**Ansible et l’automatisation des processus**

**Plan**

**Module 1 : Installer ansible, configurer la connexion et commandes ad hoc ansible**

**Installation**

* créer un lab avec LXD
* configurer SSH et python pour utiliser ansible

**configurer ansible**

* /etc ou ansible.cfg
* configuration de la connexion
* connexion SSH et autres plugins de connection
* versions de Python et d’Ansible

**L’inventaire ansible**

* gérer des groupes de machines
* L’inventaire est la source d’information principale pour Ansible

**Ansible ad-hoc et les modules de base**

* la commande ansible et ses options
* explorer les nombreux modules d’Ansible
* idempotence des modules
* exécuter correctement des commandes shell avec Ansible
* le check mode pour controller l’état d’une ressource

**TP1: Installation, configuration et prise en main avec des commandes ad-hoc**

**Module 2 : Les playbooks pour déployer une application web**

**syntaxe yaml des playbooks**

* structure d’un playbook

**modules de déploiement et configuration**

* Templates de configuration avec Jinja2
* gestion des paquets, utilisateurs et fichiers, etc.

**Variable et structures de controle**

* explorer les variables
* syntaxe jinja des variables et lookups
* facts et variables spéciales
* boucles et conditions

**Idempotence d’un playbook**

* handlers
* contrôler le statut de retour des tâches
* gestion de l’idempotence des commandes Unix

**debugging de playbook**

* verbosite
* directive de debug
* gestion des erreurs à l’exécution

**TP2: Écriture d’un playbook simple de déploiement d’une application web flask en python.**

**Module 3 : Structurer un projet, utiliser les roles**

**Complexifier notre lab en ajoutant de nouvelles machines dans plusieurs groupes.**

* modules de provisionning de machines pour Ansible
* organisation des variables de l’inventaire
* la commande ansible-inventory

**Les roles**

* Ansible Galaxy pour installer des roles.
* Architecture d’un role et bonnes pratiques de gestion des roles.

**Écrire un role et organiser le projet**

* Imports et includes réutiliser du code.
* Bonne pratiques d’organisation d’un projet Ansible
* Utiliser des modules personnalisés et des plugins pour étendre Ansible
* gestion de version du code Ansible

**TP3: Transformation de notre playbook en role et utilisation de roles ansible galaxy pour déployer une infrastructure multitiers.**

**Module 4 : Orchestrer Ansible dans un contexte de production**

**Intégration d’Ansible**

* Intégrer ansible dans le cloud un inventaire dynamique et Terraform
* Différents type d’intégration Ansible

**Orchestration**

* Stratégies : Parallélisme de l’exécution
* Délégation de tâche
* Réalisation d’un rolling upgrade de notre application web grace à Ansible
* Inverser des tâches Ansible - stratégies de rollback
* Exécution personnalisée avec des tags

**Sécurité**

* Ansible Vault : gestion des secrets pour l’infrastructure as code
* desctiver les logs des taches sensibles
* Renforcer le mode de connexion ansible avec un bastion SSH

**Exécution d’Ansible en production**

* Intégration et déploiement avec Gitlab
* Gérer une production Ansible découvrir TOWER/AWX
* Tester ses roles et gérer de multiples versions

**TP4: Refactoring de notre code pour effectuer un rolling upgrade et déploiement dans le cloud + AWX**

**Présentation d’Ansible**

**Ansible**

Ansible est un **gestionnaire de configuration** et un **outil de déploiement et d’orchestration** très populaire et central dans le monde de l'**infrastructure as code** (IaC).

Il fait donc également partie de façon centrale du mouvement **DevOps** car il s’apparente à un véritable **couteau suisse** de l’automatisation des infrastructures.

