Projet : IAM : Provisioning Hybride Simplifié

Réalisé par :

Joseline YOUEGO

Table des matières

1 Mise en place du laboratoire

Ce chapitre décrit la mise en place de l'environnement expérimental utilisé pour le projet. Il s'agit d'un laboratoire IAM basé sur Linux et OpenLDAP. L'objectif est de disposer d'une petite infrastructure virtualisée représentant une entreprise simplifiée, avec un annuaire central, une application cible et une machine de contrôle jouant le rôle de moteur IAM.

1.1 Création de l'infrastructure virtuelle

L'infrastructure repose sur trois machines virtuelles Ubuntu 20.04, créées et gérées avec Vagrant et VirtualBox. Plutôt que de configurer manuellement chaque système, l'ensemble de l'environnement est décrit dans un unique fichier `Vagrantfile`, ce qui permet de recréer le laboratoire à l'identique à partir d'un simple `vagrant up`.

Listing 1.1: Vagrantfile de l'infrastructure

```
# Vagrantfile
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/focal64" # Ubuntu 20.04

  # --- VM 1 : ANNUAIRE (Source) ---
  config.vm.define "iam-ldap" do |ldap|
    ldap.vm.hostname = "iam-ldap"
    ldap.vm.network "private_network", ip: "192.168.56.10"
    ldap.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "IAM-Lab-LDAP"
    end
    # Installation silencieuse de OpenLDAP
    ldap.vm.provision "shell", inline: <<-SHELL
      export DEBIAN_FRONTEND=noninteractive
      apt-get update
      # On dfinit le mot de passe admin  'admin' pour l'installation auto
      echo "slapd␣slapd/root_password␣password␣admin" | debconf-set-selections
      echo "slapd␣slapd/root_password_again␣password␣admin" | debconf-set-selections
      apt-get install -y slapd ldap-utils
    SHELL
  end

  # --- VM 2 : APPLICATION CIBLE (Target) ---
  config.vm.define "iam-app" do |app|
    app.vm.hostname = "iam-app"
    app.vm.network "private_network", ip: "192.168.56.20"
```

```
    app.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "IAM-Lab-APP"
    end
    app.vm.provision "shell", inline: <<-SHELL
      apt-get update
      apt-get install -y python3-pip
      pip3 install flask
    SHELL
  end

  # --- VM 3 : CONTROL CENTER (Moteur IAM) ---
  config.vm.define "iam-control" do |ctl|
    ctl.vm.hostname = "iam-control"
    ctl.vm.network "private_network", ip: "192.168.56.30"
    ctl.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "IAM-Lab-Control"
    end
    ctl.vm.provision "shell", inline: <<-SHELL
      apt-get update
      apt-get install -y python3-pip ldap-utils curl
      pip3 install ldap3 requests
      # On aide la machine  trouver les autres par leur nom
      echo "192.168.56.10 iam-ldap" >> /etc/hosts
      echo "192.168.56.20 iam-app" >> /etc/hosts
    SHELL
  end
end
```

La première machine, nommée `iam-ldap`, joue le rôle de serveur d'annuaire. Elle est adressée en `192.168.56.10` sur le réseau privé du laboratoire et reçoit automatiquement l'installation de `slapd` et des utilitaires `ldap-utils`. Le mot de passe administrateur de l'annuaire est fixé à `admin` pour faciliter les tests, ce qui est acceptable dans le cadre d'un environnement de laboratoire mais ne serait pas recevable en production.

La deuxième machine, `iam-app`, représente l'application métier qui devra, par la suite, s'intégrer avec l'annuaire. Elle est accessible en `192.168.56.20` et se voit équiper d'un environnement Python minimal (installation de `python3-pip` puis de `Flask`) pour héberger une application web simple.

Enfin, la troisième machine, `iam-control`, est utilisée comme centre de contrôle IAM. Elle est positionnée en `192.168.56.30` et dispose des outils nécessaires pour dialoguer avec l'annuaire et l'application : `ldap-utils` pour interroger OpenLDAP, `curl` pour tester les API HTTP, ainsi que les bibliothèques Python `ldap3` et `requests`. Des entrées sont ajoutées au fichier `/etc/hosts` afin que la machine puisse joindre `iam-ldap` et `iam-app` par leur nom plutôt que par leur seule adresse IP.

Après quelques minutes, les trois machines virtuelles sont opérationnelles et le laboratoire IAM est prêt pour la configuration de l'annuaire.

