Ansible

Automatiser la gestion des serveurs

Version **23.2**
Auteurs : T. Constans, F. Micaux, N. Quiniou-Briand
formation@actilis.net

Table des matières

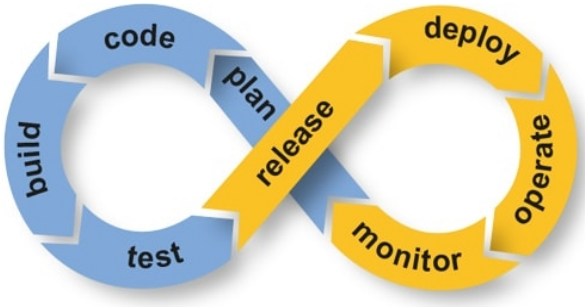
1- Présentation et positionnement

1.1- Devops & IaC : le code source de l'infrastructure

1.1.1- DevOps

Approche visant à rapprocher les équipes de développeurs (Dev) et celles d'administrateurs système (Ops) pour optimiser le cycle de vie des applications :

Travailler ensemble pour connaître / éviter un obstacle opérationnel.



Fluidifier le cycle de vie des applications :

⇒ Maîtriser les différences Dev / Prod,
⇒ Mettre en production plus fréquemment,
⇒ Permettre un retour arrière facile,
⇒ Gérer la montée en charge facilement,

Quel moyens ?

Automatiser les déploiements,
Rendre programmable l'administration système : Puppet, Chef, Ansible…
Versionner l'ensemble : Git… y compris l'infra !

1.1.2- Infrastructure as Code

Infrastructure que l'on peut mettre à disposition et administrer grâce à du code,
⇒ donc automatiquement et rapidement.

Ressemble à l'approche par shell-scripts, mais va au-delà :

⇒ S'appuie souvent sur un langage de plus haut niveau :
Polyvalence, indépendance de l'OS ou de sa version,

⇒ Peut être rejoué (idem-potence)… intégration continue…
On **décrit** une cible (recette) plutôt que de la programmer,

On s'appuie sur un processus de déploiement homogène pour construire (reconstruire) un environnement de dev / test similaire (*identique…*) à ceux d'intégration ou de production.

On considère le code de l'infrastructure… comme du code source…
À concevoir, à développer, à tester,
à versionner, à documenter…

C'est bel et bien la manière d'administrer qui a évolué…

1.2- Puppet, Chef, Saltstack... Ansible

...décrire un état dans lequel on souhaite voir les machines...

CfEngine : (1993, C) Un *master* gère des *agents* pour lesquels on décrit un état cible (*une promesse*). Un peu "Dev Friendly", son apprentissage est fastidieux.

Puppet : (2005, Ruby, C++ et Clojure) Un *master* et *des agents* (PULL) qui s'y connectent régulièrement et obtiennent un *manifest* (*une recette*) qu'ils appliquent localement. Son langage "Puppet DSL" est plutôt "Ops Friendly", et son apprentissage plus facile que celui de CfEngine.

Chef : (2009, Ruby & Erlang) Un *serveur* et des *clients* (PULL). Plutôt "Dev-Friendly" et procédural. Son langage s'appuie *directement* sur Ruby, et offre donc plus d'ouverture que Puppet.

SaltStack : (2011, PUSH, Python) initialement un outil d'exécution de code à distance. Un *master*, et des *minions* qui s'y connectent et reçoivent des messages sur un bus. Langage un peu "Ops Friendly", assez facile à apprendre.

Ansible : (2012, PUSH, Python) **Pas d'agent à installer,** il s'appuie sur **SSH**. Simple d'approche, il combine déploiement, gestion de configuration, exécution / orchestration de tâches d'administration système.

1.3- Fonctionnement d'Ansible

Ansible est un (utilise un) langage de description en YAML proche du langage naturel.

Ansible se connecte aux hôtes, y pousse de petits programmes (modules) qu'il exécute et supprime ensuite.

Les modules peuvent être hébergés sur n'importe quelle machine.

Il n'y a **pas de serveur central**, pas d'infrastructure spécifique (PKI, etc.), juste un "poste de contrôle" disposant d'un terminal et d'un éditeur de texte.

Toute machine[1] sur laquelle "**ansible**" est installé peut en piloter d'autres :

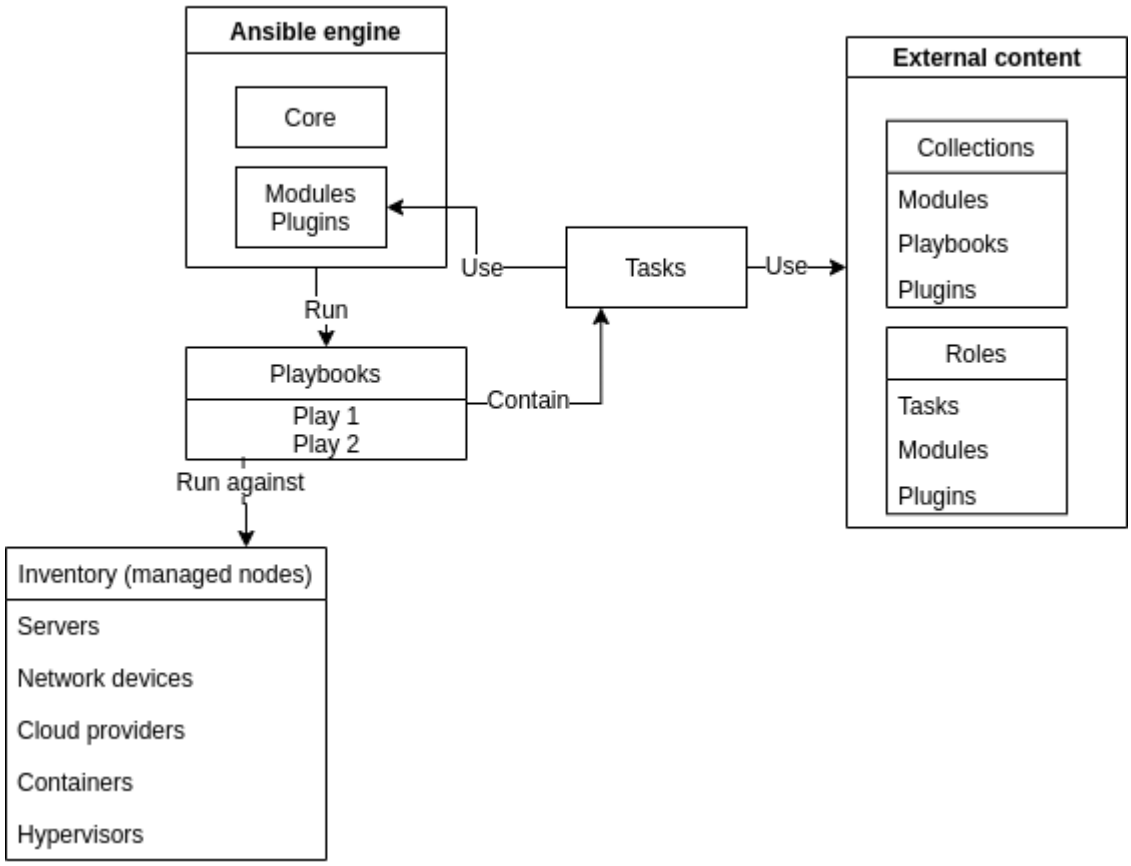
- Linux & Unix,
- Windows,
- Certains équipements réseau

