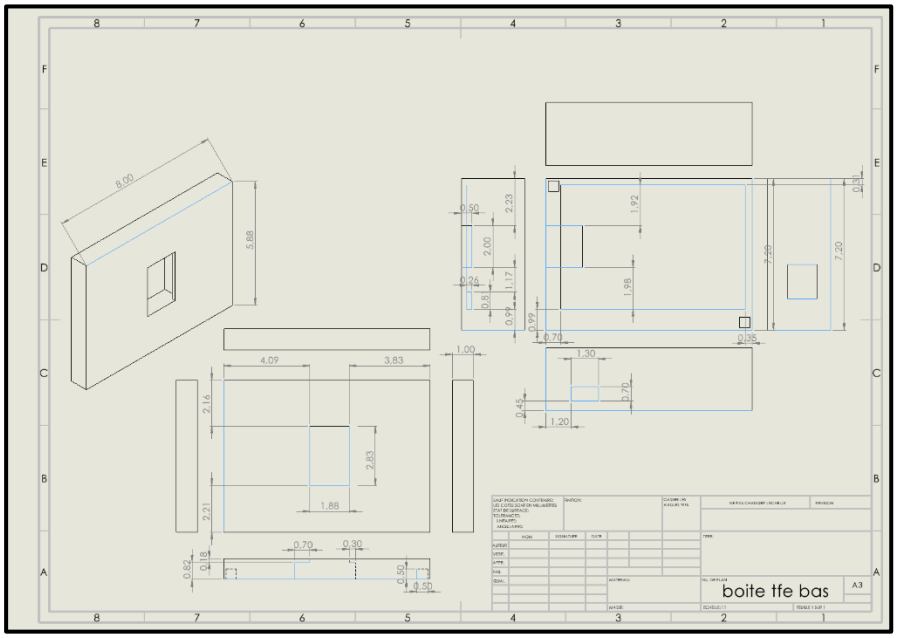


Voici la topologie de test pour le projet, d'un côté se trouvent les machines sur le réseau local qui sont virtualisées sur mon ordinateur, et de l'autre on a un autre ordinateur connecté via un partage de connexion de mon téléphone à internet. Le but de ce dernier est de tester l'application en dehors du réseau local. Chacun des ordinateurs ont un agent qui va envoyer les données au serveur RPi.

2. Modèle 3D

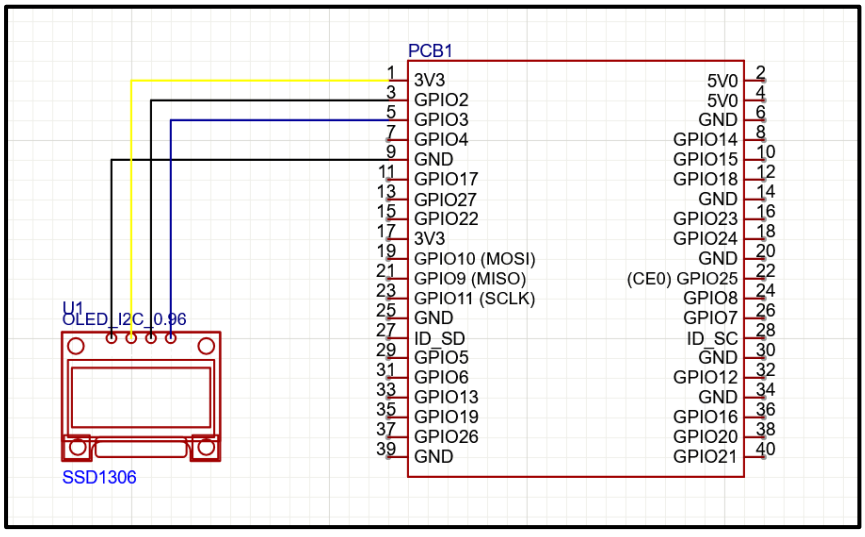
Pour le Raspberry, j'ai réalisé une boite en 3D qui me laisse attacher un écran OLED SSD1306 dessus et me permet aussi de cacher les ports inutiles améliorant l'esthétique du projet.

Le couvercle et le bas sont retenus avec des aimants, et l'écran est fixé avec du plastique.



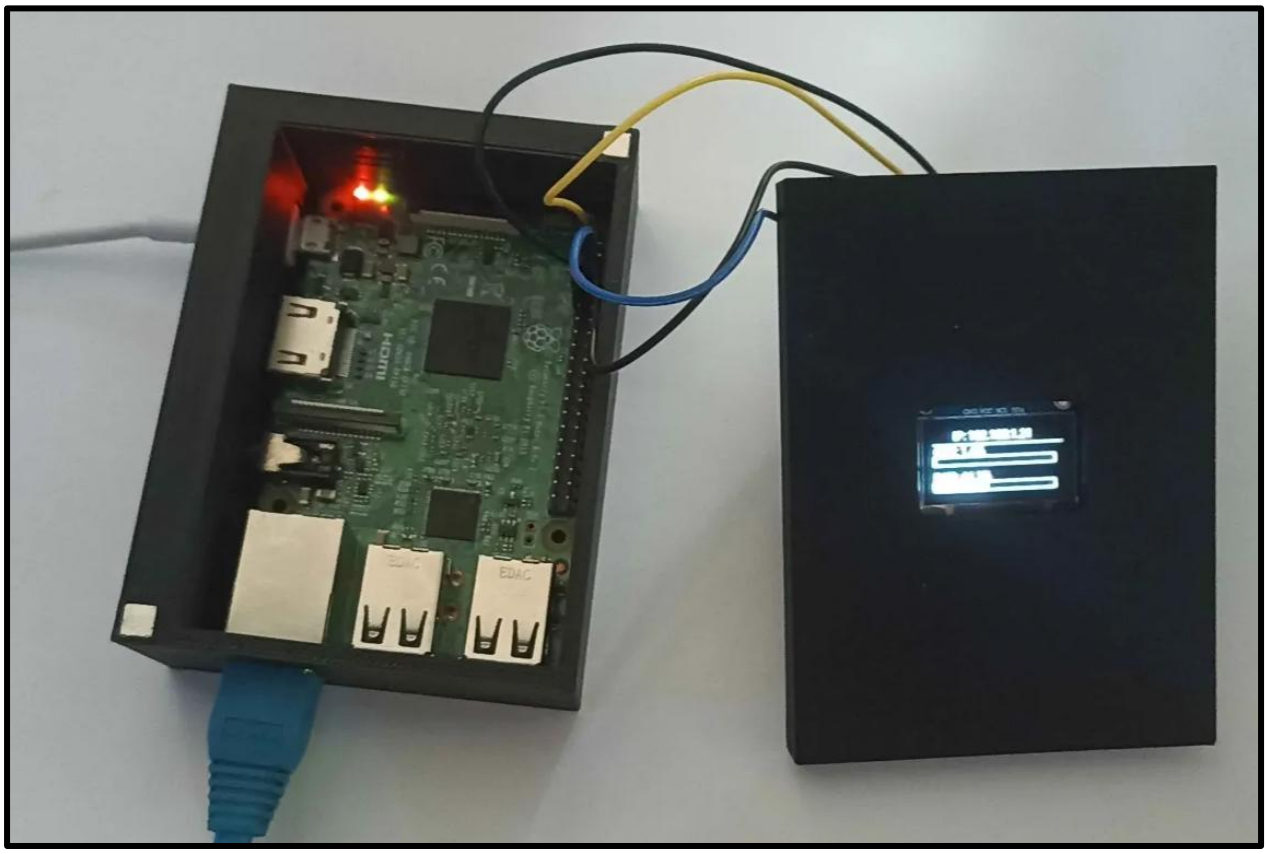
3. Schéma électrique

Ensuite, j'ai branché un écran OLED qui me permet d'afficher des informations comme l'IP et l'état du Raspberry Pi.



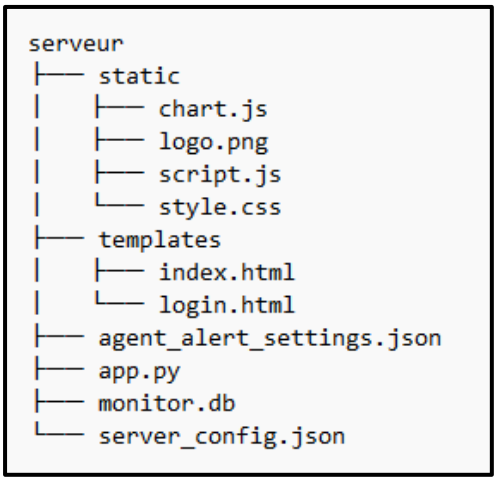
4. Photo de la maquette

Voici une photo de la maquette, on peut remarquer sur l'écran l'adresse IP du serveur, l'utilisation du processeur et de la RAM. On y voit aussi les aimant qui servent à garder la boite fermée et les câbles reliant l'écran au microordinateur.



