Université d'Artois

IUT de Béthune

Département Réseaux et Télécommunications

SAÉ5.01

Concevoir, réaliser et présenter une solution technique

par

Rémi BONNEL, Victor DAUVIN et Eliot HULEUX

Table des matières

Résumé du projet **1**

Project summary **2**

Schéma réseau du projet **3**

1 Conception et planification 4
1.1 Choix de l'hyperviseur 4
1.2 Responsabilités, répartition des tâches 4
2 Installation et configuration de l'hyperviseur 5
2.1 Caractéristiques de la machine 5
2.2 Installation de l'hyperviseur 5
3 Création des machines virtuelles modèles 10
3.1 Définition des besoins 10
3.2 Création des machines virtuelles modèles 11
4 Création de l'environnement d'administration 12
4.1 Synchronisation avec l'Active Directory 12
4.2 Scripts d'automatisation et permissions 13
5 Serveur Web pour l'utilisation étudiante 17
5.1 Choix de l'environnement 17
5.2 Création du portail de connexion 17
5.3 Tableau de bord de l'utilisateur 20
5.4 Gestion des consoles à distance 23
5.5 Lien de déconnexion 24
5.6 Créer une nouvelle machine virtuelle 24
5.7 Supprimer une machine virtuelle 29
6 Documentation 33

Résumé du projet

Ce projet a consisté à concevoir et déployer une infrastructure virtuelle robuste et sécurisée pour les travaux pratiques (TP) en utilisant l'hyperviseur Proxmox. L'objectif principal était de permettre aux étudiants d'accéder facilement et de manière sécurisée à des machines virtuelles (VM) adaptées à leurs besoins pédagogiques. Pour ce faire, nous avons installé et configuré Proxmox sur un serveur dédié, en allouant et optimisant les ressources nécessaires (CPU, mémoire, réseau, etc.). Nous avons ensuite créé plusieurs modèles de machines virtuelles basés sur des systèmes d'exploitation variés, tels que Ubuntu Desktop et Server, Windows 10 Pro 22H2, Windows 2025, Parrot OS, et Alpine Linux, chacun préconfiguré avec les logiciels et environnements requis pour les TP.

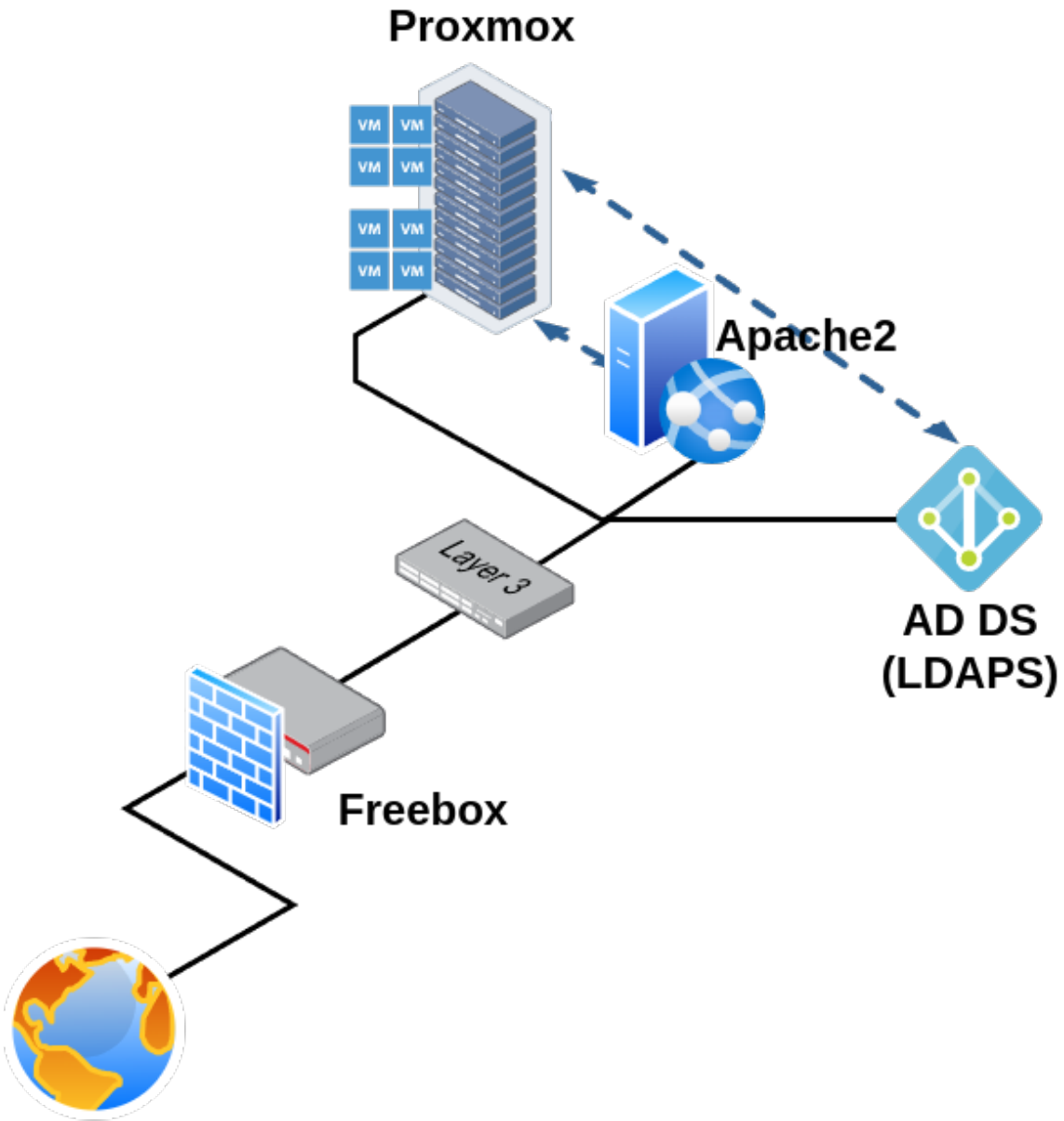
Pour l'accès à distance, nous avons mis en place une interface web intuitive via un serveur web, permettant aux étudiants de gérer les machines virtuelles : les cloner, les supprimer ou y accéder directement. Cette interface repose sur l'API Proxmox et s'intègre avec Active Directory (AD) pour assurer une gestion centralisée des utilisateurs et des droits. La liaison Proxmox/AD a été renforcée par l'utilisation du protocole LDAPS pour garantir la sécurité des échanges. Les aspects de sécurité ont été minutieusement intégrés, avec une authentification stricte et une gestion granulaire des accès pour protéger les données et l'infrastructure.

Project summary

This project involved designing and deploying a robust virtual infrastructure and secure for practical work using the Proxmox hypervisor. The main goal was to provide students with easy and secure access to virtual machines (VMs) that are tailored to their educational needs. To do this, we installed and configured Proxmox on a dedicated server, allocating and optimizing the necessary resources (CPU, memory, network, etc.). We then created several virtual machine templates based on various operating systems, such as Ubuntu Desktop and Server, Windows 10 Pro 22H2, Windows 2025, Parrot OS, and Alpine Linux, each preconfigured with the software and environments required for practical work.

For remote access, we have set up an intuitive web interface via a web server, allowing students to manage the virtual machines : clone them, create, delete or access directly. This interface is based on the Proxmox API and integrates with Active Directory (AD) to ensure centralized management of users and rights. The Proxmox/AD link has been strengthened by using the LDAPS protocol to ensure security of exchanges. Security aspects have been carefully integrated, with strict authentication and granular access management to protect data and infrastructure.

Schéma réseau du projet



1 Conception et planification

1.1 Choix de l'hyperviseur

Pour ce projet, nous sommes partis sur un hyperviseur de type 1 hébergeant la solution Proxmox.

Proxmox est une solution open-source offrant une alternative économique et flexible à VMware vSphere, particulièrement attractive depuis le rachat de VMware par Broadcom, qui a suscité des inquiétudes concernant l'augmentation des coûts et la réduction du support. Proxmox se distingue par sa simplicité de gestion, son intégration native de KVM et LXC pour la virtualisation et la conteneurisation, et son absence de frais de licence, ce qui en fait un choix privilégié pour les entreprises recherchant une indépendance technologique et un coût réduit.

Proxmox offre une interface très pédagogique, avec une facilité d'exécution, pour tout type de projet ou d'installation. De plus, le clustering est particulièrement simple, à contrario de la solution de chez VMWare.

1.2 Responsabilités, répartition des tâches

Pour gérer la planification du projet, nous avons utilisé Trello, qui s'est avéré être très facile d'utiliation.

