Chapitre 1: Vue d'ensemble

Automatisation

L'automatisation utilise la technologie pour exécuter des tâches sans intervention humaine.

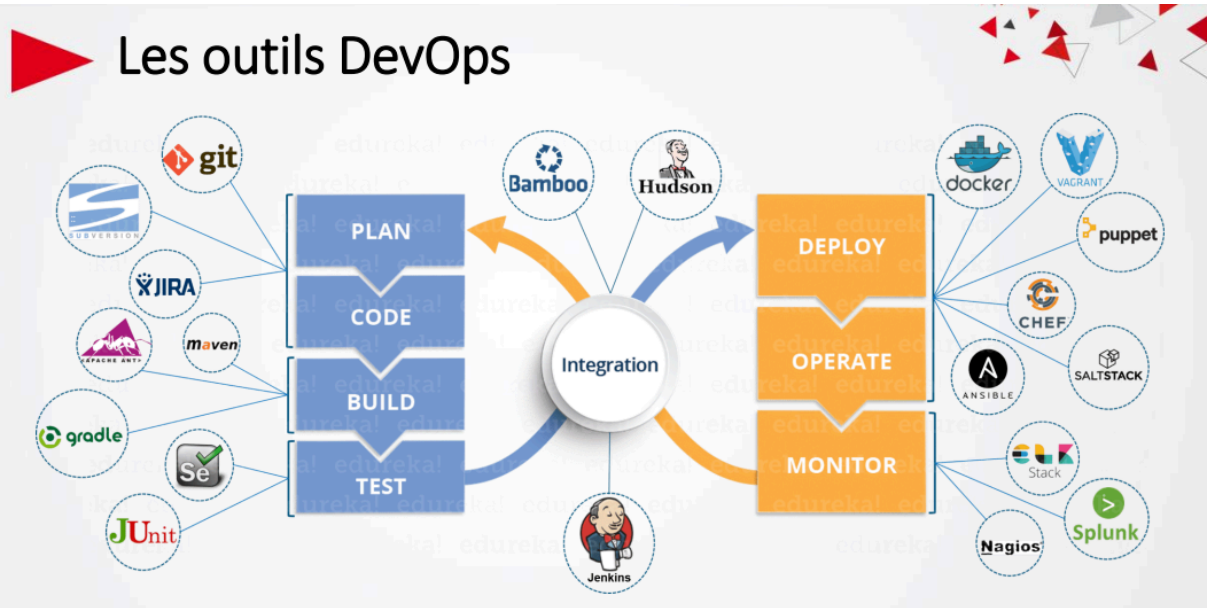
- **Informatique** : automatise les processus répétitifs.
- **Métier** : aligne processus et règles métier avec les applications.

Pourquoi automatiser ?

L'automatisation est une technologie clé utilisée dans divers domaines comme **DevOps** et les **applications cloud natives**.

DevOps : Définition

DevOps combine culture, pratiques et outils pour accélérer le développement et améliorer la compétitivité.



Phases DevOps et Outils

- **Plan** : Gestion de projet (*Jira, Trello*).
- **Code** : Contrôle de version (*GitHub, GitLab*).
- **Build** : Gestion des artefacts (*Docker Hub, Nexus*).

- **Intégration** : Automatisation CI/CD (*Jenkins, GitLab CI*).
- **Test** : Automatisation des tests (*Selenium, JUnit*).
- **Deploy & Operate** : Déploiement et infra (*Docker, Kubernetes, Terraform*).
- **Monitor** : Surveillance (*Prometheus, Grafana*).
- **Collaboration** : Communication (*Slack, Teams*).

Pratiques clés de DevOps

- **CI/CD** : Automatisation du développement et du déploiement.
- **Microservices** : Applications modulaires et indépendantes.
- **Infrastructure as Code** : Gestion automatisée des infrastructures.
- **Monitoring & Logging** : Surveillance et analyse continue.
- **Communication & Collaboration** : Travail en équipe optimisé.

CI/CD

Automatisation et surveillance continues du développement via un pipeline.

- **Intégration Continue (CI)** : Compilation, test et déploiement automatisés pour détecter rapidement les erreurs.
- **Livraison Continue (CD)** : Automatisation de la validation et de la publication du code.
- **Déploiement Continu** : Automatisation du déploiement en production.

🔷 **Différence** : La livraison continue nécessite une validation manuelle, tandis que le déploiement continu est entièrement automatisé.

Cloud Native

Le **Cloud Native** est une approche de développement exploitant les avantages du cloud. Il concerne **la manière** dont les applications sont créées et déployées, pas leur emplacement.

🔷 **Technologies clés** : DevOps, CI/CD, microservices, conteneurs.

Applications Traditionnelles vs Cloud Native

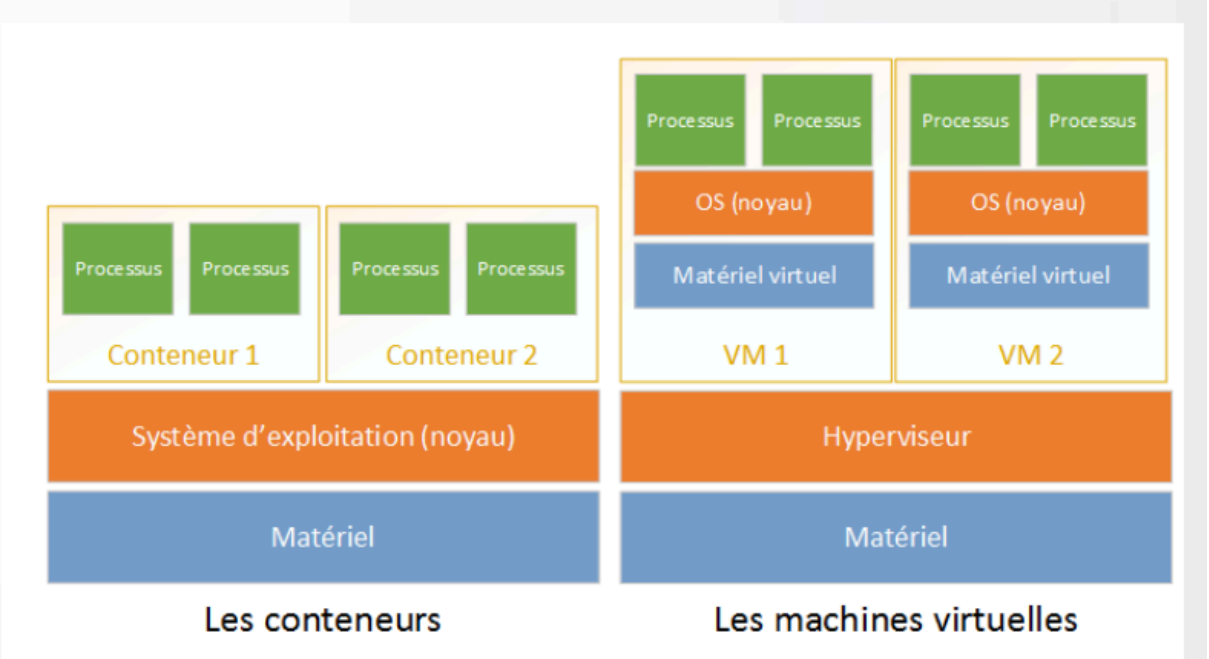
Applications Traditionnelles	Applications Cloud Native

Imprévisible	Prévisible
Dépendance à l’OS	Abstraction de l’OS
Capacité surdimensionnée	Bon dimensionnement
Isolée	Collaboration
Baisse de rendement	Déploiement continu
Dépendante	Indépendante
Mise à l’échelle manuelle	Auto-scaling
Restauration lente	Restauration rapide

Conteneurs

Un conteneur regroupe une application, ses dépendances, bibliothèques et fichiers nécessaires dans un seul package.