Les machines Linux à piloter doivent juste disposer de :

- SSH,
- Python.

1 Ansible ne s'installe pas sur une machine Windows

Ansible architecture



1.4- Architecture de Ansible

1.4.1- Communication avec les cibles

Ansible s’appuie sur ssh et suppose donc qu'on utilise une authentification par clé.

L'utilisateur cible est peu important, pourvu qu'un dispositif d'élévation de privilèges soit utilisé si on a besoin des privilèges de root,

On peut donc :

utiliser le compte root des hôtes cibles,

utiliser un autre compte et configurer sudo.

1.4.2- Fichier de configuration

On peut réaliser certains paramétrages dans un **fichier de configuration**.

Celui par défaut est **/etc/ansible/ansible.cfg**.

1.4.3- Inventaire

Ansible nécessite un fichier d'inventaire qui référence les machines gérées.

Il ressemble à ce genre de chose :

```
[webservers]
www1.example.com
www2.example.com

[dbservers]
db0.example.com
db1.example.com
```

Ce fichier au format "ini", définit
- des hôtes,
- des groupes d'hôtes,
- des groupes de groupes...

On peut affecter des variables aux hôtes ou groupes d'hôtes
- dans ce fichier d'inventaire,
  - ou dans d'autres fichiers placés dans des sous-répertoires.

Il existe des plugins pour rendre dynamique l'inventaire en collectant des informations auprès de sources comme EC2, Rackspace, OpenStack.

1.4.4- Modules

Ce sont des petits bouts de code souvent écrits en Python.

Chacun est dédié à une action précise :

- Créer ou modifier un fichier,
- Gérer un compte utilisateur (système, base de données, ...),
- Manipuler des packages des distributions (apt, yum / dnf, pacman, zypper),
- Exécuter une commande, un script shell...
- etc.

Il existe un peu plus de 3000 modules avec Ansible 2.9 : ***ansible-doc --list***

La documentation :
http://docs.ansible.com/ansible/latest/modules_by_category.html

1.4.5- Playbooks

Au travers des **playbooks**, Ansible est un langage de description simple et puissant qui permet d'orchestrer des branches entières de l'infrastructure.

Un playbook est un fichier YAML qu'on peut écrire pour un usage unique ou récurrent.

Il définit **une suite de tâches** à réaliser sur une cible.

On y retrouve :

- une ou des **cible**(s) (hôte ou groupe d'hôtes),
- des **variables**,
- des **tâches** à réaliser séquentiellement sur les cibles (en parallèle).

Un playbook doit être **idempotent** (lançable plusieurs fois)

On exécute un playbook grâce à la commande **ansible-playbook**.

1.4.6- Tasks

Une tâche est une action atomique décrite dans un playbook.

Elle peut utiliser :

- des modules,
- des variables,
- des boucles,
- des opérateurs, etc.

1.4.7- Rôles

Un rôle est un **playbook** structuré pour répondre à un besoin plus global et constitue un niveau d'abstraction et d'orchestration supérieur.

On s'en sert pour l'installation ou le paramétrage d'un système complet (au sens application N tiers).

On trouve des rôles sur le Hub Ansible.

Voir **ansible-galaxy**.

Un rôle s'utilise dans un playbook dans lequel on précise quel rôle on souhaite mettre en œuvre.

On exécute donc un rôle comme un playbook, avec la commande **ansible-playbook**.

1.4.8- Collections

Les collections sont des dossiers qui permettent de partager du code Ansible (rôles, modules et plugins).

Cette fonctionnalité a été ajoutée en version 2.8 et va servir de base à la refactorisation du code d'Ansible.

Les collections s'installent avec la commande **ansible-galaxy.**

2- Prise en main d'Ansible

2.1- Installation

2.1.1- Pré-requis

Sur les nœuds gérés par Ansible :

une connexion ssh, pas forcément en root,
Python 2 (2.7 ou supérieur) ou **Python** 3 (3.5 ou supérieur)

Sur le "contrôleur" : un système à jour.

2.1.2- Installer Ansible

Utiliser le package fourni par la distribution ou un dépôt tiers,

Debian :

- disponible dans la distribution, parfois dans les backports
- dépôt Debian fournit par le projet Ansible

Sur RHEL / CentOS 7 : EPEL fournit une version "à jour".

ou… utiliser **pip3** (package **python3-pip**), pour disposer de la dernière version.

