Ansible
Automatisation sans agent
Atelier technique DevSecOps

Thierry Gayet
thierry.gayet@gmail.com

Association LinuxMaine

25 avril 2026 à 14h

Plan I

Plan II

Comment se déroule ce module ?

Structure pédagogique

1. **Cours** : concept théorique + vocabulaire
2. **À retenir** : les points clés avant le TP
3. **TP** : mise en pratique pas à pas
4. **Checkpoint** : valider les acquis
5. Passage au thème suivant

Ce que vous allez apprendre

- Comprendre l'architecture sans agent d'Ansible
- Écrire des playbooks YAML idempotents
- Organiser le code en rôles réutilisables
- Chiffrer les secrets avec Vault
- Utiliser Galaxy pour réutiliser la communauté
- Déployer des stacks complètes (LAMP, monitoring)

Conseil

N'hésitez pas à poser des questions à chaque transition cours $\rightarrow$ TP et TP $\rightarrow$ checkpoint.

Dépôt Git & Accès au Lab

Cloner le dépôt

```
git clone https://framagit.org/linuxmaine/formations.git
cd formations/devsecops/ansible/
```

Accès Framagit

- URL : `https://framagit.org/linuxmaine/formations`
- Répertoire : `devsecops/ansible/`

Accès VM QEMU (Lab)

- SSH : `ssh -p 2222 root@localhost`
- Utilisateur : `root`
- Mot de passe : `linuxmaine`
- Clé SSH : `~/.ssh/ansible_key`

```
# Cloner et acceder aux labs
git clone https://framagit.org/linuxmaine/formations.git
cd formations/devsecops/ansible/
```

Diffusion en streaming YouTube

La séance est diffusée en direct

- **Lien YouTube Live** : `https://youtube.com/live/fkeTvScPLag`
- Posez vos questions dans le **chat** en direct
- Le replay sera disponible après la séance
- Pensez à **liker** et **vous abonner** à la chaîne LinuxMaine

Comment suivre

- Ouvrir le lien dans un navigateur
- Activer le son et la qualité HD
- Participer au chat

Après la séance

- Replay et chapitrage disponibles
- Rédiffusion partagée sur Framagit
- Commentaires ouverts sous la vidéo

Qu'est-ce que le Provisioning ?

- Processus d'**installation et de configuration automatisée** d'un système
- Infrastructure as Code (IaC) : l'infrastructure décrite dans des fichiers versionnés
- **Idempotence** : exécuter N fois = même résultat qu'une seule fois
- Deux approches : **impérative** (comment faire) vs **déclarative** (quoi obtenir)
- Deux modèles : **Push** (le serveur envoie) vs **Pull** (le client récupère)

Objectif

Remplacer les opérations manuelles par du code reproductible, versionné et testable.

Pourquoi automatiser le provisioning ?

- Le provisioning manuel est **lent** et source d'**erreurs humaines**
- Garantir la **cohérence** entre les environnements (dev, staging, prod)
- **Reproductibilité** : recréer un environnement identique à tout moment
- **Scalabilité** : provisionner 1 ou 100 serveurs avec le même effort
- **Documentation as Code** : l'infra est décrite dans des fichiers lisibles
- **Piste d'audit** complète grâce à l'historique Git (qui, quoi, quand)

Infrastructure as Code — Les principes

- Décrire l'**état désiré** (desired state) plutôt que les étapes à exécuter
- **Versionner** le code d'infrastructure dans Git
- **Revue de code** via Pull Requests avant tout changement
- **Tester** l'infrastructure (lint, dry-run, tests d'intégration)
- **Idempotence** : appliquer N fois = même résultat qu'une seule fois
- Infrastructure **immutable** vs **mutable** : remplacer plutôt que modifier

IaC — Les bénéfices concrets

Sans IaC

- Connexions SSH manuelles serveur par serveur
- Documentation Word obsolète dès le lendemain
- Serveurs “snowflake” tous différents
- “Ca marche sur ma machine” en production
- Rollback = espérer et prier

Avec IaC

- Infrastructure reproductible à 100%
- Code versionné dans Git (historique complet)
- Testable : lint, dry-run, tests d’intégration
- Auditée : chaque changement est tracé
- Rollback = `git revert` + apply

Impératif vs Déclaratif

Impératif (comment faire)

- Scripts Shell / Bash
- Ansible ad-hoc (commandes ponctuelles)
- Chef recipes (Ruby procédural)
- On décrit les étapes séquentielles
- Ordre d'exécution important

Déclaratif (quoi obtenir)

- Terraform / OpenTofu (HCL)
- Puppet manifests (DSL déclaratif)
- Ansible playbooks (YAML déclaratif)
- Salt states (YAML déclaratif)
- Le moteur calcule les actions nécessaires

Push vs Pull

Push (le contrôleur pousse)

- Ansible : connexion SSH depuis le controller
- Terraform : appel API vers les providers
- Pas d'agent à installer sur les cibles
- Exécution à la demande (on-demand)
- Idéal pour les déploiements ponctuels

Pull (les noeuds tirent)

- Puppet agent : interroge le Puppet Server
- Chef client : interroge le Chef Server
- Salt minion : écoute le Salt Master
- Agent installé sur chaque noeud
- Exécution périodique (convergence)

Historique du Provisioning

- **2005** : Puppet — premier outil moderne de gestion de configuration
- **2009** : Chef — automatisation en Ruby DSL
- **2011** : SaltStack — orchestration haute performance (ZeroMQ)
- **2012** : Ansible — simplicité, sans agent, YAML
- **2014** : Terraform — IaC multi-cloud (HCL)
- **2019** : Pulumi — IaC en langages standards (Python, Go, TS)
- **2023** : OpenTofu — fork open source de Terraform (Linux Foundation)

Tendance : de l'impératif (scripts) vers le déclaratif (IaC), du pull vers le push

Panorama des outils

Avec agent (Pull)

- **Puppet** — DSL propre, entreprise, mature
- **Chef** — Ruby DSL, flexible, complexe
- **SaltStack** — YAML, rapide (ZeroMQ), push+pull
- **CFEngine** — C, très performant, bas niveau

Sans agent (Push)

- **Ansible** — YAML, SSH, simple, très populaire
- **Terraform** — HCL, infra cloud, multi-provider (BSL)
- **OpenTofu** — Fork open source de Terraform (MPL 2.0)
- **Pulumi** — Langages standards (Python, Go, TS)

DevSecOps — Rappel

- Extension de DevOps : **Security** intégrée à chaque étape
- **Shift-Left** : la sécurité commence dès le code, pas en fin de cycle
- Automatisation des contrôles de sécurité dans le pipeline CI/CD
- Outils clés : SAST, SCA, DAST, scanning d'images, SBOM
- Le provisioning sécurisé est un pilier du DevSecOps

Lien avec le provisioning

Chaque outil de provisioning doit être intégré dans une démarche DevSecOps : lint du code IaC, scanning de sécurité, audit des configurations.

Présentation d'Ansible

- Outil d'automatisation **sans agent** : connexion via SSH (Linux) ou WinRM (Windows)
- Configuration décrite en **YAML** — lisible par tous, pas seulement les développeurs
- **Idempotent** : exécuter N fois = même résultat qu'une seule fois
- Modèle **Push** : le control node pousse la configuration vers les cibles
- Écrit en **Python** — extensible via modules et plugins
- **+30 000 modules** disponibles via Ansible Galaxy et les collections
- Utilisé par **Red Hat**, NASA, Twitter, Verizon, CERN...

Philosophie

Simple, agentless, powerful — automatiser sans complexité inutile.

Ansible — C'est quoi, en clair ?

Analogie

Imaginez que vous devez configurer 50 serveurs identiques. **Sans Ansible** : vous vous connectez un par un en SSH et tapez les commandes manuellement. **Avec Ansible** : vous écrivez une « recette» (playbook) une seule fois, et Ansible l'applique sur les 50 serveurs en parallèle.

- **SSH** : protocole de connexion sécurisée à distance (comme un tunnel chiffré)
- **YAML** : format de fichier lisible par un humain (indentation, tirets, deux-points)
- **Idempotent** : si le serveur est déjà dans l'état voulu, Ansible ne touche à rien
- **Sans agent** : pas besoin d'installer de logiciel sur les machines cibles

En résumé

Ansible = un outil qui automatise la configuration de serveurs distants via SSH, en décrivant l'état souhaité dans des fichiers YAML.