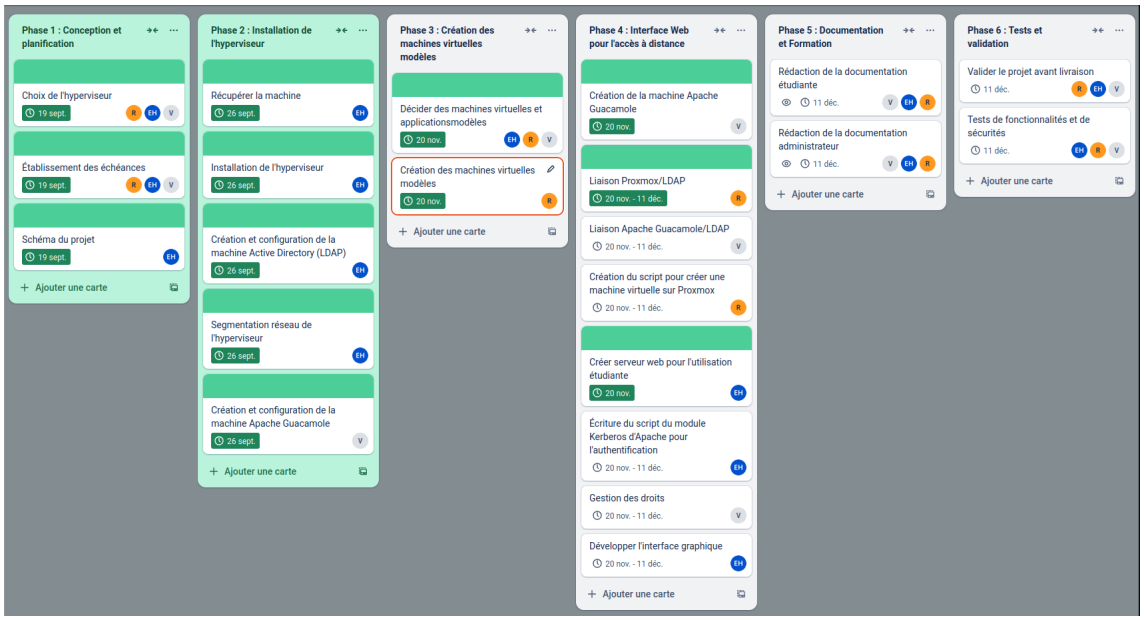


Figure 1 – *Calendrier du projet et répartition des tâches*

2 Installation et configuration de l'hyperviseur

2.1 Caractéristiques de la machine

La machine physique utilisé pour ce projet et qui sera l'hyperviseur de type 1, a comme caractéristiques :

— Un CPU Intel 6600k, 6 coeurs et 6 threads
— 64 Go de mémoire vive
— Deux disques : un de 500 Go, pour la partition système ainsi que les machines virtuelles d'administration et un autre de 1 To pour les machines virtuelles clientes

2.2 Installation de l'hyperviseur

Pour commençer, il faut télécharger l'image ISO depuis le site officiel de Proxmox.

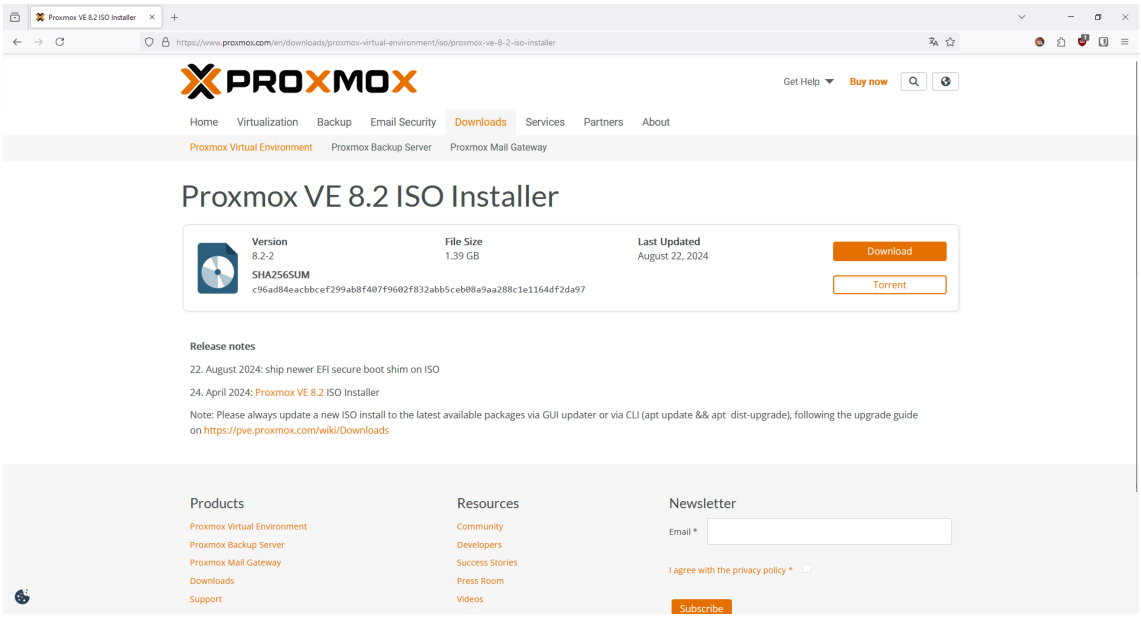


Figure 2 – *Site officiel de Proxmox*

Ensuite, il faut se munir d'une clé USB physique et la transformer en périphérique de démarrage (pour "boot" sur la clé usb). Des logiciels comme Rufus sont efficaces pour réaliser cette tâche.

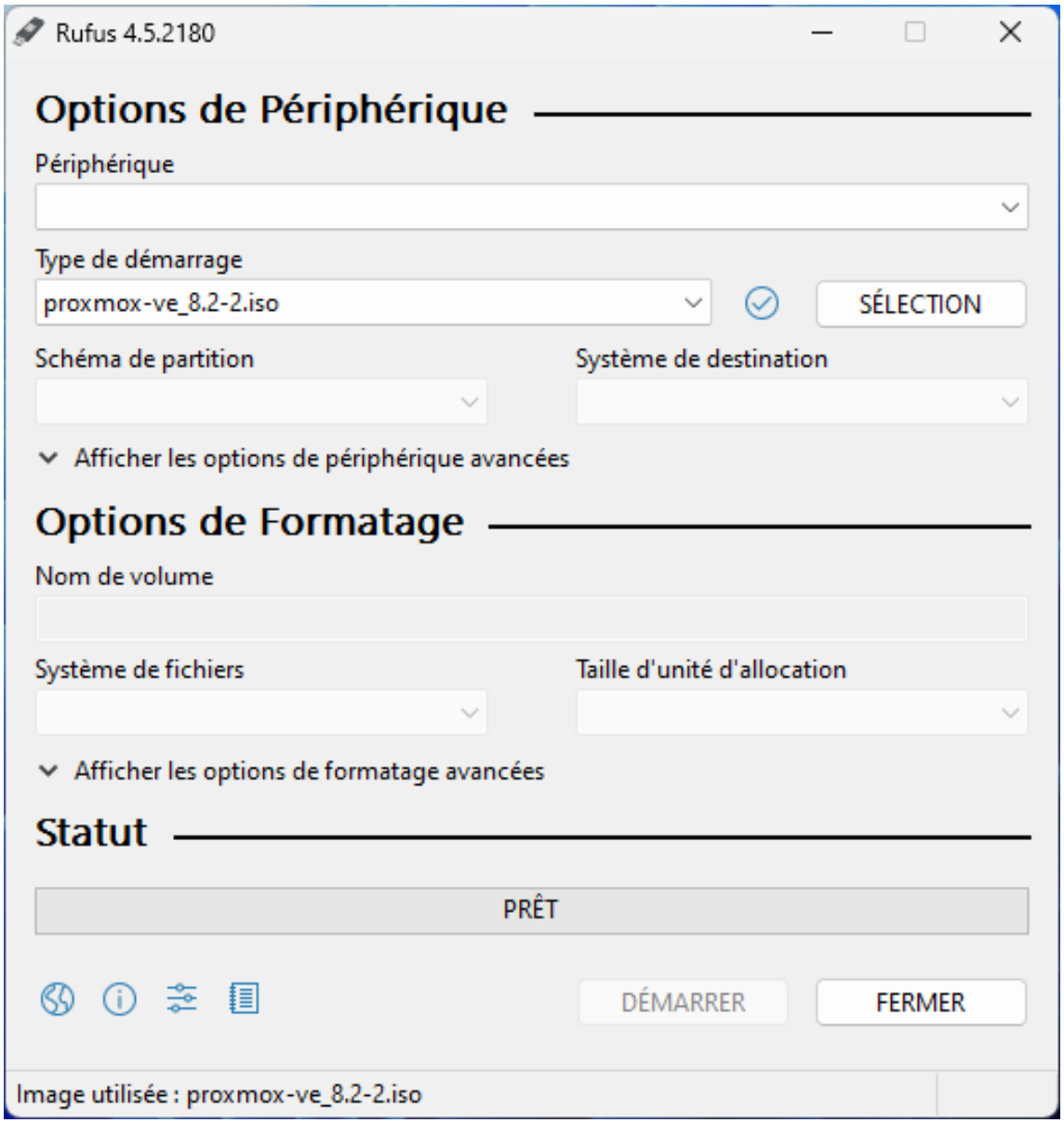


Figure 3 – *Rufus*

Maintenant, branchons la clé USB et depuis le BIOS de la machine, sélectionnons la clé USB comme disque de démarrage.