5. Organisation des fichiers

Sur le serveur, il y a plusieurs fichiers essentiels pour le bon fonctionnement du serveur.



On retrouve d'abord le dossier *static*, qui contient d'abord *chart.js* qui est le fichier qui contient les graphiques afin que mon Dashboard ne dépende pas d'un accès à internet pour afficher le graphique, ensuite on a le logo du projet (*logo.png*) qui est utilisé un peu partout sur le dashboard.

Ensuite, on y trouve le fichier *script.js*, qui contient toute la logique derrière les champs, les différents panneaux du Dashboard, l'enregistrement des paramètres, l'affichage des graphiques, données, et bien plus… C'est ce fichier qui géré la liaison entre le Dashboard & l'API Flask. Cela en fait un des fichiers les plus importants du serveur.

Et enfin, il y a le fichier *style.css* qui gère l'esthétique du Dashboard, c'est-à-dire qu'il contient les couleurs, polices, tailles et positions des différents éléments de ce dernier.

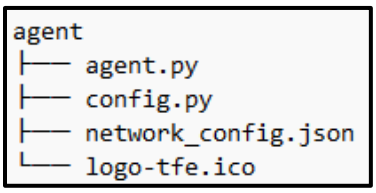
Apres cela, on retrouve le dossier *templates* qui comme le nom l'indique contient les squelettes des deux pages principales du serveur, c'est-à-dire l'écran de connexion (*login.html*) et le Dashboard en lui-même (*index.html*). Ce sont ces fichiers qui contiennent les éléments qui sont ensuite modifiées par les fichiers CSS & JS.

A cela s'ajoutent les fichiers de configuration *agent_alert_settings.json* et *server_config.json* qui contiennent des configurations pour les alertes des différents agents (comme les limites avant d'avertir l'administrateur) et la configuration du serveur (comme le port) respectivement.

Et enfin on y retrouve *app.py* et *monitor.db*. *app.py* est le fichier principal du serveur. Il contient les différentes routes API, les informations à faire passer et les fonctions du serveur.

monitor.db, lui est une base de données locale qui contient les utilisateurs et les logs.

L’agent lui, dépend de beaucoup moins de fichiers,



Le premier fichier, *agent.py* est le fichier cœur de l’agent, c’est lui qui est lancé pour récupérer constamment les informations et les envoyer au serveur via HTTP

Le deuxième fichier, *config.py* est une application Tkinter qui permet de définir les paramètres de l’agent comme l’adresse IP du serveur, son port et le délai d’envoi de données. *logo-tfe.ico* est l’icône de ce dernier.

Et *network_config.json*, lui stocke les paramètres que config.py modifie.

VII. Réalisation

1. Explication du code important - Agent

Le fonctionnement du projet est basé sur deux application, un agent qui récupère les données d'un appareil et qui les envoie vers un serveur. Et une application serveur qui reprend ces données, les formatte et les affiche sur le Dashboard et gère les alertes.

Plus précisément, l'application agent commence par lire un fichier de configuration qui contient les informations du serveur qu'elle reformatera en un lien qu'elle utilisera pour envoyer les données.

```
#lit l'adresse du serveur et le delai dans le fichier json
def get_config():
    try:
        base = os.getenv('LOCALAPPDATA') or os.path.expanduser('~')
        config_path = os.path.join(base, 'NetMonitor', 'network_config.json')

        with open(config_path, "r") as f:
            data = json.load(f)
            config = data[0] if isinstance(data, list) else data
            #construit l'url avec l'ip et le port du fichier json
            url = f"http://{config['ip']}:{config.get('port', '5000')}/update"
            interval = float(config['interval'])
            return url, interval
    except Exception:
        return "http://127.0.0.1:5000/update", 3
```

Une fois le lien reçu, l'application va essayer de récupérer les services.

```
def collect_services():
    services_dict = {}
    #récupére les services
    for service in psutil.win_service_iter():
        try:
            services_dict[service.name()] = service.status()
        #eviter d'arreter l'agent si on trv pas un service ou autre raison
        except (psutil.NoSuchProcess, psutil.AccessDenied, OSError):
            continue
    return [{"name": n, "status": s} for n, s in services_dict.items()]
```

Une fois les services récupérés, l'application va essayer de récupérer le reste des informations et de les envoyer.

```python
#identification du processeur
cpu_info = platform.processor() or "Inconnu"

#récupération des autres données
stats = {
    "os": f"{platform.system()} {platform.release()}",
    "processor": cpu_info,
    "cpu_usage": psutil.cpu_percent(interval=None),
    "ram_usage": mem.percent,
    "ram_total": round(mem.total / (1024**3), 1),
    "storage_usage": disk.percent,
    "storage_total": round(disk.total / (1024**3), 1),
    "hostname": socket.gethostname(),
    "services": services_list,
    "interval": INTERVAL
}

res = requests.post(SERVER_URL, json=stats, timeout=5)
```

S'il n'arrive pas à envoyer les données, il va afficher une notification d'erreur toutes les deux minutes tant qu'il n'arrive pas à joindre le serveur.

```python
except Exception as e:
    #renvoi d'une notification système toutes les 2 minutes en cas d'échec
    current_time = time.time()
    if current_time - last_error_notification >= 120:
        notification.notify(
            title="NetMonitor - Erreur Agent",
            message=f"Impossible de joindre le serveur de monitoring sur {SERVER_URL}.",
            timeout=10
        )
        last_error_notification = current_time
        was_disconnected = True
```

2. Explication du code important – Serveur

Le serveur, lui est beaucoup plus complexe. Il commence par charger les variables et les fonctions pour les utiliser plus tard.

```python
app = Flask(__name__)
#cle de securite pour les sessions
app.secret_key = os.environ.get('FLASK_SECRET_KEY', os.urandom(24))

#definit le répertoire de base pour les fichiers de configuration et la base de données
SERVER_BASE_DIR = os.path.dirname(os.path.abspath(__file__))

CONFIG_FILE = os.path.join(SERVER_BASE_DIR, 'server_config.json')
AGENT_SETTINGS_FILE = os.path.join(SERVER_BASE_DIR, 'agent_alert_settings.json')

def load_server_config():
    if os.path.exists(CONFIG_FILE):
        try:
            with open(CONFIG_FILE, 'r') as f:
                return json.load(f)
        except: pass
    return {
        "db_type": "sqlite",
        "sqlite_db": "monitor.db",
        "mysql_host": "localhost",
        "mysql_user": "root",
        "mysql_pass": "",
        "mysql_name": "monitor_db",
        "offline_timeout": 10,
        "log_retention_days": 7,
        "log_cleanup_enabled": True
    }
```

```python
@app.teardown_appcontext
#ferme la connexion a la fin de la requete
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()

#donne un curseur compatible entre les deux db
def get_cursor(db):
    if server_config.get('db_type') == 'mysql':
        return db.cursor(dictionary=True)
    return db.cursor()

#verifie si l'utilisateur est connecte
def login_required(f):
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user' not in session:
            return redirect(url_for('login'))
        return f(*args, **kwargs)
    return decorated_function
```