Historique d'Ansible

- **2012** : Création par **Michael DeHaan** (ex-Puppet, Cobbler)
- **2013** : Ansible 1.0 — première version stable, adoption rapide
- **2014** : Ansible Tower (interface web, RBAC, API REST)
- **2015** : **Rachat par Red Hat** pour 150M$ — intégration entreprise
- **2019** : Ansible 2.9 — dernière version monolithique
- **2020** : Séparation **ansible-core** + **collections** — architecture modulaire
- **2023** : **Ansible Lightspeed** (IA) — génération de playbooks via Watson X
- **2024** : **Event-Driven Ansible** (EDA) — automatisation réactive

De l'outil CLI simple à la plateforme d'automatisation complète (Red Hat AAP).

Architecture Ansible

Composants

- **Control Node** : machine qui exécute Ansible
- **Managed Nodes** : machines cibles (SSH)
- **Inventory** : liste des hôtes et groupes
- **Playbooks** : fichiers YAML décrivant l'état
- **Modules** : unités d'action (apt, copy, service...)
- **Roles** : structure réutilisable (tasks, handlers, vars...)

Fonctionnement

1. Lire l'**inventaire** (hôtes, groupes, variables)
2. Établir une connexion **SSH** vers chaque cible
3. **Copier** les modules Python sur la cible (`/tmp`)
4. **Exécuter** les modules sur la cible
5. Récupérer les **résultats JSON** (changed, ok, failed)
6. **Nettoyage** : suppression des fichiers temporaires

Pourquoi Ansible ?

Avantages

- **Sans agent** : rien à installer sur les cibles
- **YAML** : courbe d'apprentissage douce
- **Communauté** : +30 000 modules, Galaxy
- **Multi-cloud** : AWS, Azure, GCP, VMware...
- **Red Hat** : support entreprise (AAP)
- **Écosystème** : AWX, Molecule, ansible-lint, ARA

Limites

- **Scalabilité** : lent au-delà de 5 000 nœuds
- **Pas d'état** : pas de state file (vs Terraform)
- **Debug YAML** : erreurs parfois cryptiques
- **Windows** : support WinRM moins mature
- **Performance** : SSH séquentiel par défaut
- **Logique complexe** : Jinja2 peut devenir illisible

À retenir — Introduction

Concepts clés de cette section

- Ansible = automatisation **sans agent** via SSH (pas d'agent à maintenir)
- **Push** (vs pull de Puppet/Chef) : le Control Node initie la connexion
- **YAML + Jinja2** comme langage de description d'état
- **Idempotent par conception** : relancer ne casse rien
- Architecture : Control Node → SSH → Managed Nodes

Vocabulaire essentiel à connaître avant les TP

- **Control Node** : votre machine, elle exécute `ansible` / `ansible-playbook`
- **Managed Node** : la cible (VM, serveur) sur laquelle les actions s'appliquent
- **Inventory** : liste statique/dynamique des Managed Nodes
- **Playbook** : fichier YAML qui décrit *quoi* faire sur *qui*
- **Module** : unité d'action (apt, copy, service...)

Prochaine étape : installer l'environnement et valider la connexion Ansible.

Prérequis de la formation

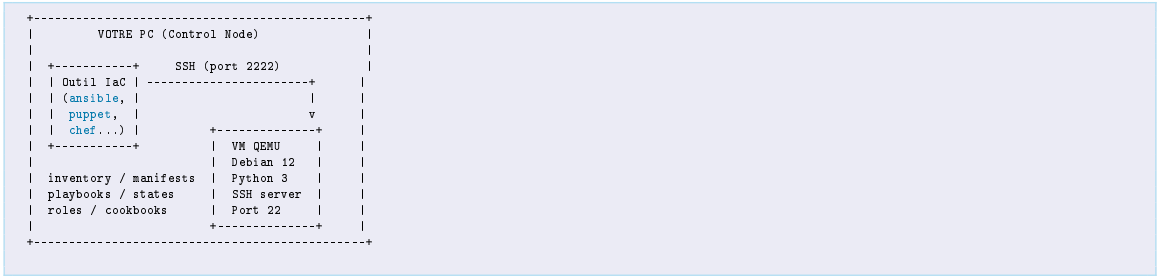
Matériel & Système

- PC avec **Linux** (Debian/Ubuntu ou Fedora)
- Minimum **4 Go de RAM** (2 Go pour la VM)
- **10 Go** d'espace disque libre
- Processeur supportant la virtualisation (**VT-x** / AMD-V)
- Accès Internet pour télécharger les images

Logiciels requis

- **QEMU/KVM** — émulateur et hyperviseur
- **cloud-image-utils** — générer le seed cloud-init
- **OpenSSH client** — connexion SSH
- **Python 3.10+** — requis par les outils modernes
- **Éditeur** : VS Code (extensions IaC) ou vim/nano
- **Git** — versionner le code d'infrastructure

Architecture du Lab



Principe

Un seul PC suffit : tout tourne en local. La VM simule un serveur distant.

QEMU — C'est quoi ?

Présentation

- **Quick EMUlator** : émulateur et virtualiseur open source
- Émule une machine complète (CPU, RAM, disque, réseau)
- Avec **KVM** : performances quasi-natives (accélération matérielle)
- Utilisé par : **libvirt**, Proxmox, OpenStack, Android Studio
- Léger : pas besoin de VirtualBox ou VMware

Concepts clés

- **qcow2** : format natif QEMU (snapshots, compression)
- **raw** : format brut (rapide, mais gros fichier)
- **qemu-img** : créer/convertir/redimensionner des images
- **Mode user** : NAT simple, pas de config réseau
- **hostfwd** : `tcp::2222-:22` redirige le port local

Pourquoi QEMU ? Léger, en ligne de commande, scriptable, idéal pour les labs automatisés.

Cloud-init — Configuration initiale des VMs

Présentation

- Outil **standard** de configuration initiale des VMs cloud
- Utilisé par AWS, GCP, Azure, OpenStack, Proxmox, QEMU
- Configure au **premier boot** : hostname, users, SSH keys, paquets
- Fichier `user-data` en YAML
- Seed ISO : `cloud-localds seed.img cloud-init.yml`

Exemple cloud-init.yml

```
#cloud-config
hostname: lab-vm
users:
  - name: deploy
    sudo: ALL=(ALL) NOPASSWD:ALL
    ssh_authorized_keys:
      - ssh-ed25519 AAAA... user@host
packages:
  - python3
  - curl
  - vim
```

Démarrage du Lab — Commandes

```
# 1. Installer QEMU et cloud-image-utils
$ sudo apt install -y qemu-system-x86 cloud-image-utils

# 2. Telecharger l'image cloud Debian 12
$ wget https://cloud.debian.org/images/cloud/bookworm/ \
    latest/debian-12-generic-amd64.qcow2

# 3. Redimensionner l'image disque
$ qemu-img resize debian-12-generic-amd64.qcow2 10G

# 4. Creer le seed cloud-init
$ cloud-localds seed.img cloud-init.yml

# 5. Lancer la VM
$ qemu-system-x86_64 -m 2G -smp 2 -nographic \
    -drive file=debian-12-generic-amd64.qcow2 \
    -drive file=seed.img,format=raw \
    -net nic -net user,hostfwd=tcp::2222-:22 &

# 6. Tester la connexion SSH
$ ssh -p 2222 deploy@localhost "hostname && id"
```

Cours — Pré-requis du lab

Composants du lab DevSecOps

- **QEMU/KVM** : virtualisation matricielle (émulation + accélération matérielle)
- **cloud-init** : provisionnement automatique au premier boot (hostname, SSH, paquets)
- **Image cloud Debian 12** : qcow2 minimal, auto-configurable
- **Clé SSH ed25519** : identité d'Ansible sur la cible
- **Port-forwarding** : `localhost:2222` → `VM:22`

Pour les débutants

Toute la suite des TP suppose que votre VM est accessible via `ssh -p 2222 root@127.0.0.1`. Si ce ping SSH ne passe pas, inutile d'aller au TP1.

Cours — Installation d'Ansible

```
# Installation via pip (recommande)
$ python3 -m pip install --user ansible
$ ansible --version
ansible [core 2.17.0]

# Installation systeme (Debian/Ubuntu)
$ sudo apt update && sudo apt install ansible
```

ansible.cfg

```
[defaults]
inventory      = ./inventory/hosts.yml
remote_user    = deploy
host_key_checking = False
forks          = 20
[privilege_escalation]
become         = True
become_method  = sudo
```

À retenir — Avant les TP 0 et 1

Chemin d'installation

1. Installer QEMU + cloud-localds
2. Générer une clé SSH ed25519
3. Démarrer la VM avec `tp00-setup.sh`
4. Attendre que cloud-init termine (60s)
5. Tester en SSH puis installer Ansible

Checks après TP 0 et 1

- `ssh -p 2222 root@127.0.0.1 hostname` → `debian-lab`
- `ansible -version` → core 2.x
- `ansible lab -m ping` → `pong`

TP 0 — Setup Lab

Objectif

Préparer une VM Debian 12 avec QEMU/KVM et cloud-init pour les TPs.