1.2 Configuration de l'annuaire OpenLDAP

La première brique fonctionnelle à mettre en place est l'annuaire LDAP, hébergé sur la machine `iam-ldap`. L'objectif est de définir un domaine logique représentant la société fictive, puis d'y créer une structure minimale d'unités organisationnelles, de groupes et d'utilisateurs.

L'accès à la machine se fait depuis le poste hôte via Vagrant :

```
vagrant ssh iam-ldap
```

L'installation automatique de `slapd` fournit une configuration par défaut qui ne correspond pas encore au domaine souhaité. Pour aligner l'annuaire sur le domaine de test `iam.lab`, l'outil `dpkg-reconfigure` est utilisé afin de reconfigurer le serveur OpenLDAP de manière interactive :

```
sudo dpkg-reconfigure slapd
```

Au cours de cette reconfiguration, le nom de domaine DNS est fixé à `iam.lab`, ce qui conduit à une base de suffixe `dc=iam,dc=lab`. L'organisation est nommée IAM Lab, et le mot de passe administrateur est positionné sur `admin`. Le backend de base de données choisi est `MDB`, et la base existante est remplacée par une nouvelle instance propre, adaptée au projet.

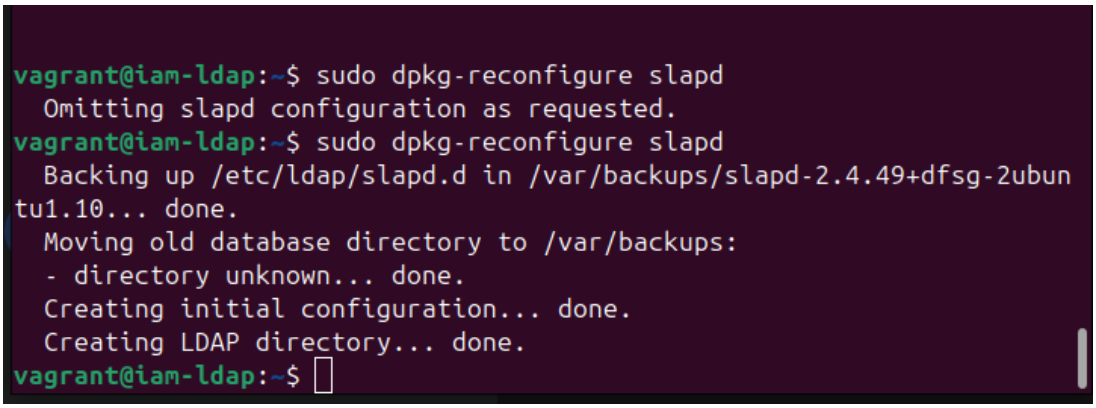

Figure 1.1: Configuration de ldap

Une fois cette étape terminée, l'annuaire est opérationnel mais encore vide. Afin de disposer de données significatives pour les scénarios d'IAM, deux utilisateurs de test sont créés : Alice Dupont et Bob Martin. La première est censée représenter une collaboratrice active du service Finance, tandis que le second illustre un compte inactif.