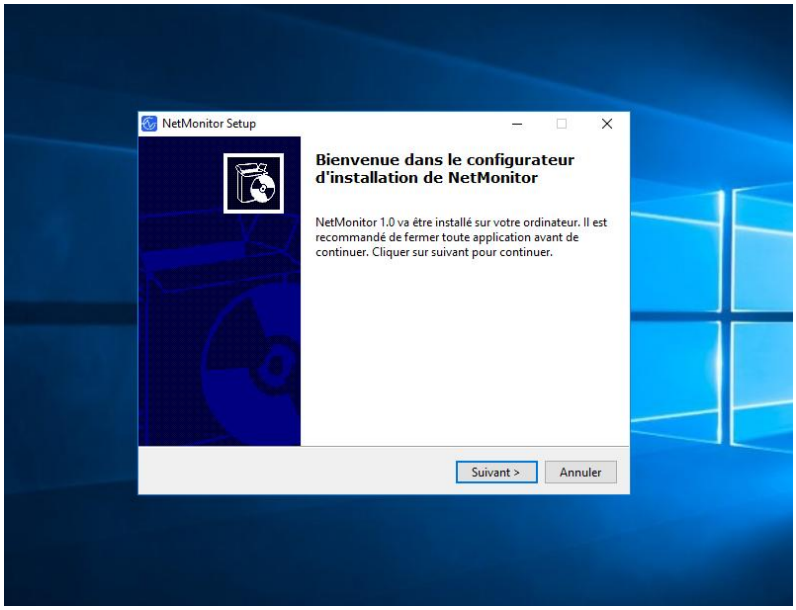
Il charge ensuite les routes API dans lesquelles il utilise ces variables

```python
#recoit les donnees de l'agent
@app.route('/update', methods=['POST'])
def update():
    data = request.json
    hostname = data.get('hostname', '').strip()
    if not hostname:
        return jsonify({"status": "error", "message": "Hostname missing"}), 400

    #ne pas compter les différences de majuscules comme hôtes différents
    real_hostname = next((k for k in agents.keys() if k.lower() == hostname.lower()), hostname)

    data['last_seen'] = time.time()

    if real_hostname not in history:
        history[real_hostname] = {"cpu": [], "ram": [], "storage": []}

    for key in ["cpu", "ram", "storage"]:
        val = data.get(f"{key}_usage", 0)
        history[real_hostname][key].append(val)
        if len(history[real_hostname][key]) > 10: history[real_hostname][key].pop(0)

    #fusion avec les données existantes pour éviter de perdre les services ou réglages
    if real_hostname in agents and not data.get('services') and agents[real_hostname].get('services'):
        data['services'] = agents[real_hostname]['services']

    #appliquer les réglages persistants
    saved_settings = next((v for k, v in all_agent_settings.items() if k.lower() == hostname.lower()), None)
    data['alert_settings'] = saved_settings or {
        "cpu_threshold": 90,
        "ram_threshold": 90,
        "disk_threshold": 90,
        "cpu_enabled": True,
        "ram_enabled": True,
        "disk_enabled": True,
        "monitored_services": []
    }

    agents[real_hostname] = data
    return jsonify({"status": "success"})
```

Par exemple, ici on voit la route « /update » qui est celle que l'agent utilise pour envoyer ses données, le serveur vérifie d'abord si un hostname est présent dans ces données et le transforme pour éviter les problèmes de majuscules. L'application vérifie ensuite si elle a déjà reçu des informations de ce nom d'hôte et réagit en fonction. Elle finit par ajouter les paramètres par défaut à l'agent s'il en à pas et finit par affecter tout cela à l'agent et renvoie à l'agent que les informations ont bien été traitées.

3. Explication des fonctionnalités

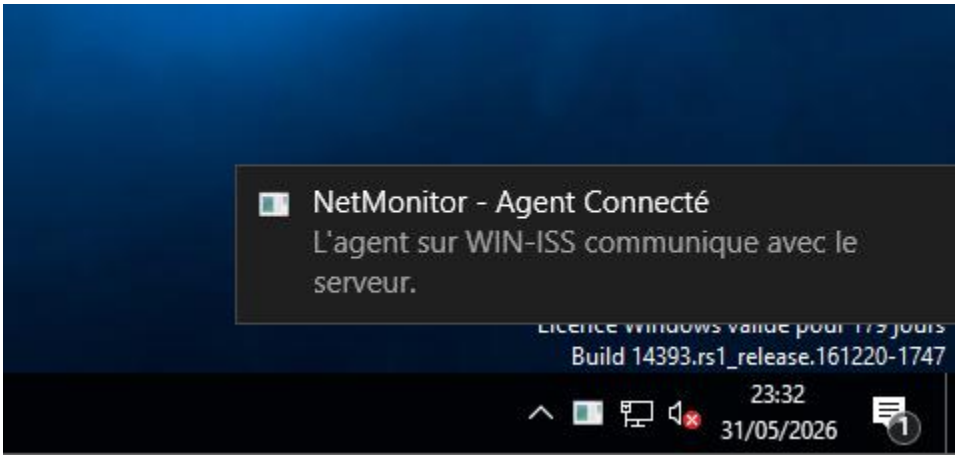
Dès que le serveur est allumé et connecté au réseau, il faut installer l'agent. Si l'hôte est sur Windows, alors on peut utiliser l'installeur interactif. Sinon, il faut configurer le service de démarrage automatique manuellement



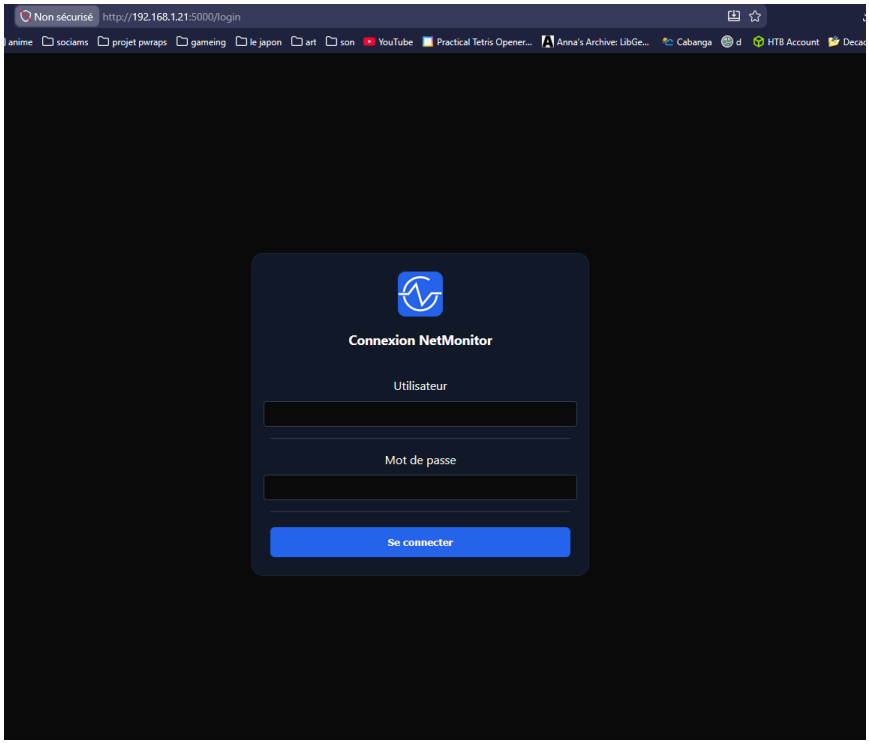

```
  GNU nano 8.4                          /etc/rc.local *
/usr/bin/python3 /home/victor2/monitoring/agent/agent.py &
```

Une fois l’agent installé, L’application de configuration se lance, il suffit d’y entrer l’IP du serveur et le délai d’envoi. L’Ip du serveur est affichée sur le boîtier.

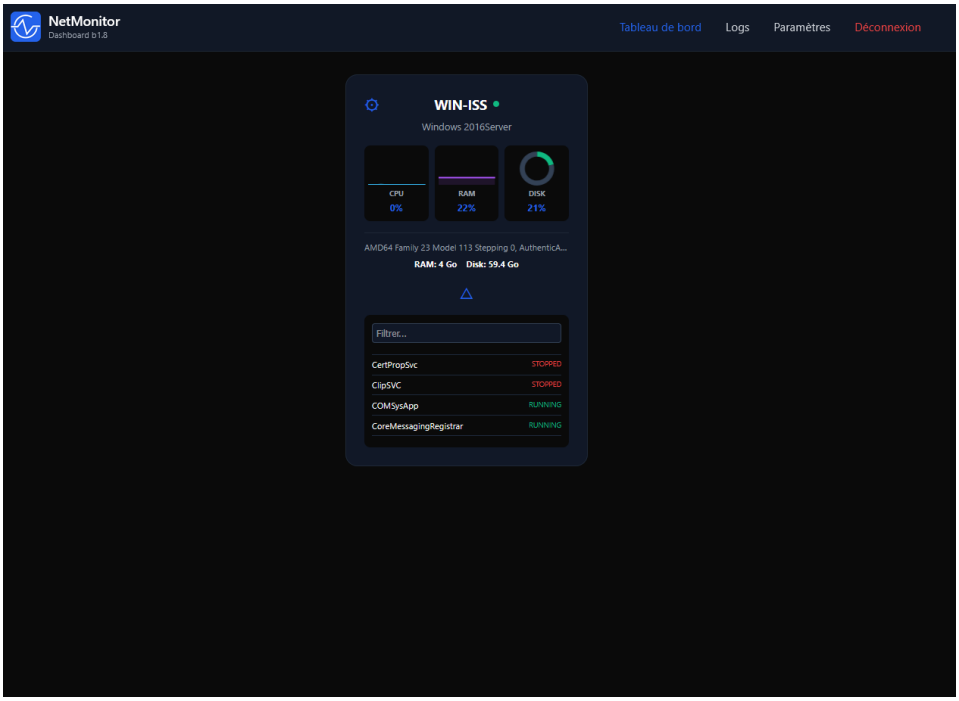
Une fois l'agent configuré, il suffit de redémarrer l'hôte et l'agent se lance automatiquement au démarrage et si la configuration est correcte, on recoit une notification de confirmation.