Nécessite gcc + libpython3-dev sous Debian

Installation sur CentOS 8 avec le package standard :

```
dnf -y install ansible
```

Installation sur CentOS 8 avec le package fourni par EPEL :

```
dnf -y install epel-release && dnf -y upgrade
dnf -y install ansible
```

Installation sur CentOS 8 avec pip3 :

```
dnf -y install python3-pip
pip3 install ansible
```

Installation avec pip3 sur Debian 10 (pour avoir la dernière version) :

```
apt update && apt install python3-pip
pip3 install ansible
```

2.2- Les commandes ansible*

Le package installe un ensemble de commandes (dans /usr/bin) :

ansible : la commande principale d'exécution d'une tâche simple.

ansible-config : gestion de la configuration d'ansible.

ansible-console : console interactive (REPL) pour exécuter des tâches Ansible.

ansible-doc : lister les modules, visualiser la documentation liée aux modules.

ansible-galaxy : gestion des rôles et des collections (dans un but de partage de code).

ansible-inventory : gestion (visualisation) de l'inventaire.

ansible-playbook : exécution des playbooks.

ansible-pull : permet d'envisager une approche "pull" au lieu de "push".

ansible-vault : gestion de coffre fort (chiffrement / déchiffrement de données).

2.3- Format de la commande ansible

ansible <host-pattern> [-m module_name] [-a args] [options]

On peut spécifier un module par "**-m**"... Le module "**ping**" n'exige pas d'argument :

```
▶ ansible localhost -m ping
  [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Le **module** par défaut est "**command**" : exécuter une commande arbitraire.
Pour ce module, il faut citer la commande en argument (**-a** "ma commande")

```
▶ ansible localhost  -a "echo coucou"       # ou  ansible localhost -m command -a "echo coucou"
  [WARNING]: provided hosts list is empty, only localhost is available

localhost | CHANGED | rc=0 >>
coucou
```

On doit citer **l'hôte ou les hôtes** sur lesquels portent les actions.
Ansible utilise un inventaire, sans lequel seul "*localhost*" est admis...

2.4- L'inventaire

Ansible travaille sur un ensemble de machines, à partir d'une liste qui constitue l'inventaire d'ansible.

Par défaut, c'est **/etc/ansible/hosts**.

On peut disposer de plusieurs inventaires et préciser celui à utiliser :

grâce au fichier de configuration (directive "inventory"),

grâce à la variable **ANSIBLE_INVENTORY**,

grâce à l'option "**-i**" sur la ligne de commande.

Il est possible de configurer cet inventaire, voire d'utiliser des inventaires dynamiques.

Le tout est que cet inventaire soit au format **INI** ou **YAML** utilisé par Ansible.

Si la valeur de la directive "inventory" est un dossier, Ansible cherchera à utiliser comme source d'inventaire tous les fichiers de ce dossier.

2.4.1- Inventaire simple

C'est une liste de machines, dans laquelle elles peuvent être groupées :

```
# les machines sans groupe sont à
# mettre en début d'inventaire
192.168.56.101
[formation]
10.1.1.100:22
10.1.1.1:22
10.1.1.2:22
10.1.1.3:22
10.1.1.4:22
10.1.1.5:22
10.1.1.6:22

[tmp]
```

ou encore ...

```
192.168.56.101
[formation]
10.1.1.100:22
10.1.1.[1:6]:22

[tmp]
192.168.10.160
```

2.4.2- Groupes de groupes

On peut définir des groupes de groupes en utilisant la directive **:children** :

```
[apache_servers]
...

[mysql_servers]
...

[lamp_servers:children]
mysql_servers
apache_servers
```

2.4.3- Variables

On peut utiliser des **variables** dans l'inventaire.

Elles s'appliquent à des machines ou à des groupes de machines.

```
[apache_servers]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

Grâce aux groupes, on peut factoriser les variables :

```
[apache_servers]
host1
host2

[apache_servers:vars]
ntp_server=ntp.example.com
proxy=proxy.example.com
```

Remarque : On peut aussi définir les variables dans des fichiers texte à créer dans des sous-répertoires nommés "**group_vars**" ou "**host_vars**" (voir page 86).

Documentation : http://docs.ansible.com/ansible/latest/intro_inventory.html

2.4.4- Connexion et identité

Dans l'exemple suivant, on suppose que l'utilisateur local est "fmicaux", et peut se connecter uniquement en tant que "root" sur la machine 172.17.1.70.

On crée l'**inventaire** suivant :

```
▶ cat >> test
[formation]
172.17.1.70
<CTRL-D>
```

Pour le test, on utilise le module "**ping**"
on précise l'inventaire par "**-i**",
et les cibles par "**all**", un nom de groupe, ou d'hôte :

```
▶ ansible -i test    all -m ping
✖ 172.17.1.70 | UNREACHABLE! => {
✖     "changed": false,
✖     "msg": "Failed to connect to the host via ssh: Warning: Permanently added '172.17.1.70' (ECDSA)
  to the list of known hosts.\r\nPermission denied (publickey,gssapi-keyex,gssapi-with-mic,password).\
  r\n",
✖     "unreachable": true
```

Le problème : **Ansible** s'identifie en tant que "fmicaux", qui n'existe pas sur la machine cible et on ne fournit pas non plus de moyen d'authentification.

La solution : spécifier l'identité à utiliser pour la connexion ssh.

Deux méthodes différentes sont possibles :

Par l'option "**-u**"... et **-k** qui permet de saisir le mot de passe associé.

```
▶ ansible -i test    all -m ping -u root -k
SSH password:
172.17.1.70 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Ou bien grâce à une variable dans l'inventaire... et **-k** pour le mot de passe.

```
▶ cat test
[formation]
172.17.1.70  ansible_ssh_user=root
```

```
▶ ansible -i test    all -m ping -k
SSH password:
172.17.1.70 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

2.5- Le fichier de configuration

Le premier qui est trouvé, dans l'ordre de recherche suivant, est utilisé, tous les autres sont ignorés :

1/ Emplacement spécifié par la variable **ANSIBLE_CONFIG**,

2/ ${PWD}/**ansible.cfg**,

3/ ${HOME}/**.ansible.cfg**,

4/ **/etc/ansible/ansible.cfg**.