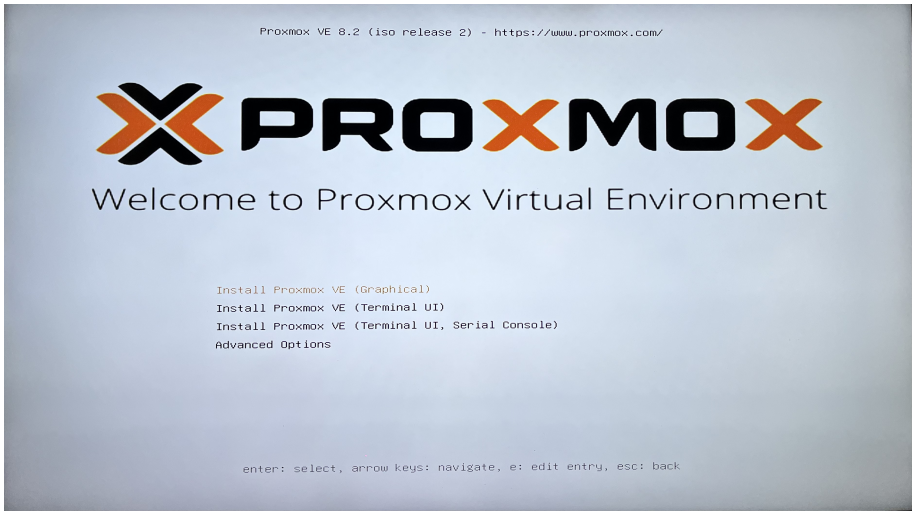


Figure 4 – *Première échange avec Proxmox*

Il faut accepter les conditions d'utilisations.

Figure 5 – *Valider les conditions d'utilisation*

Puis définir les paramètres de langue et d'horaires.

Figure 6 – *Définition de la timezone*

Il faut enfin choisir un mot de passe pour le compte root.

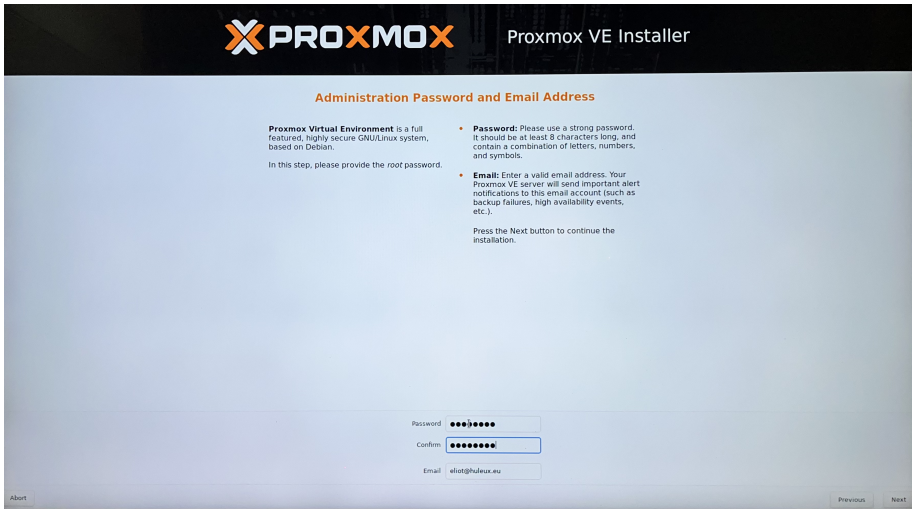


Figure 7 – *Définition de l'administration*

Ici, les paramètres réseaux. Un bail statique a été crée sur le routeur donc ici en laissons en DHCP.

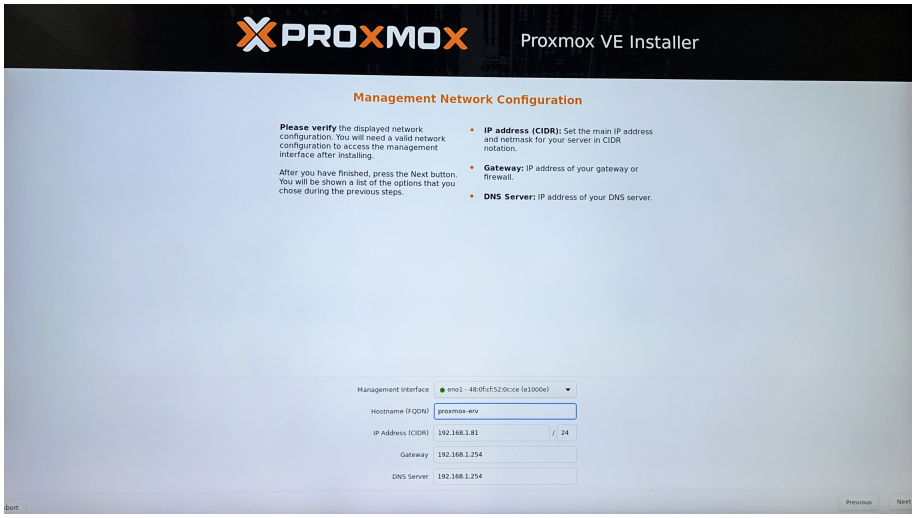


Figure 8 – *Définition des paramètres réseaux*

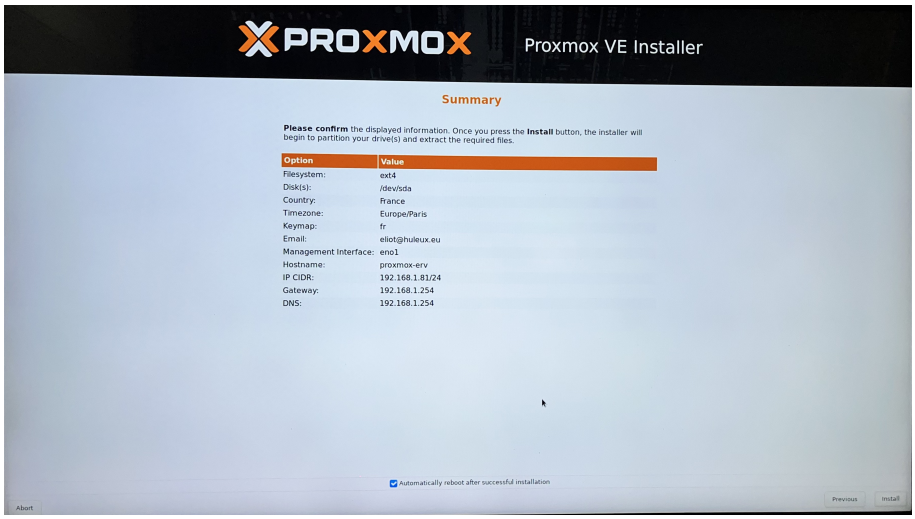


Figure 9 – *Résumé de la configuration*

Enfin, l'installateur nous récapitule la configuration que nous avons paramétré. Nous pouvons confirmer.

3 Création des machines virtuelles modèles

3.1 Définition des besoins

Pour répondre au cahier des charges du projet, nous nous sommes basés sur les TP suivis ces trois dernières années. Les machines virtuelles qui revenait le plus souvent étaient :

— Ubuntu Server & Ubuntu Desktop
— Windows Server & Windows client
— ParrotOS
— Alpine Linux

Et pour les applications et logiciels :

— Wireshark
— Cisco Packet Tracer
— GNS3
— Visual Studio Code
— Python 3
— Active Directory
— Modules sur ParrotOS (Nmap, Metasploit, BEEF, etc...)
— GNU Radio

Après réfléxion, voici le modèle convenu des machines modèles (avec applications) :

1. Ubuntu Desktop 22.04.5
   — Visual Studio Code, Wireshark, Python 3, Cisco Packet Tracer, GNS3 et GNU Radio
2. Ubuntu Server 22.04.5
   — Liberté à l'élève de choisir
3. Windows Server 2025
   — Module AD DS et DHCP
4. Windows 10 Pro 22H2

— Scilab, Waveforms et Cisco Packet Tracer

5. ParrotOS
   — Les modules de pentesting les plus communs sont déjà installés
6. Alpine Linux
   — IPBX Asterisk

3.2 Création des machines virtuelles modèles

Pour créer les VMs modèles, définissons tout d'abord les ressources qui leur seront alloués :

— Ubuntu Desktop/Server 22.04.5 : 2 coeurs, 2 GB de mémoire vive et 15 GB de disque dur
— Windows 2025 et Windows 10 Pro 22H2 : 2 coeurs, 4 GB de mémoire vive et 30 GB de disque dur
— ParrotOS : 1 coeur, 2 GB de mémoire vive, et 15 GB de disque dur
— Alpine Linux : 1 coeur, 1 GB de mémoire vive, et 10 GB de disque dur

Maintenant, sur l'hyperviseur, créons les VMs associés avec les caractéristiques définis précédemment, et la carte réseau sur *vmbr0*.