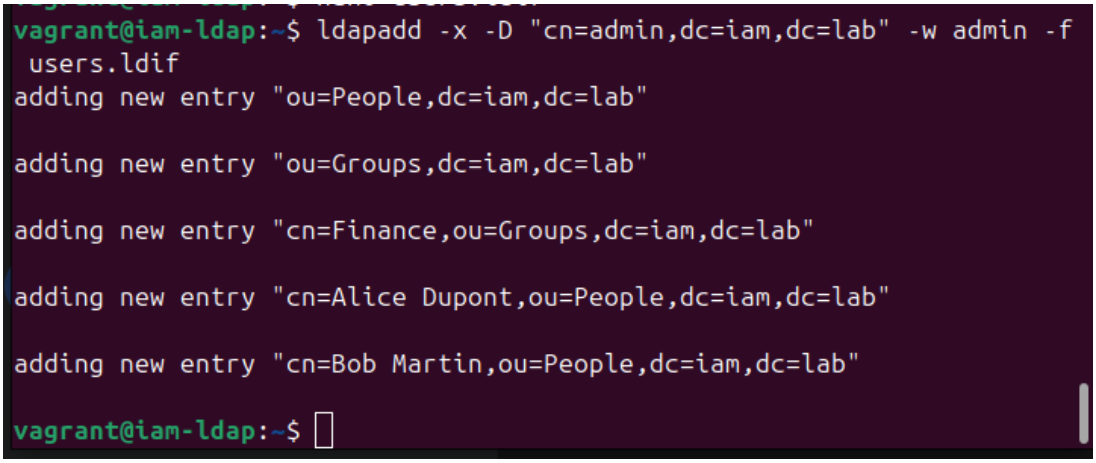

Figure 1.2: Création d'utilisateurs

La création de la structure et des entrées se fait à l'aide d'un fichier LDIF (`users.ldif`). Ce fichier définit d'abord deux unités organisationnelles, `ou=People` pour les utilisateurs et `ou=Groups` pour les groupes, puis un groupe `cn=Finance` identifié par le GID `5000`. Les entrées utilisateurs associent à Alice l'identifiant `adupont`, le groupe `Finance` et un attribut `description` positionné à `Active`. Bob reçoit l'identifiant `bmartin`, un GID distinct, et une description `Inactive`. Cette simple différenciation sur le champ `description` sera exploitée ultérieurement par le moteur IAM pour décider d'autoriser ou non l'authentification.

Le fichier LDIF est injecté dans l'annuaire au moyen de la commande suivante :

```
ldapadd -x -D "cn=admin,dc=iam,dc=lab" -w admin -f users.ldif
```

Une requête de vérification permet ensuite de s'assurer que la base contient bien les objets attendus :

```
ldapsearch -x -b "dc=iam,dc=lab" "(objectclass=*)"
```

```
vagrant@iam-ldap:~$ ldapsearch -x -b "dc=iam,dc=lab" "(objectclass=*)"
# extended LDIF
#
# LDAPv3
# base <dc=iam,dc=lab> with scope subtree
# filter: (objectclass=*)
# requesting: ALL
#

# iam.lab
dn: dc=iam,dc=lab
objectClass: top
objectClass: dcObject
objectClass: organization
o: IAM Lab
dc: iam

# admin, iam.lab
dn: cn=admin,dc=iam,dc=lab
objectClass: simpleSecurityObject
objectClass: organizationalRole
cn: admin
description: LDAP administrator

# People, iam.lab
dn: ou=People,dc=iam,dc=lab
objectClass: organizationalUnit
ou: People

# Groups, iam.lab
dn: ou=Groups,dc=iam,dc=lab
objectClass: organizationalUnit
ou: Groups

# Finance, Groups, iam.lab
dn: cn=Finance,ou=Groups,dc=iam,dc=lab
objectClass: posixGroup
cn: Finance
gidNumber: 5000

# Alice Dupont, People, iam.lab
dn: cn=Alice Dupont,ou=People,dc=iam,dc=lab
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: shadowAccount
cn: Alice Dupont
sn: Dupont
givenName: Alice
description: Active
uid: adupont
uidNumber: 1001
gidNumber: 5000
homeDirectory: /home/adupont
loginShell: /bin/bash

# Bob Martin, People, iam.lab
dn: cn=Bob Martin,ou=People,dc=iam,dc=lab
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: shadowAccount
cn: Bob Martin
sn: Martin
givenName: Bob
description: Inactive
uid: bmartin
uidNumber: 1002
gidNumber: 5001
homeDirectory: /home/bmartin
loginShell: /bin/bash
```

L'apparition des entrées correspondant à Alice et Bob dans le résultat confirme que l'annuaire OpenLDAP est correctement configuré, que la hiérarchie d'unités organisationnelles est en place et que les comptes de test sont disponibles. Le laboratoire dispose ainsi d'un Active Directory minimal mais réaliste, sur lequel les mécanismes IAM pourront être développés et testés dans les chapitres suivants.

2 Mise en place de l'application cible `iam-app`

Une fois l'annuaire OpenLDAP opérationnel, la seconde brique du laboratoire consiste à préparer une application cible jouant le rôle de service SaaS. Dans un environnement réel, cette application pourrait être une solution telle que Salesforce, ServiceNow ou un portail métier interne. Dans le cadre de ce projet, elle est simulée par une petite API web développée en Python avec Flask, déployée sur la machine virtuelle `iam-app` (192.168.56.20).

L'objectif est double. D'une part, disposer d'un point de terminaison simple acceptant des opérations de *provisioning* et de *déprovisioning* d'utilisateurs. D'autre part, faire fonctionner cette application comme un véritable service d'infrastructure, géré par `systemd`, afin de se rapprocher du comportement d'une application de production.