🔷 **Différence avec la virtualisation** : Une machine virtuelle inclut un système d'exploitation complet, tandis qu'un conteneur partage le noyau de l'OS entre plusieurs applications

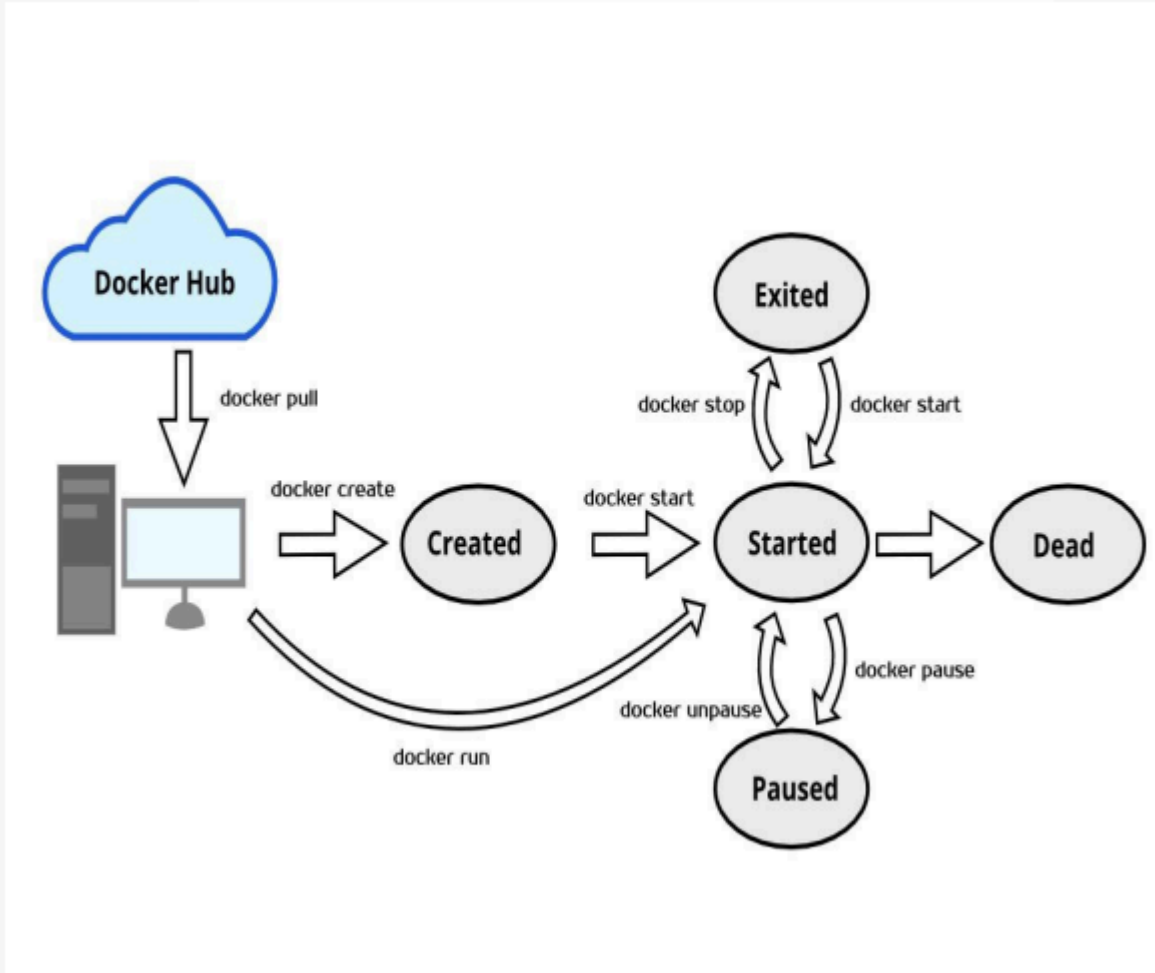


Docker fonctionne comme suit :

1. **Image** : C’est le modèle de l’application (code, dépendances, etc.).
2. **Conteneur** : Une instance de l'image qui s'exécute isolément.

3. **Commande** : `docker run` crée et démarre un conteneur.
4. **Isolation** : Le conteneur partage le noyau du système mais est séparé pour exécuter l'application.

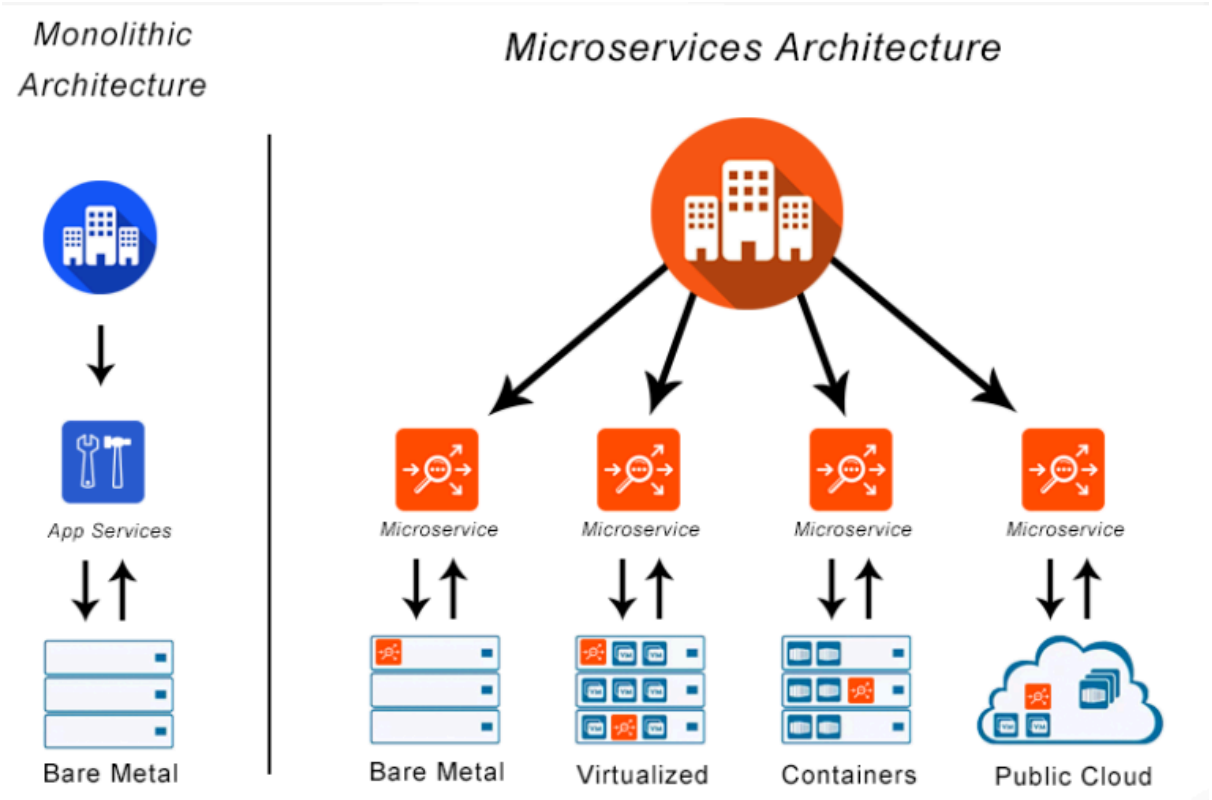
Cela permet d'exécuter des applications de manière portable et cohérente, quel que soit l'environnement.



