Ansible

- AWX
  - Démarrage - Première connexion
  - Création d'un Token API AWX et Commandes Utiles
  - Planifier un Job Template AWX via l'API

- □□ Création d'un Execution Environment (EE) personnalisé pour AWX
- □□ Guide : Installer Ansible proprement via un VENV (Virtual Environment)
- Molecule – Guide complet (Ansible Testing Framework)

AWX

AWX

Démarrage - Première connexion

? 1. Première connexion

Accède à AWX via ton service :

kubectl get svc -n awx

□□ URL = `http://<node-ip>:30080` (dans ton cas)

Login :

- user : `admin`
- password :

kubectl get secret -n awx awx-admin-password -o jsonpath="{.data.password}" | base64 -d

---

?? 2. Configuration minimale (à faire tout de suite)

Dans l'UI AWX :

? Settings ? System

- Time zone → `Europe/Paris`
- (optionnel) définir un base URL si tu exposes proprement

? 3. Créer une Credential SSH

C’est la base de tout.

? Resources ? Credentials ? Add

Type : **Machine**

Remplis :

- Name : `ssh-homelab`
- Username : ton user (`root` ou autre)
- Private key : ta clé SSH

◫◫ Exemple :

cat ~/.ssh/id_rsa

Colle la clé privée.

?? 4. Créer un Inventory

? Resources ? Inventories ? Add

- Name : `homelab`

Puis :

? Hosts ? Add

Exemple :

name: node1
variables:
ansible_host: 10.151.151.10
ansible_user: root

□□ Tu peux aussi ajouter un groupe :

- `k8s`
- `vm`
- `nas`

? 5. Créer un Project (Git)

AWX bosse avec Git, pas avec des fichiers locaux.

? Resources ? Projects ? Add

- Name : `homelab-infra`
- Source Control Type : Git
- URL : ton repo (GitHub / Gitea / GitLab)

□□ Si privé :
→ ajoute une credential Git (token ou clé SSH)

?? 6. Créer un Job Template

C'est là que tout s'exécute.

? Resources ? Templates ? Add ? Job Template

Remplis :

- Name : `Ping test`
- Inventory : `homelab`
- Project : `homelab-infra`
- Playbook : `ping.yml`
- Credentials : `ssh-homelab`

? 7. Exemple de playbook

Dans ton repo Git :

```
# ping.yml
- hosts: all
gather_facts: false
tasks:
- name: Test ping
ping:
```

?? 8. Lancer un job

Clique sur □□

□□ Si tout est bon :

- hosts OK
- SSH OK
- repo OK

➡ tu vois les résultats en live

? 9. Bonus utiles (important en homelab)

?? Variables globales (Inventory ? Variables)

```
ansible_python_interpreter: /usr/bin/python3
```

?? Tester la connectivité

Ajoute un playbook :

```
- hosts: all
tasks:
```

- command: hostname

?? Ajouter sudo

Dans Credential :

- Privilege escalation : ✔
- Method : sudo

? 10. Bonnes pratiques dès le début

□□ Tu es déjà en GitOps avec ArgoCD, donc :

- ✔ versionne :
  - inventories (via SCM)
  - projets
  - playbooks
- □ évite :
  - config manuelle non trackée

? Ce que tu viens de construire

Avec ça tu as :

- AWX = orchestrateur
- Git = source de vérité
- SSH = accès machines
- Kubernetes = infra

□□ Tu peux maintenant :

- déployer des apps
- configurer tes nodes
- gérer ton cluster

AWX

Création d'un Token API AWX et Commandes Utiles

L'API REST d'AWX permet d'automatiser et de piloter l'intégralité de votre infrastructure sans passer par l'interface web. Pour interagir avec cette API en toute sécurité, il est nécessaire de générer un **Personal Access Token (PAT)**.