Une fois les machines virtuelles disposés à être convertis en modèle, il suffit tout simplement de la convertir en modèle comme ci-dessous :

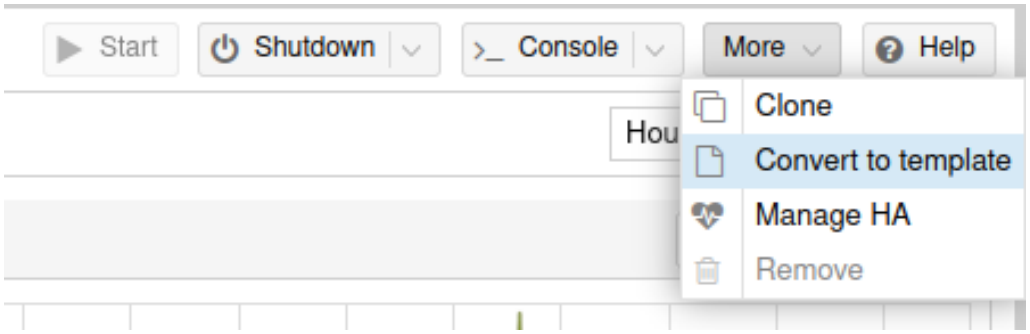


Figure 10 – *Convertir une machine virtuelle en modèle*

Et faisons de même pour toutes les machines virtuelles modèles.

4 Création de l'environnement d'administration

Pour pouvoir automatiser la gestion utilisateur, nous avons créer un annuaire Active Directory (AD DS) qui sera synchronisé avec Proxmox. Nous installerons également une authorité de certification racine sur l'Active Directory pour passer LDAP en LDAPS.

Pour le projet, nous avons décidé de travailler avec l'API de Proxmox. Nous verras cette partie dans la section d'après.

Sur la machine Proxmox, nous utiliserons *crontab* pour lancer deux scripts tous les soirs à 21h10 (après la synchronisation Active Directory/Proxmox).

4.1 Synchronisation avec l'Active Directory

Pour les utilisateurs, Proxmox travaille sous forme de royaumes. En effet, il permet de distinguer les utilisateurs de leurs sources.

Proxmox a par défaut 2 royaumes pré-disposés à être utilisé :

— pam : Linux PAM standard authentification (utilisateur hyperviseur + utilisateur machine)
— pve : Proxmox VE authentification server (utilisateur hyperviseur)

Mais à celà, nous pouvons rajouter un annuaire LDAP ou un annuaire Active Directory. Pour ce faire, il suffit d'ajouter un nouveau royaume Active Directory.

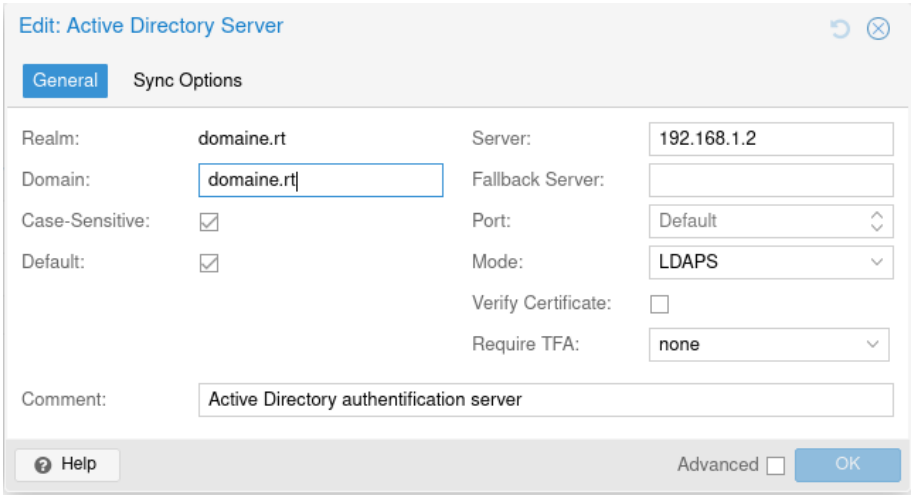


Figure 11 – *Ajout du royaume prt. 1*

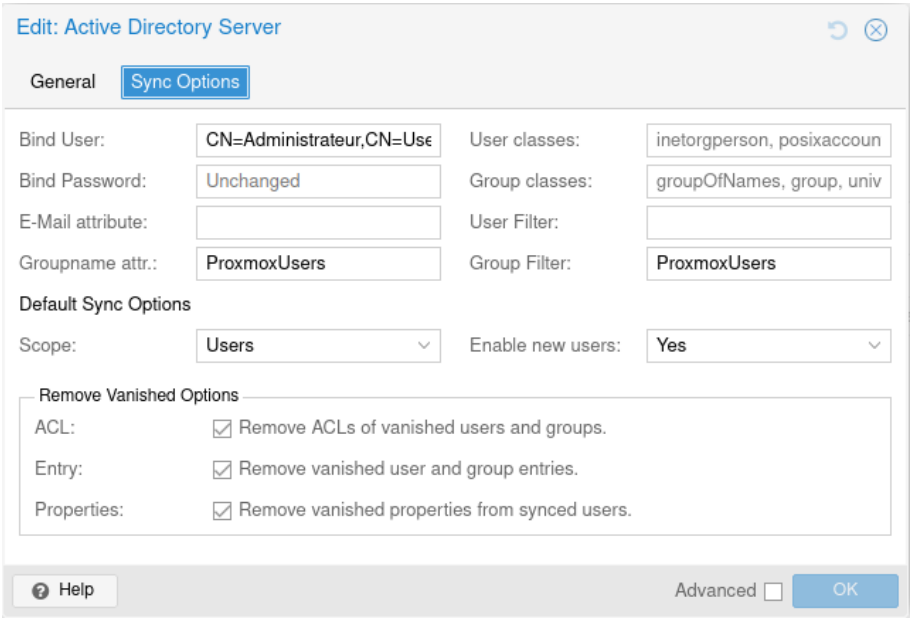


Figure 12 – *Ajout du royaume prt. 2*

Et à celà, ajoutons une synchronisation journalière tous les soirs à 21 heures.

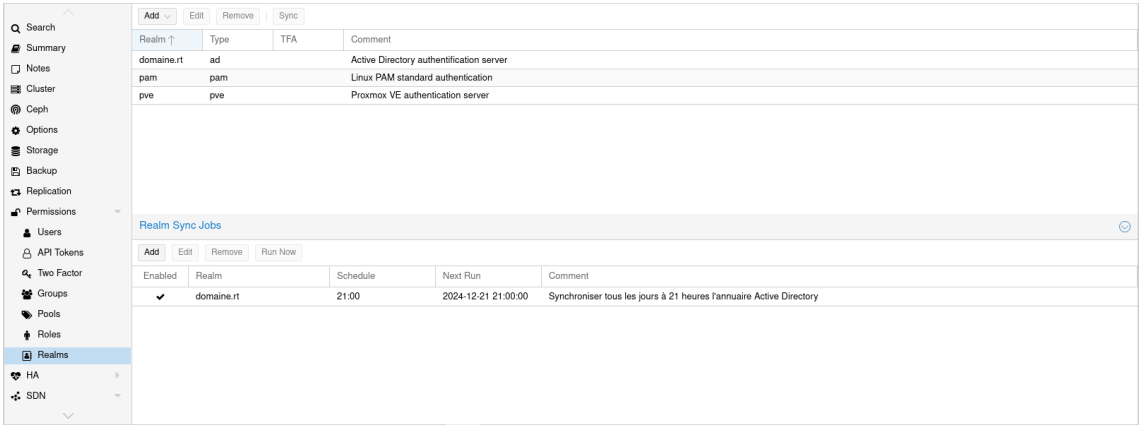


Figure 13 – *Synchronisation journalière*

4.2 Scripts d'automatisation et permissions

Sur notre projet, nous avons décidé d'intéragir avec l'API Proxmox par le biais d'un serveur web. Pour cela, nous avons donc besoin de créer ces APIs automatiquement, car si 200 utilisateurs rentrent en une fois, créer 200 APIs seraient redondants et l'efficacité passe par l'automatisation.

Pour cela, nous utiliserons crontab et deux scripts bash qui, en premier lieu, ajoute les utilisateurs du royaume *domaine.rt* au groupe Proxmox *ProxmoxUsers*, et le deuxième, créé une API pour l'utilisateur du royaume qui n'en possède pas encore une.

Pour le premier script, voici sa forme :

```bash
#!/bin/bash

DOMAIN="domaine.rt"
GROUP="ProxmoxUsers"

AD_USERS=$(pvesh get /access/users --output-format json | jq -r '.[] | select(.userid | endswith("@'"$DOMAIN"'")) | .userid')

if [ -z "$AD_USERS" ]; then
    exit 1
fi

for USER in $AD_USERS; do
    pvesh set /access/users/$USER -groups $GROUP
    if [ $? -eq 0 ]; then
        echo "$USER a été ajouté avec succès au groupe $GROUP."
    else
        echo "Erreur lors de l'ajout de $USER au groupe $GROUP."
    fi
done
```

Figure 14 – *addgroup.sh*

Le script utilise la commande *pvesh*. Elle permet d'interagir avec l'API REST de Proxmox directement depuis la ligne de commande. Elle sert à gérer et configurer les ressources (VM, conteneurs, stockage, utilisateurs, etc...) de manière automatisée.

Nous utiliserons aussi *jq* qui permet de traiter les données au format JSON, avec des critères telles que la fin de la valeur, et de trier par clé du JSON.