Avantages

• Optimisation de l'utilisation des ressources
• Rapidité de déploiement des applications et de la libération des ressources.
• Meilleur disponibilité
• Modularité.

Les microservices sont une approche de développement logiciel qui divise une application en services indépendants et faiblement couplés. Ces services communiquent via des API indépendantes du langage de programmation.



Critère	Application Monolithique	Microservices
Définition	Application unique avec toutes les fonctionnalités.	Divise l'application en services indépendants.
Modularité	Fortement couplée, modification impacte tout.	Faiblement couplée, services indépendants.
Développement	Développé dans le même codebase.	Développement par équipes distinctes, technologies variées.
Déploiement	Redéploiement complet pour toute modification.	Déploiement individuel de chaque service.
Performance	Limité par la taille et la complexité.	Meilleure utilisation des ressources.
Scalabilité	Scalabilité verticale (ajout de ressources).	Scalabilité horizontale (mise à l'échelle des services).
Gestion des pannes	Panne totale en cas de défaillance.	Pannes isolées, mais risques de propagation.
Complexité	Simple au départ, mais difficile à maintenir.	Plus complexe au début, mais facilite la maintenance.

Domaines d'application de l'automatisation informatique :

1)Gestion de configurations

- **Définition** : Gérer la description technique d'un système et ses évolutions.
- **Utilisations** :
  - Tracer les versions des informations utilisées par un système.
  - Déployer des configurations sur un parc informatique.

2)Approvisionnement de ressources :

- **Définition** : Création de machines virtuelles, réseaux, stockage et conteneurs sur des systèmes physiques ou des clouds (privé, hybride, public) via l'Infrastructure as Code (IaC).
- **Avantages de l'IaC** :
  - Standardisation
  - Gestion et contrôle des versions simplifiés
  - Réduction du temps et amélioration de la productivité
  - Fiabilité accrue
  - Amélioration des processus CI/CD

3)Déploiement d'applications : Livraison continue

- **Définition** : Pratique permettant de créer et déployer des logiciels prêts à être mis en production à tout moment.
- **Conditions** :
  - Intégration continue des modifications tout au long du développement, des tests et du déploiement.
  - Prêt à déployer dès que nécessaire.
- **Pipeline de déploiement** : Processus automatique de conception, déploiement, test et distribution des applications, initié dès qu'une modification est apportée.

4)Orchestration :

- **Définition** : Coordination et gestion des flux de travail et processus à travers divers outils, matériels et services pour offrir un service défini.
- **Fonctions** :

- Réduire les délais de traitement.
- Fiabiliser les processus à faible valeur ajoutée.
- Superviser les ressources et informer l'utilisateur.
- Offrir un service à la demande avec "capacity planning".
- Gérer le cycle de vie des ressources.

5)Sécurité et conformité :

L'automatisation des processus informatiques et réseau peut résoudre les problèmes de sécurité et de conformité. Les éléments à automatiser incluent :

- Gestion des paquets et des correctifs.
- Renforcement de l'OS selon des normes de sécurité pour assurer la cohérence.
- Identification et gestion des vulnérabilités (bilans de santé, etc.).
- Gouvernance proactive des politiques de sécurité et de conformité.

Comparaison d'Ansible, Terraform et Puppet

Caractéristique	Puppet	Ansible	Terraform
Approche	Modèle **client-serveur** (nécessite Puppet sur chaque nœud)	**Sans agent** (utilise SSH/WinRM pour communiquer)	**Déclaratif** basé sur des fichiers de configuration
Langage de Configuration	Utilise un **DSL** basé sur Ruby	Utilise **YAML** pour les playbooks	Utilise **HCL (HashiCorp Configuration Language)**
Configuration	Nécessite un **Puppet master** et des agents	**Pas d'architecture master/agent**, fonctionne directement via SSH ou WinRM	**Pas d'agent**, déclare l'infrastructure dans des fichiers de configuration
Facilité d'utilisation	Plus complexe, nécessite l'apprentissage du **Puppet DSL**	Plus facile à utiliser, **YAML** est plus simple à apprendre	**Relativement facile**, mais nécessite une bonne compréhension des concepts d'infrastructure

État	Suivi de l'état du système, garantit qu'il correspond à la configuration définie	Ne suit pas l'état, exécute des tâches de manière idempotente basées sur le playbook	**Gère l'état** de l'infrastructure, garantit la cohérence de l'infrastructure
Cas d'utilisation	**Configuration à grande échelle** et **maintenance continue** des systèmes	**Automatisation du déploiement**, des configurations et des tâches ad-hoc	**Provisionnement d'infrastructure**, gestion des **ressources cloud** et de l'**infrastructure as code**
Courbe d'apprentissage	Plus raide, nécessite l'apprentissage du **Puppet DSL** (basé sur Ruby)	Plus facile à apprendre grâce à **YAML**	Modérée, nécessite une compréhension de la gestion d'infrastructure et des concepts cloud
Communication	Modèle **Pull** : les agents récupèrent la configuration du Puppet master	Modèle **Push** : Ansible envoie les configurations directement aux serveurs	**Déclare l'infrastructure** via des fichiers, mais nécessite l'exécution de commandes pour appliquer les changements
Vitesse	Plus lent en raison de la communication agent-serveur et de l'infrastructure plus large	Plus rapide car il est **sans agent** et utilise directement SSH	**Rapide**, car Terraform s'exécute principalement en local, sans agent
Gère les dépendances	Oui, mais complexe et parfois difficile à maintenir	Non, mais il peut être intégré avec des outils comme **Ansible Galaxy** pour gérer des dépendances de rôles	Oui, Terraform gère automatiquement les dépendances entre ressources
Idempotence	Oui, garantit que l'état final est cohérent à chaque exécution	Oui, assure que les tâches sont idempotentes selon les playbooks	Oui, Terraform s'assure que l'infrastructure est toujours à l'état désiré,

			même après plusieurs exécutions

- **Choisissez Puppet** si vous avez des besoins complexes en termes de gestion de configurations à grande échelle avec suivi d'état.
- **Choisissez Ansible** si vous recherchez une solution simple et directe pour l'automatisation sans trop de complexité.
- **Choisissez Terraform** si vous travaillez avec des infrastructures cloud et que vous souhaitez gérer des ressources de manière déclarative et reproductible.