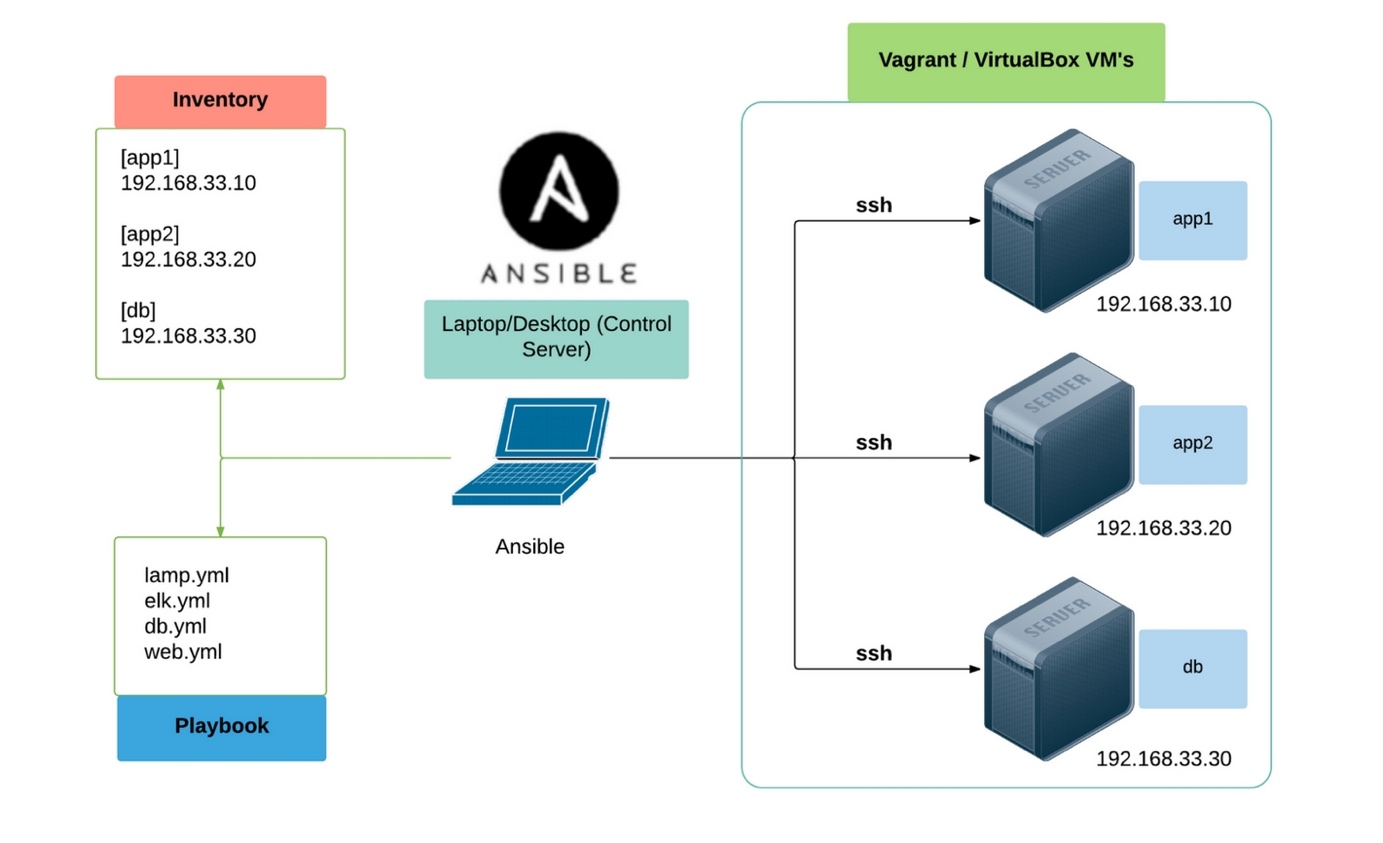
**Histoire**

Ansible a été créé en **2012** (plus récent que ses concurrents Puppet et Chef) autour d’une recherche de **simplicité** et du principe de configuration **agentless**.

Très orienté linux/opensource et versatile il obtient rapidement un franc succès et s’avère être un couteau suisse très adapté à l’automatisation DevOps et Cloud dans des environnements hétérogènes.

Red Hat rachète Ansible en 2015 et développe un certain nombre de produits autour (Ansible Tower, Ansible container avec Openshift).

**Architecture : simplicité et portabilité avec ssh et python**



Ansible est **agentless** c’est à dire qu’il ne nécessite aucun service/daemon spécifique sur les machines à configurer.

La simplicité d’Ansible provient également du fait qu’il s’appuie sur des technologies linux omniprésentes et devenues universelles.

* **ssh** : connexion et authentification classique avec les comptes présents sur les machines.
* **python** : multiplateforme, un classique sous linux, adapté à l’admin sys et à tous les usages.

De fait Ansible fonctionne efficacement sur toutes les distributions linux, debian, centos, ubuntu en particulier (et maintenant également sur Windows).

**Ansible pour la configuration**

Ansible est **semi-déclaratif** c’est à dire qu’il s’exécute **séquentiellement** mais idéalement de façon **idempotente**.

Il permet d’avoir un état descriptif de la configuration:

* qui soit **auditable**
* qui peut **évoluer progressivement**
* qui permet d'**éviter** que celle-ci ne **dérive** vers un état inconnu

**Ansible pour le déploiement et l’orchestration**

Peut être utilisé pour des **opérations ponctuelles** comme le **déploiement**:

* vérifier les dépendances et l’état requis d’un système
* récupérer la nouvelle version d’un code source
* effectuer une migration de base de données (si outil de migration)
* tests opérationnels (vérifier qu’un service répond)

**Ansible à différentes échelles**

Les cas d’usages d’Ansible vont de …:

* petit:
  + … un petit playbook (~script) fournit avec le code d’un logiciel pour déployer en mode test.
  + … la configuration d’une machine de travail personnelle.
  + etc.
* moyen:
  + … faire un lab avec quelques machines.
  + … déployer une application avec du code, une runtime (php/jav etc) et une base de données à migrer.
  + etc.
* grand:
  + … gestion de plusieurs DC avec des produits multiples.
  + … gestion multi-équipes et logging de toutes les opérations grâce à Ansible Tower.
  + etc.

**Ansible et Docker**

Ansible est très complémentaire à docker:

* Il permet de provisionner des machines avec docker ou kubernetes installé pour ensuite déployer des conteneurs.
* Il permet une orchestration simple des conteneur avec le module docker\_container.

Plus récemment avec l’arrivé d'Ansible container il est possible de construire et déployer des conteneurs docker avec du code ansible. Cette solution fait partie de la stack Red Hat Openshift. Concrètement le langage ansible remplace (avantageusement ?) le langage Dockerfile pour la construction des images Docker.

**Partie 1, Installation, configuration et commandes ad hoc.**

Pour l’installation plusieurs options sont possibles:

* Avec le gestionnaire de paquet de la distribution ou homebrew sur OSX:
  + version généralement plus ancienne (2.4 ou 2.6)
  + facile à mettre à jour avec le reste du système
  + Pour installer une version récente on il existe des dépots spécifique à ajouter: exemple sur ubuntu: sudo apt-add-repository --yes --update ppa:ansible/ansible
* Avec pip le gestionnaire de paquet du langage python: sudo pip3 install
  + installe la dernière version stable (2.8 actuellement)
  + commande d’upgrade spécifique sudo pip3 install ansible --upgrade
  + possibilité d’installer facilement une version de développement pour tester de nouvelles fonctionnalité ou anticiper les migrations.

Pour voir l’ensemble des fichier installé par un paquet pip3 :

pip3 show -f ansible | less

Pour tester la connexion aux serveurs on utilise la commande ad hoc suivante. ansible all -m ping

**Les inventaires statiques**

Il s’agit d’une liste de machines sur lesquelles vont s’exécuter les modules Ansible. Les machines de cette liste sont:

* Classées par groupe et sous groupes pour être désignables collectivement (exp executer telle opération sur)
* La méthode connexion est précisée soit globalement soit pour chaque machine.
* Des variables peuvent être définies pour chaque machine ou groupe pour contrôler dynamiquement par la suite la configuration ansible.
* Classées par groupe et sous groupes pour être désignables collectivement (exp executer telle opération sur)
* La méthode connexion est précisée soit globalement soit pour chaque machine.
* Des variables peuvent être définies pour chaque machine ou groupe pour contrôler dynamiquement par la suite la configuration ansible.