Sur Proxmox, définissons les permissions :

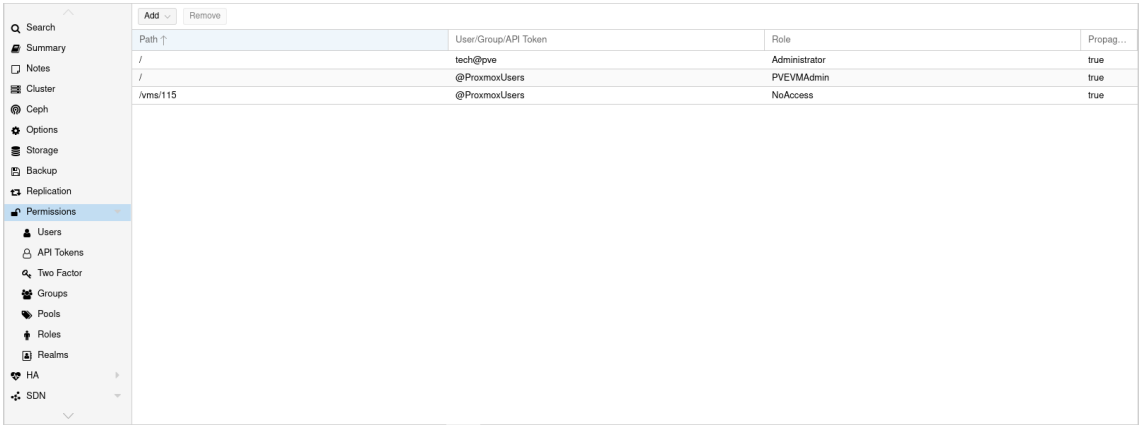


Figure 15 – *Permissions des utilisateurs*

Sur la capture d'écran, nous pouvons regarder les permissions suyivantes :

— Droits d'administrateur sur tech@pve : c'est un utilisateur crée pour l'administration pour éviter l'utilisation de root

— Le droit *PVEVMAdmin* pour le groupe *ProxmoxUsers* : ce droit permet au utilisateur du royaume *domaine.rt* de pouvoir créer des VMs, cloner, supprimer. Le nécessaire pour une utilisation cliente.

— La restriction *NoAccess* sur la machine virtuelle 115 pour le groupe *ProxmoxUsers* : La VM 115 étant l'Active Directory, le groupe ne pourra donc rien faire dessus, ni la supprimer, ni l'éteindre, ni la cloner, toutes tentatives d'accès sont interdites pour ce groupe.

Et pour le deuxième script, nous avons décidé pour la nomenclature, de l'appeler par le nom d'utilisateur sans le @ et le royaume, ce qui nous permettra d'automatiser les recherches dans le script :

```
#!/bin/bash

DOMAIN="domaine.rt"

AD_USERS=$(pvesh get /access/users --output-format json | jq -r '.[] | select(.userid | test("@"$DOMAIN"$")) | .userid')


if [ -z "$AD_USERS" ]; then
    exit 1
fi

for USER in $AD_USERS; do
    USERNAME=$(echo "$USER" | cut -d '@' -f 1)

    USER_TOKENS=$(pvesh get /access/users/$USER/token --output-format json | jq -r '.[].tokenid')

    if echo "$USER_TOKENS" | grep -q "$USER!$USERNAME"; then
        echo "L'utilisateur $USER a déjà un jeton API nommé $USERNAME."
    else
        echo "Création d'un jeton API pour l'utilisateur $USERNAME..."
        pvesh create /access/users/$USER/token/$USERNAME --comment "Token créé automatiquement" --privsep 0 --expire 0
        if [ $? -eq 0 ]; then
            echo "Jeton API créé avec succès pour l'utilisateur $USERNAME."
        else
            echo "Erreur lors de la création du jeton API pour l'utilisateur $USERNAME."
        fi
    fi

done
```

Figure 16 – *token.sh*

pvesh est ici encore réutilisé pour créer le token avec *create* et lister avec *get*. Pour lancer ces scripts automatiquement, nous avons utilisé crontab. Les scripts sont lancés tous les soirs à 21h10 (10 minutes après la synchronisation Proxmox/Active Directory).

Pour pouvoir utiliser crontab :

```
1
crontab -e
```

```
10 21 * * * /home/scripts/addgroup.sh >> /var/log/addgroup.log 2>&1
10 21 * * * /home/scripts/token.sh
```

Figure 17 – *crontab*

5 Serveur Web pour l'utilisation étudiante

5.1 Choix de l'environnement

Pour réaliser cette interface Web, nous sommes partis sur un serveur web Apache2. Les utilisateurs s'authentifiront auprès d'un portail de connexion, qui créera 3 cookies, et ces derniers permettront d'accéder à un dashboard, ainsi que l'accès à la fenêtre à distance noVNC, et il pourra créer de nouvelles machines virtuelles ou supprimer les leurs. Enfin, un guide d'utilisation leur sera mis à disposition.

5.2 Création du portail de connexion

Pour que les utilisateurs s'authentifie au près de Proxmox, nous allons intéragir avec les tokens créé automatiquement, et vu précédemment.

Pour s'y faire, nous utiliserons en backend, du nodes.js, qui lancera une requête GET auprès de Proxmox (le port 8006). Ensuite, nous créérons 3 cookies :

— PVEAuthCookie : qui identifie l'utilisateur et ses permissions
— CSRFPreventionToken : pour la sécurité web
— username : qui servira pour le clonage de machines virtuelles

Pour initialiser le serveur, nous créérons l'environnement grâce à la commande :

2
```
npm init -y
```

Voici le code en backend :

```
const express = require('express');
const bodyParser = require('body-parser');
const fetch = require('node-fetch');
const https = require('https');
const fs = require('fs');
const cors = require('cors');
const axios = require("axios");
const cookieParser = require('cookie-parser');

const options = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.crt"),
};

const app = express();
const PORT = 3000;
const httpsAgent = new https.Agent({
  rejectUnauthorized: false,
});

app.use(cors({
  origin: 'https://192.168.1.81',
  credentials: true,
}));
app.use(express.json());
app.use(cookieParser());


app.use(bodyParser.json());
```

Figure 18 – *Configuration du serveur node.js*

Le serveur est configuré en HTTPS et les requêtes cross-origin sont autorisés pour le serveur Proxmox à lui-seul. Le serveur sera accessible depuis le port 3000 et il faudra donc accepter le certificat auto-signé (les certificats auto-signés ont été vu au S4).

Les modules utilisés :

— express : framework pour créer le serveur
— body-parser : analyse les requêtes JSON
— node-fetch et axios : effectuent des requêtes HTTP/S
— HTTPS : gère le serveur HTTPS
— fs : lit les fichiers (certificat/clé)
— cors : autorise les requêtes cross-origin
— cookie-parser : gère les cookies

Pour les installer, il faut simplement d'utiliser la commande suivante :

3

npm install *nom_du_module*

Installons *pm2* avec npm qui est un module permettant d'exécuter les serveurs node.js

en arrière-plan.

Pour lancer le serveur :

4

```
pm2 start app.js
```

Créons ensuite la route d'authentification :

```
app.post('/authenticate', async (req, res) => {
    const { username, password } = req.body;

    try {
         const ticketResponse = await axios.post(
            "https://192.168.1.81:8006/api2/json/access/ticket",
            new URLSearchParams({
                username: username,
                password: password,
            }),
            { httpsAgent }
        );

        const { ticket, CSRFPreventionToken } = ticketResponse.data.data;

        res.cookie('PVEAuthCookie', ticket, {
            path: '/',
            httpOnly: false,
            secure: false,
            maxAge: 1000 * 60 * 60 * 24,
        });

        res.cookie('CSRFPreventionToken', CSRFPreventionToken, {
            path: '/',
            httpOnly: false,
            secure: false,
            maxAge: 1000 * 60 * 60 * 24,
        });

        res.cookie('username', username, {
            path: '/',
            httpOnly: false,
            secure: false,
            maxAge: 1000 * 60 * 60 * 24,
        });


        res.status(200).send({ message: 'Authentification réussie' });
    } catch (error) {
        console.error('Erreur lors de l\'authentification :', error.response?.data || error.message);
        res.status(401).send({ message: 'Nom d\'utilisateur ou mot de passe incorrect' });
    }
});

https.createServer(options, app).listen(PORT, () => {
    console.log("https://localhost:${PORT}");
});
```

Figure 19 – *Route d'authentification*

Le plus important étant le code en frontend :

```
const form = document.getElementById('authForm');
const errorMessage = document.getElementById('errorMessage');

form.addEventListener('submit', async (e) => {
  e.preventDefault();

const username = document.getElementById('username').value + "@domaine.rt";
const password = document.getElementById('password').value;

try {
  const response = await fetch('https://192.168.1.81:3000/authenticate', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    credentials: 'include',
    body: JSON.stringify({ username, password }),
  });

  if (response.ok) {
    window.location.href = '/dashboard';
  } else {
      const error = await response.json();
      errorMessage.textContent = error.message || 'Erreur d\'authentification';
      errorMessage.style.display = 'block';
   }
  } catch (err) {
      errorMessage.textContent = 'Erreur de connexion au serveur';
      errorMessage.style.display = 'block';
    }
});
```

Figure 20 – *Frontend pour l'authentification*

Lors de la soumission, il intercepte l'événement pour empêcher le rechargement de la page, récupère les valeurs des champs de saisie pour créer un identifiant complet (en ajoutant le royaume domaine.rt) et un mot de passe, puis envoie ces données au serveur (le serveur node.js sur le port 3000) via une requête POST au format JSON. Si la réponse du serveur est positive (code d'état 200 OK), l'utilisateur est redirigé vers la page /dashboard.