Une fois l'appareil connecté, on peut accéder au Dashboard en entrant l'IP du serveur ainsi que son port dans n'importe quel navigateur au sein du réseau.

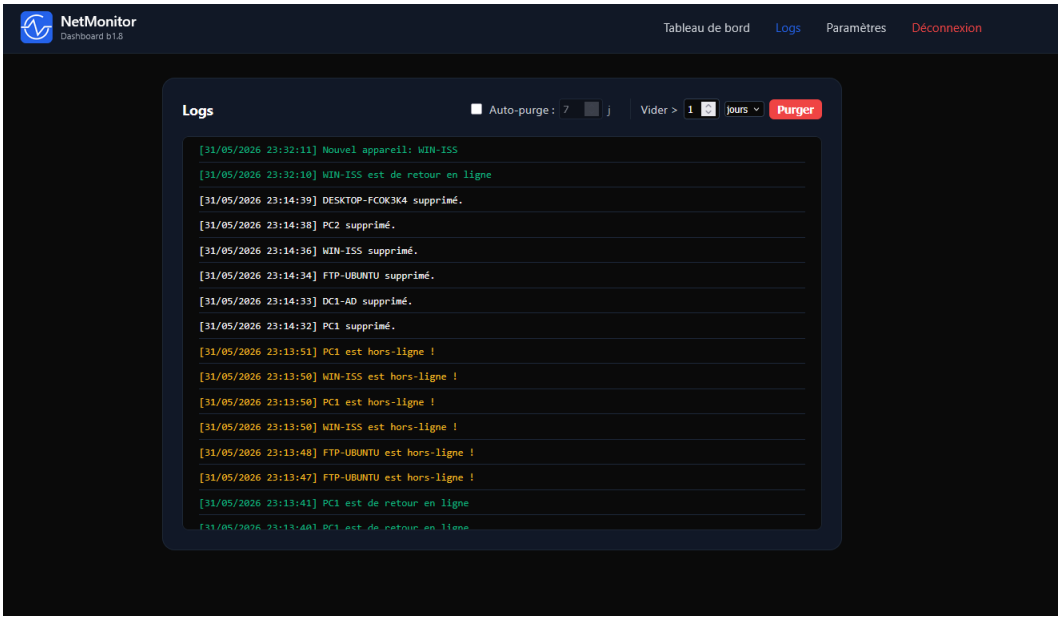


L’Hote se trouve maintenant sur le Dashboard et on peut voir son état.

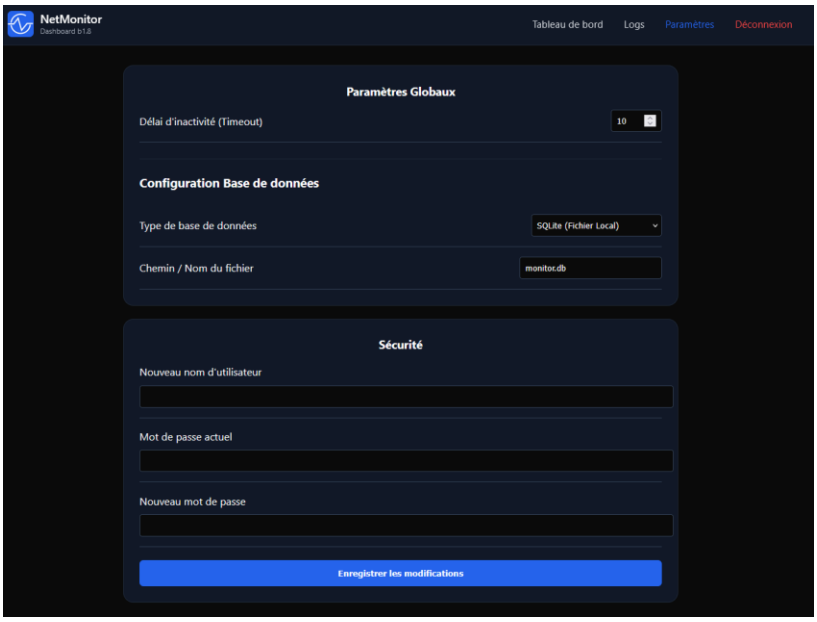


Sur le dashboard, on a acces aux logs de tous les événements importants comme l’ajout et suppression d’appareils, alertes, changements de paramétres, de mot de passe et de database.

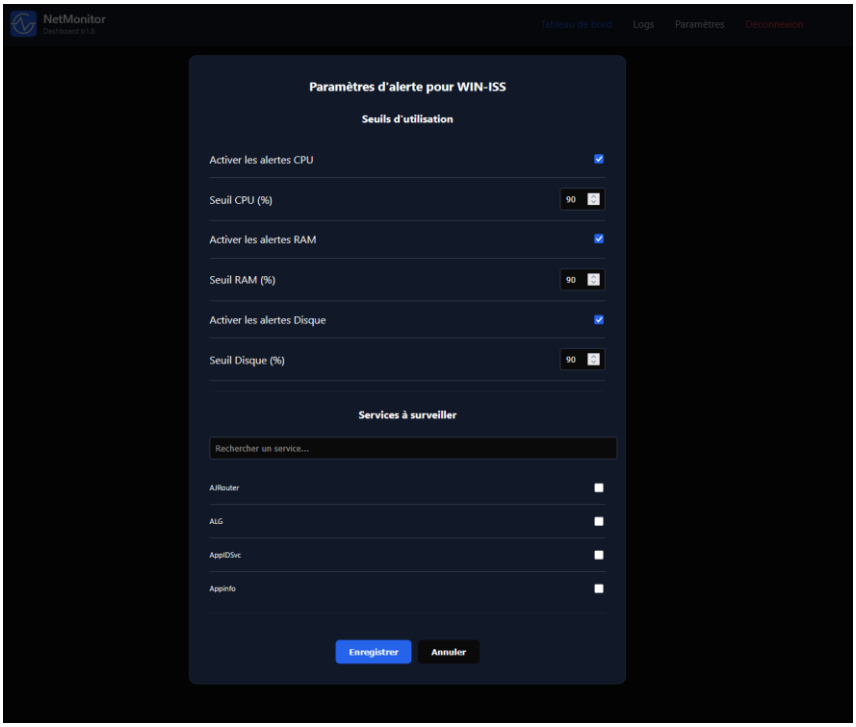
On a aussi l’option d’effacer les logs en fonction de leur date, manuellement ou automatiquement.



Dans les paramétres globaux, on peut changer la base de données qu'on utilise, le délai d'inactivité avant qu'un hôte soit considéré comme « hors ligne » et les identifiants du compte utilisé pour se connecter.



Et enfin, sur chaque appareil, on peut définir quoi surveiller, à partir de quand et quels services surveiller.



4. Difficultés Rencontrées

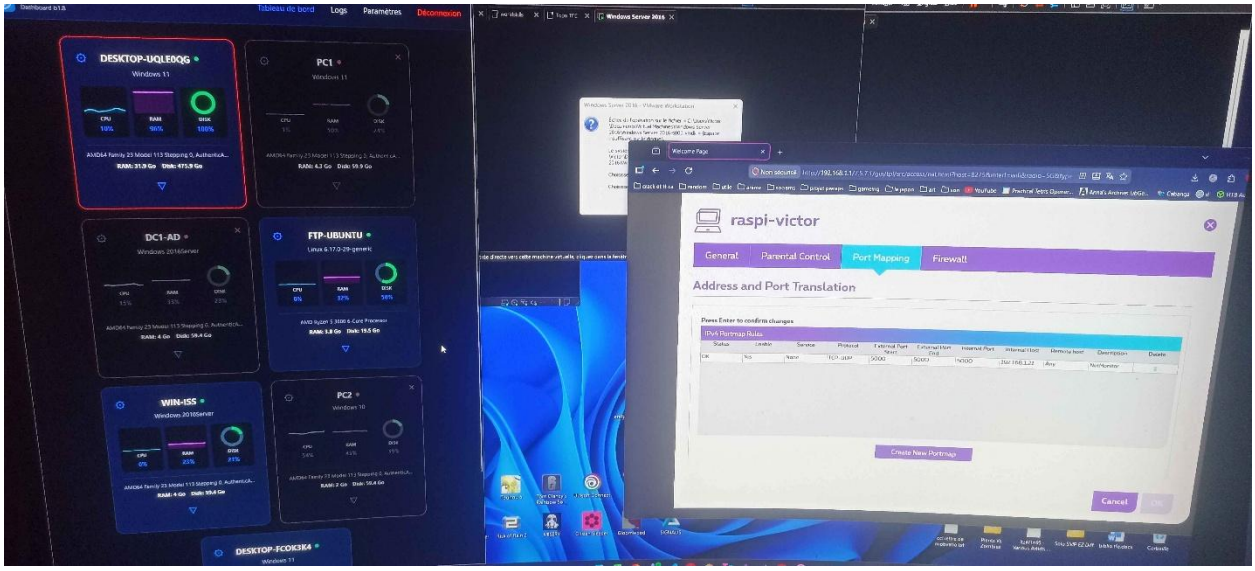
Durant la conception du TFE, j'ai rencontré plusieurs problèmes auxquels j'ai dû remédier.