Son format est similaire aux fichiers ".ini" :
des sections : [...]
des affectations de variables

Exemple :

```
▶ cat ansible.cfg
[defaults]
inventory = ./hosts
```

Documentation : http://docs.ansible.com/ansible/latest/intro_configuration.html
Principales directives en page suivante.

Directive	Description	Valeur "idéale" / Remarque
forks	Nb de machines cibles traitées en parallèle.	Très supérieure à la valeur par défaut pour de meilleures performances
ask_pass	Équivalent de l'option -k (ansible nous demandera systématiquement le mot de passe de connexion)	True si on n'utilise pas l'authentification par clé ssh.
host_key_checking	Contrôle si l'on vérifie ou pas les clés d'hôtes ssh.	False pour éviter les confirmations lors d'une 1ère connexion.
remote_user	Identité utilisée pour les connexions sur les machines cibles (équivalent de l'option -u)	À définir si l'on a un compte unique dédié aux opérations ansible
become	Contrôle l'utilisation ou pas d'un mécanisme d'élévation de privilèges	True si l'on se connecte avec un compte non privilégié (ce qui est conseillé)
become_ask_pass	Équivalent de l'option -K . Si activé, ansible nous demande systématiquement le mot de passe become/sudo	True si un mot de passe est nécessaire pour l'élévation de privilèges sur la cible.
retry_files_enabled	Contrôle la création des fichiers retry contenant les machines sur lesquelles un playbook a échoué	*False* en mode de mise au point des playbook, puis True une fois en production
gathering	Contrôle la récupération des *facts.*	*Smart* si on utilise les facts, *explicit* sinon. Voir aussi "mise en cache des facts"
callbacks_enabled	Liste des plugins appelés à des moments clés de l'exécution d'un playbook	Les plugins ansible.posix.*timer* et ansible.posix.*profile_tasks* permettent d'avoir un récapitulatif des temps d'exécution des tâches d'un playbook.
log_path	Spécifie un fichier de log dans lequel le résultat de l'exécution d'ansible sera collecté.	Ce fichier doit pouvoir être modifié par l'opérateur ansible, et doit être régulièrement archivé.
pipelining	Améliore encore le temps d'exécution d'un play.	True si possible. Nécessite une configuration sudo (requiretty désactivé) sur les cibles.

2.6- Configuration des nœuds

Il n'y a **RIEN** à faire sur les nœuds... à part...

...si les connexions "root" sont interdites sur les nœuds...

On doit alors créer un compte utilisateur dédié à ansible sur les nœuds...

Et gérer l'authentification ssh vers ce compte.

Dans les faits, beaucoup d'actions nécessitent des privilèges...

On utilisera alors l'option "**--become**" dans ces commandes ansible.

À condition que "sudo" soit configuré sur les nœuds !

On doit alors configurer sudo sur les nœuds.

3- Les commandes Ad-Hoc

3.1- Parallélisme et commandes shell

Une commande Ad-Hoc est quelque chose que l'on souhaite faire rapidement, sans passer par la mise en place d'un playbook ou d'un rôle complet.

La commande Ad-Hoc est au playbook ce que le one-liner est au script shell. On privilégie la rapidité de mise en œuvre à la réutilisabilité ultérieure.

Avec la commande **ansible** il est possible d'exécuter une commande, mais également n'importe quel module ansible, sur tout ou partie de notre inventaire de machines.

Pour fonctionner, il est nécessaire que ssh et/ou sudo soient correctement configurés. On pourra utiliser les options **--ask-pass**, **--ask-become-pass**, **--user**, si nécessaire.

option	Signification et cas d'utilisation
--user / -u	Compte utilisateur à utiliser pour initier la connexion ssh
--ask-pass / -k	Ansible demandera le mot de passe de connexion (dans le cas où l'authentification par clé n'est pas en place)
--become	Utilisation de l'escalade de privilége (en général sudo)
--ask-become-pass / -K	Ansible demandera le mot de passe sudo

Le résultat est coloré afin de mettre en valeur le statut de la commande :

```
SUCCESS : la commande s'est déroulée correctement mais n'a pas entraîné de modification sur la cible.
```

```
CHANGED : la commande s'est déroulée correctement. Des modifications ont été apportées à la machine cible.
```

```
FAILED : la commande ne s'est pas exécutée correctement.
```

Le module "**command**" retournera toujours un résultat `CHANGED` sur un succès (non idempotent).

3.2- Remarques concernant le nom des modules

Il est désormais recommandé de référencer les modules en utilisant leur Full Qualified Collection Name, à la fois via les commandes AdHoc et lors de l'écriture de playbooks / rôles

Les référencer par leur nom court sera à terme déprécié.

Le FQCN d'un module est indiqué en haut à gauche de la page de doc (ansible-doc).

3.3- Commandes shell

Le principal cas d'utilisation des commandes Ad-Hoc est l'exécution d'une commande sur un ensemble de machines.

Ainsi, la commande suivante va lancer la commande *uptime* sur toutes les machines du groupe **test**, défini dans le fichier d'inventaire :

```
▶ ansible test -a "uptime"
✖ machine1 | UNREACHABLE! => {
✖     "changed": false,
✖     "msg": "Failed to connect to the host via ssh: ssh: connect to host machine1 port 22:
  Connection refused\r\n",
✖     "unreachable": true
✖ }
✔
✔ machine2 | CHANGED   | rc=0 >>
✔  09:50:26 up 109 days, 17:11,  2 users,  load average: 0,58, 0,43, 0,47
```

Note : ici, le module n'est pas précisé (pas d'option -m) :
c'est donc le module par défaut[2] ***command*** qui est utilisé.
on lui passe la commande voulue en argument (**-a**).