Étapes

1. Téléchargement de l'image cloud Debian 12
2. Redimensionnement du disque qcow2 à 10 Go
3. Génération du seed ISO cloud-init
4. Lancement de QEMU avec port-forward SSH
5. Vérification SSH vers la VM

```
# Telecharger l'image cloud Debian 12
$ wget https://cloud.debian.org/images/cloud/bookworm/ \
    latest/debian-12-generic-amd64.qcow2
# Creer la VM avec cloud-init (user: deploy, SSH key)
$ qemu-img resize debian-12-generic-amd64.qcow2 10G
$ cloud-localds seed.img cloud-init.yml
$ qemu-system-x86_64 -m 2G -smp 2 -nographic \
    -drive file=debian-12-generic-amd64.qcow2 \
    -drive file=seed.img,format=raw \
    -net nic -net user,hostfwd=tcp::2222-:22
# Verification SSH
$ ssh -p 2222 deploy@localhost "hostname && id"
```

TP 1 — Installation Ansible

Objectif

Installer Ansible, configurer `ansible.cfg` et vérifier la connexion.

Étapes

1. Installer Ansible via pip (+ ansible-lint)
2. Créer un dossier projet `~/tp-ansible/`
3. Rédiger `ansible.cfg` (inventaire par défaut, SSH)
4. Rédiger `inventory/hosts.yml`
5. Lancer le premier `ansible all -m ping`

```
# Installer Ansible via pip
$ python3 -m pip install --user ansible ansible-lint
$ ansible --version

# Creer le fichier de configuration
$ mkdir -p ~/tp-ansible && cd ~/tp-ansible
$ cat > ansible.cfg << 'EOF'
[defaults]
inventory = ./inventory/hosts.yml
remote_user = deploy
host_key_checking = False
EOF

# Premier test de connexion
$ mkdir inventory
```

Checkpoint — Mise en place

Vous devez pouvoir

- Lancer `./tp00-setup.sh -bg` et voir la VM démarrer
- Vous connecter par SSH avec la clé `~/.ssh/ansible_key`
- Exécuter `ansible lab -m ping` qui retourne pong
- Lire `ansible -version` et repérer le `config file` utilisé

Si ça ne marche pas

- Port 2222 occupé : `ss -tlnp | grep :2222`
- Permission denied : `chmod 600 ~/.ssh/ansible_key`
- Host key changed : `ssh-keygen -R '[127.0.0.1]:2222'`
- `config file = None` : cd dans le répertoire du projet

Cours — Inventaire

Format INI

```
[webservers]
web1 ansible_host=10.0.0.11
web2 ansible_host=10.0.0.12

[databases]
db1 ansible_host=10.0.0.21

[prod:children]
webservers
databases
```

Format YAML

```
all:
  children:
    webservers:
      hosts:
        web1:
          ansible_host: 10.0.0.11
        web2:
          ansible_host: 10.0.0.12
    databases:
      hosts:
        db1:
          ansible_host: 10.0.0.21
```

Variables : `host_vars/web1.yml`, `group_vars/webservers.yml`

Pour les débutants

L'inventaire est comme un **carnet d'adresses** : il liste toutes les machines que vous voulez gérer, regroupées par fonction (web, base de données…).

À retenir — Inventaire

Les 3 briques

- **Hôtes** : entrées avec `ansible_host`, `ansible_port`...
- **Groupes** : rassemblent des hôtes par fonction
- **Variables** : par hôte (`host_vars/`) ou par groupe (`group_vars/`)

Groupes implicites

- `all` : tous les hôtes de l'inventaire
- `ungrouped` : hôtes sans groupe explicite
- `<group>:children` : groupe de groupes
- `<group>:vars` : variables partagées par le groupe

Commandes ad-hoc

- `ansible <pattern> -m <module> -a "<args>"`
- `-b` = become (sudo) ; `-i` = inventaire ; `-l` = limiter

Pattern avancés : `web:&prod`, `all:!dev`, `~web\d+`

TP 2 — Inventaire & Commandes ad-hoc

Objectif

Maîtriser l'inventaire et les commandes ad-hoc.

Étapes

1. Lister l'inventaire en YAML avec `ansible-inventory`
2. Tester la connectivité avec `ping` et `setup`
3. Installer un paquet avec `apt` et `-b`
4. Cibler un groupe spécifique
5. Valider l'idempotence en relançant `apt`

```
# Lister les hotes de l'inventaire
$ ansible-inventory --list --yaml

# Commandes ad-hoc
$ ansible all -m ping
$ ansible all -m shell -a "uname -a"
$ ansible all -m setup -a "filter=ansible_os_family"
$ ansible all -m apt -a "name=curl state=present" -b

# Grouper les hotes
$ ansible webservers -m shell -a "systemctl status nginx"
$ ansible databases -m shell -a "pg_isready"
```

`-m` = module, `-a` = arguments, `-b` = become (sudo).

Checkpoint — Inventaire

Vous savez

- Lire `ansible-inventory -graph`
- Distinguer inventaire INI et YAML
- Utiliser `[group:vars]` vs `[group]`
- Exécuter 5+ modules ad-hoc (ping, setup, shell, apt, user)
- Ajouter `-b` uniquement quand c'est nécessaire

Pièges courants

- `command` interpréte les `$VAR` ? **Non**, utilisez `shell` pour les pipes / redirections
- `apt` sans `-b` à la 1re exécution → `Failed to lock apt`
- Mauvais nom de groupe → `No hosts matched`

Cours — Modules essentiels

Système & Fichiers

- **apt / yum / dnf** : gestion paquets
- **copy** : copier des fichiers vers la cible
- **file** : créer/supprimer fichiers, liens, permissions
- **template** : copier avec rendu Jinja2
- **lineinfile** : modifier une ligne dans un fichier
- **command / shell** : commandes brutes

Services & Réseau

- **service / systemd** : démarrer/arrêter services
- **user / group** : gestion utilisateurs
- **uri** : requêtes HTTP (health checks)
- **cron** : planifier des tâches
- **git** : cloner des dépôts
- **docker_container** : gestion Docker

Conseil

Privilégier les modules dédiés (`apt`) plutôt que `shell` pour garantir l'idempotence.

À retenir — Modules

Comment fonctionne un module ?

1. Ansible copie le module Python sur la cible (`/tmp/.ansible/...`)
2. Le module s'exécute **sur la cible** avec les arguments fournis
3. Il renvoie un résultat JSON : `changed: true/false`, `failed: true/false`
4. Ansible nettoie le fichier temporaire

Module idéal = idempotent

- **apt/user/file/copy/service** : idempotents (vérifient l'état)
- **command/shell** : *toujours* `changed` (sauf `changed_when:/creates:`)
- **wait_for** : synchronisation d'événements (port ouvert...)
- **uri** : vérification HTTP

Antipattern

- `shell: apt install ...` au lieu de `apt: name=... state=present`

TP 3 — Modules

Objectif

Pratiquer les modules essentiels : apt, user, file, copy, service, uri.

Étapes

1. Installer un paquet (`apt`)
2. Créer un utilisateur (`user`)
3. Créer un répertoire avec droits (`file`)
4. Copier un fichier (`copy`)
5. Démarrer un service (`service`)
6. Vérifier un endpoint HTTP (`uri`)

```
# Installer un paquet
$ ansible all -m apt -a "name=htop state=present" -b
# Creer un utilisateur
$ ansible all -m user -a "name=webadmin shell=/bin/bash \
    groups=sudo" -b
# Creer un repertoire
$ ansible all -m file -a "path=/opt/app state=directory \
    owner=webadmin mode=0755" -b
# Copier un fichier
$ ansible all -m copy -a "src=motd.txt \
    dest=/etc/motd" -b
# Demarrer un service
$ ansible all -m service -a "name=ssh state=started \
```

Checkpoint — Modules

Vous devez pouvoir

- Utiliser `apt`, `user`, `file`, `copy`, `service`, `uri`
- Observer `changed=true` au premier run puis `changed=false` au second
- Prévisualiser avec `-check -diff` (dry-run)
- Choisir le bon module plutôt que `shell`

Erreurs classiques

- Oubli de `-b` sur `apt` / `service` / `file` en `/etc`
- `mode: 0755` au lieu de `mode: "0755"` (string)
- `uri` avant `wait_for` → service pas prêt

Cours — Playbooks, anatomie

site.yml

```yaml
---
- name: Configurer les serveurs web
  hosts: webservers
  become: true
  vars:
    http_port: 80
  tasks:
    - name: Installer nginx
      ansible.builtin.apt:
        name: nginx
        state: present
        update_cache: true
    - name: Deployer la page d'accueil
      ansible.builtin.template:
        src: index.html.j2
        dest: /var/www/html/index.html
      notify: Recharger nginx
  handlers:
    - name: Recharger nginx
      ansible.builtin.service:
        name: nginx
        state: reloaded
```

Lire un playbook

Un playbook se lit de haut en bas : **qui** (hosts) → **avec quels droits** (become) → **quelles variables** (vars) → **quelles actions** (tasks) → **quelles réactions** (handlers).