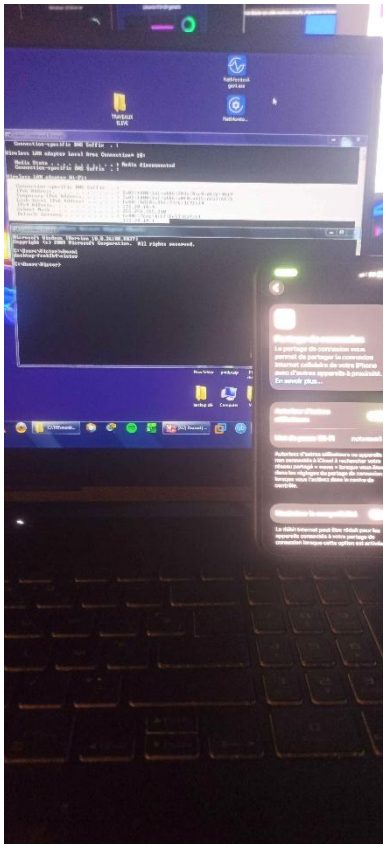
Difficulté	**Solution Apportée**
Gestion des données (Les données des agents restaient dans la mémoire vive (RAM) du serveur donc un redémarrage aurait effacé tout les paramétres & les logs).	Enregistrer les paramétres dans des fichiers locaux et donner l'option d'utiliser des bases de données externes pour les logs, assurant la redondance de ces derniers.
Installation de l'agent compliquée (besoin d'installer les bibliothèques & python) et besoin de relancer l'agent manuellement à chaque redémarrage de l'hôte.	Compacter le programme et ses bibliothèques en un fichier exécutable et l'ajouter à un installeur interactif afin de le mettre automatiquement dans le dossier « Démarrage » du système.
Accéder à l'application en dehors du réseau local	Configurer le « Port Forwarding » sur mon routeur afin de faire communiquer le serveur avec un agent en dehors du réseau.
Faire fonctionner les graphiques hors-ligne (Les graphiques était téléchargés depuis internet)	Installer localement les graphiques (fichier Chart.JS)

VIII. Tests & validation

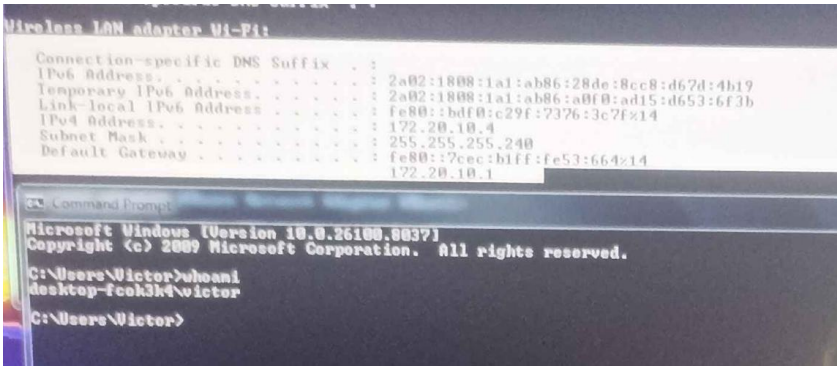
1. Implémentation au sein de la topologie exemple

Pour s'assurer que le projet fonctionne comme prévu, j'ai testé le projet en configurant l'environnement décrit sur la topologie.





On peut apercevoir les 7 appareils présents sur le Dashboard et le port-Forwarding configuré pour connecter le 7e appareil qui est en dehors du réseau avec le serveur.



2. Résultats obtenus

Tout fonctionnait parfaitement, les agents envoyaient tous leurs informations rapidement et les appareils surchargés (par exemple, mon desktop qui faisait tourner les 6 VM était affiché en rouge vu que son utilisation de mémoire RAM dépassait 96%). Aussi, pendant que je testais la topologie, mon desktop est tombé à court de stockage ce qui a fait arrêter 3 des machines virtuelles qui sont aussitôt apparues comme « hors-ligne » sur mon Dashboard, prouvant donc que mon projet est bel et bien fonctionnel.

3. Bugs rencontrés

En testant mon application j'ai rencontré plusieurs bugs, la plupart étant un problème dans la gestion des données coté serveur. Par exemple je ne pouvais pas décocher un certain service sur un hôte linux car mon projet l'avait scanné en double, redémarrer le serveur à réglé cela mais j'ai passé pas mal de temps à essayer d'en trouver la source dans le code.

A part des petits bugs comme ceux-ci, je n'ai pas vraiment rencontré de bugs qui m'ont vraiment marqué.

IX. Conclusion

1. Résumé du projet

Pour conclure, ce projet est une application de monitoring réseau constitué d'une partie serveur et d'une autre partie agent, la partie agent étant une application python qui récolte les données d'un ordinateur comme son stockage restant, l'utilisation de la mémoire vive, et bien plus… et qui les envoie à un serveur qui lui, les récupéré et les formatte sous forme de graphiques et de nombres visibles sur un dashboard accessible sur n'importe quel navigateur au sein du réseau et en dehors si le port Forwarding est configuré.

Sur ce dashboard, l'administrateur réseau peut configurer des alertes si un service venait à s'arrêter ou si une utilisation anormale d'un des composants était détectée. Le serveur garde une trace de tout ces éléments sous forme d'un log qui se trouve soit localement sous forme d'un fichier .db ou sous forme d'une base de données externe.

2. Objectifs Atteinds ?

Personnellement, j'estime que j'ai réalisé le travail qu'on m'a demandé de réaliser, le serveur et l'agent fonctionnent et je pense qu'il y a tout ce qu'il faut de disponible sur le dashboard. Donc j'estime avoir atteind mes objectifs.

3. Compétences acquises

Pendant le développement de ce projet, j'ai appris à :

- Réaliser une application à base d'API REST
- Créer un fichier .exe à base d'un programme python
- Utiliser de nouveaux logiciels (PrusaSlicer, WinSCP, InstallForge)
- Configurer le Port Forwarding
- Règles de base pour sécuriser une application
- Correctement utiliser les outils d'intelligence artificielle

4. Améliorations Possibles

Vu que j'ai commencé à travailler sur cette application tard dans l'année, j'ai dû finir certaines choses rapidement et donc omettre des fonctionnalités que j'aurais aimé implémenter comme

l’envoi d’alertes par mail via SMTP, rajouter la possibilité d’ajouter des appareils en agentless comme des routeurs (vérification par ICMP), Créer un script d’installation d’agent pour Linux, et un paquet d’autre choses.