Exemple :

[all:vars]

ansible\_ssh\_user=elie

ansible\_python\_interpreter=/usr/bin/python3

[awx\_nodes]

awxnode1 node\_state=started ansible\_host=10.164.210.101 container\_image=centos\_ansible\_20190901

[dbservers]

pgnode1 node\_state=started ansible\_host=10.164.210.111 container\_image=centos\_ansible\_20190901

pgnode2 node\_state=started ansible\_host=10.164.210.112 container\_image=centos\_ansible\_20190901

[appservers]

appnode1 node\_state=started ansible\_host=10.164.210.121 container\_image=centos\_ansible\_20190901

appnode2 node\_state=started ansible\_host=10.164.210.122 container\_image=centos\_ansible\_20190901

Les inventaires peuvent également être au format YAML (plus lisible mais pas toujours intuitif) ou JSON (pour les machines).

**Configuration**

Ansible se configure classiquement au niveau global dans le dossier /etc/ansible/ dans lequel on retrouve en autre l’inventaire par défaut et des paramètre de configuration.

Ansible est très fortement configurable pour s’adapter à des environnement contraints. Liste des paramètre de configuration:

Alternativement on peut configurer ansible par projet avec un fichier ansible.cfg présent à la racine. Toute commande ansible lancée à la racine du projet récupère automatiquement cette configuration.

**La commande ansible**

* version minimale : ansible <groupe\_machine> -m <module> -a <arguments\_module>
* ansible all -m ping: Permet de tester si les hotes sont joignables et ansible utilisable (SSH et python sont présents et configurés).
* version plus complète : ansible <groupe\_machine> --inventory <fichier\_inventaire> --become -m <module> -a <arguments\_module>

**Les modules Ansible**

Ansible fonctionne grâce à des modules python téléversés sur sur l’hôte à configurer puis exécutés. Ces modules sont conçus pour être cohérents et versatiles et rendre les tâches courantes d’administration plus simples.

Il en existe pour un peu toute les tâches raisonnablement courantes : un slogan Ansible “Batteries included” ! Plus de 1300 modules sont intégrés par défaut.

* ping: un module de test Ansible (pas seulement réseau comme la commande ping)
* yum/apt: pour gérer les paquets sur les distributions basées respectivement sur Red Hat ou Debian.

... -m yum -a "name=openssh-server state=present"

* systemd (ou plus générique service): gérer les services/daemons d’un système.

... -m systemd -a "name=openssh-server state=started"

* user: créer des utilisateurs et gérer leurs options/permission/groupes
* file: pour créer, supprimer, modifier, changer les permission de fichiers, dossier et liens.
* shell: pour exécuter des commandes unix grace à un shell

**Option et documentation des modules**

La documentation des modules Ansible se trouve à l’adresse <https://docs.ansible.com/ansible/latest/modules/file_module.html>

Chaque module propose de nombreux arguments pour personnaliser son comportement:

exemple: le module file permet de gérer de nombreuses opérations avec un seul module en variant les arguments.

Il est également à noter que la plupart des arguments sont facultatifs.

* cela permet de garder les appel de modules très succints pour les taches par défaut
* il est également possible de rendre des paramètres par défaut explicites pour augmenter la clarté du code.

Exemple et bonne pratique: toujours préciser state: present même si cette valeur est presque toujours le défaut implicite.

**Commençons le TP1**

**Cours 2 - Les playbooks Ansible, modules de base, variables et structures de contrôle**

Les commandes ad-hoc sont des appels directs de modules Ansible qui fonctionnent de façon idempotente mais ne présente pas les avantages du code qui donne tout son intérêt à l’IaC:

* texte descriptif écrit une fois pour toute
* logique lisible et auditable
* versionnable avec git
* reproductible et incrémental

La dimension incrémentale du code rend en particulier plus aisé de construire une infrastructure progressivement en la complexifiant au fur et à mesure plutôt que de devoir tout plannifier à l’avance.

Le playbook est une sorte de script ansible, c’est à dire du code. Le nom provient du football américain : il s’agit d’un ensemble de stratégies qu’une équipe a travaillé pour répondre aux situations du match. Elle insiste sur la versatilité de l’outil.

**Syntaxe yaml**

Les playbooks ansible sont écrits au format **YAML**.

* YAML est basé sur les identations à base d’espaces (2 espaces par indentation en général). Comme le langage python.
* C’est un format assez lisible et simple à écrire bien que les indentations soient parfois difficiles à lire.
* C’est un format assez flexible avec des types liste et dictionnaires qui peuvent s’imbriquer.
* Le YAML est assez proche du JSON (leur structures arborescentes typées sont isomorphes) mais plus facile à écrire.

A quoi ça ressemble ?

**Une liste**

- 1

- Poire

- "Message à caractère informatif"

**Un dictionnaire**

clé1: valeur1

clé2: valeur2

clé3: 3

**Un exemple imbriqué plus complexe**

marché: # debut du dictionnaire global "marché"

lieu: Crimée Curial

jour: dimanche

horaire:

unité: "heure"

min: 9

max: 14 # entier

fruits: #liste de dictionnaires décrivant chaque fruit

- nom: pomme

couleur: "verte"

pesticide: avec #les chaines sont avec ou sans " ou '

# on peut sauter des lignes dans interrompre la liste ou le dictionnaire en court

- nom: poires

couleur: jaune

pesticide: sans

légumes: #Liste de 3 éléments

- courgettes

- salade

- potiron

#fin du dictionnaire global

Pour mieux visualiser l’imbrication des dictionnaires et des listes en YAML on peut utiliser un convertisseur YAML -> JSON : <https://www.json2yaml.com/>.

Notre marché devient:

{

"marché": {

"lieu": "Crimée Curial",

"jour": "dimanche",

"horaire": {

"unité": "heure",

"min": 9,

"max": 14

},

"fruits": [

{

"nom": "pomme",

"couleur": "verte",

"pesticide": "avec"

},

{

"nom": "poires",

"couleur": "jaune",

"pesticide": "sans"

}

],

"légumes": [

"courgettes",

"salade",

"potiron"

]

}

}

Observez en particulier la syntaxe assez condensée de la liste “fruits” en YAML qui est une liste de dictionnaires.