Une fois authentifié, voici les cookies que nous récupérons :

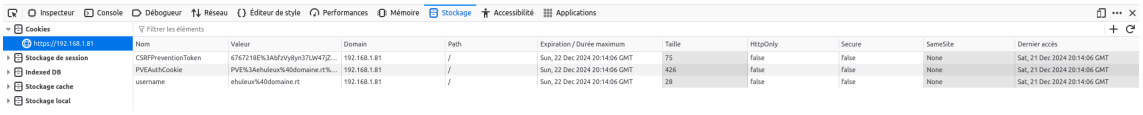


Figure 21 – *Cookies obtenu suite à l'authentification*

5.3 Tableau de bord de l'utilisateur

Une fois que l'utilisateur a réussi à s'authentifier, il arrive sur son tableau de bord, qui est vide. Tout est normal !

En effet, le tableau de bord est dynamique et recherche les VMs en fonction de son nom. Cette nomenclature a été choisi. Un système de tags existe également sur les ma-

chines virtuelles, mais notre avons préféré cette nomenclature suivante :

username-nom_vm, par exemple : ehuleux-essaiVM

Dans le dashboard, l'utilisateur retrouvera sous forme de cards ses machines virtuelles de la manière suivante :

— Nom de la machine virtuelle
— L'ID de la machine virtuelle
— Le nombre de coeurs
— La taille de la mémoire vive
— La taille du disque dur

En exemple concret, voici à quoi ressemblerai l'utilisateur avec la VM nommée *essai* :

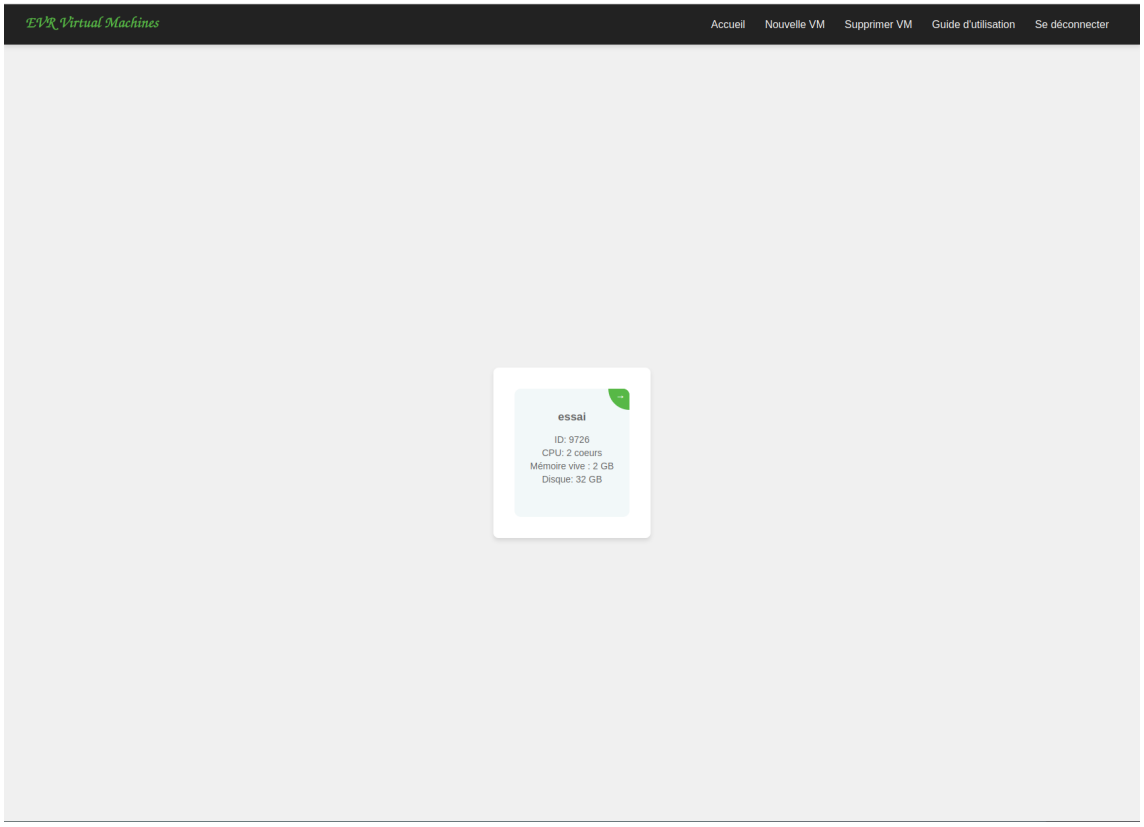


Figure 22 – *Dashboard d'un utilisateur*

Pour arriver à ce résultat ci, nous avons avons recréé un serveur node.js avec la même configuration que le précédent, récupérant les cookies et faisant une requête curl

et traiter la sortie en JSON. Voici à quoi ressemble la route :

```
app.get('/vms', async (req, res) => {
  try {
    const pveAuthCookie = req.cookies.PVEAuthCookie;
    const username = req.cookies.username;

    if (!pveAuthCookie || !username) {
      return res.status(401).send("Authentification requise");
    }

    const userPrefix = username.split('@')[0];

    const resourcesResponse = await axios.get(
      "https://192.168.1.81:8006/apt2/json/cluster/resources",
      {
        headers: {
          Cookie: `PVEAuthCookie=${pveAuthCookie}`,
        },
        httpsAgent,
      }
    );

    const vms = resourcesResponse.data.data
      .filter(vm => vm.name && vm.name.startsWith(userPrefix))
      .map(vm => ({
        vmid: vm.vmid,
        name: vm.name,
        maxcpu: vm.maxcpu,
        maxmem: (vm.maxmem / 1024 / 1024 / 1024).toFixed(2),
        maxdisk: (vm.maxdisk / 1024 / 1024 / 1024).toFixed(2),
      }));

    res.json(vms);
  } catch (error) {
    console.error("Erreur lors de la récupération des VMs :", error);
    res.status(500).send("Erreur 500");
  }
});
```

Figure 23 – *Route pour récupérer les VMs au nom de l'utilisateur*

Sur le code, nous pouvons voir que la sortie est en JSON et que nous venons l'exploiter en triant les informations et retenir que ce qui nous intéresse. Nous avons reformuler les tailles car par défaut, elles sortent en octet. Nous les avons donc recalculer en Go.

Le serveur node.js renvoie au serveur web le JSON final comme ceci :

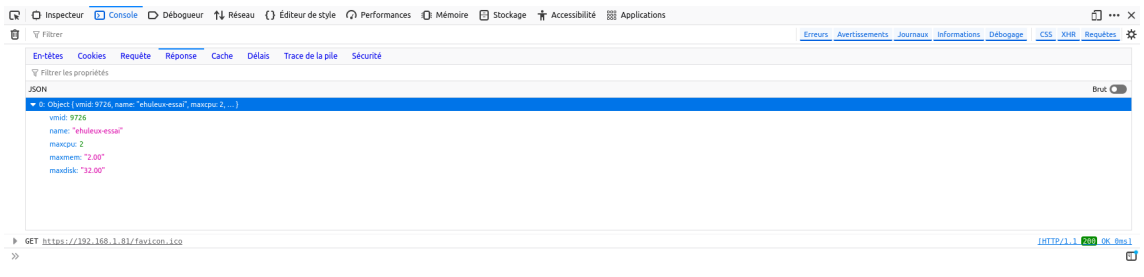


Figure 24 – *Sortie final en JSON*

Maintenant, du côté frontend, voici comment nous l'avons exploité :

```
<div id="vmContainer" class="container"></div>

<script>
    async function fetchVMs() {
        try {
            const response = await fetch('https://192.168.1.81:4000/vms', {credentials: 'include'});
            const vms = await response.json();

            const container = document.getElementById('vmContainer');
            container.innerHTML = ''; // Clear previous content

            vms.forEach(vm => {
                const splitName = vm.name.split('-');
                const displayName = splitName[splitName.length - 1];

                const card = document.createElement('div');
                card.classList.add('card');
                card.innerHTML = `
                    <a class="card1" target="_blank" href="https://192.168.1.81:8006/?console=kvm&novnc=1&vmid=${vm.vmid}&vmname=${vm.name}&node=proxmox-erv&resize=off&cmd=">
                        <p id="vmName">${displayName}</p>
                        <p id="vmCharacteristics" class="small">
                            ID: ${vm.vmid} <br>
                            CPU: ${vm.maxcpu} coeurs <br>
                            Mémoire vive : ${Math.round(vm.maxmem)} GB<br>
                            Disque: ${Math.round(vm.maxdisk)} GB
                        </p>
                        <div class="go-corner" href="#">
                            <div class="go-arrow">→</div>
                        </div>
                    </a>
                `;
                container.appendChild(card);
            });
        } catch (error) {
            console.error(error);
        }
    }

    fetchVMs();
</script>
```

Figure 25 – *Frontend*

Les cards sont gérés dynamiquement avec un remplissage des balises par les valeurs. Nous pouvons aperçevoir une balise de lien qui renvoie dynamiquement le lien de fenêtre à distance noVNC. Nous verrons ceci dans la partie d'après.