Outil	Type	Explication
Terraform	**Immuable (Immutable)**	Terraform recrée ou remplace les ressources pour atteindre l'état désiré. Il ne modifie pas directement les ressources existantes.
Ansible	**Muable (Mutable)**	Ansible applique directement les configurations sur les ressources existantes, en modifiant ou mettant à jour les systèmes sans les supprimer ou recréer.

Chapitre3: Initiation à l'outils Ansible

1- Historique et Caractéristiques

- **Création** : 2012 (AnsibleWorks), racheté par RedHat en 2015.
- **Langage** : Python.
- **Compatibilité** : Linux, Unix, macOS, Windows.
- **Licence** : Open source.
- **Version commerciale** : Ansible Tower.
- **Architecture** : Sans agent, utilise SSH.
- **Langages de configuration** : JSON et YAML.

2- Ansible vs. Ansible Tower

- **Ansible** : Ligne de commande (CLI).
- **Ansible Tower** : Interface graphique, contrôle d'accès, API REST, tableau de bord.

🔷 **Tower facilite l’automatisation, mais la CLI reste plus flexible.**

3-Architecture d’Ansible

1- Éléments clés

- **Nœuds** : Machines contrôlées par Ansible.
- **Inventaire** : Fichier listant les nœuds et leurs caractéristiques.
- **Playbooks** : Scripts YAML définissant les tâches automatisées.

2- Types de nœuds

- **Nœud de contrôle** : Machine où Ansible est installé, orchestre les opérations.
- **Nœud géré** : Machine cible administrée via SSH.

3- Inventaire

- Contient les IP, groupes et variables des nœuds.
- Permet d’organiser et structurer les déploiements.

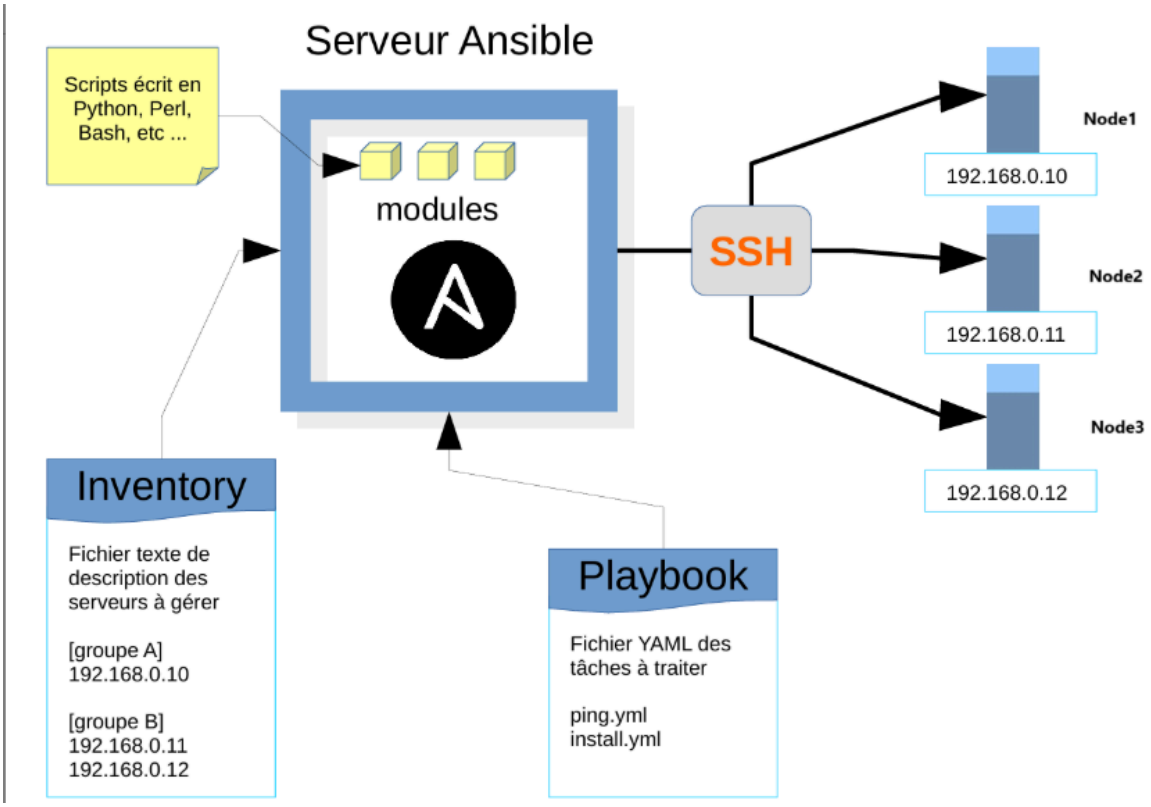
4- Playbooks et modules

- **Playbooks** : Définissent les tâches à exécuter sur les nœuds.
- **Modules** : Effectuent des actions spécifiques (installation, gestion de services).
  - **Exécution locale** : Sur le nœud de contrôle.
  - **Exécution distante** : Sur les nœuds gérés.

5- Pourquoi SSH ?

- Sécurise la communication entre nœuds.
- Évite l’installation d’agents.
- Facilite l’administration sur des infrastructures variées.

🔷 **Ansible simplifie l’automatisation grâce à son architecture sans agent et son utilisation d’SSH.**



4-Playbook vs Play

1- Différence entre Playbook et Play

- **Playbook** : Contient un ou plusieurs Plays.
- **Play** : Définit les actions à exécuter sur des machines cibles.

2- Structure d'un Play

- **Nom (name)** : Description du Play.
- **become** : Exécution avec privilèges (`true` ou `false`).
- **Machines cibles (hosts)** : Sur quelles machines exécuter le Play.
- **Tâches (tasks)** : Liste ordonnée des actions à exécuter.

3- Points clés

- Un **Playbook** peut regrouper plusieurs **Plays** (Play 1, Play 2, Play 3).
- Un **Play** peut cibler une seule machine ou un groupe.

4- Exemple de Playbook

```
---
- name: Play1
  become: true
  hosts: web_servers
  tasks:
    # Tâches

- name: Play2
  hosts: app_server
  tasks:
    # Tâche

- name: Play3
  hosts: db_servers
  tasks:
    # Tâche
```

🔷 **Un Playbook organise plusieurs Plays pour automatiser des tâches sur différentes machines.**

6-Les tâches dans Ansible

1- Définition

- Une **tâche** est une action exécutée dans un **Play**.
- Chaque tâche utilise un **module Ansible**.
- Un **module** contient des **attributs** pour définir son comportement.

2- Structure d'une tâche dans un Playbook

```
- name: Nom du Play
  hosts: Nom des hôtes
  become: true/false
  tasks:
    - name: Nom de la tâche
```

```
    module:
      attribut1: valeur
      attribut2: valeur
```

3- Exemples de modules

🔷 Module `apt` (Gestion des paquets - Debian/Ubuntu)

```
- name: Installer Apache
  apt:
    name: apache2
    state: present  # Installe si absent, ne fait rien sinon
```

🔷 Module `service` (Gestion des services)

```
- name: Démarrer Apache et l'activer au démarrage
  service:
    name: apache2
    state: started
    enabled: yes
```

🔷 Module `copy` (Copie de fichiers)

```
- name: Copier un fichier de configuration
  copy:
    src: /etc/ansible/app_config.conf  # Fichier source
    dest: /etc/app_config.conf  # Destination sur l'hôte distant
```

🔷 Module `file` (Gestion des fichiers et dossiers)

```
- name: Créer un fichier test.txt
  file:
    path: /tmp/test.txt
    state: touch  # Crée un fichier vide
```

Les tâches permettent d'automatiser l'installation, la gestion des services, la copie de fichiers et la création de répertoires dans Ansible.