X. Bibliographie

Prosseek. (2010, 23 juin). *How to get the system info with Python?* Stack Overflow. https://stackoverflow.com/questions/3103178/how-to-get-the-system-info-with-python

- *Apprendre quoi utiliser et comment récupérer les informations de l'ordinateur avec python*

How to send HTTP requests with Python. (2025, 27 juin). IONOS Digital Guide. https://www.ionos.com/digitalguide/websites/web-development/python-requests-module/

- *Site web expliquant la bibliothèque python « requests », utilisé pour savoir comment envoyer des demandes HTTP avec python*

JSON Syntax. (s. d.). W3Schools.com. https://www.w3schools.com/js/js_json_syntax.asp

- *Site utilisé pour savoir comment formatter en JSON pour savoir comment écrire le fichier de configuration.*

Quickstart — Flask Documentation (3.1.X). (s. d.). PalletProjects. https://flask.palletsprojects.com/en/stable/quickstart/#a-minimal-application

- *Site utilisé pour apprendre les bases de flask et comment mettre des variables python dans une page HTML*

Qwen3-Max (2026). Alibaba Cloud. https://chat.qwen.ai/

- *Aide avec le code (plan général du code de l'application), code de la page web Dashboard », juger les avantages et inconvénients de l'utilisation de SSH ou HTTP pour le projet.*

Burnel, F., & Burnel, F. (2023, 17 juillet). *Installer et configurer IIS 10 sur Windows Server 2022 | IT-Connect. IT-Connect.* *https://www.it-connect.fr/installer-et-configurer-iis-10-sur-windows-server-2022/*

- *Site web avec un guide utilisé pour apprendre la configuration du serveur ISS (ftp&web) sur Windows Server.*

GordonBpdZenith. (2014, 25 novembre). *Simulate WAN / Route Between 2 Virtual networks (Vmware).* Server Fault. *https://serverfault.com/questions/646978/simulate-wan-route-between-2-virtual-networks-vmware*

- *Recherches sur comment simuler deux réseaux connectés via internet avec des machines virtuelles VMWare.*

Claude Sonnet 4.6 (2026). Anthropic. https://claude.ai/

- *Aide pour trouver quoi utiliser pour réaliser certaines tâches comme l'installeur interactif, aide pour voir quel VPN utiliser pour simuler le port-Forwarding*

Gemini 3 (2026). Google. https://gemini.google.com

- *Utilisé pour générer la plupart du code HTML, CSS, et JS du projet & du site WEB comme base.*

Zhou. (2023, 24 juillet). How to install and use the Chart.js package locally? Stack Overflow. https://stackoverflow.com/questions/76753442/how-to-install-and-use-the-chart-js-package-locally

- *Comment stocker les graphiques localement*

Solicus. (2025). *InstallForge Documentation. InstallForge.* Consulté le 28 juin 2026, sur *https://docs.installforge.net/*

- *Documentation d'InstallForge utilisée pour voir comment faire démarrer une application au lancement du PC*

Dexter Industries. (s.d.). Five ways to run a program on your Raspberry Pi at startup. Dexter Industries. https://www.dexterindustries.com/howto/run-a-program-on-your-raspberry-pi-at-startup/

- *Voir comment lancer des applications au démarrage d'un micro-ordinateur Raspberry PI.*

Proximus. (s.d.). *Redirection des ports sur votre b-box ou Internet Box*. Proximus. Consulté le 30 juin 2026 *https://www.proximus.be/support/fr/id_sfaqr_ports_mapping/particuliers/support/internet/internet-a-la-maison/parametres-avances/redirection-des-ports-sur-votre-b-box-ou-internet-box.html*

- *Apprendre comment ouvrir un port sur les routeurs Proximus.*

XI. Annexes

Code de l’agent Windows

```python
import time
import requests
import psutil
import platform
import socket
import json
import os
from plyer import notification

#lit l'adresse du serveur et le delai dans le fichier json
def get_config():
    try:
        base = os.path.expanduser('~')
        config_path = os.path.join(base, 'NetMonitor', 'network_config.json')

        with open(config_path, "r") as f:
            data = json.load(f)
            config = data[0] if isinstance(data, list) else data
            #construit l'url avec l'ip et le port du fichier json
            url = f"http://{config['ip']}:{config.get('port', '5000')}/update"
            interval = float(config['interval'])
            return url, interval
    except Exception:
        return "http://127.0.0.1:5000/update", 3

def collect_services():
    services_dict = {}
    #récupère les services
    for proc in psutil.process_iter(['name', 'status']):
        try:
            name, status = proc.info['name'], proc.info['status']
            if name not in services_dict or status == 'running':
                services_dict[name] = status
        #éviter d'arreter l'agent si on trv pas un service ou autre raison
        except (psutil.NoSuchProcess, psutil.AccessDenied):
            continue
    return [{"name": n, "status": s} for n, s in services_dict.items()]

#boucle d'envoi des donnees
def main():
    SERVER_URL, INTERVAL = get_config()
    #print(f"Monitoring active. Target: {SERVER_URL}")

    #initialisation du compteur CPU (évite d'envoyer 0.0 au premier message)
    psutil.cpu_percent(interval=None)

    last_error_notification = 0
    was_disconnected = True
    last_service_update = 0

    while True:
        try:
            mem, disk = psutil.virtual_memory(), psutil.disk_usage('/')
            current_time = time.time()

            #on collecte les services immédiatement au démarrage, puis toutes les 5s
            services_list = []
            if current_time - last_service_update >= 5:
                services_list = collect_services()
                last_service_update = current_time

            #identification du processeur
            cpu_info = platform.processor() or "Inconnu"
            try:
                with open("/proc/cpuinfo", "r") as f:
                    for line in f:
                        if "model name" in line:
                            cpu_info = line.split(":")[1].strip()
                            break
            except Exception:
                pass

            #récupération des autres données
            stats = {
                "os": f"{platform.system()} {platform.release()}",
                "processor": cpu_info,
                "cpu_usage": psutil.cpu_percent(interval=None),
                "ram_usage": mem.percent,
                "ram_total": round(mem.total / (1024**3), 1),
                "storage_usage": disk.percent,
                "storage_total": round(disk.total / (1024**3), 1),
                "hostname": socket.gethostname(),
                "services": services_list,
                "interval": INTERVAL
            }

            res = requests.post(SERVER_URL, json=stats, timeout=5)

            #debug
            #print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Pushed (Status: {res.status_code})")

            #notification de connexion initiale ou de reconnexion
            if was_disconnected:
                notification.notify(
                    title="NetMonitor - Agent Connecté",
                    message=f"L'agent sur {socket.gethostname()} communique avec le serveur.",
                    timeout=10
                )
                was_disconnected = False

            #réinitialise le délai de notification si la connexion réussit pr eviter le spam
            last_error_notification = 0

        except Exception as e:
            #renvoi d'une notification système toutes les 2 minutes en cas d'échec
            current_time = time.time()
            if current_time - last_error_notification >= 120:
                notification.notify(
                    title="NetMonitor - Erreur Agent",
                    message=f"Impossible de joindre le serveur de monitoring sur {SERVER_URL}.",
                    timeout=10
                )
                last_error_notification = current_time
                was_disconnected = True

            #debug
            #print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Transmission error: {e}")

        time.sleep(INTERVAL)

if __name__ == "__main__":
    main()
```

Code de l’agent Linux

```python
import time
import requests
import psutil
import platform
import socket
import json
import os
from plyer import notification

#lit l'adresse du serveur et le delai dans le fichier json
def get_config():
    try:
        base = os.getenv('LOCALAPPDATA') or os.path.expanduser('~')
        config_path = os.path.join(base, 'NetMonitor', 'network_config.json')

        with open(config_path, "r") as f:
            data = json.load(f)
            config = data[0] if isinstance(data, list) else data
            #construit l'url avec l'ip et le port du fichier json
            url = f"http://{config['ip']}:{config.get('port', '5000')}/update"
            interval = float(config['interval'])
            return url, interval
    except Exception:
        return "http://127.0.0.1:5000/update", 3

def collect_services():
    services_dict = {}
    #récupère les services
    for service in psutil.win_service_iter():
        try:
            services_dict[service.name()] = service.status()
        #éviter d'arreter l'agent si on trv pas un service ou autre raison
        except (psutil.NoSuchProcess, psutil.AccessDenied, OSError):
            continue
    return [{"name": n, "status": s} for n, s in services_dict.items()]

#boucle d'envoi des donnees
def main():
    SERVER_URL, INTERVAL = get_config()
    #print(f"Monitoring active. Target: {SERVER_URL}")

    #initialisation du compteur CPU (évite d'envoyer 0.0 au premier message)
    psutil.cpu_percent(interval=None)

    last_error_notification = 0
    was_disconnected = True
    last_service_update = 0

    while True:
        try:
            mem, disk = psutil.virtual_memory(), psutil.disk_usage('C:')
            current_time = time.time()

            #on collecte les services immédiatement au démarrage, puis toutes les 5s
            services_list = []
            if current_time - last_service_update >= 5:
                services_list = collect_services()
                last_service_update = current_time

            #identification du processeur
            cpu_info = platform.processor() or "Inconnu"

            #récupération des autres données
            stats = {
                "os": f"{platform.system()} {platform.release()}",
                "processor": cpu_info,
                "cpu_usage": psutil.cpu_percent(interval=None),
                "ram_usage": mem.percent,
                "ram_total": round(mem.total / (1024**3), 1),
                "storage_usage": disk.percent,
                "storage_total": round(disk.total / (1024**3), 1),
                "hostname": socket.gethostname(),
                "services": services_list,
                "interval": INTERVAL
            }

            res = requests.post(SERVER_URL, json=stats, timeout=5)

            #debug
            #print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Pushed (Status: {res.status_code})")

            #notification de connexion initiale ou de reconnexion
            if was_disconnected:
                notification.notify(
                    title="NetMonitor - Agent Connecté",
                    message=f"L'agent sur {socket.gethostname()} communique avec le serveur.",
                    timeout=10
                )
                was_disconnected = False

            #réinitialise le délai de notification si la connexion réussit pr eviter le spam
            last_error_notification = 0

        except Exception as e:
            #renvoi d'une notification système toutes les 2 minutes en cas d'échec
            current_time = time.time()
            if current_time - last_error_notification >= 120:
                notification.notify(
                    title="NetMonitor - Erreur Agent",
                    message=f"Impossible de joindre le serveur de monitoring sur {SERVER_URL}.",
                    timeout=10
                )
                last_error_notification = current_time
                was_disconnected = True

            #debug
            #print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Transmission error: {e}")

        time.sleep(INTERVAL)

if __name__ == "__main__":
    main()
```