5.4 Gestion des consoles à distance

Grâce aux cookies créés lors de l'authentification, l'utilisateur a le droit sur les consoles noVNC de ses VMs. Proxmox crée ses URLs de console noVNC du même principe à chaque fois :

5

https ://@IpProxmox :8006/ ?console=kvm&novnc=1&vmid=vmID&vmname=nom_vm&node=nom_hyperviseur&resize=off&cmd=

Profitons en pour l'insérer dynamiquement comme ceci :

6

https ://192.168.1.81 :8006/ ?console=kvm&novnc=1&vmid=vmID&vmname=nom_vm&node=proxmox-erv&resize=off&cmd=

Et comme nous récupérons l'ID de la machine virtuelle et son nom, nous pouvons donc créer le lien dynamiquement sur la card.

5.5 Lien de déconnexion

Sur la navbar, l'utilisateur peut se déconnecter. Ce bouton casse les cookies et le redirige vers la page de connexion. Pour le faire, nous avons donc, côté backend, créer une route qui casse les cookies et fait une redirection et le code en frontend, renvoie vers cette route.

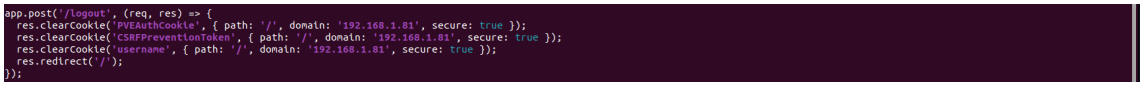


Figure 26 – *Code backend pour la déconnexion*

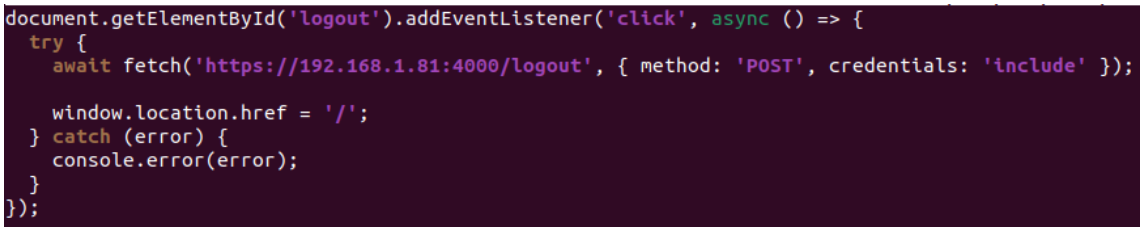


Figure 27 – *Code frontend pour la déconnexion*

5.6 Créer une nouvelle machine virtuelle

Pour créer des nouvelles machines virtuelles, nous n'avons pas utiliser node.js mais un serveur Flask.

Afin de permettre aux utilisateurs de créer et de supprimer une machine virtuelle dans leur espace, nous avons utilisé un serveur Flask, hébergé sur deux containers Docker. Les Dockers sont installés directement sur la machine Proxmox. Nous avons effectués le choix de conteneuriser ces deux applications sur deux containers différents afin de permettre une continuité de service sans risque d'affecter l'hyperviseur en cas de plantage d'un script, ou de mauvaise manipulation.

Pour le clonage de machines virtuelles, le Dockerfile :

7

```
FROM python :3.9-slim

WORKDIR /app
COPY requirements.txt /app/

RUN pip install –no-cache-dir -r requirements.txt

COPY . /app

EXPOSE 5000

CMD ["python", "app.py"]
```

Le *requirements.txt* :

8

```
flask
flask-cors
proxmoxer
requests
flask_session
flask-session
```

Et le *docker-compose.yml* :

```
version: '3.9'
services:
  app:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/app
      - /etc/apache2/ssl:/app/ssl:ro
```

Figure 28 – *docker-compose.yml*

Enfin, voici l'*app.py* qui permet de cloner une machine virtuelle :

```python
from flask import Flask, request, jsonify, render_template, make_response
from flask_cors import CORS
from proxmoxer import ProxmoxAPI
import time
import re
import random

app = Flask(__name__)
CORS(app)
app.secret_key = '9755b41abd0729d33c8199f19351f5d06e357f9f71fb14b1'

proxmox = ProxmoxAPI(
    '192.168.1.81',
    user='root@pam',
    token_name='vmcreator',
    token_value='94ddd076-2df8-43f5-b0db-c0a26138f051',
    verify_ssl=False
)

NODE = "proxmox-erv"
templates = [
    {'name': 'Ubuntu Desktop 22.04', 'id': 206},
    {'name': 'ParrotOS', 'id': 103},
    {'name': 'Alpine Linux', 'id': 210},
]

@app.route('/add-vm')
def add_vm():
    return render_template('add/index.html', templates=templates)

@app.route('/options', methods=['GET'])
def get_options():
    template_names = [template['name'] for template in templates]
    return jsonify({
        "templates": template_names
    })

def wait_for_task_completion(task_id):
    while True:
        task_status = proxmox.nodes(NODE).tasks(task_id).status.get()
        if task_status.get("status") == "stopped":
            if task_status.get("exitstatus") == "OK":
                return True
            else:
                raise Exception(f"Tâche {task_id} échouée : {task_status.get('exitstatus')}")
        time.sleep(1)

def generate_random_id():
    return random.randint(500, 9999)
```

Figure 29 – *Code Flask prt. 1*

```python
@app.route('/clone-vm', methods=['POST'])
def clone_vm():
    data = request.json
    try:
        template_name = data['template_name']
        target_name = data['name']
        username = request.cookies.get('username')
        username_clean = re.sub(r'[^a-zA-Z0-9-]', '', username.split('%40')[0])
        if username_clean:
            target_name = f"{username_clean}-{target_name}"
        target_id = generate_random_id()
        vm_password = "progtr00"
        template = next((template for template in templates if template['name'] == template_name), None)
        if not template:
            return jsonify({"error": "Template non trouvé"}), 400
        clone_task = proxmox.nodes(NODE).qemu(template['id']).clone.post(
            newid=target_id,
            name=target_name
        )
        wait_for_task_completion(clone_task.get("data"))
        proxmox.nodes(NODE).qemu(target_id).config.post(
            cores=2,
            memory=2048,
            net0="virtio,bridge=vmbr0"
        )
        proxmox.nodes(NODE).qemu(target_id).set.post(
            password=vm_password
        )

        return jsonify({"message": "VM clonée avec succès"}), 200

    except Exception as e:
        return jsonify({"message": "VM clonée avec succès"}), 200

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

Figure 30 – *Code Flask prt. 2*

Enfin, voici l'HTML qui permet de présenter ceci :

```
    <script>
        function goBack() {
            window.history.back();
        }

        async function createVM() {
            const templateName = document.getElementById('templateName').value;
            const vmName = document.getElementById('vmName').value;

            const response = await fetch('/clone-vm', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    template_name: templateName,
                    name: vmName
                })
            });

            const result = await response.json();
            alert(result.message);
        }
    </script>
</head>
<body>
    <div class="back-arrow" onclick="goBack()"></div>

    <div class="container">
        <h1>Clonage d'une VM</h1>
        <form onsubmit="event.preventDefault(); createVM();">
            <label for="templateName">Template:</label>
            <select id="templateName">
                {% for template in templates %}
                    <option value="{{ template.name }}">{{ template.name }}</option>
                {% endfor %}
            </select>

            <label for="vmName">Nom de la VM:</label>
            <input type="text" id="vmName" required>

            <button type="submit">Créer la VM</button>
        </form>
    </div>
</body>
</html>
```

Figure 31 – *HTML pour cloner une machine virtuelle*

Le serveur web Flask se connecte à l'API Proxmox, puis liste les templates grâce à l'ID associé, vérifie l'état du clonage en cours, génère aléatoirement un ID pour la nouvelle, récupère le cookie utilisateur pour pouvoir écrire dynamiquement le nom de la machine virtuelle. Le clonage se lance suite à une requête POST.

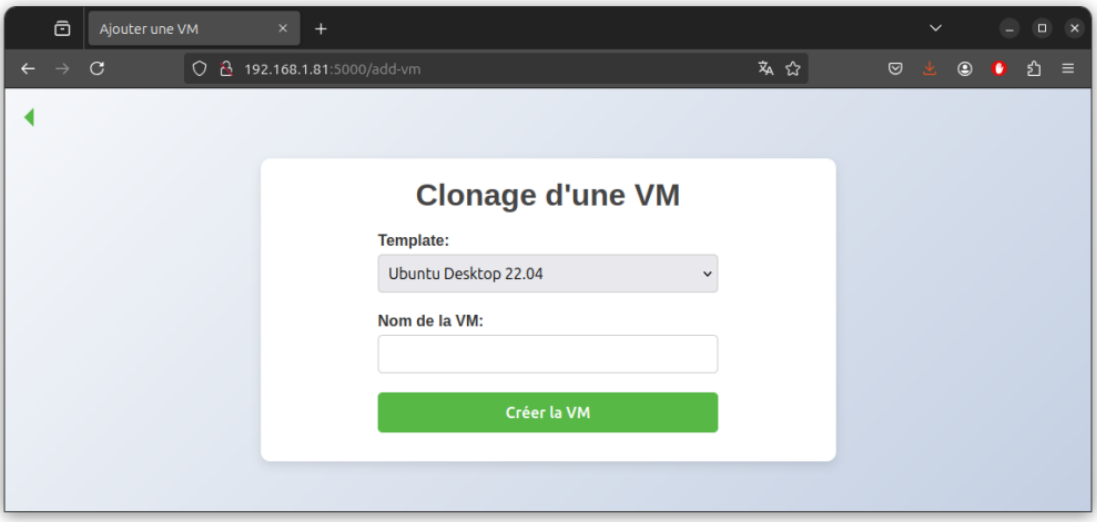


Figure 32 – *Page web*

5.7 Supprimer une machine virtuelle

Même principe, un Docker hébergeant un serveur Flask :

9