2.1 Implémentation de l'API Flask

L'application est installée dans le répertoire `/opt` de la machine `iam-app`. Le fichier `dummy_app.py` définit une API REST minimale exposant trois opérations : consultation de la liste des utilisateurs, création ou mise à jour d'un utilisateur, et suppression d'un utilisateur. Une journalisation est mise en place dans le fichier `/var/log/dummy_app.log`, ce qui permettra de tracer les actions déclenchées ultérieurement par le moteur IAM.

Le code complet de l'application est présenté ci-dessous.

Listing 2.1: Application Flask simulant la cible SaaS

```
from flask import Flask, request, jsonify
import logging

# Configuration des logs : tout ce qui se passe sera crit dans /var/log/dummy_app.log
logging.basicConfig(filename='/var/log/dummy_app.log', level=logging.INFO,
                    format='%(asctime)s␣-␣%(message)s')

app = Flask(__name__)

# Simule une base de donnes d'utilisateurs en mmoire
users_db = {}

# Endpoint pour voir qui est dans l'app (GET)
@app.route('/api/users', methods=['GET'])
def get_users():
    return jsonify(users_db)
```

```
# Endpoint pour crer ou mettre  jour un utilisateur (POST)
@app.route('/api/users/<uid>', methods=['POST'])
def create_user(uid):
    data = request.json
    # On stocke l'utilisateur dans le dictionnaire
    users_db[uid] = data
    # On crit dans le journal
    logging.info(f"[PROVISIONING] Ajout/Update de l'utilisateur {uid} (Role: {data.get
    ('role')})")
    return jsonify({"status": "success", "msg": f"User {uid} created"}), 201

# Endpoint pour supprimer un utilisateur (DELETE)
@app.route('/api/users/<uid>', methods=['DELETE'])
def delete_user(uid):
    if uid in users_db:
        del users_db[uid]
        logging.info(f"[DE-PROVISIONING] Suppression de l'utilisateur {uid}")
        return jsonify({"status": "success", "msg": "User deleted"}), 200

    logging.warning(f"[ERROR] Tentative de suppression choue : {uid} inconnu.")
    return jsonify({"error": "User not found"}), 404

if __name__ == '__main__':
    # L'app coute sur toutes les interfaces (0.0.0.0) port 5000
    app.run(host='0.0.0.0', port=5000)
```

Ce script ne persiste pas les données dans une base, mais utilise un dictionnaire en mémoire. Cette approche est suffisante pour démontrer la logique de provisionnement et de déprovisionnement orchestrée par le moteur IAM dans les chapitres suivants, tout en gardant le comportement de l'API facilement observable via les journaux.

2.2 Intégration comme service `systemd`

Afin que l'application fonctionne comme un service système classique, elle est intégrée à `systemd` au moyen d'un fichier d'unité dédié, `/etc/systemd/system/dummy-app.service`. Cela permet de démarrer automatiquement le service au boot de la machine, de le relancer en cas de crash et de l'administrer avec les commandes habituelles (`start`, `stop`, `status`).

Le contenu du fichier de service est le suivant.

Listing 2.2: Service `systemd` pour l'application cible

```
[Unit]
Description=Dummy IAM Target Application
After=network.target

[Service]
User=root
WorkingDirectory=/opt
ExecStart=/usr/bin/python3 /opt/dummy_app.py
Restart=always

[Install]
```

```
WantedBy=multi-user.target
```

Après création de ce fichier, la configuration de `systemd` est rechargée puis le service est démarré et activé au démarrage de la machine à l'aide des commandes suivantes, exécutées sur `iam-app` :

```
sudo systemctl daemon-reload
sudo systemctl start dummy-app
sudo systemctl enable dummy-app
```

La figure 2.1 présente l'état du service après démarrage, tel qu'affiché par la commande `systemctl status dummy-app`.

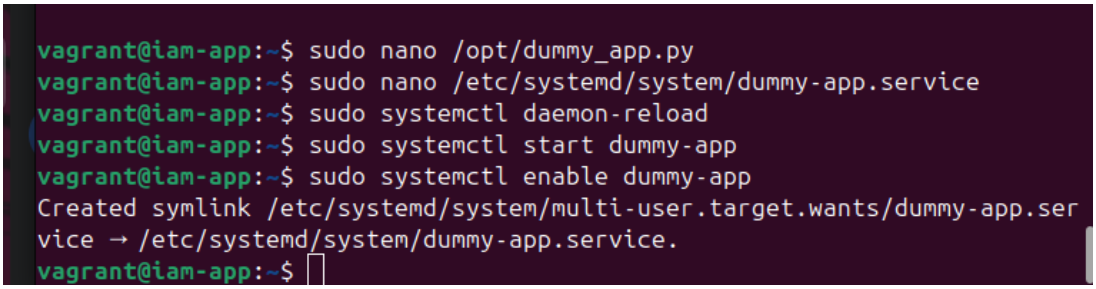

Figure 2.1: Vérification du service `dummy-app` via `systemctl status`.