---

1. Créer un Token d'Accès dans AWX

La génération du token se fait une seule fois depuis l'interface web d'AWX :

1. Connectez-vous à l'interface web d'AWX avec votre compte administrateur.
2. Dans le menu latéral de gauche, allez dans **Access** > **Users**.
3. Cliquez sur votre nom d'utilisateur (ex: *admin*).
4. Allez dans l'onglet **Tokens** en haut de la page.
5. Cliquez sur le bouton **Add** (Ajouter).
6. Remplissez le formulaire :
   - **Description :** Donnez un nom clair (ex: *Token Script Bash Homelab*).
   - **Scope (Portée) :** Sélectionnez **Write** (si vous avez besoin de lancer des jobs) ou **All** .
7. Cliquez sur **Save**.

> ⚠️**ATTENTION :** Le Token généré (une longue chaîne de caractères) ne s'affichera **qu'une seule fois**. Copiez-le immédiatement et conservez-le dans un endroit sûr (comme un gestionnaire de mots de passe ou Vaultwarden).

---

2. Comment utiliser le Token

Pour chaque requête API (souvent via `curl`), vous devez passer ce token dans les en-têtes (Headers) HTTP pour prouver votre identité.

```
# Structure de base de l'authentification
curl -H "Authorization: Bearer VOTRE_TOKEN_ICI" https://awx.domaine.lan/api/v2/...
```

3. Antisèche (Cheat Sheet) des requêtes API utiles

Voici quelques commandes Bash prêtes à l'emploi. Pensez à adapter `$AWX_HOST` et `$TOKEN` avec vos valeurs.

A. Trouver l'ID d'un Job Template

Avant de lancer un playbook, vous avez besoin de connaître son ID. Cette commande liste tous les templates avec leur nom et leur ID.

```
curl -s -k -H "Authorization: Bearer $TOKEN" \
     -X GET "$AWX_HOST/api/v2/job_templates/" \
     | jq '.results[] | "ID: \(.id) - Nom: \(.name)"'
```

Note : L'outil `jq` est utilisé ici pour formater la sortie JSON de manière lisible.

B. Lancer un Job Template immédiatement

Permet de déclencher un playbook à la demande (très utile pour l'intégrer dans d'autres scripts ou pipelines).

```
TEMPLATE_ID="42" # Remplacez par votre ID

curl -s -k -X POST "$AWX_HOST/api/v2/job_templates/$TEMPLATE_ID/launch/" \
     -H "Authorization: Bearer $TOKEN" \
     -H "Content-Type: application/json" \
     -d '{}'
```

C. Lancer un Job Template avec une Limit spécifique

Idéal si vous voulez exécuter un playbook générique (comme "Setup SSH") sur une seule machine spécifique via l'API.

```
TEMPLATE_ID="42"
MACHINE_CIBLE="bookstack"

curl -s -k -X POST "$AWX_HOST/api/v2/job_templates/$TEMPLATE_ID/launch/" \
     -H "Authorization: Bearer $TOKEN" \
     -H "Content-Type: application/json" \
     -d '{
         "limit": "'"$MACHINE_CIBLE"'"
     }'
```

D. Vérifier le statut du dernier Job exécuté

Permet de savoir si le job s'est bien passé (successful) ou s'il a échoué (failed).

```
TEMPLATE_ID="42"

curl -s -k -H "Authorization: Bearer $TOKEN" \
     -X GET "$AWX_HOST/api/v2/job_templates/$TEMPLATE_ID/jobs/?order_by=-id&page_size=1" \
     | jq '.results[0] | "Job ID: \(.id) - Statut: \(.status)"'
```

AWX

Planifier un Job Template AWX via l'API

L'interface graphique d'AWX (Tower) peut parfois s'avérer capricieuse ou peu intuitive pour configurer des tâches planifiées complexes. Utiliser l'API REST via une simple commande `curl` permet de créer des planifications précises, rapides et reproductibles.