```
FROM python :3.9-slim

WORKDIR /app
COPY requirements.txt /app/

RUN pip install –no-cache-dir -r requirements.txt

COPY . /app

EXPOSE 5001

CMD ["python", "app.py"]
```

Le *requirements.txt* :

10

Flask==2.2.3

Flask-Cors==3.0.10

proxmoxer==1.3.1

Werkzeug==2.2.3

requests==2.28.2

Et le *docker-compose.yml* :

```yaml
version: '3.9'
services:
  deleter:
    build: .
    ports:
      - "5001:5001"
    volumes:
      - .:/app
      - /etc/apache2/ssl:/app/ssl:ro
```

Figure 33 – *docker-compose.yml*

L'*app.py* et la partie web :

```python
from flask import Flask, request, jsonify, render_template, make_response
from flask_cors import CORS
from proxmoxer import ProxmoxAPI
import time
import re

app = Flask(__name__)
CORS(app)
app.secret_key = '9755b41abd0729d33c8199f19351f5d06e357f9f71fb14b1'
proxmox = ProxmoxAPI(
    '192.168.1.81',
    user='root@pam',
    token_name='vmcreator',
    token_value='94ddd076-2df8-43f5-b0db-c0a26138f051',
    verify_ssl=False
)

NODE = "proxmox-erv"


def wait_for_task_completion(task_id):
    while True:
        task_status = proxmox.nodes(NODE).tasks(task_id).status.get()
        if task_status.get("status") == "stopped":
            if task_status.get("exitstatus") == "OK":
                return True
            else:
                raise Exception(f"Tâche {task_id} échouée : {task_status.get('exitstatus')}")
        time.sleep(1)

def extract_username_from_cookie(cookie_value):
    try:
        parts = cookie_value.split('%3A')
        if len(parts) >= 2:
            return parts[1]
    except Exception:
        pass
    return None

@app.route('/delete-vm')
def delete_vm_form():
    return render_template('delete/index.html')
```

Figure 34 – *Configuration Flask prt. 1*

```python
@app.route('/del', methods=['POST'])
def delete_vm():
    data = request.json
    try:
        vm_id = data.get('vm_id')
        if not vm_id:
            return jsonify({"error": "ID de VM manquant"}), 400
        username = request.cookies.get('username')
        if not username:
            return jsonify({"error": "Cookie utilisateur manquant"}), 400
        username_clean = re.sub(r'[^a-zA-Z0-9-]', '', username.split('%40')[0])
        if not username_clean:
            return jsonify({"error": "Impossible d'extraire le nom d'utilisateur du cookie"}), 400
        vm_config = proxmox.nodes(NODE).qemu(vm_id).config.get()
        vm_name = vm_config.get('name')
        if not vm_name:
            return jsonify({"error": "Impossible de récupérer le nom de la VM"}), 404
        if not vm_name.startswith(f"{username_clean}-"):
            return jsonify({"error": "Permission refusée. Cette VM n'appartient pas à l'utilisateur connecté"}), 403
        delete_task = proxmox.nodes(NODE).qemu(vm_id).delete()
        wait_for_task_completion(delete_task.get("data"))
        return jsonify({"message": f"VM {vm_name} supprimée avec succès"}), 200

    except Exception as e:
        return jsonify({"message": f"VM {vm_name} supprimée avec succès"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001, debug=True)
```

Figure 35 – *Configuration Flask prt. 2*

```
    <script>
        // Fonction pour retourner en arrière
        function goBack() {
            window.history.back();
        }

        // Gestion du formulaire pour supprimer une VM
        async function handleDeleteVm(event) {
            event.preventDefault();

            const vmId = document.getElementById('vmId').value;
            const responseDiv = document.getElementById('response');

            try {
                const response = await fetch('/del', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({ vm_id: vmId })
                });

                const result = await response.json();

                if (response.ok) {
                    responseDiv.innerHTML = `<div class="alert alert-success">${result.message}</div>`;
                } else {
                    responseDiv.innerHTML = `<div class="alert alert-danger">Erreur: ${result.error}</div>`;
                }
            } catch (error) {
                responseDiv.innerHTML = `<div class="alert alert-danger">Une erreur est survenue: ${error.message}</div>`;
            }
        }
    </script>
</head>
<body>
    <div class="back-arrow" onclick="goBack()"></div>

    <div class="container">
        <h1>Supprimer une Machine Virtuelle</h1>
        <form id="deleteVmForm" onsubmit="handleDeleteVm(event)">
            <label for="vmId">ID de la VM</label>
            <input type="text" id="vmId" placeholder="Entrez l'ID de la VM" required>
            <button type="submit">Supprimer</button>
        </form>
        <div id="response"></div>
    </div>
```

Figure 36 – *HTML dynamique pour supprimer une machine virtuelle*

Le rendu web est de cette manière :

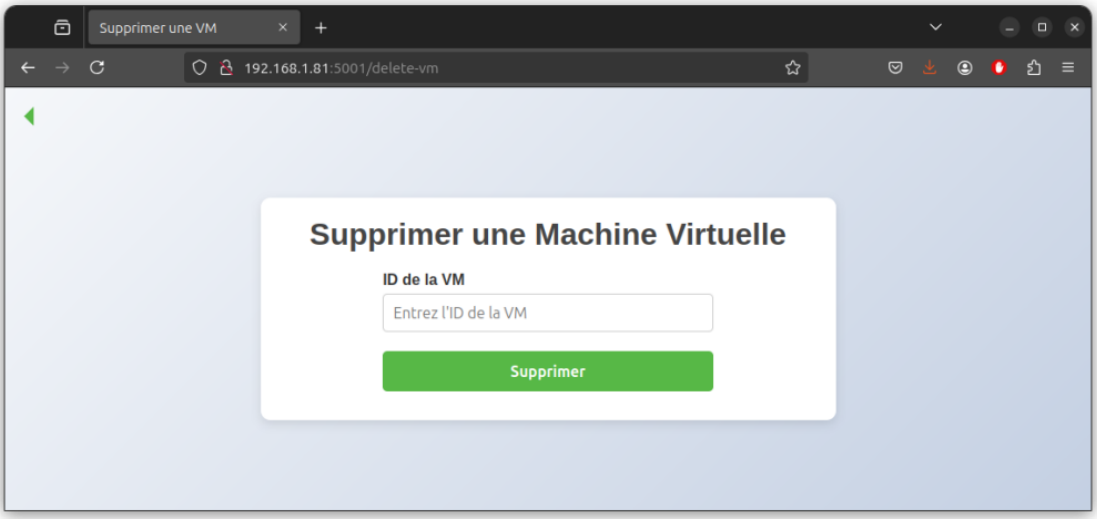


Figure 37 – *Interface web*

Ce script s'exécute lorsque la requête POST est envoyée depuis une page web pour demander la suppression d'une VM. Tout d'abord, il récupère l'ID de la VM à supprimer depuis les données envoyées dans la requête. Ensuite, il lit le cookie utilisateur pour identifier la session de l'utilisateur connecté.

Le script extrait et nettoie le nom d'utilisateur depuis le cookie, en retirant les caractères spéciaux. Il vérifie ensuite si la VM correspond bien à l'utilisateur en s'assurant que le nom de la VM commence par le préfixe *utilisateur-*. Si la VM ne correspond pas à l'utilisateur, une erreur de permission est renvoyée.

Après cette validation, le script initialise la tâche de suppression en appelant l'API Proxmox avec l'ID de la VM. Le processus de suppression démarre via une requête POST. Une fois la suppression terminée, le script attend que Proxmox confirme la fin de la tâche grâce à la fonction wait_for_task_completion. Si tout se passe correctement, une confirmation de suppression est renvoyée à l'utilisateur, accompagnée du nom de la VM supprimée.

6 Documentation

Le manuel d'utilisation utilisateur est disponible sur la navbar, étant accessible depuis l'URL :

https ://www.canva.com/design/DAGYu-I2fnQ/ca5_NYSeT2yFzZRVxvxLlQ/view

Un utilisateur *tech* a été crée pour Proxmox afin de deboguer si besoin.
Fichiers de logs pour les erreurs ou soucis d'un élève :

1. Apache2
   - Accès : /var/log/apache2/access.log
   - Erreurs : /var/log/apache2/error.log
2. Proxmox
   - Interface Web : /var/log/pveproxy/access.log
   - Gestion des VMs : /var/log/pvedaemon/pvedaemon.log
   - Cluster : /var/log/pve-cluster.log
   - /var/log/syslog
3. Active Directory
   - Directory Service Log : Logs des services AD (via eventvwr.msc)
   - Security Log : Journaux d'authentification et de droits
   - System Log : Erreurs système et matérielles