Code de l’outil de configuration

```python
import tkinter as tk
from tkinter import messagebox
import json
import os
#initialise la fenetre de configuration
root = tk.Tk()
root.title("NetMonitor - Configuration de l'agent")
# définit la taille de la fenêtre
root.geometry("350x170")

# Dossier de données de l'agent (AppData)
def get_agent_data_dir():
    base = os.getenv('LOCALAPPDATA') or os.path.expanduser('~')
    path = os.path.join(base, 'NetMonitor')
    os.makedirs(path, exist_ok=True)
    return path

DATA_DIR = get_agent_data_dir()
CONFIG_FILE_PATH = os.path.join(DATA_DIR, "network_config.json")

# Icône (chemin absolu par rapport au script pour l'installation)
icon_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), "logo-tfe.ico")
if os.path.exists(icon_path):
    try:
        root.iconbitmap(icon_path)
    except tk.TclError:
        pass # Ignore l'erreur si l'icône ne peut pas être chargée
# cadre principal qui s'etend pour remplir la fenetre
frame = tk.Frame(root, padx=10, pady=10, relief="groove", borderwidth=2)
frame.pack(expand=True, fill="both")

# configuration des colonnes pour que le contenu s'etire
frame.columnconfigure(1, weight=1)

#texte pour afficher le statut de l'enregistrement
status_label = tk.Label(frame, text="", fg="green")
status_label.grid(row=4, column=0, columnspan=3)

#sauvegarde les reglages dans le fichier
def save_to_file():
    try:
#prepare les donnees de configuration
        config_data = [{
            "ip": ip.get(),
            "port": port.get(),
            "interval": float(interval.get())
        }]
#enregistre les donnees dans le fichier json
        with open(CONFIG_FILE_PATH, "w") as f:
            json.dump(config_data, f, indent=4)
        status_label.config(text="")
        messagebox.showinfo("Succès", "Configuration enregistrée !")

#gestion d'erreur
    except ValueError:
        status_label.config(text="Erreur: L'intervalle doit être un nombre.", fg="red")
    except IOError as error:
        messagebox.showerror("Erreur", f"Erreur d'écriture : {error}")
        status_label.config(text="Erreur d'écriture.", fg="red")
    except Exception as error:
        messagebox.showerror("Erreur", f"Une erreur inattendue est survenue : {error}")
        status_label.config(text="Erreur inattendue.", fg="red")

#titres des champs
tk.Label(frame, text="IP du serveur").grid(row= 0, column=0, sticky="e", pady=5)
tk.Label(frame, text="Port du serveur").grid(row=1, column=0, sticky="e", pady=5)
tk.Label(frame, text="Intervalle d'envoi").grid(row=2, column=0, sticky="e", pady=5)

#zone de saisie pour l'adresse ip
ip = tk.Entry(frame, width=25)
ip.grid(row=0, column=1, columnspan=1, sticky="w", pady=5)

#zone de saisie pour le port
port = tk.Entry(frame, width=10)
port.grid(row=1, column=1, sticky='w', columnspan=1, pady=5)
port.insert(0, "5000")

#zone de saisie pour le delai
interval_frame = tk.Frame(frame)
interval_frame.grid(row=2, column=1, sticky="w", pady=5)
interval = tk.Entry(interval_frame, width=10)
interval.pack(side="left")
tk.Label(interval_frame, text="s").pack(side="left")

#bouton pour valider les changements
btn = tk.Button(frame, text="Enregistrer", command=save_to_file)
btn.grid(row=5, column=0, columnspan=4, pady=10, sticky="s")

root.mainloop()
```

Code du serveur (1/3)

```python
from flask import Flask, render_template, request, jsonify, session, redirect, url_for, g
from werkzeug.security import generate_password_hash, check_password_hash
import time
import sqlite3
import functools
import json
import os
try:
    import mysql.connector # Nécessite 'pip install mysql-connector-python'
except ImportError:
    mysql = None

app = Flask(__name__)
#cle de securite pour les sessions
app.secret_key = os.environ.get('FLASK_SECRET_KEY', os.urandom(24))

#definit le répertoire de base pour les fichiers de configuration et la base de données
SERVER_BASE_DIR = os.path.dirname(os.path.abspath(__file__))

CONFIG_FILE = os.path.join(SERVER_BASE_DIR, 'server_config.json')
AGENT_SETTINGS_FILE = os.path.join(SERVER_BASE_DIR, 'agent_alert_settings.json')

#chargement des config
def load_server_config():
    if os.path.exists(CONFIG_FILE):
        try:
            with open(CONFIG_FILE, 'r') as f:
                return json.load(f)
        except: pass
    return {
        "db_type": "sqlite",
        "sqlite_db": "monitor.db",
        "mysql_host": "localhost",
        "mysql_user": "root",
        "mysql_pass": "",
        "mysql_name": "monitor_db",
        "offline_timeout": 10,
        "log_retention_days": 7,
        "log_cleanup_enabled": True
    }
#enregistrement des config
def save_server_config(config):
    with open(CONFIG_FILE, 'w') as f:
        json.dump(config, f, indent=4)

#chargement des config des alertes agent
def load_agent_alert_settings():
    if os.path.exists(AGENT_SETTINGS_FILE):
        try:
            with open(AGENT_SETTINGS_FILE, 'r') as f:
                return json.load(f)
        except: pass
    return {}
#enregistrement des config alertes agent
def save_agent_alert_settings(settings):
    with open(AGENT_SETTINGS_FILE, 'w') as f:
        json.dump(settings, f, indent=4)

#chargement de la configuration du serveur
server_config = load_server_config()
#chargement de la configuation des alertes des agents
all_agent_settings = load_agent_alert_settings()

def get_pl():
    # Marqueur de paramètre dynamique (? pour SQLite, %s pour MySQL)
    return '?' if server_config.get('db_type') == 'sqlite' else '%s'

#données actuelles des agents
agents = {}
#historique des mesures
history = {}

#recupere la connexion sqlite
def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        if server_config.get('db_type') == 'mysql':
            if mysql is None:
                raise ImportError("Le module 'mysql-connector-python' est requis pour MySQL.")
            db = g._database = mysql.connector.connect(
                host=server_config.get('mysql_host'),
                user=server_config.get('mysql_user'),
                password=server_config.get('mysql_pass'),
                database=server_config.get('mysql_name')
            )
        else:
            # Récupère le chemin et utilise "monitor.db" s'il est vide ou mal configuré
            db_path = server_config.get("sqlite_db", "monitor.db").strip()
            if not db_path:
                db_path = "monitor.db"

            # Force l'utilisation d'un chemin absolu basé sur le dossier NetMonitor dans AppData
            if not os.path.isabs(db_path):
                db_path = os.path.join(SERVER_BASE_DIR, db_path)

            # S'assure que le répertoire parent existe (évite l'erreur si le dossier est manquant)
            os.makedirs(os.path.dirname(db_path), exist_ok=True)

            db = g._database = sqlite3.connect(db_path)
            db.row_factory = sqlite3.Row
    return db

def cleanup_old_logs(days):
    # Supprime les logs plus vieux que le nombre de jours spécifié
    if days <= 0: return 0
    limit_time = time.time() - (days * 86400)
    # Format ISO pour correspondre au stockage JavaScript (YYYY-MM-DD...)
    limit_iso = time.strftime('%Y-%m-%dT%H:%M:%S', time.gmtime(limit_time))

    PL = get_pl()
    db = get_db()
    cursor = get_cursor(db)
    cursor.execute(f"DELETE FROM logs WHERE timestamp < {PL}", (limit_iso,))
    db.commit()
    return cursor.rowcount

@app.teardown_appcontext
def close_connection(exception):
    #ferme la connexion a la fin de la requete
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()

def get_cursor(db):
    # Retourne un curseur compatible (dictionnaire pour MySQL)
    if server_config.get('db_type') == 'mysql':
        return db.cursor(dictionary=True)
    return db.cursor()


def login_required(f):
    #verifie si l'utilisateur est connecte
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user' not in session:
            return redirect(url_for('login'))
        return f(*args, **kwargs)
    return decorated_function

@app.route('/')
@login_required
def index():
```