**Structure d’un playbook**

---

- name: premier play # une liste de play (chaque play commence par un tiret)

hosts: serveur\_web # un premier play

become: yes

gather\_facts: false # récupérer le dictionnaires d'informations (facts) relatives aux machines

vars:

logfile\_name: "auth.log"

var\_files:

- mesvariables.yml

pre\_tasks:

- name: dynamic variable

set\_fact:

mavariable: "{{ inventory\_hostname + 'prod' }}" #guillemets obligatoires

roles:

- flaskapp

tasks:

- name: installer le serveur nginx

apt: name=nginx state=present # syntaxe concise proche des commandes ad hoc mais moins lisible

- name: créer un fichier de log

file: # syntaxe yaml extensive : conseillée

path: /var/log/{{ logfile\_name }} #guillemets facultatifs

mode: 755

- import\_tasks: mestaches.yml

handlers:

- systemd:

name: nginx

state: "reloaded"

- name: un autre play

hosts: dbservers

tasks:

...

* Un playbook commence par un tiret car il s’agit d’une liste de plays.
* Un play est un dictionnaire yaml qui décrit un ensemble de taches ordonnées en plusieurs sections. Un play commence par préciser sur quelles machines il s’applique puis précise quelques paramètres faculatifs d’exécution comme become: yes pour l’élévation de privilège (section hosts).
* La section hosts est obligatoire. Toutes les autres sections sont **facultatives** !
* La section tasks est généralement la section principale car elle décrit les taches de configuration à appliquer.
* La section tasks peut être remplacée ou complétée par une section roles et des sections pre\_tasks post\_tasks
* Les handlers sont des tâches conditionnelles qui s’exécutent à la fin (post traitements conditionnels comme le redémarrage d’un service)

**Ordre d’execution**

1. pre\_tasks
2. roles
3. tasks
4. post\_tasks
5. handlers

Les roles ne sont pas des tâches à proprement parler mais un ensemble de tâches et ressources regroupées dans un module un peu comme une librairie developpement. Cf. cours 3.

**bonnes pratiques de syntaxe**

* Indentation de deux espaces.
* Toujours mettre un name: qui décrit lors de l’execution la tache en court : un des principes de l’IaC est l’intelligibilité des opérations.
* Utiliser les arguments au format yaml (sur plusieurs lignes) pour la lisibilité, sauf s’il y a peu d’arguments

Pour valider la syntaxe il est possible d’installer et utiliser ansible-linter sur les fichiers YAML.

**Imports et includes**

Il est possible d’importer le contenu d’autres fichiers dans un playbook:

* import\_tasks: importe une liste de tâches (atomiques)
* import\_playbook: importe une liste de play contenus dans un playbook.

Les deux instructions précédentes désignent un import **statique** qui est résolu avant l’exécution.

Au contraire, include\_tasks permet d’intégrer une liste de tâche **dynamiquement** pendant l’exécution

Par exemple:

vars:

apps:

- app1

- app2

- app3

tasks:

- include\_tasks: install\_app.yml

loop: "{{ apps }}"

Ce code indique à Ansible d’executer une série de tâches pour chaque application de la liste. On pourrait remplacer cette liste par une liste dynamique. Comme le nombre d’import ne peut pas facilement être connu à l’avance on **doit** utiliser include\_tasks.

**Élévation de privilège**

L’élévation de privilège est nécessaire lorsqu’on a besoin d’être root pour exécuter une commande ou plus généralement qu’on a besoin d’exécuter une commande avec un utilisateur différent de celui utilisé pour la connexion on peut utiliser:

* Au moment de l’exécution l’argument --become en ligne de commande avec ansible, ansible-console ou ansible-playbook.
* La section become: yes
  + au début du play (après hosts) : toutes les tâches seront executée avec cette élévation par défaut.
  + après n’importe quelle tâche : l’élévation concerne uniquement la tâche cible.
* Pour executer une tâche avec un autre utilisateur que root (become simple) ou celui de connexion (sans become) on le précise en ajoutant à become: yes, become\_user: username

**Variables Ansible**

Ansible utilise en arrière plan un dictionnaire contenant de nombreuses variables.

Pour s’en rendre compte on peut lancer : ansible <hote\_ou\_groupe> -m debug -a "msg={{ hostvars }}"

Ce dictionnaire contient en particulier:

* des variables de configuration ansible (ansible\_user par exemple)
* des facts c’est à dire des variables dynamiques caractérisant les systèmes cible (par exemple ansible\_os\_family) et récupéré au lancement d’un playbook.
* des variables personnalisées (de l’utilisateur) que vous définissez avec vos propre nom généralement en **snake\_case**.

**Jinja2 et variables dans les playbooks et rôles (fichiers de code)**

La plupart des fichiers Ansible (sauf l’inventaire) sont traités avec le moteur de template python JinJa2.

Ce moteur permet de créer des valeurs dynamiques dans le code des playbooks, des roles, et des fichiers de configuration.