À retenir — Playbooks

Structure

- **Play** : groupe de tasks pour un groupe d'hôtes
- **Task** : appelle un module avec des arguments
- **Handler** : tâche déclenchée uniquement sur `notify`
- **Vars** : paramètres du play (surchageables)

Règles de base

- Toujours donner un `name:` à chaque task (lisibilité)
- Préférer les FQCN : `ansible.builtin.apt` (pas `apt`)
- `become: true` au niveau du play si la plupart des tasks exigent root
- Tester avec `-check -diff` avant d'appliquer

Cycle de vie

- `pre_tasks` → `roles` → `tasks` → `post_tasks` → handlers (fin de play)

TP 4 — Premier Playbook

Objectif

Écrire un playbook pour installer nginx et déployer une page. Tester l'idempotence.

Étapes

1. Écrire `playbooks/nginx.yml`
2. Lancer `ansible-playbook playbooks/nginx.yml`
3. Vérifier avec `curl http://localhost`
4. Relancer : observer `changed=0`
5. Essayer `-check -diff` après modification

playbooks/nginx.yml

```yaml
---
- name: Installer et configurer nginx
  hosts: webservers
  become: true
  tasks:
    - name: Installer nginx
      ansible.builtin.apt:
        name: nginx
        state: present
        update_cache: true
    - name: Deployer la page d'accueil
      ansible.builtin.copy:
        content: "<h1>Hello LinuxMaine !</h1>"
        dest: /var/www/html/index.html
    - name: Demarrer nginx
      ansible.builtin.service:
        name: nginx
        state: started
        enabled: true
```

Lancer 2 fois : la 2e exécution = **changed=0** (idempotence).

Checkpoint — Playbooks

Vous maîtrisez

- Écrire un playbook YAML valide
- Lire `PLAY RECAP` (ok, changed, unreachable, failed…)
- Vérifier l'idempotence (`changed=0` au 2^e run)
- Utiliser `-syntax-check` et `-check -diff`

Cours — Variables, Facts, Jinja2

Priorité des variables (du moins au plus prioritaire) : command line → role defaults → inventory → group_vars → host_vars → play vars → task vars → extra vars (-e)

group_vars/webservers.yml

```
---
http_port: 80
app_name: "monapp"
packages:
  - nginx
  - curl
```

Utilisation des facts et filtres Jinja2

```
- name: Afficher l'OS
  debug:
    msg: "OS: {{ ansible_distribution }} {{ ansible_distribution_version }}"
- name: Filtre Jinja2
  debug:
    msg: "{{ packages | join(', ') | upper }}"
```

À retenir — Variables

Les 5 endroits utiles en pratique

1. `defaults/main.yml` du rôle : priorité basse, facile à surcharger
2. `group_vars/<group>.yml` : partagé par un groupe
3. `host_vars/<host>.yml` : spécifique à un hôte
4. `vars:` dans le play : override local
5. `-e "k=v"` sur la CLI : override maximum

Facts

- Collectés par `gather_facts` (défaut: yes)
- Préfixe `ansible_*` : `ansible_distribution`, `ansible_memtotal_mb`...
- Accès Jinja2 : `{{ ansible_hostname }}`

Pièges

- `"false"` (string) vs `false` (bool) : ne pas quoter les booléens
- `inventory_hostname` (inventaire) vs `ansible_hostname` (fact collecté)

TP 5 — Variables

Objectif

Utiliser group_vars, facts et override avec -e.

Étapes

1. Créer `group_vars/webservers.yml`
2. Écrire un playbook qui utilise `app_name` et `packages`
3. Lancer le playbook, observer
4. Surcharger avec `-e "app_name=superapp"`
5. Comparer : qui a gagné la priorité ?

group_vars/webservers.yml

```yaml
---
http_port: 80
app_name: "monapp"
packages: [nginx, curl, htop]
```

playbooks/variables.yml

```yaml
- hosts: webservers
  become: true
  tasks:
    - debug:
        msg: "{{ app_name }} sur {{ ansible_hostname }}"
    - apt:
        name: "{{ item }}"
        state: present
      loop: "{{ packages }}"
```

```
# Override avec extra vars
$ ansible-playbook playbooks/variables.yml \
    -e "app_name=superapp"
```

Checkpoint — Variables

Vous savez

- Placer une variable au bon niveau (groupe, hôte, play, CLI)
- Utiliser un fact dans une tâche
- Déboguer avec `debug: var=x`
- Interpréter la priorité lorsque plusieurs niveaux définissent la même variable

Cours — Conditions
beginframe[fragile]Cours — Conditions & Boucles Boucles

Conditions avec when

```
- name: Installer sur Debian uniquement
  ansible.builtin.apt:
    name: nginx
    state: present
  when: ansible_os_family == "Debian"
```

Boucles avec loop

```
- name: Creer des utilisateurs
  ansible.builtin.user:
    name: "{{ item.name }}"
    groups: "{{ item.groups }}"
  loop:
    - { name: alice, groups: sudo }
    - { name: bob, groups: docker }
```

Autres : `loop` + `dict2items`, `with_fileglob`, `until` (retry).

À retenir — Conditions & Boucles

when

- **Pas de** `{{ }}` dans `when:` (Jinja2 implicite)
- Combinable : `when: a and (b or c)`
- Tests : `is defined`, `is not none`, `in`, `not in`

loop

- `loop: [a, b, c]` ou `loop: "{{ list }}"`
- `dict2items` pour itérer sur un dict
- `loop_control: { label: "{{ item.name }}" }` pour la lisibilité
- `until / retries / delay` pour les boucles conditionnelles

Combinaisons

- `loop + when` : filtrer avant exécution
- Attention : `name: [list]` dans apt est plus rapide qu'un `loop`

TP 6 — Conditions

beginframe[fragile]TP 6 — Conditions & Boucles Boucles

Objectif

Hardening serveur avec conditions `when` et boucles `loop`.

Étapes

1. Désactiver une liste de services via `loop`
2. Filtrer par OS avec `when: ansible_os_family == "Debian"`
3. Configurer des paramètres `sysctl` via `dict2items`
4. Vérifier avec `systemctl is-enabled` et `sysctl -a`

playbooks/hardening.yml

```yaml
- hosts: all
  become: true
  tasks:
    - name: Desactiver les services inutiles
      ansible.builtin.service:
        name: "{{ item }}"
        state: stopped
        enabled: false
      loop: [bluetooth, cups, avahi-daemon]
      when: ansible_os_family == "Debian"
      ignore_errors: true
    - name: Configurer sysctl
      ansible.posix.sysctl:
        name: "{{ item.key }}"
        value: "{{ item.value }}"
        sysctl_set: true
      loop: "{{ sysctl_params | dict2items }}"
      vars:
        sysctl_params:
          net.ipv4.ip_forward: "0"
          net.ipv4.conf.all.rp_filter: "1"
```

Checkpoint — Conditions & Boucles

Vous maîtrisez

- `when` pour OS, facts, variables, résultats de tâche
- `loop` avec liste simple et dict
- `loop_control: label` pour la lisibilité
- `until` / `retries` pour les attentes

Cours — Handlers

Notifications et handlers

```
tasks:
  - name: Deployer la configuration nginx
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
      validate: "nginx -t -c %s"
    notify:
      - Valider nginx
      - Recharger nginx

handlers:
  - name: Valider nginx
    ansible.builtin.command: nginx -t
  - name: Recharger nginx
    ansible.builtin.service:
      name: nginx
      state: reloaded
```

Les handlers s'exécutent **une seule fois** en fin de play. Forcer : `meta: flush_handlers`.

À retenir — Handlers

Pourquoi ?