2.3 Vérification fonctionnelle de l'API

Une fois le service démarré, un premier test fonctionnel est réalisé directement depuis la machine `iam-app` à l'aide de `curl`. L'appel suivant interroge l'endpoint `/api/users` sur le port `5000` :

```
curl http://localhost:5000/api/users
```

La réponse obtenue est un objet JSON vide `{}`, ce qui est cohérent avec l'état initial de l'application, aucun utilisateur n'ayant encore été provisionné. La capture de ce test est illustrée figure 2.2.

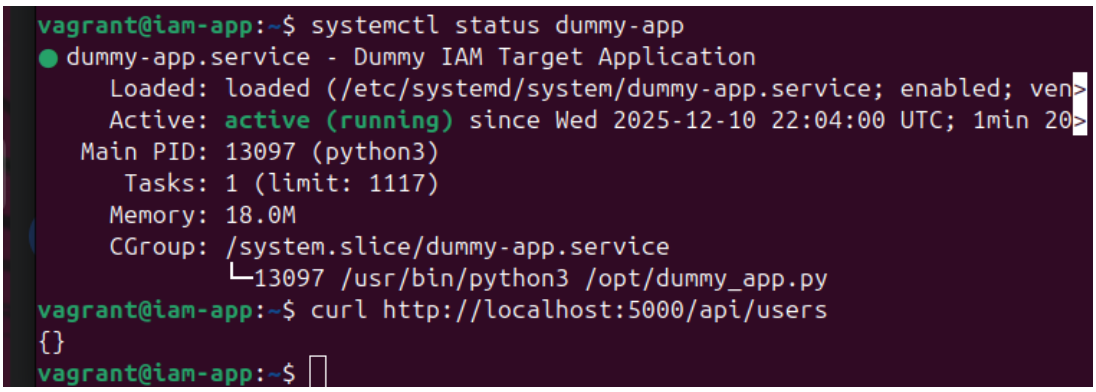

Figure 2.2: Réponse initiale de l'API cible `/api/users` interrogée via `curl`.

À ce stade, l'application cible se comporte comme un service SaaS simplifié. Elle expose une interface HTTP claire pour créer, mettre à jour et supprimer des comptes utilisateurs, et journalise chaque opération dans un fichier de log. Elle constitue désormais une destination idéale pour tester, au chapitre suivant, le cerveau IAM déployé sur la machine `iam-control`, chargé de synchroniser l'état de l'annuaire LDAP avec les comptes présents dans l'application.

3 Moteur de provisioning et chaîne IAM de bout en bout

Les deux premières briques du laboratoire sont désormais en place : l'annuaire OpenLDAP sur `iam-ldap`, qui joue le rôle de source de vérité pour les identités, et l'application Flask sur `iam-app`, qui simule une application SaaS cible. Ce dernier chapitre décrit la mise en place du moteur de provisioning sur la troisième machine `iam-control`, ainsi que le scénario de test qui permet de valider le fonctionnement complet de la chaîne IAM.

3.1 Rôle de la machine `iam-control`

La machine virtuelle `iam-control` (192.168.56.30) incarne le cerveau de l'architecture. C'est sur ce nœud que s'exécute le script de réconciliation chargé de :

- lire l'état des comptes dans l'annuaire LDAP (`iam-ldap`) ;
- appliquer des règles de transformation (construction de l'identifiant technique, mapping groupe → rôle) ;
- appeler l'API de l'application cible (`iam-app`) pour créer, mettre à jour ou supprimer les comptes.

Les bibliothèques Python `ldap3` et `requests` ont été installées lors du provisioning de la machine. Une simple vérification permet de constater leur présence :

```
vagrant ssh iam-control
python3 --version
pip3 list | grep -E "(ldap3|requests)"
```

3.2 Implémentation du moteur de réconciliation

Le cœur logique du projet est regroupé dans un unique script Python, `provisioning_engine.py`, placé sur la machine `iam-control`. Ce script se connecte à l'annuaire LDAP, extrait la liste des utilisateurs, applique des règles de mapping puis appelle l'API de l'application cible pour refléter l'état du référentiel.