* Les variables écrites au format {{ mavariable }} sont remplacées par leur valeur provenant du dictionnaire d’exécution d’Ansible.
* Des filtres (fonctions de transformation) permettent de transformer la valeur des variables: exemple : {{ hostname | default('localhost') }} (Voir plus bas)

**Jinja2 et les variables dans les fichiers de templates**

Les fichiers de templates (.j2) utilisés avec le module template, généralement pour créer des fichiers de configuration peuvent **contenir des variables** et des **filtres** comme les fichier de code (voir au dessus) **mais également** d’autres constructions jinja2 comme:

* Des if : {% if nginx\_state == 'present' %}...{% endif %}.
* Des boucles for : {% for host in groups['appserver'] %}...{% endfor %}.
* Des inclusions de templates {% include 'autre\_fichier\_template.j2' %}

**Définition des variables**

On peut définir et modifier la valeur des variables à différents endroits du code ansible:

* La section vars: du playbook.
* Un fichier de variables appelé avec var\_files:
* L’inventaire : variables pour chaque machine ou pour le groupe.
* Dans des dossier extension de l’inventaire group\_vars, host\_bars
* Dans le dossier defaults des roles (cf partie sur les roles)
* Dans une tache avec le module set\_facts.
* A runtime au moment d’appeler la CLI ansible avec --extra-vars "version=1.23.45 other\_variable=foo"

Lorsque définies plusieurs fois, les variables ont des priorités en fonction de l’endroit de définition. L’ordre de priorité est plutôt complexe: https://docs.ansible.com/ansible/latest/user\_guide/playbooks\_variables.html#variable-precedence-where-should-i-put-a-variable

En résumé la règle peut être exprimée comme suit: les variables de runtime sont prioritaires sur les variables dans un playbook qui sont prioritaires sur les variables de l’inventaire qui sont prioritaires sur les variables par défaut d’un role.

* Bonne pratique: limiter les redéfinitions de variables en cascade (au maximum une valeur par défaut, une valeur contextuelle et une valeur runtime) pour éviter que le playbook soit trop complexe et difficilement compréhensible et donc maintenable.

**Remarques de syntaxe**

* groups.all et groups['all'] sont deux syntaxes équivalentes pour désigner les éléments d’un dictionnaire.

**variables spéciales**

<https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html>

Les plus utiles:

* hostvars: dictionaire de toute les variables rangées par hote de l’inventaire.
* ansible\_host: information utilisée pour la connexion (ip ou domaine).
* inventory\_hostname: nom de la machine dans l’inventaire.
* groups: dictionnaire de tous les groupes avec la liste des machines appartenant à chaque groupe.

Pour explorer chacune de ces variables vous pouvez utiliser le module debug en mode adhoc ou dans un playbook:

ansible <hote\_ou\_groupe> -m debug -a "msg={{ ansible\_host }}"

ou encore:

ansible <hote\_ou\_groupe> -m debug -a "msg={{ groups.all }}"

**Facts**

Les facts sont des valeurs de variables récupérées au début de l’exécution durant l’étape **gather\_facts** et qui décrivent l’état courant de chaque machine.

* Par exemple, ansible\_os\_family est un fact/variable décrivant le type d’OS installé sur la machine. Elle n’existe qu’une fois les facts récupérés.

! Lors d’une **commande adhoc** ansible les **facts** ne sont pas récupérés : la variable ansible\_os\_family ne sera pas disponible.

La liste des facts peut être trouvée dans la documentation et dépend des plugins utilisés pour les récupérés: <https://docs.ansible.com/ansible/latest/user_guide/playbooks_vars_facts.html>

**Structures de controle Ansible (et non JinJa2)**

**La directive when**

Elle permet de rendre une tâche conditionnelle (une sorte de if)

- name: start nginx service

systemd:

name: nginx

state: started

when: ansible\_os\_family == 'RedHat'

Sinon la tache est sautée (skipped) durant l’exécution.

**La directive loop:**

Cette directive permet d’executer une tache plusieurs fois basée sur une liste de valeur:

<https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html>

exemple:

- hosts: localhost

tasks:

- name: exemple de boucle

debug:

msg: "{{ item }}"

loop:

- message1

- message2

- message3

On peut également controler cette boucle avec quelques paramètres:

- hosts: localhost

vars:

messages:

- message1

- message2

- message3

tasks:

- name: exemple de boucle

debug:

msg: "message numero {{ num }} : {{ message }}"

loop: "{{ messages }}"

loop\_control:

loop\_var: message

index\_var: num

Cette fonctionnalité de boucle était anciennement accessible avec le mot clé with\_items: qui est maintenant déprécié.

**Filtres Jinja**

Pour transformer la valeur des variables à la volée lors de leur appel on peut utiliser des filtres (jinja2) :

* par exemple on peut fournir une valeur par défaut pour une variable avec filtre default: {{ hostname | default('localhost') }}
* Un autre usage courant des filtres est de reformater et filtrer des listes et dictionnaires de paramètre. Ces syntaxes sont peut intuitives. Vous pouvez vous entrainer en regardant ces tutoriels:
  + <https://www.tailored.cloud/devops/how-to-filter-and-map-lists-in-ansible/>
  + <https://www.tailored.cloud/devops/advanced-list-operations-ansible/>

La liste complète des filtres ansible se trouve ici : <https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html>

**Debugger un playbook.**

Avec Ansible on dispose d’au moins trois manières de debugger un playbook:

* Rendre la sortie verbeuse (mode debug) avec -vvv.
* Utiliser une tache avec le module debug : debug msg="{{ mavariable }}".
* Utiliser la directive debugger: always ou on\_failed à ajouter à la fin d’une tâche. L’exécution s’arrête alors après l’exécution de cette tâche et propose un interpreteur de debug.

Les commandes et l’usage du debugger sont décris dans la documentation: <https://docs.ansible.com/ansible/latest/user_guide/playbooks_debugger.html>

**Cours 3 - Organiser un projet modulariser son code avec les roles**

**Organisation d’un dépot de code Ansible**

Voici, extrait de la documentation Ansible sur les “Best Practice”, l’une des organisations de référence d’un projet ansible de configuration d’une infrastructure:

production # inventory file for production servers

staging # inventory file for staging environment

group\_vars/

group1.yml # here we assign variables to particular groups

group2.yml

host\_vars/

hostname1.yml # here we assign variables to particular systems

hostname2.yml

site.yml # master playbook

webservers.yml # playbook for webserver tier

dbservers.yml # playbook for dbserver tier

roles/

common/ # this hierarchy represents a "role"