---

1. Le Script de Planification

Copiez ce code dans un terminal Linux (ou dans un script bash) pour créer la planification.
Attention à bien modifier les variables avant de l'exécuter.

```
#!/bin/bash

# ========================================
# Variables de configuration
# ========================================
AWX_HOST="https://awx.numericare.fr"
TOKEN="ton_token_api_ici"
TEMPLATE_ID="42"

# ========================================
# Appel API
# ========================================
curl -X POST "$AWX_HOST/api/v2/job_templates/$TEMPLATE_ID/schedules/" \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d '{
        "name": "Conversion Media Quotidienne",
        "description": "Exécution automatique 2 fois par jour",
        "rrule": "DTSTART:20240101T030000Z
RRULE:FREQ=DAILY;INTERVAL=1;BYHOUR=3,15;BYMINUTE=0",
        "extra_data": {},
        "inventory": null
```

```
    }'
```

2. Explication des Variables

Voici comment récupérer et adapter les valeurs nécessaires pour que la commande fonctionne avec votre environnement :

Variable	Description & Comment la trouver
`AWX_HOST`	L'URL de base de votre serveur AWX. *Exemple : https://awx.mon-homelab.lan*
`TOKEN`	Votre jeton d'accès personnel. Pour le générer dans AWX : allez dans **Users** > Cliquez sur votre utilisateur > Onglet **Tokens** > **Add** (Cochez "Write" ou "All").
`TEMPLATE_ID`	L'identifiant unique du playbook à lancer. Pour le trouver, ouvrez votre Job Template dans AWX et regardez l'URL : `/templates/job_template/42/details`. Ici, l'ID est 42.

3. Comprendre la syntaxe `rrule` (La Planification)

Le paramètre le plus complexe de la requête JSON est le `rrule` (Recurrence Rule). Il utilise le standard iCalendar. Voici comment le décoder et le modifier :

> **Syntaxe utilisée :** `DTSTART:20240101T030000Z RRULE:FREQ=DAILY;INTERVAL=1;BYHOUR=3,15;BYMINUTE=0`

- **DTSTART:** Définit la date et l'heure de début de validité de la règle (au format UTC, indiqué par le `Z`). Mettre une date dans le passé fonctionne très bien pour que la règle soit active immédiatement.
- **FREQ=DAILY :** La fréquence de base. Peut être remplacé par `HOURLY` (Toutes les heures), `WEEKLY` (Toutes les semaines), ou `MONTHLY` (Tous les mois).
- **INTERVAL=1 :** S'applique tous les *X* jours (car FREQ=DAILY). Si on met 2, le script tournera un jour sur deux.
- **BYHOUR=3,15 :** Exécute la tâche à 03h00 et à 15h00 **(Heure UTC)**. Attention au décalage horaire selon votre fuseau !
- **BYMINUTE=0 :** S'assure que l'exécution se fait à l'heure pile (00 minute).

Autres exemples de `rrule` utiles :

Toutes les 4 heures :	`... RRULE:FREQ=HOURLY;INTERVAL=4`
Tous les lundis à 2h du matin :	`... RRULE:FREQ=WEEKLY;INTERVAL=1;BYDAY=MO;BYHOUR=2;BYMINUTE=0`

? Création d'un Execution Environment (EE) personnalisé pour AWX

Cette documentation explique comment créer, builder et déployer un environnement d'exécution (EE) sur mesure pour AWX afin d'inclure des dépendances spécifiques (Proxmox, Docker Compose v2) non présentes dans l'image de base.

> **Contexte technique :**
>
> - **Base OS :** CentOS Stream 9 (Image officielle AWX-EE)
> - **Registry :** Harbor (docker-registry.numericare.fr)
> - **Outils :** Docker CLI, Ansible-Galaxy

1. Pourquoi un EE personnalisé ?

L'image de base `awx-ee:latest` est minimaliste. Pour piloter l'infrastructure Numericare, nous avons besoin de :

- **proxmoxer :** Librairie Python pour l'API Proxmox.
- **community.docker (v3.4.0+) :** Pour le support de `docker_compose_v2`.
- **community.proxmox :** Pour la gestion des LXC et VMs.