Le code complet est reproduit ci-après.

Listing 3.1: Moteur de provisioning IAM sur `iam-control`

```
import ldap3
import requests
import sys

# --- A. CONFIGURATION ---
# Adresses des deux autres VMs (dfinies dans /etc/hosts par Vagrant)
LDAP_HOST = 'iam-ldap'
API_HOST  = 'iam-app'

# Identifiants pour lire l'annuaire
LDAP_USER = 'cn=admin,dc=iam,dc=lab'
LDAP_PWD  = 'admin'

# URL de l'API de l'application cible
APP_URL   = f'http://{API_HOST}:5000/api/users'


# --- B. FONCTIONS UTILITAIRES ---

def get_ldap_users():
    """Se␣connecte␣␣l'AD␣et␣ rcupre ␣tous␣les␣utilisateurs␣humains"""
    print(f"[INFO]␣Connexion␣au␣serveur␣LDAP␣{LDAP_HOST}...")

    # Connexion simple
    server = ldap3.Server(LDAP_HOST, get_info=ldap3.ALL)
    conn = ldap3.Connection(server, LDAP_USER, LDAP_PWD, auto_bind=True)

    # Recherche : On veut les objets 'inetOrgPerson' dans l'unit 'ou=People'
    # On demande des attributs prcis : Prnom, Nom, Description (Statut), GID (Groupe)
    conn.search('ou=People,dc=iam,dc=lab', '(objectClass=inetOrgPerson)',
                attributes=['cn', 'givenName', 'sn', 'description', 'gidNumber'])

    users = []
    for entry in conn.entries:
        users.append({
            'cn': str(entry.cn),           # Nom complet
            'first': str(entry.givenName), # Prnom
            'last': str(entry.sn),         # Nom de famille
            'status': str(entry.description), # Notre champ 'Statut' (Active/Inactive)
            'gid': str(entry.gidNumber)    # ID du groupe (5000=Finance, 5001=IT)
        })
    return users

def generate_uid(first, last):
    """Rgle␣de␣nommage␣:␣Premire␣lettre␣prnom␣+␣Nom␣(ex:␣a.dupont)"""
    # .strip() enlve les espaces inutiles, .lower() met en minuscule
    clean_first = first.strip().lower()
    clean_last = last.strip().lower()
    return f"{clean_first[0]}.{clean_last}"

def determine_role(gid):
    """Matrice␣de␣Rles␣:␣Convertit␣un␣Groupe␣technique␣(GID)␣en␣Rle␣Business"""
    if gid == '5000':
        return 'FINANCIAL_CONTROLLER' # Si groupe Finance
```

```
    if gid == '5001':
        return 'IT_SUPPORT'           # Si groupe IT
    return 'USER_STANDARD'            # Par dfaut


# --- C. MOTEUR DE RCONCILIATION (BOUCLE PRINCIPALE) ---

def reconcile():
    print("---␣Dmarrage␣du␣Cycle␣de␣Provisioning␣---")

    # 1. Lecture de la Source
    try:
        users = get_ldap_users()
    except Exception as e:
        print(f"[ERREUR]␣Impossible␣de␣lire␣le␣LDAP␣:␣{e}")
        return

    print(f"[INFO]␣{len(users)}␣identits␣trouves␣dans␣l'annuaire.")

    # 2. Traitement pour chaque utilisateur
    for u in users:
        # Calcul de l'ID unique
        uid = generate_uid(u['first'], u['last'])

        # CAS 1 : L'utilisateur est ACTIF -> On PROVISIONNE (Cration/Mise  jour)
        if u['status'] == 'Active':
            role = determine_role(u['gid'])

            # Prparation des donnes pour l'API cible
            payload = {
                "fullname": u['cn'],
                "role": role,
                "email": f"{uid}@company.com"
            }

            print(f"[PROVISION]␣Traitement␣de␣{uid}␣(Rle:␣{role})...")
            try:
                # Appel POST vers l'application
                r = requests.post(f"{APP_URL}/{uid}", json=payload)
                if r.status_code == 201:
                    print("␣␣␣␣->␣Succs␣(Compte␣cr/mis␣ ␣jour).")
                else:
                    print(f"␣␣␣␣->␣Erreur␣API␣:␣{r.status_code}")
            except Exception as e:
                print(f"␣␣␣␣->␣Erreur␣de␣connexion␣API␣:␣{e}")

        # CAS 2 : L'utilisateur est INACTIF -> On DPROVISIONNE (Suppression)
        elif u['status'] == 'Inactive':
            print(f"[DE-PROVISION]␣Dpart␣dtect␣pour␣{uid}.␣Suppression␣des␣accs...")
            try:
                # Appel DELETE vers l'application
                r = requests.delete(f"{APP_URL}/{uid}")
                if r.status_code == 200:
                    print("␣␣␣␣->␣Succs␣(Compte␣supprim).")
                elif r.status_code == 404:
                    print("␣␣␣␣->␣Info␣:␣Le␣compte␣n'existait␣dj␣plus.")
```

```
                else:
                    print(f"␣␣␣->␣Erreur␣API␣:␣{r.status_code}")
            except Exception as e:
                print(f"␣␣␣->␣Erreur␣de␣connexion␣API␣:␣{e}")

    print("---␣Fin␣du␣Cycle␣---")

if __name__ == '__main__':
    reconcile()
```