... # role code

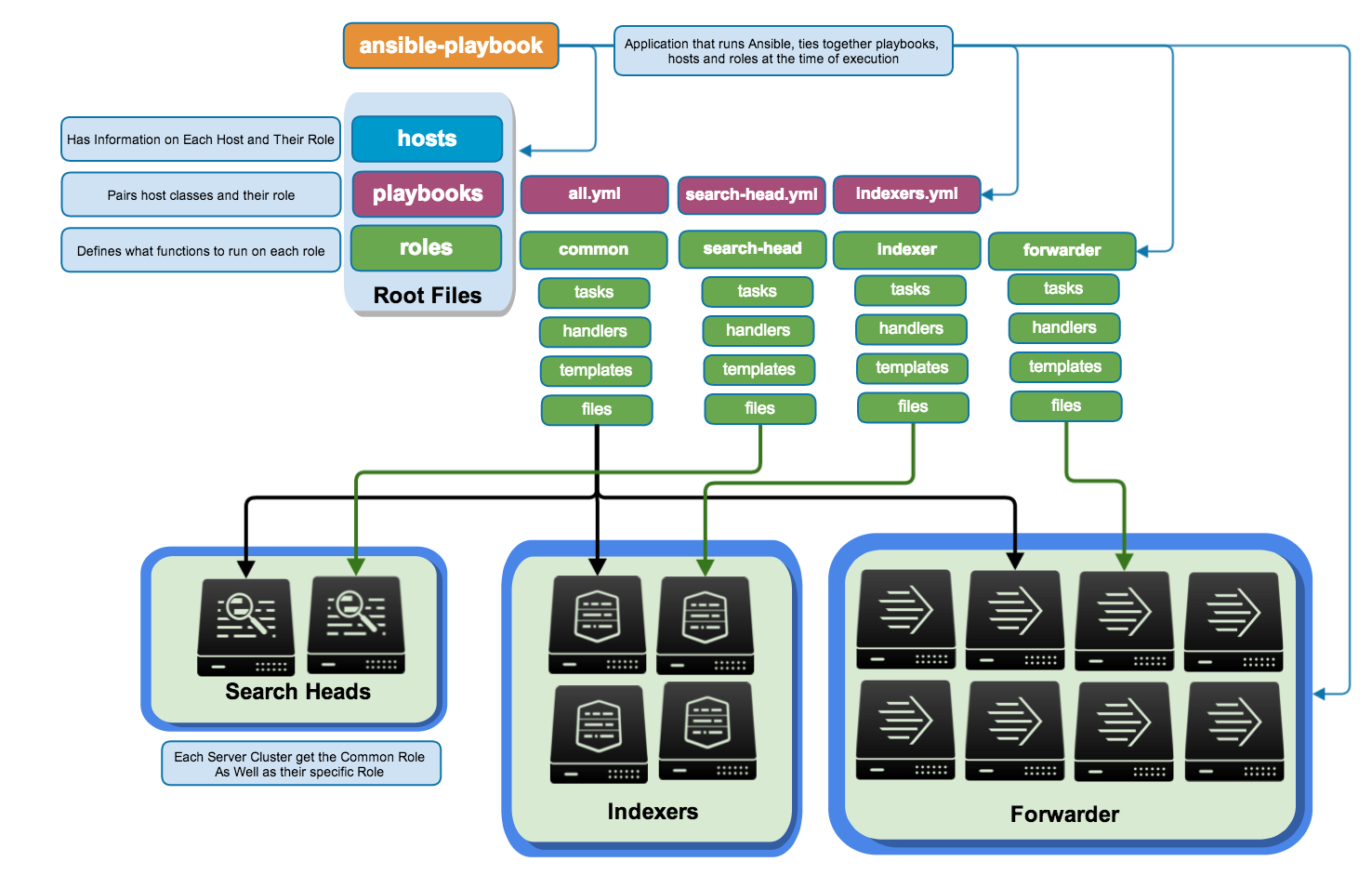
webtier/ # same kind of structure as "common" was above, done for the webtier role

monitoring/ # ""

fooapp/ # ""

Plusieurs remarques:

* Chaque environnement (staging, production) dispose d’un inventaire ce qui permet de préciser à runtime quel environnement cibler avec l’option --inventaire production.
* Chaque groupe de serveurs (tier) dispose de son playbook
  + qui s’applique sur le groupe en question.
  + éventuellement définit quelques variables spécifiques (mais il vaut mieux les mettre dans l’inventaire ou les dossiers cf suite).
  + Idéalement contient un minimum de tâches et plutôt des roles (ie des tâches rangées dans une sorte de module)
* Pour limiter la taille de l’inventaire principal on range les variables communes dans des dossiers group\_vars et host\_vars. On met à l’intérieur un fichier <nom\_du\_groupe>.yml qui contient un dictionnaire de variables.
* On cherche à modulariser au maximum la configuration dans des roles c’est à dire des modules rendus génériques et specifique à un objectif de configuration.
* Ce modèle d’organisation correspond plutôt à la **configuration** de base d’une infrastructure (playbooks à exécuter régulièrement) qu’à l’usage de playbooks ponctuels comme pour le déploiement. Mais, bien sur, on peut ajouter un dossier playbooks ou operations pour certaines opérations ponctuelles. (cf cours 4)
* Si les modules de Ansible (complétés par les commandes bash) ne suffisent pas on peut développer ses propre modules ansible.
  + Il s’agit de programmes python plus ou moins complexes
  + On les range alors dans le dossier library du projet ou d’un role et on le précise éventuellement dans ansible.cfg.
* Observons le role Common : il est utilisé ici pour rassembler les taches de base des communes à toutes les machines. Par exemple s’assurer que les clés ssh de l’équipe sont présentes, que les dépots spécifiques sont présents etc.