7-Définition des variables dans un Playbook Ansible

1- Définition des variables dans `vars`

- Les variables sont définies sous `vars` dans un play.
- Elles sont limitées à ce play spécifique et ne peuvent pas être utilisées dans un autre play du même playbook.

Exemple :

```
- name: Premier Play
  hosts: web_servers
  vars:
    message: "Bonjour depuis le premier play"
  tasks:
    - name: Afficher le message
      debug:
        msg: "{{ message }}"

- name: Deuxième Play
  hosts: web_servers
  tasks:
    - name: Afficher le message (Échoue ici)
      debug:
        msg: "{{ message }}"  # Erreur : la variable n'est pas définie
```

2- Partager des variables avec `vars_files`

Si une variable doit être utilisée dans plusieurs plays, il faut la définir dans un fichier séparé et l'inclure avec `vars_files`.

Exemple :

📄 **Fichier `vars.yml`**

```
message: "Bonjour depuis le fichier de variables"
```

📄 **Playbook Ansible**

```
- name: Premier Play
  hosts: web_servers
  vars_files:
    - vars.yml
  tasks:
    - name: Afficher le message
      debug:
        msg: "{{ message }}"

- name: Deuxième Play
  hosts: web_servers
  vars_files:
    - vars.yml
  tasks:
    - name: Afficher le message
      debug:
        msg: "{{ message }}"  # Réutilisation de la variable sans erreur
```

📌 **Avec `vars_files`, la variable est accessible dans tous les plays où elle est incluse.**

8-Handlers Ansible : Automatisation des Actions Post-Changement

📌 Qu'est-ce qu'un handler en Ansible ?

Un **handler** est une tâche spéciale déclenchée uniquement lorsqu'une autre tâche le notifie. Il est souvent utilisé pour **redémarrer des services** ou exécuter une action après une modification.

⚙️ Principes des Handlers

- **Déclenchement sélectif** : Activé uniquement lorsqu'une tâche notifie un changement.
- **Référencé par plusieurs tâches** : Un même handler peut être notifié par plusieurs tâches.
- **Exécution unique** : Il ne s'exécute qu'une seule fois, même s'il est notifié plusieurs fois.
- **Ordre d'exécution** : Il est exécuté à la fin du play, après toutes les tâches.

📝 Exemple : Définition et Notification d'un Handler

📄 Playbook Ansible

```
- name: Exemple de Handlers
  hosts: localhost
  tasks:
    - name: Créer un fichier test1
      file:
        path: /tmp/test1.txt
        state: touch
      notify: Afficher un message

    - name: Créer un fichier test2
      file:
        path: /tmp/test2.txt
        state: touch
      notify: Afficher un message

  handlers:
    - name: Afficher un message
```

```
    debug:
      msg: "Le fichier de test a été créé !"
```

10-Ansible Templating avec le module `template`

Le module `template` permet de copier un fichier Jinja2 (`.j2`) depuis la machine de contrôle vers l'hôte distant, en remplaçant dynamiquement les variables.

Différence entre `template` et `copy`

Module	Fonctionnalité
`copy`	Copie un fichier statique **sans modification**
`template`	Génère un fichier **avec remplacement des variables** grâce à **Jinja2**

✔️ **Avantage du `template`** : Permet d'adapter dynamiquement les fichiers en fonction des variables et des informations système collectées.

11-Ansible Loops : Pourquoi utiliser des boucles ?

Les boucles permettent d'exécuter une tâche plusieurs fois avec différentes valeurs, évitant ainsi la répétition de code et facilitant la maintenance du playbook.

```
- name: Ajouter plusieurs utilisateurs avec une boucle
  hosts: worker
  tasks:
    - name: Créer des utilisateurs
      user:
        name: "{{ item }}"
        state: present
      loop:
        - user1
        - user2
        - user3
```

12-Ansible Conditions : Utilisation de `when`

Les conditions dans un playbook contrôlent l'exécution des tâches. La directive `when` permet d'exécuter une tâche uniquement si la condition est vraie.

Exemple : Installer Apache selon l'OS

```
- name: Installer Apache
  hosts: workers
  tasks:
    - name: Sur Ubuntu
      apt:
        name: apache2
        state: present
      when: ansible_os_family == "Debian"

    - name: Sur CentOS
      yum:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"
```

Explication :

- La première tâche s'exécute sur Ubuntu (`Debian`).
- La seconde tâche s'exécute sur CentOS (`RedHat`).

Si la condition `when` n'est pas remplie, la tâche est ignorée, mais l'exécution continue.

13-Commandes Ad-Hoc Ansible

1. **Principe**
   - Les commandes ad-hoc permettent d'effectuer des tâches rapidement sur des nœuds distants, sans avoir à écrire de playbook.
   - Elles sont utiles pour des actions ponctuelles sur plusieurs hôtes.
2. **Syntaxe des Commandes Ad-Hoc**
   - La commande de base est :

```
ansible <groupe_inventaire> -m <nom_module> -a "clé=valeur clé=val
```

```
eur"
```

- **Explications** :
  - `<groupe_inventaire>` : Le groupe d'hôtes ciblé.
  - `m <nom_module>` : Le module Ansible à utiliser.
  - `a "clé=valeur"` : Les arguments sous forme de paires clé-valeur.

Exemple pour redémarrer un service Apache :

```
ansible webservers -m service -a "name=apache2 state=restarted"
```

14-Comprendre les Rôles Ansible et leurs Bénéfices

Qu'est-ce qu'un rôle Ansible ?

Un rôle Ansible est une manière structurée d'organiser les différentes parties d'un playbook. Chaque rôle est constitué de répertoires et de fichiers bien définis, permettant une gestion plus claire et modulaire du code Ansible. Les rôles favorisent la réutilisabilité et améliorent la maintenabilité des configurations.

Avantages des Rôles Ansible

- **Organisation claire** : Structure les tâches, variables, fichiers et templates de manière logique.
- **Réutilisabilité** : Les rôles peuvent être réutilisés dans différents playbooks ou projets.
- **Maintenance simplifiée** : Facilite les mises à jour et modifications en centralisant les configurations dans des répertoires dédiés.

Structure des Rôles Ansible

Généralement, un rôle Ansible se compose des répertoires et fichiers suivants :

1. **tasks/** : Contient les tâches principales à exécuter.
2. **handlers/** : Tâches exécutées en réponse à un changement.
3. **files/** : Fichiers statiques à copier sur les hôtes.
4. **templates/** : Fichiers de templates Jinja2 pour générer des configurations.

5. **vars/** : Variables spécifiques au rôle.
6. **defaults/** : Valeurs par défaut des variables.
7. **tests/** : Tests de validation pour s'assurer du bon fonctionnement du rôle.
8. **meta/** : Informations sur le rôle, comme les dépendances.
9. **README.md** : Documentation du rôle, expliquant son utilisation.