2 directive de configuration *module_name*

3.4- Transferts de fichiers

Cette commande va copier le fichier /etc/vimrc depuis la machine locale vers les machines cibles.

```
▶ ansible test -m ansible.builtin.copy -a 'src=/etc/vimrc dest=/etc/vimrc owner=root group=root
mode=0644'
✔ 100.0.0.100 | CHANGED => {
✔     "changed": true,
✔     "checksum": "bffdfda911f5ba0719de9156e6c23d9582b50289",
✔     "failed": false,
✔     "gid": 0,
✔     "group": "root",
✔     "mode": "0644",
✔     "owner": "root",
✔     "path": "/etc/vimrc",
✔     "secontext": "system_u:object_r:etc_t:s0",
✔     "size": 1982,
✔     "state": "file",
✔     "uid": 0
✔ }
```

On utilise ici le module ***copy***.

C'est un moyen rapide de déployer un fichier de configuration, un script vers un ensemble de machine.

3.5- Les packages avec yum/apt

Plutôt que d'utiliser les commandes shell pour gérer les paquets, il est préférable d'utiliser le module correspondant au gestionnaire de paquet utilisé (ici ***yum***).

```
▶ ansible test -m ansible.builtin.yum -a "name=vim state=latest"
100.0.0.100 | SUCCESS => {
    "changed": false,
    "failed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "All packages providing vim are up to date",
        ""
    ]
}
▶ ansible  test -m ansible.builtin.yum -a "name=nano state=absent"
✔ 100.0.0.100 | CHANGED => {
✔     "changed": true,
✔     "failed": false,
✔     "msg": "",
✔     "rc": 0,
✔     "results : [ .. ]
✔ }
▶ ansible test -m ansible.builtin.yum -a "name=* state=latest"
100.0.0.100 | SUCCESS => {
    "changed": false,
    "failed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "Nothing to do here, all packages are up to date"
```

3.6- Les packages - indépendamment de la plate-forme

Le module *package* permet de s'affranchir de la distribution (et donc du gestionnaire de packages) installée sur la cible.

Il s'utilise de la même manière que le module yum ou apt.

C'est un bon candidat si l'on souhaite travailler sur des cibles hétérogènes.

Inconvénient[3] : souvent les paquets n'ont pas le même nom d'une distribution à l'autre.

```
▶ ansible test -m ansible.builtin.package -a "name=vim state=latest"
100.0.0.100 | SUCCESS => {
    "changed": false,
    "failed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "All packages providing vim are up to date",
        ""
    ]
}
```

3 non insurmontable, on le verra plus tard

3.7- Les utilisateurs et les groupes

Ici on utilise le module group pour s'assurer que le groupe "example" existe.

```
▶ ansible test -m ansible.builtin.group -a "name=example state=present"
✔ 100.0.0.100 | CHANGED => {
✔     "changed": true,
✔     "failed": false,
✔     "gid": 1003,
✔     "name": "example",
✔     "state": "present",
✔     "system": false
✔ }
```

Le module ***user*** permet de gérer utilisateurs sur un ensemble de machines.

```
▶ ansible test -m ansible.builtin.user -a "name=tconstans groups=example,wheel home=/home/tom
  password={{ '123Soleil'|password_hash('sha512') }}"
✔ 100.0.0.100 | CHANGED => {
✔     "append": false,
✔     "changed": true,
✔     "comment": "",
✔     "failed": false,
✔     "group": 1002,
✔     "groups": "example,wheel",
✔     "home": "/home/tom",
✔     "move_home": false,
✔     "name": "tconstans",
✔     "password": "NOT_LOGGING_PASSWORD",
✔     "shell": "/bin/bash",
✔     "state": "present",
✔     "uid": 1002
✔ }
```

La construction "{{ '123Soleil'|password_hash('sha512') }}" est un filtre[4] permettant de générer un hash à partir d'un chaîne de caractère. Les précautions d'usage vis-à-vis de l'utilisation d'un mot de passe en clair en ligne de commande s'imposent.

4 https://docs.ansible.com/ansible/latest/playbooks_filters.html#hash-filters

3.8- Les services

Les services peuvent être gérés par le module ***service***, fonctionnant sur les machines utilisant upstart, systemd, BSD init, sysV init, openRC et Solaris SMF.

Le module ***systemd*** est réservé aux distributions Linux récentes utilisant ce nouveau mode de gestion des services.

```
▶ ansible test -m ansible.builtin.service -a "name=crond state=restarted"
✔ 100.0.0.100 | CHANGED => {
✔     "changed": true,
✔     "name": "crond",
✔     "state": "started",
✔     "status": {
✔ [..]
▶ ansible test -m ansible.builtin.service -a "name=ntpd enabled=false"
✔ 100.0.0.100 | CHANGED => {
✔     "changed": true,
✔     "enabled": false,
✔     "name": "ntpd",
✔     "status": {
✔         "ActiveEnterTimestamp": "mar. 2017-11-07 09:13:35 CET",
✔ [..]
▶ ansible test -m ansible.builtin.systemd -a "name=ntpd state=stopped enabled=false"
100.0.0.100 | CHANGED => {
    "changed": false,
    "enabled": false,
    "name": "ntpd",
    "state": "stopped",
    "status": {
[..]
```

3.9- Lineinfile

Le module lineinfile permet de s'assurer de la présence, ou de l'absence d'une ligne dans une fichier.
Il peut même être utilisé pour réaliser des "rechercher et remplacer" dans un fichier.
Un module idéal pour affiner (ou détruire) ses fichiers de configuration !

```
ansible cibles -m ansible.builtin.lineinefile -a 'path=/etc/motd create=yes line="machine gérée par ansible"'
```

Voir aussi cette page pour plus de cas d'utilisation: https://www.middlewareinventory.com/blog/ansible-lineinfile-examples/

4- Les playbooks

4.1- Syntaxe YAML

Ref: https://docs.ansible.com/ansible/latest/YAMLSyntax.html

YAML est un langage qui à l'avantage d'être à la fois exploitable informatiquement et lisible / éditable facilement par tout un chacun.