**Roles Ansible**

**Objectif:**

* Découper les tâches de configuration en sous ensembles réutilisables (une suite d’étapes de configuration).
* Ansible est une sorte de langage de programmation et l’intéret du code est de pouvoir créer des fonction regroupées en librairies et les composer. Les roles sont les “librairies/fonction” ansible en quelque sorte.
* Comme une fonction un role prend généralement des paramètres qui permettent de personnaliser son comportement.
* Tout le nécessaire doit y être (fichiers de configurations, archives et binaires à déployer, modules personnels dans library etc.)
* Remarque ne pas confondre **modules** et **roles** : file est un module geerlingguy.docker est un role. On **doit** écrire des roles pour coder correctement en Ansible, on **peut** écrire des modules mais c’est largement facultatif car la plupart des actions existent déjà.
* Présentation d’un exemple de role : <https://github.com/geerlingguy/ansible-role-docker>
  + Dans la philosophie Ansible on recherche la généricité des roles. On cherche à ajouter des paramètres pour que le rôle s’adapte à différents cas (comme notre playbook flask app).
  + Une bonne pratique: préfixer le nom des paramètres par le nom du role exemple docker\_edition.
  + Cependant la généricité est nécessaire quand on veut distribuer le role ou construire des outils spécifiques qui serve à plus endroit de l’infrastructure mais elle augmente la complexité.
  + Donc pour les roles internes on privilégie la simplicité.
  + Les roles contiennent idéalement un fichier README en décrire l’usage et un fichier meta/main.yml qui décrit la compatibilité et les dépendanice en plus de la licence et l’auteur.
  + Il peuvent idéalement être versionnés dans des dépots à part et installé avec ansible-galaxy

**Structure d’un rôle**

Un role est un dossier avec des sous dossiers conventionnels:

roles/

common/ # this hierarchy represents a "role"

tasks/ #

main.yml # <-- tasks file can include smaller files if warranted

handlers/ #

main.yml # <-- handlers file

templates/ # <-- files for use with the template resource

ntp.conf.j2 # <------- templates end in .j2

files/ #

foo.sh # <-- script files for use with the script resource

vars/ #

main.yml # <-- variables associated with this role

defaults/ #

main.yml # <-- default lower priority variables for this role

meta/ #

main.yml # <-- role dependencies

library/ # roles can also include custom modules

module\_utils/ # roles can also include custom module\_utils

lookup\_plugins/

On constate que les noms des sous dossiers correspondent souvent à des sections du playbook. En fait le principe de base est d’extraire les différentes listes de taches ou de variables dans des sous-dossier

* Remarque : les fichier de liste **doivent nécessairement** s’appeler **main.yml**" (pas très intuitif)
* Remarque2 : main.yml peut en revanche importer d’autre fichiers aux noms personnalisés (exp role docker de geerlingguy)
* Le dossier defaults contient les valeurs par défaut des paramètres du role. Ces valeurs ne sont jamais prioritaires (elles sont écrasées par n’importe quelle redéfinition)
* Le fichier meta/main.yml est facultatif mais conseillé et contient des informations sur le role
  + auteur
  + license
  + compatibilité
  + version
  + dépendances à d’autres roles.
* Le dossier files contient les fichiers qui ne sont pas des templates (pour les module copy ou sync, script etc).

**Ansible Galaxy**

C’est le store de roles officiel d’Ansible : <https://galaxy.ansible.com/>

C’est également le nom d’une commande ansible-galaxy qui permet d’installer des roles et leurs dépendances depuis internet. Un sorte de gestionnaire de paquet pour ansible.

Elle est utilisée généralement sour la forme ansible install -r roles/requirements.yml -p roles <nom\_role> ou plus simplement ansible-galaxy install <role> mais installe dans /etc/ansible/roles.

Tous les rôles ansible sont communautaires (pas de roles officiels) et généralement stockés sur github.

Mais on peut voir la popularité la qualité et les tests qui garantissement la plus ou moins grande fiabilité du role

Il existe des roles pour installer un peu n’importe quelle application serveur courante aujourd’hui. Passez du temps à explorer le web avant de développer quelque chose avec Ansible

**Installer des roles avec requirements.yml**

Conventionnellement on utilise un fichier requirements.yml situé dans roles pour décrire la liste des roles nécessaires à un projet.

- src: geerlingguy.repo-epel

- src: geerlingguy.haproxy

- src: geerlingguy.docke

# from GitHub, overriding the name and specifying a specific tag

- src: https://github.com/bennojoy/nginx

version: master

name: nginx\_role

* Ensuite pour les installer on lance: ansible-galaxy install -r roles/requirements.yml -p roles.

**Cours 4 - Ansible en production, sécurité et Cloud**

**Execution d’Ansible en production**

L’intérêt d’un outil d’installation idempotent comme Ansible est de pouvoir exécuter de façon régulière et automatiser l’execution d’Ansible pour s’assurer de la conformité de l’infrastructure avec le code.

Une production Ansible est généralement un serveur spécial (parfois appelé un ansible master) depuis lequel le code peut être exécuté, ponctuellement ou de préférence régulièrement (2x par jours par exemple).

Le serveur Ansible s’assure également que les exécutions sont correctement logguées et que les DevOps peuvent par la suite s’assurer que les différentes exécutions se sont déroulées correctement et éventuellement lire les logs d’execution pour diagnostiquer les erreurs.

**Différentes solutions de serveur de production Ansible**

* Ansible Tower/AWX : La solution “officielle” pour exécuter ansible en production promue par RedHat. AWX est l’upstream open source de Tower. Cette solution est assez lourde à déployer et n’exécute que du Ansible (peu versatile) mais elle a été prouvé adapté pour des très grosses production pilotées principalement par Ansible.
* Un serveur master Linux simple pour executer Ansible en CLI ou en Cron : plus léger et versatile mais ne propose par de dashboard pour afficher l’état de de l’infrastructure
* Rundeck: une solution générique pour exécuter des Jobs d’infrastructure qui s’intègre plutôt correctement avec Ansible.
* Jenkins: souvent associé à la CI/CD, Jenkins est en réalité un serveur générique pour exécuter des Jobs automatiquement et à la demande. Il propose un plugin Ansible intéssant et permet de consulté les logs d’exécution et d’avoir une vue globale des dernières exécutions à travers des dashboard. Il est très flexible mais assez complexe à configurer correctement.

Nous allons pour le dernier TP de ce module utiliser Jenkins pour exécuter Ansible. Ainsi nous pouvons découvrir un peu en avance Jenkins qui est complexe et important pour la fin du cursus.