Cette implémentation suit une logique simple mais représentative d'un moteur IAM de production. La fonction `get_ldap_users()` interroge l'unité `ou=People,dc=iam,dc=lab` et reconstruit une liste de dictionnaires Python contenant les attributs pertinents (nom, prénom, statut, groupe). La fonction `generate_uid()` applique une règle de nommage basique pour produire un identifiant technique de type `a.dupont`. La fonction `determine_role()` traduit un identifiant de groupe technique (`gidNumber`) en rôle métier, par exemple `FINANCIAL_CONTROLLER` pour le groupe `5000`.

La fonction `reconcile()` constitue la boucle principale. Pour chaque identité, elle décide soit de provisionner (appel HTTP `POST` vers l'API de la cible avec les informations utilisateur), soit de déprovisionner (appel HTTP `DELETE`), en fonction de la valeur de l'attribut `description` (`Active` ou `Inactive`) définie dans l'annuaire.

3.3 Exécution du cycle de provisioning

Le test final consiste à exécuter ce moteur de réconciliation et à vérifier que l'application cible reflète correctement l'état de l'annuaire.

Depuis la machine `iam-control`, le script est lancé par la commande :

```
python3 provisioning_engine.py
```

La sortie console affiche les différentes étapes du cycle, par exemple :

```
--- Démarrage du Cycle de Provisioning ---
[INFO] Connexion au serveur LDAP iam-ldap...
[INFO] 2 identités trouvées dans l'annuaire.
[PROVISION] Traitement de a.dupont (Rôle: FINANCIAL_CONTROLLER)...
   -> Succès (Compte créé/mis à jour).
[DE-PROVISION] Départ détecté pour b.martin. Suppression des accès...
   -> Succès (Compte supprimé).
--- Fin du Cycle ---
```

La figure 3.1 illustre un exemple d'exécution complète du moteur de provisioning, avec les messages d'information et de succès affichés dans le terminal.

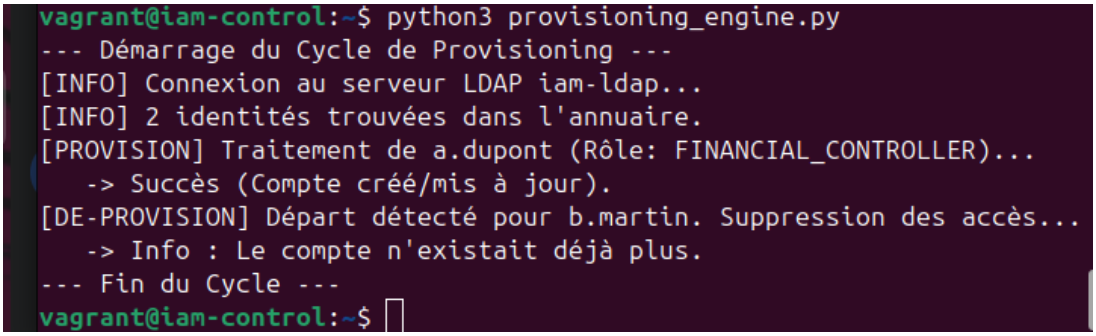

Figure 3.1: Exécution du moteur de provisioning sur `iam-control`.

3.4 Validation sur l'application cible

Afin de vérifier le résultat du cycle de provisioning, l'API de l'application cible est interrogée à partir de `iam-control`. Un simple appel HTTP permet de récupérer la liste des utilisateurs connus par la cible :

```
curl http://iam-app:5000/api/users
```

La réponse attendue est un dictionnaire JSON ne contenant que le compte d'Alice, Bob étant marqué comme inactif dans l'annuaire et donc supprimé de l'application :

```
{
  "a.dupont": {
    "email": "a.dupont@company.com",
    "fullname": "Alice Dupont",
    "role": "FINANCIAL_CONTROLLER"
  }
}
```

La figure 3.2 présente la réponse obtenue, qui confirme le bon fonctionnement du moteur de provisioning : l'état de la cible est aligné sur celui de la source.