- Un handler s'exécute **uniquement** si au moins une tâche l'a notifié
- Dédoublonnage automatique : N notifications → 1 exécution
- S'exécute en **fin de play** (ou après `meta: flush_handlers`)

Antipattern

- `service: state=restarted` directement dans les tasks (pas idempotent)
- À la place : `notify: restart X` + handler

Pour aller plus loin

- `listen: <name>` pour découpler notify des noms
- `force_handlers: yes` pour ne pas perdre un handler après un échec
- `validate:` avec `%s` pour tester la config avant reload

TP 7 — Handlers

Objectif

Configuration nginx avec validation et rechargement via handlers.

Étapes

1. Écrire un playbook déployant `nginx.conf` et un vhost
2. Ajouter `notify: Recharger nginx`
3. Ajouter un handler `Recharger nginx`
4. Lancer 1re fois : handler exécuté
5. Lancer 2e fois : handler *non* exécuté (rien n'a changé)
6. Modifier une variable → handler rappelé

playbooks/handlers.yml

```yaml
- hosts: webservers
  become: true
  tasks:
    - name: Deployer nginx.conf
      ansible.builtin.template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf
        validate: 'nginx -t -c %s'
      notify: Recharger nginx
    - name: Deployer le vhost
      ansible.builtin.template:
        src: templates/vhost.conf.j2
        dest: /etc/nginx/sites-available/default
      notify: Recharger nginx
  handlers:
    - name: Recharger nginx
      ansible.builtin.service:
        name: nginx
        state: reloaded
```

Le handler ne s'exécute qu'**une fois**, même si notifié 2 fois.

Checkpoint — Handlers

Vous savez

- Lier une tâche à un handler via `notify:`
- Distinguer `reloaded` (sans interruption) vs `restarted`
- Utiliser `validate:` pour ne déployer qu'une config valide
- Forcer un handler immédiatement avec `meta: flush_handlers`

Cours — Templates Jinja2

templates/vhost.conf.j2

```
# {{ ansible_managed }}
server {
    listen {{ http_port }};
    server_name {{ server_name }};
    root {{ doc_root }};
{% for loc in locations %}
    location {{ loc.path }} {
        proxy_pass {{ loc.backend }};
    }
{% endfor %}
{% if ssl_enabled | default(false) %}
    listen 443 ssl;
    ssl_certificate {{ ssl_cert }};
{% endif %}
}
```

Syntaxe : `{{ var }}` = expression, `{% %}` = logique, `{# #}` = commentaire.

À retenir — Jinja2 & Templates

3 balises à retenir

- `{{ expr }}` : affiche une valeur (remplacée par le rendu)
- `{% if %}`, `{% for %}`, `{% set %}` : logique
- `{# ... #}` : commentaire Jinja2 (**invisible** dans le rendu)

Filtres utiles

- `| default(x)`, `| upper`, `| join(', ')`
- `| selectattr(...)`, `| map(attribute='x')`
- `| to_json`, `| to_nice_yaml`, `| hash('sha256')`

Module template

- `src:` relatif à `templates/` (du playbook ou du rôle)
- Options utiles : `backup: true`, `validate: "cmd %s"`
- Le rendu se fait **côté Control Node**, puis copié sur la cible

TP 8 — Templates

Objectif

Générer une configuration nginx multi-site dynamique avec Jinja2.

Étapes

1. Écrire `templates/vhost.conf.j2` avec `for` sur `sites`
2. Définir la liste `sites:` en vars (3 sites au moins)
3. Utiliser le module `template` dans le playbook
4. Vérifier le fichier généré sur la cible
5. Ajouter un 4e site et relancer — rien d'autre à modifier

templates/vhost.conf.j2

```
# {{ ansible_managed }}
{% for site in sites %}
server {
    listen {{ site.port | default(80) }};
    server_name {{ site.domain }};
    root {{ site.root }};
    access_log /var/log/nginx/{{ site.domain }}.log;
}
{% endfor %}
```

Playbook (extrait)

```
  vars:
    sites:
      - { domain: "site1.local", root: "/var/www/site1" }
      - { domain: "site2.local", root: "/var/www/site2",
          port: 8080 }
```

Checkpoint — Templates

Vous pouvez

- Lire et écrire un template Jinja2 avec `for` et `if`
- Utiliser un filtre (`default`, `upper`, `join`...)
- Déployer un fichier avec `validate:` pour sécuriser
- Expliquer pourquoi le rendu se fait côté Control Node

Cours — Rôles

Structure d'un rôle

```
roles/webserver/
  tasks/main.yml      # Taches principales
  handlers/main.yml   # Handlers
  templates/           # Fichiers Jinja2
  files/               # Fichiers statiques
  vars/main.yml       # Variables du role
  defaults/main.yml   # Valeurs par defaut
  meta/main.yml       # Dependances, Galaxy info
```

Utilisation dans un playbook

```
- hosts: webservers
  roles:
    - role: webserver
      vars:
        http_port: 8080
    - role: geerlingguy.docker
```

Créer un squelette : `ansible-galaxy role init mon_role`

Pour les débutants

Un rôle est comme un **kit prêt à l'emploi** : au lieu de tout mettre dans un seul playbook, on organise les tâches, templates et variables dans une structure standardisée et réutilisable.

À retenir — Rôles

Principes

- Un rôle = **une** responsabilité (nginx, mariadb, firewall…)
- `defaults/` = surchargeable ; `vars/` = interne
- Les templates/fichiers sont résolus *automatiquement* (pas de préfixe)
- Préfixer les variables : `<role>_<var>` pour éviter les collisions

Appel

- `roles: - role: webserver` (statique, au parse)
- `include_role:` (dynamique, pendant l'exécution)
- `import_role:` (statique, au parse)

Création

- `ansible-galaxy role init mon_role` crée le squelette standard
- `meta/main.yml` : métadonnées + dépendances

TP 9 — Rôles

Objectif

Créer un rôle `webserver` complet et réutilisable.

Étapes

1. `ansible-galaxy role init roles/webserver`
2. Remplir `tasks/main.yml` (install, config, enable)
3. Mettre `http_port` dans `defaults/main.yml`
4. Appeler le rôle depuis un playbook avec surcharge de `http_port`
5. Vérifier l'idempotence et la réutilisabilité

```
$ ansible-galaxy role init roles/webserver
```

roles/webserver/tasks/main.yml

```
---
- name: Installer nginx
  ansible.builtin.apt:
    name: nginx
    state: present
    update_cache: true
- name: Deployer la configuration
  ansible.builtin.template:
    src: vhost.conf.j2
    dest: /etc/nginx/sites-available/default
  notify: Recharger nginx
- name: Activer nginx
  ansible.builtin.service:
    name: nginx
    state: started
    enabled: true
```

Utilisation : `roles: [webserver]` dans le playbook.

Checkpoint — Rôles

Vous savez

- Créer un rôle avec `ansible-galaxy role init`
- Surcharger les `defaults` depuis le playbook
- Appeler un rôle via `roles:`, `include_role` ou `import_role`
- Organiser un projet en repérant ce qui est réutilisable

Cours — Vault, gestion des secrets

Chiffrement AES-256 pour protéger les données sensibles.

```
# Chiffrer un fichier entier
$ ansible-vault encrypt group_vars/prod/secrets.yml

# Chiffrer une seule variable
$ ansible-vault encrypt_string 'S3cr3t!' --name 'db_pass'

# Editer un fichier chiffré
$ ansible-vault edit secrets.yml

# Executer un playbook avec vault
$ ansible-playbook site.yml --vault-password-file .vault_pass
```

Variable chiffrée inline

```
db_pass: !vault |
  $ANSIBLE_VAULT;1.1;AES256
  38613034326237393831...
```

Pour les débutants

Vault chiffre vos mots de passe et clés API pour qu'ils puissent être stockés dans Git **sans risque**. Seul celui qui a le mot de passe Vault peut les déchiffrer.

À retenir — Vault

Deux approches

- `encrypt` : chiffrer tout un fichier (vault complet)
- `encrypt_string` : chiffrer une variable au sein d'un fichier clair

Convention de projet

- `group_vars/<env>/vars.yml` (clair) + `vault.yml` (chiffré)
- `.vault_pass` dans `.gitignore` (toujours)
- `no_log: true` pour masquer les secrets dans la sortie

CI/CD

- `$VAULT_PASSWORD` en variable secrète
- `-vault-password-file` ou `ANSIBLE_VAULT_PASSWORD_FILE`
- Rotation régulière avec `ansible-vault rekey`

TP 10 — Vault

Objectif

Chiffrer des secrets avec Ansible Vault et `-vault-password-file`.