2. Le Dockerfile de construction

Nous utilisons la méthode manuelle (Dockerfile) plutôt qu'Ansible-Builder pour éviter les conflits de dépendances lourdes (oVirt, etc.) et optimiser l'espace disque.

```
FROM quay.io/ansible/awx-ee:latest


# Passage en root pour l'installation système
USER root
```

```
# 1. Installation de PIP et des librairies Python requises
RUN dnf install -y python3-pip && \
    python3 -m pip install --no-cache-dir proxmoxer requests

# 2. Installation des collections Ansible dans le répertoire GLOBAL
# Note : le flag -p est crucial pour que l'utilisateur non-root d'AWX y accède
RUN ansible-galaxy collection install community.docker:'>=3.4.0' -p
/usr/share/ansible/collections --upgrade && \
    ansible-galaxy collection install community.proxmox -p /usr/share/ansible/collections

# Repassage sur l'utilisateur AWX par défaut (UID 1000)
USER 1000
```

3. Build et Push vers Harbor

Commandes à exécuter depuis le nœud de build (ex: `squall`) :

```
# 1. Connexion à la registry
docker login docker-registry.numericare.fr

# 2. Construction de l'image
docker build -t docker-registry.numericare.fr/private/custom-awx-ee:v1.0 .

# 3. Envoi sur Harbor
docker push docker-registry.numericare.fr/private/custom-awx-ee:v1.0
```

4. Configuration dans AWX

Une fois l'image sur Harbor, il faut la déclarer dans l'interface AWX :

1. **Créer le Credential :**
   - Type : *Container Registry*
   - URL : `https://docker-registry.numericare.fr`
2. **Créer l'Execution Environment :**
   - Image : `docker-registry.numericare.fr/private/custom-awx-ee:v1.0`
   - Pull : `Always pull`
   - Registry Credential : Sélectionner celui créé à l'étape précédente.

3. **Assigner :** Modifier le *Job Template* pour utiliser ce nouvel EE.

?? Troubleshooting & Pièges

1. Espace disque saturé (No space left on device)

Le build d'images crée beaucoup de cache. Sur le nœud de build, lancer régulièrement :

```
docker system prune -af
```

2. ModuleNotFoundError (permissions)

Si Ansible ne trouve pas une collection alors qu'elle est installée, vérifier qu'elle a été installée avec `USER root` dans `/usr/share/ansible/collections` et non dans `/root/.ansible`.

3. Erreur de syntaxe Dockerfile

Attention aux espaces après l'anti-slash (`\ `). Il est préférable de mettre les commandes `RUN` sur une seule ligne pour éviter les erreurs `command not found`.

? Guide : Installer Ansible proprement via un VENV (Virtual Environment)

> **Pourquoi un VENV global ?**

Les distributions modernes (Debian 12+, Ubuntu 24.04+) bloquent l'installation de paquets Python système avec `pip` (erreur *externally-managed-environment*). Installer Ansible via `apt` fournit souvent une version obsolète. La solution est un VENV placé dans `/opt/`, accessible à tous les utilisateurs via des liens symboliques.

1. Installation des prérequis système

En tant qu'utilisateur `root`, installez Python et le gestionnaire d'environnements :

```
apt update
apt install -y python3-pip python3-venv
```

2. Création de l'environnement virtuel

Nous allons créer le dossier de l'environnement dans `/opt/ansible-venv` :

```
python3 -m venv /opt/ansible-venv
```

3. Installation d'Ansible et des dépendances

On utilise le binaire `pip` contenu dans notre VENV pour installer Ansible ainsi que les librairies nécessaires à notre infrastructure (Proxmox, K8s, Docker) :

```
/opt/ansible-venv/bin/pip install --upgrade pip
/opt/ansible-venv/bin/pip install ansible ansible-core proxmoxer requests kubernetes docker
```