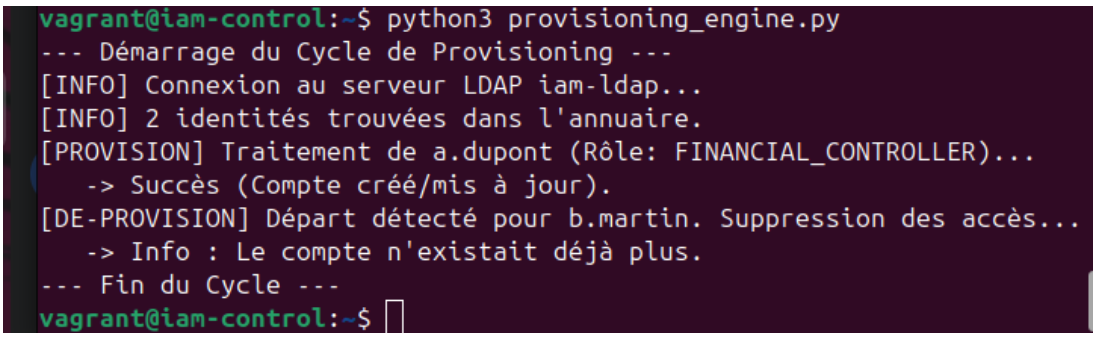

Figure 3.2: Résultat de la réconciliation sur l'API cible `/api/users`.

En complément, il est possible d'ouvrir un second terminal sur `iam-app` et de suivre en temps réel le journal applicatif :

```
vagrant ssh iam-app
sudo tail -f /var/log/dummy_app.log
```

À chaque exécution du script de provisioning, les entrées de log montrent les opérations de création, de mise à jour ou de suppression de comptes effectuées par l'API. Cette observation directe constitue une preuve supplémentaire du comportement attendu de la chaîne IAM.

3.5 Bilan du laboratoire

Au terme de ce projet, une chaîne IAM complète a été mise en place dans un environnement de laboratoire entièrement reproductible. L'objectif initial était de mieux comprendre les défis techniques de l'intégration d'applications en contexte entreprise. Pour cela, un mini-réseau a été construit autour de trois serveurs Linux : un annuaire OpenLDAP jouant le rôle d'Active Directory, une application cible exposant une API REST (Flask) et un serveur de contrôle exécutant le moteur de provisioning.

Ce laboratoire a permis de mettre en œuvre de bout en bout le cycle de vie JML (*Joiner, Mover, Leaver*) :

- côté *Joiner*, la création d'un utilisateur dans l'annuaire LDAP est automatiquement détectée par le script de réconciliation, qui génère un identifiant technique, calcule son rôle et crée le compte correspondant dans l'application cible ;
- côté *Mover*, une matrice de rôles simple a été implémentée : l'appartenance au groupe `Finance` dans LDAP se traduit par l'attribution d'un rôle applicatif renforcé (`FINANCIAL_CONTROLLER`) dans la cible, illustrant le lien entre groupes techniques et droits métiers (RBAC) ;
- côté *Leaver*, le passage d'un compte en état `Inactive` dans l'annuaire entraîne automatiquement le déprovisioning sur l'application, avec suppression des accès et journalisation des opérations.

Sur le plan technique, ce travail a permis de consolider l'usage de bibliothèques Python telles que `ldap3` pour l'interrogation de l'annuaire et `requests` pour l'appel aux APIs REST, ainsi que l'intégration de services applicatifs gérés par `systemd`. Surtout, l'expérience a mis en évidence l'importance d'une source de vérité fiable pour les identités (ici l'annuaire LDAP) et du rôle central joué par le moteur de provisioning dans la cohérence globale des droits.

La structure en trois machines virtuelles, la journalisation systématique et le recours à des composants standards (`OpenLDAP`, `Flask`, `ldap3`, `requests`, `systemd`) offrent enfin une base solide pour des extensions futures : intégration de plusieurs applications cibles, déclenchement en quasi temps réel via des événements, ou encore remplacement du script par une solution IAM industrielle tout en conservant la même logique d'architecture.