Un fichier YAML commence systématiquement par **---**

Concernant Ansible, tout fichier commence généralement par une liste de paires :
clé / valeurs

Les membres d'une même liste sont déterminés par:
1. leur niveau d'indentation
2. le fait qu'il commence par "**-** "

Dans l'exemple suivant, nous avons une liste "repas" composée de 4 éléments.
"sucre: yes" est un argument de l'élément "dessert"

```
---
repas:
 - entrée
 - plat
 - fromage
 - dessert
   sucre: yes
```

4.2- Introduction aux playbooks

Ref:
https://docs.ansible.com/ansible/latest/playbooks_intro.html#intro-to-playbooks

Les ***playbooks*** constituent un outil bien plus puissant que les commandes Ad-Hoc.

Ils permettent

d'organiser l'exécution de tâches,

de définir leurs paramètres d'exécution,

et de spécifier les machines sur lesquelles ces tâches vont s'exécuter.

Un ***playbook*** est un ensemble de *plays*, elles-même constituées de tâches.

Un ***playbook*** correspond en général à une fonction / service.

4.3- Définition des tâches / plays

Un **play** est constitué:

1. d'un nom
2. d'une liste de machines cibles
3. d'une liste de tâches

```
---
- name:  installation et configuration de vim
  hosts: test

  tasks:
  - name: return true
    ansible.builtin.command: /bin/true

  - name: vim installation - CentOS
    ansible.builtin.yum:
      name: vim-enhanced
      state: latest

  - name: vim configuration
    ansible.builtin.copy:
      src: "/etc/vimrc"
      dest: "/etc/vimrc"
      owner: root
      group: root
      mode: 644
```

4.4- Exécuter un playbook

L'exécution d'un playbook se fait via la commande **ansible-playbook**

```
▶ ansible-playbook vim.yml
PLAY [installation et configuration de vim]

TASK [Gathering Facts]
ok: [100.0.0.100]

TASK [return true]
changed: [100.0.0.100]

TASK [vim installation - CentOS]
changed: [100.0.0.100]

TASK [vim configuration]
changed: [100.0.0.100]

PLAY RECAP
100.0.0.100                : ok=4    changed=3    unreachable=0    failed=0
```

L'option **-v** (que l'on peut répéter) augmente la verbosité de la commande.

L'option **--syntax-check** permet de vérifier la validité de nos playbooks.

▶ **Remarque** : en installant le paquet *cowsay* vous aurez une surprise !

4.5- Tester un playbook en dry-run

L'option **--check** permet de lancer une simulation. Les modules le supportant indiqueront ce qu'ils vont faire, sans le faire réellement. Les modules ne supportant pas ce mode se contenteront de "faire semblant".

```
▶ ansible-playbook vim.yml --check -v
PLAY [installation et configuration de vim]

TASK [Gathering Facts]
ok: [100.0.0.100]

TASK [return true]
skipping: [100.0.0.100] => {"changed": false, "msg": "skipped, running in check mode"}

TASK [vim installation - CentOS]
changed: [100.0.0.100] => {"changed": true, "changes": {"installed": ["vim-enhanced"], "updated": []},
"msg": "", "rc": 0, "results": []}

TASK [vim configuration]
changed: [100.0.0.100] => {"changed": true}
```

À partir de la version 2.2, on peut forcer une tâche à s'exécuter en mode de vérification via la directive **check_mode: yes**

Il est possible de déterminer si on est en mode simulation en testant la variable **ansible_check_mode**.

4.6- Les modules et les tâches

Ref: https://docs.ansible.com/ansible/latest/modules_intro.html
https://docs.ansible.com/ansible/latest/playbooks_keywords.html

C'est à travers les **modules** qu'Ansible fait le travail qu'on lui demande. Ils sont utilisés dans la définition d'une tâche ou peuvent être directement utilisés via une commande Ad-Hoc.

```
- name: mise à jour des serveurs
  ansible.builtin.yum:
   name: '*'
   state: latest
```

```
▶ ansible test -m ansible.builtin.yum -a "name=* state=latest"
```

La liste des modules peut être obtenue via la commande **ansible-doc -l**

```
▶ ansible-doc -l
a10_server              Manage A10 Networks AX/SoftAX/Thunder/vThunder devices' server object.
a10_server_axapi3       Manage A10 Networks AX/SoftAX/Thunder/vThunder devices
a10_service_group       Manage A10 Networks AX/SoftAX/Thunder/vThunder devices' service groups.
a10_virtual_server      Manage A10 Networks AX/SoftAX/Thunder/vThunder devices' virtual servers.
accelerate              Enable accelerated mode on remote node
aci_aep                 Manage attachable Access Entity Profile (AEP) on Cisco ACI fabrics
(infra:AttEntityP)
aci_ap                  Manage top level Application Profile (AP) objects on Cisco ACI fabrics (fv:Ap)
...
yum                     Manages packages with the `yum' package manager
yum_repository          Add or remove YUM repositories
```

La même commande permet d'obtenir la documentation,

```
▶ ansible-doc yum
...
= name
        Package name, or package specifier with version, like `name-1.0'. If a
        previous version is specified, the task also needs to turn `allow_downgrade'
        on. See the `allow_downgrade' documentation for caveats with downgrading
        packages. When using state=latest, this can be '*' which means run `yum -y
        update'.  You can also pass a url or a local path to a rpm file (using
        state=present). To operate on several packages this can accept a comma
        separated list of packages or (as of 2.0) a list of packages.
        (Aliases: pkg)[Default: None]
...
```

Le préfixe **=** désigne un paramètre **obligatoire**.

On notera également la notion d'alias, de valeur par défaut et de la version d'Ansible à partir de laquelle le paramètre a été ajouté.