Fichier `main.yml`

Le fichier `main.yml` est central pour le bon fonctionnement d'un rôle. Il définit les sections principales (comme `tasks`) du rôle, permettant une meilleure organisation et lisibilité du playbook.

Par exemple, la section **tasks** est centralisée dans le fichier `/etc/ansible/roles/<role_name>/tasks/main.yml`, ce qui rend le playbook plus propre et facile à gérer.

Intégration des rôles dans un playbook

Méthodes d'intégration des rôles :

1. **Appel classique avec la section `roles`**
   - Dans le playbook, sous la section `roles`, vous mentionnez les rôles à utiliser.

```
- hosts: all
  roles:
    - mon_role
```

2. **Appel dynamique de rôle : `include_role`**
   - Permet d'inclure un rôle au moment de l'exécution.

```
tasks:
  - name: Inclure un rôle
    include_role:
      name: mon_role
```

3. **Appel statique de rôle : `import_role`**
   - Charge un rôle avant l'exécution du playbook.

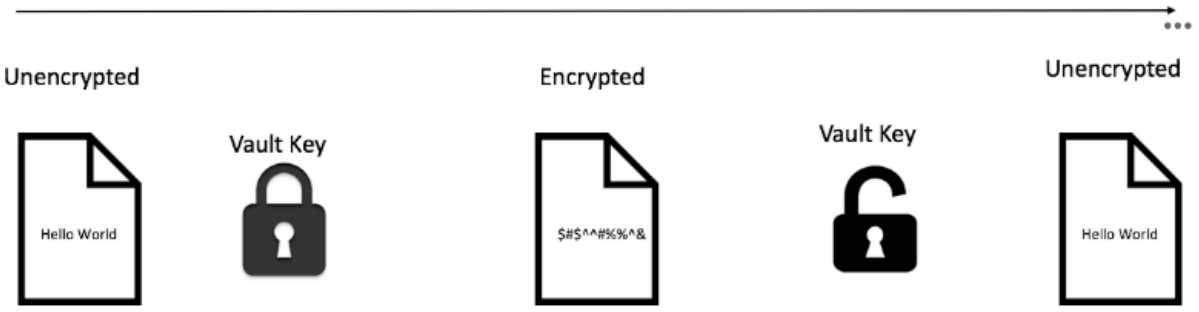
```
tasks:
  - name: Importer un rôle
    import_role:
      name: mon_role
```

Note :

`include_role` et `import_role` sont des modules et doivent être utilisés dans les tâches d'un playbook.

15-Qu'est-ce qu'Ansible Vault ?

Ansible Vault permet de chiffrer les informations sensibles (mots de passe, clés, etc.) dans les playbooks pour assurer leur sécurité. Il utilise un chiffrement symétrique avec une seule clé pour chiffrer et déchiffrer les données.



Encrypter un Playbook avec Ansible Vault

1. Chiffrement du Playbook

Pour sécuriser un playbook, utilisez la commande suivante :

`ansible-vault encrypt nom_playbook.yml`

Lors de l'exécution, vous saisissez un mot de passe qui chiffre immédiatement le fichier, le rendant illisible sans ce mot de passe.

```
esprit@esprit-virtual-machine:/etc/ansible$ sudo ansible-vault encrypt myplaybook.yml
New Vault password:
Confirm New Vault password:
Encryption successful
esprit@esprit-virtual-machine:/etc/ansible$ sudo cat myplaybook.yml
$ANSIBLE_VAULT;1.1;AES256
39323366616461303965626435636132336132313330646237653636616335653866313835646461
34386132613161373966393434386666626535616232393 20a333231373363643764393837333330
38363035343233643035366461346434653832373866663733363064383137613438306636613130
30323765663437323 00a3635346531646532643565613163633764646436656164356338383736 34
37626461356265646131326235656332323465656331393263636461333865636436646634633362
32393232343266613266326333656235633634633363346236646634323662313333353234663438
34316261393962353463386161333234613963656230636338653961666237333561316634613936
63393966396132636231333763643661313232373432383239613465333862633135653934333332
37343662633462616665663263633434383537393939613733666262343935393866633733366630
61623330343831333535316563316662393735376536636333343761626531643539363765663732
37313362643536663263356233666432383930363263383938616138353438623133393063363530
37376334353832376231336165393839613263353638303435336337383233336666646362653964
34303362313762656163306331303862613737326562333037336563336530636263
```

2. Exécution d'un Playbook Chiffré

Lorsque vous tentez d'exécuter un playbook chiffré, Ansible renverra une erreur, car le fichier est illisible sans déchiffrement.

Vous obtiendrez une erreur similaire à :

```
ERROR! Attempting to decrypt but no vault secrets found
```

Cela signifie qu'Ansible a détecté un fichier chiffré mais n'a pas reçu le mot de passe pour le déchiffrer, rendant ainsi l'exécution impossible.

3. Déchiffrement lors de l'exécution

Solution : Fournir le mot de passe lors de l'exécution

Pour exécuter un playbook contenant des fichiers chiffrés, il faut **fournir le mot de passe de déchiffrement** en utilisant l'option `--ask-vault-pass` ou un fichier contenant le mot de passe (`--vault-password-file`).

Méthode 1 : Demander le mot de passe au moment de l'exécution

```
ansible-playbook -i inventory.ini myplaybook.yml --ask-vault-pass
```

Explication :

- `i inventory.ini` : Spécifie l'inventaire utilisé.
- `myplaybook.yml` : Le playbook à exécuter.
- `-ask-vault-pass` : Demande le mot de passe pour déchiffrer les fichiers chiffrés.

Méthode 2 : Utiliser un fichier contenant le mot de passe

Si vous ne voulez pas saisir le mot de passe à chaque exécution, vous pouvez le stocker dans un fichier (ex. `vault_pass.txt`) et l'utiliser ainsi :

```
ansible-playbook -i inventory.ini myplaybook.yml --vault-password-file vau
lt_pass.txt
```

💡 **Attention** :

Assurez-vous que le fichier contenant le mot de passe a des permissions restreintes (`chmod 600 vault_pass.txt`) pour éviter toute fuite de données sensibles.

Encrypter une Variable du Playbook avec Ansible Vault

1. **Création d'un Fichier pour la Variable Sensible**
   - Créez un fichier séparé, par exemple `secrets.yml`, pour stocker la variable sensible :

     ```
     touch secrets.yml
     ```

2. **Déplacement de la Variable**
   - Ajoutez la variable sensible dans le fichier `secrets.yml` :

     ```
     secret_password: azerty123
     ```

   - Supprimez cette variable du playbook pour éviter qu'elle y reste en clair.
3. **Chiffrement du Fichier avec Ansible Vault**
   - Chiffrez le fichier `secrets.yml` avec la commande suivante :

```
ansible-vault encrypt secrets.yml
```

- Vous serez invité à entrer un mot de passe pour le chiffrement.