4. Création des liens symboliques (La Magie ?)

Pour éviter de devoir "activer" le venv à chaque fois, on crée des liens symboliques dans `/usr/local/bin/`. Ainsi, n'importe quel utilisateur tapant `ansible` utilisera la version du VENV :

```
ln -s /opt/ansible-venv/bin/ansible /usr/local/bin/ansible
ln -s /opt/ansible-venv/bin/ansible-playbook /usr/local/bin/ansible-playbook
ln -s /opt/ansible-venv/bin/ansible-galaxy /usr/local/bin/ansible-galaxy
ln -s /opt/ansible-venv/bin/ansible-vault /usr/local/bin/ansible-vault
```

5. Utilisation et Vérification

Basculez sur votre utilisateur standard et vérifiez que le chemin pointe bien vers `/opt/` :

```
ansible --version
```

Note : Pour mettre à jour Ansible plus tard, il suffira de relancer : `/opt/ansible-venv/bin/pip install --upgrade ansible`

Molecule – Guide complet (Ansible Testing Framework)

Objectif : Comprendre et utiliser Molecule pour tester des rôles Ansible de manière automatisée, reproductible et fiable.

---

1. Qu’est-ce que Molecule ?

Molecule est un framework de test pour **rôles Ansible**. Il permet de vérifier automatiquement qu’un rôle fonctionne correctement dans un environnement simulé (Docker, Podman, LXC ou VM).

Il fait partie de l’écosystème Ansible et est utilisé pour :

- tester des rôles Ansible
- valider des infrastructures de manière reproductible
- automatiser les tests CI/CD
- éviter les régressions en production

---

2. Pourquoi utiliser Molecule ?

Sans Molecule :

- Tests manuels des rôles
- Erreurs découvertes en production
- Aucune reproductibilité

Avec Molecule :

- Tests automatisés
- Environnements jetables
- Validation systématique des rôles

---

3. Le cycle Molecule (IMPORTANT)

Molecule fonctionne selon un cycle standard :

```
create → converge → verify → destroy
```

Étape	Rôle	Explication
Create	Créer l'environnement	Container ou VM vierge
Converge	Appliquer le rôle	Exécution Ansible
Verify	Tester	Validation du résultat
Destroy	Nettoyer	Suppression de l'environnement

Commande globale : `molecule test` exécute tout automatiquement.

---

4. Architecture d'un rôle avec Molecule

```
role/
 └── molecule/
      └── default/
           ├── molecule.yml
           ├── converge.yml
           ├── verify.yml
```

- `molecule.yml` → configuration du scénario
- `converge.yml` → exécution du rôle
- `verify.yml` → tests automatisés

---

5. Installation

```
pip install molecule molecule-docker
# ou
pip install molecule molecule-podman
```

---

6. Création d'un rôle + Molecule

```
ansible-galaxy init my_role
cd my_role
molecule init scenario default
```

7. Exemple molecule.yml

```
driver:
  name: docker

platforms:
  - name: instance
    image: geerlingguy/docker-ubuntu2204-ansible
    privileged: true

provisioner:
  name: ansible

verifier:
  name: ansible
```

8. Exemple converge.yml

```
- name: Converge
  hosts: all
  tasks:
    - name: Include role
      include_role:
        name: my_role
```

9. Exemple verify.yml

```
- name: Verify nginx installation
  hosts: all
  tasks:

    - name: Check nginx version
      command: nginx -v
      register: result
      changed_when: false

    - name: Assert nginx is installed
      assert:
```

```
        that:
          - result.rc == 0
```

10. Commandes essentielles

```
molecule create
molecule converge
molecule verify
molecule destroy
molecule test
```

11. Debug

```
molecule login
molecule destroy
molecule test --debug
```

12. Drivers disponibles

- Docker (standard)
- Podman (rootless)
- LXC (possible mais complexe)
- VM (Vagrant)

Attention : Docker dans LXC nécessite nesting=1 côté Proxmox.

13. Bonnes pratiques

- 1 rôle = 1 scénario Molecule
- toujours ajouter verify.yml
- tester plusieurs distributions si possible
- intégrer CI/CD

14. Résumé

Molecule transforme un rôle Ansible en système testable automatiquement.

- ✔ Moins d’erreurs en production
- ✔ Tests reproductibles
- ✔ Infra industrialisée