La documentation est également disponible en ligne :
https://docs.ansible.com/ansible/latest/modules_by_category.html

Valeur de retour

Ref: https://docs.ansible.com/ansible/latest/common_return_values.html

Chaque module renvoie un ensemble de valeurs, au format JSON, que l'on peut récupérer dans une variable ou simplement afficher (en utilisant l'option **-v**)

```
< TASK [mysql : install mysql community repo] >

ok: [100.0.0.100] => {"changed": false, "failed": false, "msg": "", "rc": 0, "results": [""]}
```

On distingue :

les valeurs standard renvoyées systématiquement par tous les modules :
changed, failed, stdout, ...

et les valeurs spécifiques à chaque module :
que l'on retrouve dans la documentation du module.

On peut utiliser ces valeurs, pour, par exemple n'exécuter la prochaine tâche que si la précédente a provoqué un changement, que l'on peut vérifier en examinant la valeur de retour de **changed**.

Exemple :

```
- hosts: cibles

  tasks:
    - name: print PATH variable and register values
      ansible.builtin.shell: echo $PATH
      register: shell_result
      ignore_errors: True

    # Using -v will do the same but on one line
    - name: show register values
      ansible.builtin.debug:
        var: shell_result
```

Résultat

```
TASK [show register value]

ok: [node1-c7] => {
    "shell_result": {
        "changed": true,
        "cmd": "echo $PATH",
        [..]
        "failed": false,
        "rc": 0,
        "stderr": "",
        "stderr_lines": [],
        "stdout": "/sbin:/bin:/usr/sbin:/usr/bin",
        "stdout_lines": [
            "/sbin:/bin:/usr/sbin:/usr/bin"
        ]
    }
}
```

Niveau de support des modules

```
AUTHOR: Stephen Fromm (@sfromm)
        METADATA:
          status:
          - stableinterface
          supported_by: core
```

Le *status* et le niveau de support (*supported_by*) de chaque module sont clairement définis dans la documentation.

Les modules **core** sont intégrés à Ansible.

Les modules **network** sont développés et supportés par l'équipe Network d'Ansible.

Les modules **certified** sont maintenus par les partenaires du projet Ansible.

Les modules **community** sont développés et fournis par la communauté. Leur stabilité n'est pas garantie, mais ils ont été considérés de qualité suffisante pour être inclus dans l'installation de base d'Ansible.

Documentation sur le cycle de vie des modules :
https://docs.ansible.com/ansible/latest/dev_guide/module_lifecycle.html#module-lifecycle

4.6.1- Mettre au point un playbook

Démarche :

1/ Décomposer chaque étape de la procédure en une tâche simple,

2/ Identifier le module permettant de réaliser l'opération,, par ex pour l'installation et la configuration de vim :

1. installer le paquet vim-enhanced ⇒ module **yum**
2. copier le fichier de configuration vers la cible ⇒ module **copy**

3/ Identifier les paramètres des modules à utiliser (documentation),

4/ Rassembler les tâches dans un playbook, et y spécifier :

Le nom
Les machines cibles
Les informations de connexion et de compte.

5/ Lancer l'exécution du playbook avec l'option **--syntax-check** pour s'assurer de l'absence d'erreur de syntaxe.

6/ Lancer le playbook en mode simulation (option –check)

7/ Lancer le playbook sur une environnement de test, puis en production.

Exemple :

▶ **cat /etc/ansible/playbooks/vim.yml**

```
---
- name: installation et configuration de vim
  hosts: test
  tasks:
  - name: vim installation - CentOS
    ansible.builtin.vim:
      name: vim-enhanced
      state: latest

  - name: vim configuration
    ansible.builtin.copy:
      src: "/etc/vimrc"
      dest: "/etc/vimrc"
      owner: root
      group: root
      mode: 644
```

▶ **ansible-playbook /etc/ansible/playbooks/vim.yml**

```
PLAY [installation et configuration de vim]

TASK [Gathering Facts]
ok: [100.0.0.100]

TASK [vim installation - CentOS]
ok: [100.0.0.100]

TASK [vim configuration]
ok: [100.0.0.100]

PLAY RECAP
100.0.0.100                : ok=3    changed=0    unreachable=0    failed=0
```

4.7- Exécution step by step, saut de tâches

L'option **--start-at-task="nom"** permet de démarrer un playbook à partir d'une tâche particulière.

On peut également utiliser l'option **--tags** pour sélectionner les tâches associées à une ou plusieurs étiquettes.

L'option **--step** permet de demander avant chaque tâche si l'on souhaite l'exécuter.

```
▶ ansible-playbook vim.yml --step
PLAY [installation et configuration de vim]]
Perform task: TASK: Gathering Facts (N)o/(y)es/(c)ontinue:
```

4.8- Gestion des erreurs

Ref: https://docs.ansible.com/ansible/latest/playbooks_error_handling.html

Ansible exécute chaque tâche d'un playbook *en parallèle* sur les hôtes concernés (directive "hosts"). Il attend d'avoir le résultat de chaque tâche, pour chaque hôte avant de poursuivre.

Si une erreur est détectée pour un hôte, Ansible continue l'exécution du *playbook* (pas uniquement le play) sans cet hôte. Le nom de la machine en erreur sera enregistré dans un fichier *retry*.

Ce fichier pourra être utilisé via l'option **--limit @fichier.retry** de manière à rejouer le playbook uniquement sur les machines en erreur.

Il est possible d'inhiber la création de ces fichiers *retry* (au moins pendant la phase de conception des playbooks) via la directive de configuration ***retry_file_enabled*** dans **/etc/ansible/ansible.cfg**, section **[defaults]**.

Ignorer les erreurs

Si l'on souhaite poursuivre le traitement même si la tâche échoue, on peut rajouter la directive directive ***ignore_errors: yes*** dans la tâche.

Définir soi-même la gestion d'erreurs

Ansible détecte le bon fonctionnement des commandes et des modules en examinant leur code de retour.

On peut modifier ce comportement et en se basant sur d'autres éléments renvoyés par la commande ou le module.