4. **Référencement du Fichier dans le Playbook**
   - Modifiez le playbook pour inclure le fichier chiffré avec la directive `vars_files` :

```
vars_files:
  - secrets.yml
```

5. **Exécution du Playbook**
   - Lors de l'exécution, Ansible demandera le mot de passe Vault pour déchiffrer le fichier `secrets.yml` et utiliser la variable dans les tâches :

```
ansible-playbook -i myinventory myplaybook.yml --ask-vault-pass
```

   - Si vous omettez l'option `-ask-vault-pass`, Ansible renverra une erreur indiquant que le mot de passe est nécessaire pour déchiffrer le fichier Vault.

Chapitre4 : Gestion des Variables avec Ansible

1. Types de Variables

- ***Liste*** :Pour créer une variable avec plusieurs valeurs, nous pouvons utiliser la syntaxe des listes YAML :

```
vars:
  region:
    - northeast
    - southeast
    - midwest
```

  - Référencer une liste complète : `{{ regions }}`
  - Accéder à un élément spécifique : `{{ regions[1] }}`

- ***Dictionnaire :*** consiste à stocker les paires clé-valeur dans des variables sous forme de dictionnaires. Par exemple :

```
vars:
  users:
    admin: carole
    developer: peter
    tester: sophie
```

- Référencer tout le dictionnaire : `{{ users }}`
- Accéder à une clé spécifique : `{{ users.admin }}`
- ***Structure Imbriquée :***
  - Combinaison de listes et dictionnaires pour des structures complexes. Exemple :

```
vars:
  users:
    - name: carole
      role: admin
      department: IT
    - name: peter
      role: tester
      department: QA
```

  - Référencer une structure imbriquée complète : `{{ users }}`
  - Accéder à un élément spécifique dans une structure imbriquée : `{{ users[2].role }}`

Variables Enregistrées

- Les variables registred sont crées à partir des sorties de playbooks en utilisant le mot clé register.
- Stockées en RAM, accessibles uniquement dans le playbook et sur le host en cours.

```
- hosts: localhost
  tasks:
    - name: Enregistrer une variable
      command: whoami
      register: utilisateur

    - name: Afficher la variable
      debug:
        msg: "L'utilisateur est {{ utilisateur.stdout }}"
```

Ansible Facts

- Informations sur la machine distante (OS, IP, CPU, etc.).
- Accessibles via la variable `ansible_facts`.

Pour voir ces donnée en brute, il est possible d'utiliser la commande:

$ ansible <hostname> -m ansible.builtin.setup

- Par début, la collecte des facts est lancée au début de chaque play . Pour la désactivée, il suffit de rajouter cette ligne au niveau du play: `gather_facts: no`.

Voici un exemple simple d'utilisation des **Ansible Facts** :

```
- hosts: localhost
  gather_facts: yes  # Active la collecte des facts
  tasks:
    - name: Afficher le nom de l'OS
      debug:
        msg: "Le système d'exploitation est {{ ansible_facts['os_family'] }}"
```

Variables Magiques

- Variables spéciales définies par Ansible, non modifiables par l'utilisateur.
- **Exemples** :
  - `ansible_play_hosts` : Liste des hosts dans le play courant.
  - `hostvars` : Dictionnaire des variables pour tous les hosts de l'inventaire.

Voici un exemple d'utilisation des **variables magiques** dans Ansible :

```
- hosts: localhost
  tasks:
    - name: Afficher tous les hôtes du playbook
      debug:
        msg: "Les hôtes dans ce play sont : {{ ansible_play_hosts }}"

    - name: Afficher les variables d'un hôte spécifique
      debug:
        msg: "Les variables de localhost : {{ hostvars['localhost'] }}"
```

Variables d'Inventaire

- **Définition** : Variables spécifiques à des hosts ou groupes de hosts.
- **Emplacements** :
  - Fichier d'inventaire.
  - Fichiers `host_vars` pour des hosts spécifiques.
  - Fichiers `group_vars` pour des groupes de hosts.

Voici un exemple d'utilisation des **variables d'inventaire** dans Ansible :

Définir les variables dans un fichier d'inventaire (`inventory.ini`)

```
[webservers]
web1 ansible_host=192.168.1.10 ansible_user=ubuntu app_port=8080
```

Utiliser ces variables dans un playbook (`playbook.yml`)

```
- hosts: webservers
  tasks:
    - name: Afficher la variable app_port
      debug:
        msg: "L'application tourne sur le port {{ app_port }}"
```

Explication :

1. **Dans** `inventory.ini` → La variable `app_port=8080` est définie pour `web1`.
2. **Dans le playbook** → `{{ app_port }}` récupère cette valeur.

Utilité : Permet de personnaliser le déploiement selon les machines.

Priorité des Variables dans Ansible

Ordre de Priorité (du plus élevé au plus bas) :

1-Extra_Vars

2-Task_Vars / registred_vars

3-Role_Vars

4-Playbook_Vars

5-Host_vars

6-Inventory_vars

7-Group_vars

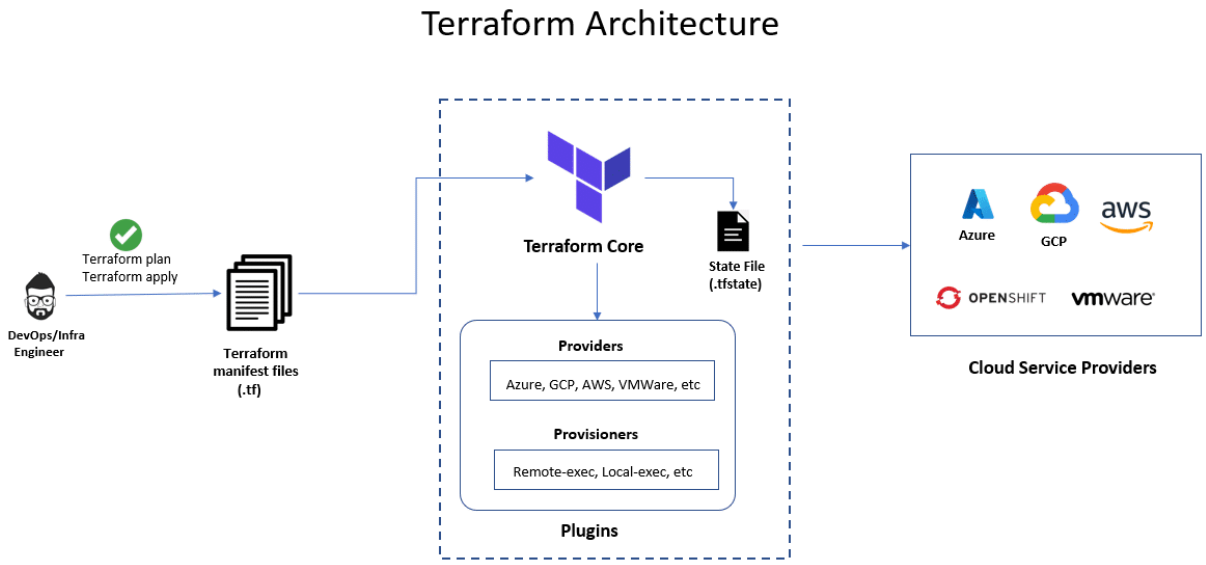
8-Role_default_vars

Ansible choisit la valeur de la variable en fonction de sa priorité et privilégie les variables définies plus récemment ou plus spécifiques.