**Sécurité**

Les problématiques de sécurité linux ne sont pas résolue magiquement par Ansible. Tous le travail de réflexion et de sécurisation reste identique mais peut comme le reste être mieux controllé grace à l’approche déclarative de l’infrastructure as code.

Si cette problématique des liens entre Ansible et sécurité vous intéresse : Security automation with Ansible

Il est à noter tout de même qu’Ansible est généralement apprécié d’un point de vue sécurité car il n’augmente pas (vraiment) la surface d’attaque de vos infrastructure : il est basé sur ssh qui est éprouvé et ne nécessite généralement pas de réorganisation des infrastructures.

Pour les cas plus spécifiques et si vous voulez éviter ssh, Ansible est relativement agnostique du mode de connexion grâce aux plugins de connexions (voir ci-dessous).

**Authentification et SSH**

Un bonne pratique importante : changez le port de connexion ssh pour un port atypique. Ajoutez la variable ansible\_ssh\_port=17728 dans l’inventaire.

Il faut idéalement éviter de créer un seul compte ansible de connexion pour toutes les machines:

* difficile à bouger
* responsabilité des connexions pas auditable (auth.log + syslog)

Il faut utiliser comme nous avons fait dans les TP des logins ssh avec les utilisateurs humain réels des machines et des clés ssh. C’est à dire le même modèle d’authentification que l’administration traditionnelle.

**Les autres modes de connexion**

Le mode de connexion par défaut de Ansible est SSH cependant il est possible d’utiliser de nombreux autres modes de connexion spécifiques :

* Pour afficher la liste des plugins disponible lancez ansible-doc -t connection -l.
* Une autre connexion courante est ansible\_connection=local qui permet de configurer la machine locale sans avoir besoin d’installer un serveur ssh.
* Citons également les connexions ansible\_connexion=docker et ansible\_connexion=lxd pour configurer des conteneurs linux ainsi que ansible\_connexion= pour les serveurs windows
* Les questions de sécurités de la connexion se posent bien sur différemment selon le mode de connexion utilisés (port, authentification, etc.)
* Pour débugger les connexions et diagnotiquer leur sécurité on peut afficher les détails de chaque connection ansible avec le mode de verbosité maximal (network) en utilisant le paramètre -vvvv.

**Variables et secrets**

Le principal risque de sécurité lié à Ansible comme avec Docker et l’IaC en général consiste à laisser trainer des secrets (mot de passe, identités de clients, api token, secret de chiffrement / migration etc.) dans le code ou sur les serveurs (moins problématique).

Attention : les dépôt git peuvent cacher des secrets dans leur historique. Pour chercher et nettoyer un secret dans un dépôt l’outil le plus courant est BFG : <https://rtyley.github.io/bfg-repo-cleaner/>

**Désactiver le logging des informations sensibles**

Ansible propose une directive no\_log: yes qui permet de désactiver l’affichage des valeurs d’entrée et de sortie d’une tâche.

Il est ainsi possible de limiter la prolifération de données sensibles.

**Ansible vault**

Pour éviter de divulguer des secrets par inadvertance, il est possible de gérer les secrets avec des variables d’environnement ou avec un fichier variable externe au projet qui échappera au versionning git, mais ce n’est pas idéal.

Ansible intègre un trousseau de secret appelé , **Ansible Vault** permet de chiffrer des valeurs **variables par variables** ou des **fichiers complets**. Les valeurs stockées dans le trousseaux sont déchiffrée à l’exécution après dévérouillage du trousseau.

* ansible-vault create /var/secrets.yml
* ansible-vault edit /var/secrets.yml ouvre $EDITOR pour changer le fichier de variables.
* ansible-vault encrypt\_file /vars/secrets.yml pour chiffrer un fichier existant
* ansible-vault encrypt\_string monmotdepasse permet de chiffrer une valeur avec un mot de passe. le résultat peut être ensuite collé dans un fichier de variables par ailleurs en clair.

Pour déchiffrer il est ensuite nécessaire d’ajouter l’option --ask-vault-pass au moment de l’exécution de ansible ou ansible-playbook

Il existe également un mode pour gérer plusieurs mots de passe associés à des identifiants.

**Ansible dans le cloud**

L’automatisation Ansible fait d’autant plus sens dans un environnement d’infrastructures dynamique:

* L’agrandissement horizontal implique de résinstaller régulièrement des machines identiques
* L’automatisation et la gestion des configurations permet de mieux contrôler des environnements de plus en plus complexes.

Il existe de nombreuses solutions pour intégrer Ansible avec les principaux providers de cloud (modules ansible, plugins d’API, intégration avec d’autre outils d’IaC Cloud comme Terraform ou Cloudformation).

**Inventaires dynamiques**

Les inventaires que nous avons utilisés jusqu’ici implique d’affecter à la main les adresses IP des différents noeuds de notre infrastructure. Cela devient vite ingérable.

La solution ansible pour le pas gérer les IP et les groupes à la main est appelée inventaire dynamique ou inventory plugin. Un inventaire dynamique est simplement un programme qui renvoie un JSON respectant le format d’inventaire JSON ansible, généralement en contactant l’api du cloud provider ou une autre source.

$ ./inventory\_terraform.py

{

"\_meta": {

"hostvars": {

"balancer0": {

"ansible\_host": "104.248.194.100"

},

"balancer1": {

"ansible\_host": "104.248.204.222"

},

"awx0": {

"ansible\_host": "104.248.204.202"

},

"appserver0": {

"ansible\_host": "104.248.202.47"

}

}

},

"all": {

"children": [],

"hosts": [

"appserver0",

"awx0",

"balancer0",

"balancer1"

],

"vars": {}

},

"appservers": {

"children": [],

"hosts": [

"balancer0",

"balancer1"

],

"vars": {}

},

"awxnodes": {

"children": [],

"hosts": [

"awx0"

],

"vars": {}

},

"balancers": {

"children": [],

"hosts": [

"appserver0"

],

"vars": {}

}

}%

On peut ensuite appeler ansible-playbook en utilisant ce programme plutôt qu’un fichier statique d’inventaire: ansible-playbook -i inventory\_terraform.py configuration.yml