Code du serveur (2/3)

```python
@app.route('/')
@login_required
def index():
    #affiche la page du tableau de bord
    return render_template('index.html')

def init_db():
    #initialise les tables et le compte admin
    with app.app_context():
        PL = get_pl()
        db = get_db()
        cursor = get_cursor(db)
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS logs (
                timestamp VARCHAR(255),
                message TEXT,
                type VARCHAR(50))''')
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                username VARCHAR(255) PRIMARY KEY,
                password TEXT)''')

        cursor.execute("SELECT COUNT(*) as total FROM users")
        result = cursor.fetchone()
        count = result['total'] if isinstance(result, dict) else result[0]
        if count == 0:
            hashed_pw = generate_password_hash("admin")
            cursor.execute(f"INSERT INTO users (username, password) VALUES ({PL}, {PL})", ("admin", hashed_pw))
            db.commit()

#recoit les donnees de l'agent
@app.route('/update', methods=['POST'])
def update():
    data = request.json
    hostname = data.get('hostname', '').strip()
    if not hostname:
        return jsonify({"status": "error", "message": "Hostname missing"}), 400

    #ne pas compter les différences de majuscules comme hôtes différents
    real_hostname = next((k for k in agents.keys() if k.lower() == hostname.lower()), hostname)

    data['last_seen'] = time.time()

    if real_hostname not in history:
        history[real_hostname] = {"cpu": [], "ram": [], "storage": []}

    for key in ["cpu", "ram", "storage"]:
        val = data.get(f"{key}_usage", 0)
        history[real_hostname][key].append(val)
        if len(history[real_hostname][key]) > 10: history[real_hostname][key].pop(0)

    #fusion avec les données existantes pour éviter de perdre les services ou réglages
    if real_hostname in agents and not data.get('services') and agents[real_hostname].get('services'):
        data['services'] = agents[real_hostname]['services']

    #appliquer les réglages persistants
    saved_settings = next((v for k, v in all_agent_settings.items() if k.lower() == hostname.lower()), None)
    data['alert_settings'] = saved_settings or {
        "cpu_threshold": 90,
        "ram_threshold": 90,
        "disk_threshold": 90,
        "cpu_enabled": True,
        "ram_enabled": True,
        "disk_enabled": True,
        "monitored_services": []
    }

    agents[real_hostname] = data
    return jsonify({"status": "success"})

@app.route('/api/stats')
@login_required
def api_stats():
    #envoie les stats actuelles en json
    return jsonify({
        "agents": agents,
        "history": history,
        "current_time": time.time()
    })

@app.route('/api/delete/<hostname>', methods=['DELETE'])
@login_required
def delete_agent(hostname):
    #supprime un agent du suivi et ses reglages persistants
    # Recherche insensible à la casse pour la suppression
    target_key = next((k for k in agents.keys() if k.lower() == hostname.lower()), None)
    if target_key:
        del agents[target_key]
        if target_key in history: del history[target_key]
        if target_key in all_agent_settings: del all_agent_settings[target_key]
        save_agent_alert_settings(all_agent_settings)

    return jsonify({"status": "success"})

@app.route('/api/alert_settings/<hostname>', methods=['POST'])
@login_required
def update_alert_settings(hostname):
    settings = request.json
    hostname = hostname.strip()

    # Met à jour la config persistante (en gardant la casse existante si possible)
    existing_key = next((k for k in all_agent_settings.keys() if k.lower() == hostname.lower()), hostname)
    all_agent_settings[existing_key] = settings
    save_agent_alert_settings(all_agent_settings)

    # Met à jour l'agent en mémoire
    for agent_name in agents:
        if agent_name.lower() == hostname.lower():
            agents[agent_name]['alert_settings'] = settings

    return jsonify({"status": "success", "message": f"Alert settings updated for {hostname}"})

@app.route('/api/logs', methods=['POST'])
@login_required
def add_log_entry():
    #enregistre un nouveau log
    log_data = request.json
    if not all(k in log_data for k in ('timestamp', 'message', 'type')):
        return jsonify({"status": "error", "message": "Missing log data"}), 400

    PL = get_pl()
    db = get_db()
    cursor = get_cursor(db)
    cursor.execute(f"INSERT INTO logs (timestamp, message, type) VALUES ({PL}, {PL}, {PL})",
                   (log_data['timestamp'], log_data['message'], log_data['type']))
    db.commit()
    return jsonify({"status": "success"})

@app.route('/api/logs', methods=['GET'])
@login_required
def get_logs():
    #lit les logs dans la base
    limit = request.args.get('limit', 100, type=int)
    PL = get_pl()
    db = get_db()
    cursor = get_cursor(db)
    cursor.execute(f"SELECT timestamp, message, type FROM logs ORDER BY timestamp DESC LIMIT {PL}", (limit,))
    return jsonify([dict(row) for row in cursor.fetchall()])
```

Code du serveur (3/3)

```python
@app.route('/api/logs/cleanup', methods=['POST'])
@login_required
def api_cleanup_logs():
    # Route pour le nettoyage (auto ou manuel avec paramètres)
    data = request.json or {}
    days = data.get('days')

    if days is None: # Cas de l'auto-nettoyage
        if not server_config.get('log_cleanup_enabled', True):
            return jsonify({"status": "skipped", "message": "Auto-cleanup disabled"})
        days = server_config.get('log_retention_days', 7)

    deleted_count = cleanup_old_logs(days)
    return jsonify({"status": "success", "deleted": deleted_count})

@app.route('/login', methods=['GET', 'POST'])
def login():
    #gere la connexion utilisateur
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        PL = get_pl()
        db = get_db()
        cursor = get_cursor(db)
        cursor.execute(f"SELECT password FROM users WHERE username = {PL}", (username,))
        user = cursor.fetchone()

        if user and check_password_hash(user['password'], password):
            session['user'] = username
            return redirect(url_for('index'))

        return render_template('login.html', error="Identifiants invalides")
    return render_template('login.html')

@app.route('/logout')
def logout():
    #deconnecte l'utilisateur
    session.pop('user', None)
    return redirect(url_for('login'))

@app.route('/api/change_password', methods=['POST'])
@login_required
def change_password():
    #change le nom d'utilisateur et/ou le mot de passe admin
    data = request.json
    new_username = data.get('new_username', '').strip()
    current_pw = data.get('current_password')
    new_pw = data.get('new_password')

    if not current_pw or not new_pw:
        return jsonify({"status": "error", "message": "Données manquantes"}), 400

    old_username = session['user']
    PL = get_pl()
    db = get_db()
    cursor = get_cursor(db)
    cursor.execute(f"SELECT password FROM users WHERE username = {PL}", (old_username,))
    user = cursor.fetchone()

    if user and check_password_hash(user['password'], current_pw):
        hashed_pw = generate_password_hash(new_pw)
        final_username = new_username if new_username else old_username
        cursor.execute(f"UPDATE users SET username = {PL}, password = {PL} WHERE username = {PL}",
                       (final_username, hashed_pw, old_username))
        db.commit()
        session['user'] = final_username
        return jsonify({"status": "success"})

    return jsonify({"status": "error", "message": "Mot de passe actuel incorrect"}), 401

@app.route('/api/server_settings', methods=['GET', 'POST'])
@login_required
def manage_server_settings():
    global server_config
    if request.method == 'POST':
        new_settings = request.json
        for key in server_config:
            if key in new_settings:
                if key in ['sqlite_db', 'db_type'] and not str(new_settings[key]).strip():
                    continue

                if key == 'mysql_pass' and new_settings[key] == '********':
                    continue
                server_config[key] = new_settings[key]

        save_server_config(server_config)
        # On force le renouvellement de la connexion DB
        db = getattr(g, '_database', None)
        if db:
            db.close()
            g._database = None
        return jsonify({"status": "success"})

    #pour le get, on renvoie une copie avec les mots de passe masqués
    display_config = server_config.copy()
    if display_config.get('mysql_pass'):
        display_config['mysql_pass'] = '********'

    return jsonify(display_config)

init_db()
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```