Chap5 : Initiation à l'outil Terraform

Terraform est un outil d'infrastructure as code qui utilise une approche déclarative. L'utilisateur définit l'état souhaité de l'infrastructure, et Terraform se charge de le mettre en œuvre. Il utilise **HCL (HashiCorp Configuration Language)**, un langage simple et lisible.

Ses points forts incluent la **compatibilité avec plus de 125 fournisseurs d'infrastructure**, comme AWS, Azure, Google Cloud, et VMware.



Architecture Terraform :

- **Terraform Core** : C'est le noyau qui orchestre l'infrastructure définie dans les fichiers de configuration. Il prend en entrée deux éléments :
  1. **Fichiers de configuration (.tf)** : Ils décrivent l'infrastructure à créer, ses ressources, leurs propriétés et les fournisseurs de services.
  2. **Terraform State (terraform.tfstate)** : Il contient l'état actuel de l'infrastructure déployée, permettant à Terraform de comparer et d'élaborer un plan d'exécution pour synchroniser l'infrastructure réelle avec la configuration.
- **Terraform State** : Il garde l'état des ressources et peut être stocké localement ou à distance (ex. : S3, Terraform Cloud). Il assure la synchronisation entre la configuration et les ressources réelles.
- **Providers** : Ce sont des plugins qui permettent à Terraform d'interagir avec des services comme AWS, Azure, GCP, etc. Chaque provider gère des ressources spécifiques. Par exemple, pour une ressource AWS EC2, Terraform utilisera le plugin AWS pour interagir avec l'API AWS.

Workflow de Terraform :

1. **Init (Initialisation)** :

   `terraform init` initialise le répertoire de travail, télécharge les plugins nécessaires (comme AWS, Azure), et prépare l'environnement Terraform pour les étapes suivantes.

2. **Validate (Validation)** :

   `terraform validate` vérifie la syntaxe et la structure des fichiers de configuration `.tf`, s'assurant que les ressources sont bien définies et que les variables sont déclarées.

3. **Plan (Création du plan)** :

   `terraform plan` analyse l'état actuel de l'infrastructure et génère un plan d'exécution pour aligner l'infrastructure avec la configuration souhaitée, permettant à l'utilisateur de vérifier les modifications avant de les appliquer.

4. **Apply (Application des changements)** :

   `terraform apply` applique le plan en créant, modifiant ou supprimant des ressources pour correspondre à la configuration. Les changements sont ensuite enregistrés dans le fichier Terraform State pour maintenir la synchronisation.

5. **Destroy (Destruction des ressources)** :

   `terraform destroy` supprime toutes les ressources gérées par Terraform, nettoyant l'environnement, particulièrement utile dans les environnements de test ou pour éviter des coûts inutiles.

Terraform vs Ansible

Critère	Terraform	Ansible
Objectif principal	Provisionnement d'infrastructure (création, modification, suppression des ressources)	Gestion de la configuration des machines et des applications
Approche	Déclarative (définir l'état souhaité de l'infrastructure)	Impérative (définir les tâches à exécuter)
Langage	HCL (HashiCorp Configuration Language)	YAML (Yet Another Markup Language)
Exécution	Gère l'état de l'infrastructure, crée des plans et applique des modifications	Exécute des tâches sur des serveurs existants sans garder un état permanent
Utilisation	Déploiement et gestion de ressources cloud et infrastructures	Automatisation de la configuration et gestion des applications

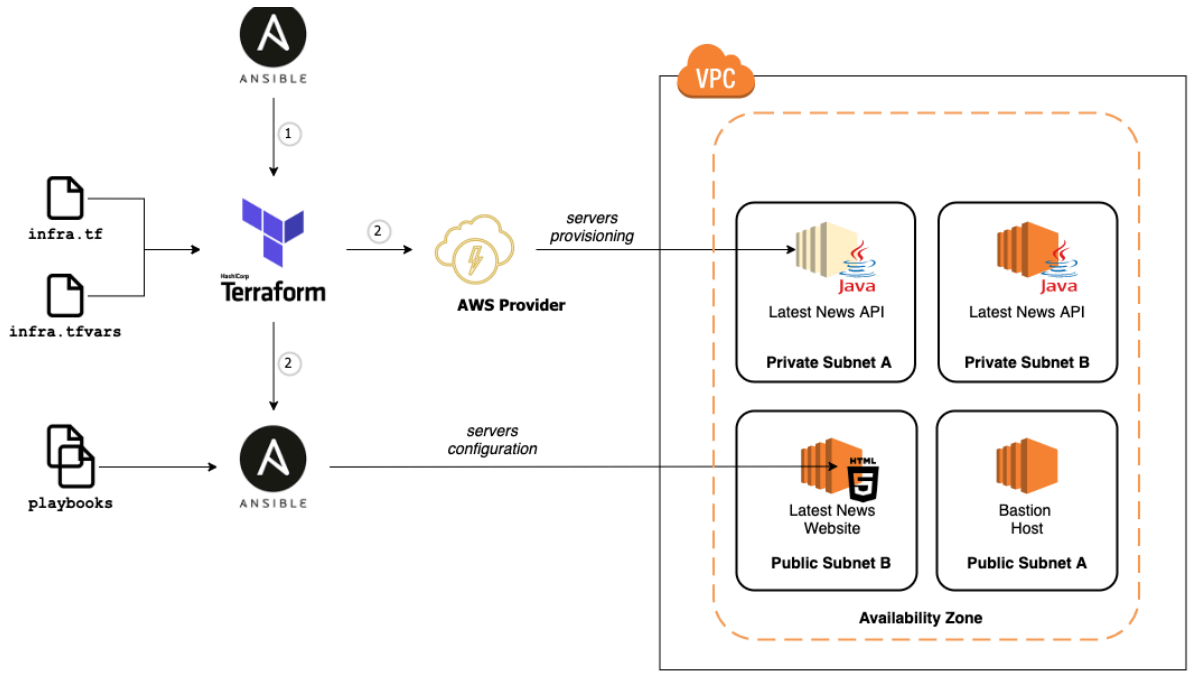
Gestion des dépendances	Gère les dépendances entre ressources automatiquement	Les dépendances doivent être gérées manuellement dans les playbooks
Idéal pour	Provisionnement d'infrastructure (cloud, machines virtuelles, réseaux)	Configuration des serveurs et déploiement d'applications

Quand choisir Terraform ou Ansible ?

Terraform Si vous avez besoin de créer ou gérer l'infrastructure (serveurs, réseaux, bases de données, Ansible Si votre objectif est de configurer et de gérer des systèmes après leur provisionnement

Ansible et Terraform : Complémentarité

Terraform crée l'infrastructure (réseaux, serveurs). Ensuite, Ansible configure les serveurs (installation de logiciels, déploiement d'applications).



Immuable (Immutable) signifie que **Terraform** ne modifie pas directement les ressources existantes, mais plutôt **les recrée ou les remplace** pour atteindre l'état désiré.

Explication :

- **Immuabilité** signifie que les ressources sont considérées comme **non modifiables** après leur création.

- Si une modification est nécessaire, Terraform **détruit** et **recrée** la ressource au lieu de la mettre à jour directement.