Étapes

1. Créer un fichier `.vault_pass` (perms 600, dans `.gitignore`)
2. Créer `group_vars/prod/secrets.yml` en clair
3. Chiffrer le fichier avec `ansible-vault encrypt`
4. Écrire un playbook qui utilise `db_password` et `api_key`
5. Lancer avec `-vault-password-file .vault_pass`

```
# Créer le fichier de mot de passe
$ echo "MonSuperSecret" > .vault_pass
$ chmod 600 .vault_pass
$ echo ".vault_pass" >> .gitignore

# Chiffrer un fichier de secrets
$ ansible-vault encrypt group_vars/prod/secrets.yml \
    --vault-password-file .vault_pass

# Exécuter avec vault
$ ansible-playbook site.yml \
    --vault-password-file .vault_pass
```

group_vars/prod/secrets.yml (avant chiffrement)

```
---
db_password: "P@ssw0rd!"
api_key: "sk-abc123def456"
```

Checkpoint — Vault

Vous maîtrisez

- `ansible-vault create/view/edit/encrypt/decrypt/rekey/encrypt_string`
- Exécuter un playbook avec `-ask-vault-pass` ou `-vault-password-file`
- Structurer `vars.yml` + `vault.yml`
- Ne jamais commiter `.vault_pass`

Cours — Tags

Définir des tags

```
tasks:
  - name: Installer les paquets
    ansible.builtin.apt:
      name: "{{ item }}"
    loop: [nginx, curl]
    tags: [install, packages]
  - name: Configurer nginx
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    tags: [config, nginx]
```

Utilisation des tags

```
$ ansible-playbook site.yml --tags "config"
$ ansible-playbook site.yml --skip-tags "install"
$ ansible-playbook site.yml --list-tags
```

À retenir — Tags

Tags spéciaux

- `always` : toujours exécuté (même si `--tags` restreint)
- `never` : jamais exécuté sauf appel explicite
- `tagged` / `untagged` / `all` : méta-sélecteurs

Conventions recommandées

- `install`, `config`, `service`, `deploy`, `verify`, `cleanup`
- Marquer le `cleanup` avec `never` pour éviter les accidents
- Tagger au niveau des *phases*, pas de chaque tâche

Cas d'usage

- Redéployer uniquement la config : `--tags config`
- Cycle CI : `--tags verify`
- Désinstallation : `--tags cleanup`

TP 11 — Tags

Objectif

Utiliser les tags pour exécuter sélectivement des tâches.

Étapes

1. Tagger des tâches par phase (`install`, `config`, `deploy`)
2. Ajouter une tâche `cleanup` avec tags `[never, cleanup]`
3. Lancer `-list-tags` puis `-tags config`
4. Observer que `cleanup` ne s'exécute pas sans appel explicite

Playbook avec tags

```
tasks:
  - name: Installer les paquets
    apt: name={{ item }} state=present
    loop: [nginx, curl]
    tags: [install, packages]
  - name: Configurer nginx
    template: src=nginx.conf.j2 dest=/etc/nginx/nginx.conf
    tags: [config]
  - name: Deployer l'application
    copy: src=app/ dest=/var/www/html/
    tags: [deploy]
```

```
$ ansible-playbook site.yml --tags "config,deploy"
$ ansible-playbook site.yml --skip-tags "install"
$ ansible-playbook site.yml --list-tags
```

Checkpoint — Tags

Vous utilisez

- `-tags`, `-skip-tags`, `-list-tags`
- Les tags spéciaux `always` et `never` à bon escient
- Des conventions de nommage cohérentes

Cours — Galaxy
beginframe[fragile]Cours — Galaxy & Collections Collections

Installer des rôles et collections

```
# Installer un role depuis Galaxy
$ ansible-galaxy role install geerlingguy.docker

# Installer une collection
$ ansible-galaxy collection install community.docker
```

requirements.yml

```
---
roles:
  - name: geerlingguy.docker
    version: "7.1.0"
  - name: geerlingguy.certbot
collections:
  - name: community.docker
    version: ">=3.0.0"
  - name: ansible.posix
```

Installation : `ansible-galaxy install -r requirements.yml`

À retenir — Galaxy

Rôle vs Collection

- **Rôle** : une fonction (installer Docker, Nginx...)
- **Collection** : ensemble de rôles, modules, plugins sous un namespace

Bonnes pratiques

- Toujours **figer la version** : `version: "7.1.0"`
- Lire le `defaults/main.yml` du rôle pour connaître les variables
- Préférer les rôles bien maintenus (geerlingguy, debops...)

Installation

- Rôles : `~/.ansible/roles/`
- Collections : `~/.ansible/collections/ansible_collections/`
- CI : utiliser `requirements.yml` versionné dans Git

TP 12 — Galaxy

Objectif

Installer Docker via le rôle `geerlingguy.docker` depuis Galaxy.

Étapes

1. Écrire `requirements.yml` avec `geerlingguy.docker`
2. `ansible-galaxy install -r requirements.yml`
3. Écrire un playbook qui utilise le rôle
4. Surcharger `docker_users` et `docker_compose_version`
5. Vérifier avec `docker -version` et `docker run hello-world`

requirements.yml

```
---
roles:
  - name: geerlingguy.docker
    version: "7.1.0"
```

```
$ ansible-galaxy install -r requirements.yml
```

playbooks/docker.yml

```
---
- hosts: all
  become: true
  vars:
    docker_users: [deploy]
    docker_compose_version: "v2.24.0"
  roles:
    - geerlingguy.docker
  tasks:
    - name: Verifier Docker
      command: docker --version
      changed_when: false
```

Checkpoint — Galaxy

Vous savez

- Écrire un `requirements.yml` avec versions figées
- Installer et appeler un rôle communautaire
- Surcharger ses variables par défaut
- Distinguer rôle et collection

Cours — Gestion d'erreurs

block / rescue / always

```
- name: Deploiement avec rollback
  block:
    - name: Deployer la nouvelle version
      ansible.builtin.copy:
        src: app-v2/
        dest: /opt/app/
    - name: Redemarrer le service
      ansible.builtin.service:
        name: myapp
        state: restarted
  rescue:
    - name: Rollback vers l'ancienne version
      ansible.builtin.copy:
        src: app-v1/
        dest: /opt/app/
    - name: Notifier l'equipe
      ansible.builtin.debug:
        msg: "ECHEC deploiement - rollback effectue"
  always:
    - name: Verifier le service
      ansible.builtin.uri:
        url: "http://localhost:8080/health"
```

À retenir — Gestion d'erreurs

Équivalence try/catch/finally

- `block:` = essai (try)
- `rescue:` = capture d'échec (catch) À chaque échec dans le block
- `always:` = toujours exécuté (finally)

Autres mécanismes

- `ignore_errors:` `true` : ignorer un échec ponctuel
- `failed_when:` : définir ses propres critères d'échec
- `changed_when:` : contrôler le statut changed
- `any_errors_fatal:` `true` / `max_fail_percentage`

Lire le PLAY RECAP

- `rescued` : échec récupéré par un `rescue`
- `ignored` : échec ignoré par `ignore_errors`
- `failed` : échec fatal

TP 13 — Gestion d'erreurs

Objectif

Déploiement avec rollback automatique via block/rescue/always.

Étapes

1. Sauvegarder la version courante dans `/opt/app.bak/`
2. Déployer la nouvelle version dans `/opt/app/`
3. Lancer un health check HTTP
4. En cas d'échec → `rescue:` restaure la version précédente
5. Toujours → `always:` nettoie la sauvegarde
6. Utiliser `serial: 1` pour déployer h^ote par hôte

playbooks/deploy_safe.yml

```
- hosts: webservers
  become: true
  serial: 1
  tasks:
    - block:
        - name: Sauvegarder la version actuelle
          copy: src=/opt/app/ dest=/opt/app.bak/ remote_src=yes
        - name: Deployer v2
          copy: src=app-v2/ dest=/opt/app/
        - name: Health check
          uri: url=http://localhost:8080/health status_code=200
      rescue:
        - name: Rollback
          copy: src=/opt/app.bak/ dest=/opt/app/ remote_src=yes
        - name: Alerte
          debug: msg="ROLLBACK effectue sur {{ inventory_hostname }}"
      always:
        - name: Cleanup backup
          file: path=/opt/app.bak state=absent
```

Checkpoint — Gestion d'erreurs

Vous savez

- Écrire un pattern déploiement-rollback
- Combiner `ignore_errors` et `failed_when`
- Interpréter `rescued` vs `ignored` dans le PLAY RECAP
- Utiliser `serial` pour le rolling deploy

Cours — Debugging

Modules de debug

```
- name: Afficher une variable
  ansible.builtin.debug:
    var: ansible_default_ipv4.address
- name: Verifier une condition
  ansible.builtin.assert:
    that:
      - ansible_memtotal_mb >= 1024
      - ansible_distribution == "Debian"
    fail_msg: "Prerequis non remplis"
```

Niveaux de verbosité

```
$ ansible-playbook site.yml -v      # taches
$ ansible-playbook site.yml -vv     # + input
$ ansible-playbook site.yml -vvv    # + connexion
$ ansible-playbook site.yml -vvvv   # + SSH debug
$ ansible-playbook site.yml --check --diff  # dry-run
```

À retenir — Debugging

3 outils de base

- `debug: msg=...` / `var=...` : afficher
- `register:` + `debug: var=r` : capturer puis inspecter
- `assert: that: [conds]` : valider des invariants

Niveaux de verbosité — quand ?

- `-v` : détails des modules (`stdout`, `rc`)
- `-vv` : config, chemins YAML
- `-vvv` : commandes SSH complètes (problèmes réseau)
- `-vvvv` : transferts SFTP, plugins

Outils complémentaires

- `ansible-lint` : bonnes pratiques
- `yamllint` : syntaxe YAML
- `-check -diff` : dry-run avec affichage des diffs
- `-start-at-task "X"` : reprendre à une tâche

TP 14 — Debugging

Objectif

Construire un playbook de diagnostic système complet.

Étapes

1. Afficher les facts système (hostname, OS, RAM, CPU)
2. Capturer la sortie d'une commande avec `register`
3. Valider les prérequis avec `assert`
4. Lancer en `-vvv` pour observer la connexion SSH
5. Tester en `-check -diff`

playbooks/diagnostic.yml

```
- hosts: all
  tasks:
    - name: Infos systeme
      debug:
        msg: "{{ ansible_hostname }} | {{ ansible_distribution }}
              {{ ansible_distribution_version }} |
              RAM: {{ ansible_memtotal_mb }}MB |
              CPU: {{ ansible_processor_vcpus }}"
    - name: Espace disque
      shell: df -h / | tail -1
      register: disk
      changed_when: false
    - debug: var=disk.stdout
    - name: Verifier prerequis
      assert:
        that:
          - ansible_memtotal_mb >= 512
          - ansible_processor_vcpus >= 1
        fail_msg: "Prerequis non remplis"
```

Lancer avec `-vvv` pour le debug détaillé.

Checkpoint — Debugging

Vous savez

- Choisir la bonne verbosité pour chaque problème
- Écrire un playbook de diagnostic système
- Utiliser `assert` pour fail-fast
- Inspecter une variable capturée via `register`

Stratégies & Performance

Stratégies d'exécution

- **linear** (défaut) : tâche par tâche, tous les hôtes
- **free** : chaque hôte avance à son rythme
- **serial** : par lots (rolling update)
- Exemple : `serial: "30%"` ou `serial: [1, 5, 10]`

Optimisations

- **forks** : parallélisme (défaut 5, recommandé 20-50)
- **pipelining** : réduit les transferts SSH
- **async** + **poll** : tâches longues non bloquantes
- **Mitogen** : accélère 2–7x l'exécution
- **Fact caching** : éviter `gather_facts` répétitif

Conseil

Pour +500 hôtes : `forks=50`, `pipelining=True`, fact caching Redis/JSON.

Plugins & IHM

Qualité & Tests

- **ansible-lint** : linter de bonnes pratiques
- **Molecule** : tests unitaires de rôles (Docker, Vagrant)
- **Mitogen** : accélérateur d'exécution (2–7x)
- **ARA** : enregistrement et reporting des runs
- **yamllint** : validation syntaxique YAML

Interfaces graphiques

- **AWX** : version open source de Tower
- **Ansible Tower / AAP** : version entreprise Red Hat
- **Semaphore UI** : IHM légère open source
- **Rundeck** : orchestration multi-outils
- **Jenkins** : intégration via plugin Ansible

Pipeline DevSecOps

`yamllint` → `ansible-lint` → `molecule test` → `ansible-playbook -check` → deploy

Pourquoi ces 4 derniers TP ?

Objectif : consolidation

- Mobiliser **tous** les concepts précédents dans un cas réel
- Déployer une vraie stack de production (LAMP, hardening, monitoring)
- Pratiquer l'orchestration multi-rôles
- Préparer le passage en production

Progression

- **TP 15** : stack LAMP complète (app web dynamique)
- **TP 16** : hardening CIS (sécurisation système)
- **TP 17** : observabilité (Prometheus + Grafana)
- **TP 18** : *infra-as-code* — tout orchestrer en un seul `site.yml`

Ces TP utilisent les rôles, variables, vault, tags, templates et handlers déjà vus.

TP 15 — Stack LAMP

Objectif

Déployer une pile LAMP complète (Linux, Nginx, MariaDB, PHP-FPM) avec vérification.

Étapes principales

1. Installer `nginx`, `mariadb-server`, `php-fpm`, `php-mysql`, `python3-pymysql`
2. Démarrer MariaDB puis créer base + utilisateur via `community.mysql`
3. Détecter dynamiquement la version PHP-FPM installée
4. Déployer un vhost Nginx qui transmet les `.php` à PHP-FPM
5. Vérifier avec `curl -H 'Host: tp15.linuxmaine.local'`

Concepts mobilisés

- Modules `apt`, `service`, `template`, `copy`, `community.mysql.*`
- Handlers pour reloader Nginx
- Variables par environnement (dev/prod)
- **Vault** pour les mots de passe (suggestion : extension)

TP 16 — Hardening CIS

Objectif

Appliquer un profil de sécurisation inspiré du benchmark CIS Debian 12.

Étapes principales

1. Durcir `sshd_config` avec `lineinfile` + `validate: sshd -t`
2. Installer et configurer **fail2ban** (jail SSH)
3. Activer le pare-feu `ufw` (allow 22, 80)
4. Appliquer un profil `sysctl` (IP forward, rp_filter)
5. Valider avec `lynis audit system`

Concepts mobilisés

- `lineinfile` / `blockinfile` pour les modifs ponctuelles
- `template` + `validate:` pour ne pas se verrouiller
- Tags par catégorie (`network`, `auth`, `logging`)

TP 17 — Monitoring (Prometheus + Grafana)

Objectif

Déployer une stack d'observabilité complète.

Étapes principales

1. Déployer `node_exporter` sur la cible
2. Installer et configurer **Prometheus** (scrape du node_exporter)
3. Installer **Grafana** + provisionner datasource et dashboard
4. Tester une requête PromQL (CPU, RAM, disque)
5. Vérifier le dashboard dans le navigateur

Concepts mobilisés

- Templates Jinja2 pour les dashboards et la conf Prometheus
- Handlers pour reloader les services après changement de config
- `wait_for` pour synchroniser le démarrage des services

TP 18 — Infrastructure complète (synthèse finale)

Objectif

Combiner tous les rôles précédents dans un unique `site.yml`.

Étapes principales

1. Créer une arborescence `roles/` multi-rôles (common, hardening, web, db, monitoring)
2. Écrire `site.yml` qui orchestre tous les rôles
3. Utiliser `group_vars/<env>/` pour différencier dev/prod
4. Appliquer **vault** sur les secrets
5. Tester avec `-check -diff` puis déployer
6. Mettre en place un pipeline CI : lint + molecule + deploy

Concepts mobilisés

- **Tous** les concepts précédents : inventaire, playbooks, variables, rôles, handlers, tags, vault, gestion d'erreurs, debugging
- Stratégies (`serial`, `free`) pour l'orchestration
- Pipeline DevSecOps (`yamllint` → `ansible-lint` → `molecule` → `-check` → deploy)

Checkpoint — Ateliers de synthèse

Vous êtes opérationnel si

- Vous pouvez déployer une pile LAMP sans aide (TP 15)
- Vous savez durcir un serveur Debian selon CIS (TP 16)
- Vous avez déployé une stack Prometheus/Grafana (TP 17)
- Vous avez un `site.yml` unique qui déploie **toute** l'infra (TP 18)

Prochaines étapes

- Explorer **Molecule** pour les tests unitaires de rôles
- Passer la certification **RHCE** (Red Hat Certified Engineer)
- Découvrir **AAP** / **AWX** pour l'orchestration centralisée
- S'intéresser à **Event-Driven Ansible** et **Lightspeed**

Ansible-core 2.17+ — Quoi de neuf ?

- **Python 3.10+** requis sur le control node (fin du support Python 3.9)
- Amélioration de la **gestion des erreurs** et messages d'avertissement
- **Nouveaux modules** : cloud natif, Kubernetes, conteneurs
- Support amélioré de **Windows** via OpenSSH (remplacement de WinRM)
- **Execution Environments** : conteneurs standardisés pour l'exécution (image Docker avec ansible-core + collections + dépendances Python)

Pour les débutants

Depuis 2020, Ansible est découpé en deux parties : **ansible-core** (le moteur minimal) et les **collections** (les modules). C'est comme un navigateur web (le noyau) auquel on ajoute des extensions (les collections).

Architecture ansible-core + Collections

ansible-core (le noyau)

- Moteur d'exécution des playbooks
- Gestion de l'inventaire
- Connexions SSH / WinRM
- Modules de base : `copy`, `file`, `debug`, `command`
- `pip install ansible-core`

Collections (les modules)

- `community.general` : modules génériques
- `ansible.posix` : modules Linux/POSIX
- `community.docker` : gestion Docker
- `amazon.aws` : cloud AWS
- `kubernetes.core` : Kubernetes

```
# Installer une collection
$ ansible-galaxy collection install community.docker
# Utiliser un module de collection dans un playbook
# -> community.docker.docker_container au lieu de docker_container
$ ansible-galaxy collection list    # Voir les collections installees
```

Event-Driven Ansible (EDA) — Principe

Qu'est-ce que c'est ?

- **Automatisation réactive** : Ansible réagit aux événements en temps réel
- Trois composants : **source** → **condition** → **action**
- Intégré dans **Red Hat AAP 2.4+**

Sources d'événements

- Webhooks (GitHub, GitLab)
- Alertes monitoring (Prometheus, Zabbix)
- Messages Kafka / MQTT
- Alertes cloud (AWS CloudWatch, Azure Monitor)
- Fichiers (inotify)

Actions possibles

- Lancer un **playbook** Ansible
- Exécuter un **module** directement
- Créer un **ticket** (Jira, ServiceNow)
- Envoyer une **notification** (Slack, email)
- **Débugger** (afficher l'événement)

Cas d'usage

- CPU $> 90\%$ → scale-out auto
- Git push → déploiement auto
- CVE détectée → patching

Event-Driven Ansible — Exemple de Rulebook

rulebook.yml

```
---
- name: Reagir aux alertes Prometheus
  hosts: all
  sources:
    - ansible.eda.alertmanager:
        host: 0.0.0.0
        port: 5000
  rules:
    - name: Redemarrer le service en surcharge
      condition: event.alert.labels.severity == "critical"
      action:
        run_playbook:
          name: playbooks/restart-service.yml
          extra_vars:
            target: "{{ event.alert.labels.instance }}"
    - name: Notifier l'equipe
      condition: event.alert.labels.severity == "warning"
      action:
        debug:
          msg: "Alerte warning sur {{ event.alert.labels.instance }}"
```

Installation

`pip install ansible-rulebook` puis `ansible-rulebook -rulebook rulebook.yml`

Ansible Lightspeed (IA) — Principe

Qu'est-ce que c'est ?

- **Assistant IA** intégré dans VS Code
- Propulsé par **IBM watsonx Code Assistant**
- Modèle spécialement entraîné sur des millions de playbooks Ansible
- Disponible via **Red Hat AAP** ou extension VS Code gratuite

Comment ça marche ?

1. Vous écrivez le **nom de la tâche** en langage naturel
2. Lightspeed **génère le code YAML** correspondant
3. Vous validez, modifiez ou rejetez la suggestion
4. **Content matching** : attribution aux auteurs originaux (respect des licences)

Pour les débutants

C'est comme GitHub Copilot, mais spécialisé pour Ansible. Vous décrivez ce que vous voulez faire en français ou anglais, et l'IA écrit le code YAML pour vous. **Toujours vérifier le code généré !**

Ansible Lightspeed — Exemple concret

Vous tapez le nom de la tâche :

```
- name: Installer et configurer nginx avec un vhost SSL
```

Lightspeed génère automatiquement :

```
  ansible.builtin.apt:
    name: nginx
    state: present
    update_cache: true
- name: Deployer le certificat SSL
  ansible.builtin.copy:
    src: files/ssl/cert.pem
    dest: /etc/ssl/certs/site.pem
    mode: '0644'
- name: Deployer la configuration vhost
  ansible.builtin.template:
    src: templates/vhost-ssl.conf.j2
    dest: /etc/nginx/sites-available/default
  notify: Recharger nginx
```

Bonnes pratiques

Toujours valider : `ansible-lint` → `-check -diff` → `molecule test` avant de déployer du code généré par IA en production.

Refcard Ansible — Commandes

Exécution

- `ansible all -m ping` — test connectivité
- `ansible all -m setup` — collecter les facts
- `ansible-playbook site.yml` — exécuter un playbook
- `ansible-playbook -C -D` — dry-run + diff
- `ansible-playbook -e var=val` — surcharge variable
- `ansible-playbook -tags install` — exécution sélective
- `ansible-playbook -limit web` — restreindre les cibles

Gestion

- `ansible-inventory -list` — lister l'inventaire
- `ansible-galaxy install role` — installer un rôle
- `ansible-galaxy collection install ns.col` — collection
- `ansible-vault encrypt file` — chiffrer un fichier
- `ansible-vault edit file` — éditer un secret
- `ansible-lint playbook.yml` — analyse statique
- `ansible-doc module_name` — documentation module

Refcard Ansible — Modules essentiels

Système

- `apt` / `yum` / `dnf` — paquets
- `service` / `systemd` — services
- `user` / `group` — utilisateurs
- `file` — fichiers, répertoires, liens
- `copy` — copier des fichiers
- `template` — déployer un Jinja2
- `lineinfile` — modifier une ligne
- `cron` — tâches planifiées

Avancé

- `uri` — requêtes HTTP
- `git` — cloner un dépôt
- `docker_container` — conteneurs Docker
- `command` / `shell` — commandes brutes
- `debug` — afficher des variables
- `assert` — vérifier des conditions
- `wait_for` — attendre un port/fichier
- `set_fact` — définir un fact dynamique

Lexique (1/2)

Termes généraux

SSH Secure Shell — protocole de connexion sécurisée à distance

YAML Format de données lisible (indentation, clé: valeur)

JSON Format de données structurées (utilisé par les API)

API Interface de programmation entre deux logiciels

CLI Interface en ligne de commande (terminal)

Idempotence N exécutions = même résultat qu'une seule

IaC Infrastructure as Code — infra décrite en fichiers

CI/CD Intégration et déploiement continus

Termes Ansible

Control Node Machine qui exécute Ansible

Managed Node Machine cible gérée via SSH

Playbook Fichier YAML décrivant l'état désiré

Play Ensemble de tasks pour un groupe d'hôtes

Task Unité d'action dans un playbook

Module Unité d'exécution (apt, copy, service...)

Role Structure réutilisable (tasks, vars, templates)

Inventory Liste des hôtes et groupes gérés

Handler Tâche déclenchée par notification

Facts Variables collectées sur les hôtes

Lexique (2/2)

Termes Ansible (suite)

Vault Chiffrement AES-256 des secrets

Galaxy Dépôt communautaire de rôles/collections

Collection Paquet de modules, rôles, plugins

Template Fichier Jinja2 rendu dynamiquement

Jinja2 Moteur de templates Python (`{{ var }}`)

Tag Étiquette pour exécution sélective

Callback Plugin qui modifie la sortie d'Ansible

Become Élévation de privilèges (sudo)

Register Stocker le résultat d'une tâche

Notify Déclencher un handler après un changement

Termes DevSecOps

SAST Analyse statique de sécurité du code

DAST Analyse dynamique de sécurité

SCA Analyse de composition logicielle

Shift-Left Sécurité dès le développement

SBOM Nomenclature des composants logiciels

EDA Event-Driven Ansible (automatisation réactive)

ansible-lint Linter de bonnes pratiques Ansible

Molecule Framework de test de rôles

AWX IHM open source pour Ansible (Tower)

AAP Ansible Automation Platform (Red Hat)

Références

Documentation officielle

- https://docs.ansible.com — Documentation officielle
- https://galaxy.ansible.com — Rôles et collections
- https://github.com/ansible/ansible — Code source
- https://www.ansible.com/blog — Blog officiel
- https://www.redhat.com/en/technologies/management/ansible — Red Hat AAP

Outils & Communauté

- https://ansible.readthedocs.io/projects/lint/ — ansible-lint
- https://ansible.readthedocs.io/projects/molecule/ — Molecule
- https://jinja.palletsprojects.com — Jinja2
- https://github.com/ansible/awx — AWX
- https://semaphoreui.com — Semaphore UI
- https://ara.recordsansible.org — ARA
- https://owasp.org/www-project-devsecops-guideline/ —

Merci !

Questions ?

thierry.gayet@gmail.com

Association LinuxMaine — 25 avril 2026