Dans l'exemple ci-dessous, on considère que la tâche 1 échoue si le terme "FAILED" est détecté sur la sortie d'erreur et la tâche échoue si son code de retour est 0 ou >= 2.

```
- name: Fail task when the command error output prints FAILED
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"

- name: Fail task when both files are identical
  raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

4.8.1- Tout annuler en cas d'erreur

Il peut être parfois nécessaire d'interrompre l'ensemble du playbook dans le cas de l'échec d'une tâche particulièrement critique.

Pour cela on rajoute la directive ***any_errors_fatal*** dans la définition du « play ».

De manière un peu plus souple la directive ***max_fail_percentage*** permet de spécifier un pourcentage de machines en échec qui entraînera l'interruption du play s'il est atteint.

4.8.2- diff

L'option **--diff** permet d'afficher les différences entre avant et après.

Elle peut s'utiliser conjointement à l'option **--check**. Là encore, les modules doivent supporter cette option. C'est principalement utile pour les opérations manipulant les fichiers. Attention, le résultat obtenu peut être volumineux !

4.8.3- debug

À la place d'une tâche on peut utiliser le module debug dont le paramètre ***msg*** permet d'afficher des messages sur la console.

```
tasks :
- name: affiche un message
  ansible.builtin.debug:
    msg: "début du play {{ ansible_check_mode }}"

- name: vim installation - CentOS
  ansible.builtin.yum:
    name: vim-enhanced
    state: latest
```

```
▶ ansible-playbook test.yml --check

PLAY [test]
TASK [affiche un message]
ok: [100.0.0.100] => {
    "msg": "début du play  True"
}

TASK [vim installation - CentOS]
ok: [100.0.0.100]
```

Le paramètre ***var*** permet d'examiner une structure de données complète :

```
- ansible.builtin.command: /bin/true
  register: result
  ignore_errors: True

- ansible.builtin.debug:
    var: result
```

```
▶ ansible-playbook test.yml --check
PLAY [test]

TASK [test : command]
skipping: [100.0.0.100]

TASK [test : debug]
ok: [100.0.0.100] => {
    "result": {
        "changed": false,
        "failed": false,
        "msg": "skipped, running in check mode",
        "skipped": true
    }
}
```

5- Écrire du code modulaire

5.1- Handlers et Notifications

Ref: https://docs.ansible.com/ansible/latest/playbooks_intro.html#handlers-running-operations-on-change

Les ***handlers*** permettent de déclencher une action sur réception d'un évènement.

On les déclenche par un système de notification.

On peut ainsi provoquer le redémarrage d'un service uniquement si la tâche de configuration du-dit service a effectivement modifié celui-ci.

5.1.1- Définition d'un handler

Un handler est une tâche qui s'exécute à réception d'un événement.

```
handlers:
  - name: reload firewalld
    ansible.builtin.service: name="firewalld" state="reloaded"

  - name: reload apache
    ansible.builtin.service: name="httpd" state="reloaded"
```

5.1.2- Appel d'un handler par notification

```
tasks:
- name: configure apache
  ansible.builtin.template: src="vhost.conf" dest="/etc/httpd/conf.d/vhost.conf" ...
  notify: reload apache
```

La directive ***notify:*** va entraîner l'exécution du handler "**reload apache**" si la tâche "**configure apache**" a modifié la machine cible.

Dans l'exemple, si la tâche "**configure apache**" ne modifie pas le fichier *vhost.conf* (parce que nous l'avons déjà exécutée auparavant), le service Apache ne sera pas relancé.

5.1.3- Déclenchement par souscription à un message

On peut également spécifier une directive ***listen "message"***.

À réception de "message" tous les handlers écoutant "message" seront exécutés.

Cela simplifie l'appel de plusieurs handlers.

5.1.4- Exécution des handlers

- Un handler s'exécute **à la fin** du *play* après que toutes les tâches aient été exécutées
- Un handler n'est exécuté qu'*une seule fois* dans un play même s'il a été "notifié" à plusieurs reprises
- Les handlers sont toujours exécutés dans l'ordre dans lequel ils sont définis et non pas dans l'ordre d'appel.

Attention : dans le cas d'une erreur dans un play, certains changements peuvent avoir été appliqués sans que les services aient été redémarrés.

5.2- Les rôles et les includes

5.2.1- Includes

Ref: https://docs.ansible.com/ansible/latest/playbooks_reuse.html

Les directives d'inclusion sont essentielles pour la mise au point de playbooks réutilisables.

Plutôt que d'avoir un gros playbook monolithique, on va plutôt utiliser une multitude de fichiers que l'on pourra utiliser et recombiner selon nos besoins, en utilisant ces fonctionnalités d'importation, d'inclusion et de rôles (que nous aborderons plus tard).

On aura ainsi un ***playbook "maître"*** (site.yml) qui se contente d'inclure les fichiers dans lesquels sont définies les tâches que l'on souhaite exécuter. Cela nous permet de constituer petit à petit une bibliothèque de tâches réutilisables.

C'est ce playbook maître qui sera appelé à l'aide de ***ansible-playbook***.

```
▶ cat site.yml
---
- hosts: cibles
  tasks:
  - ansible.builtin.import_tasks: vim.yml
  - ansible.builtin.import_tasks: epel.yml


▶ cat epel.yml
-   name: EPEL repo definition
    ansible.builtin.yum: name="epel-release" state=present
    when: ansible_distribution=="CentOS"

▶ cat vim.yml
---
- name: vim installation - Debian
  tags: [vim,debian]
  ansible.builtin.apt: name=vim state=present
  when: ansible_distribution=='Debian'

- name: vim installation - CentOS
  ansible.builtin.yum: name=vim-enhanced state=present
  tags: [vim,CentOS]
  when: ansible_distribution=='CentOS'

- name: vim configuration
  tags: [vim]
  ansible.builtin.template: src="/etc/ansible/Srv/vimrc" dest="/etc/vimrc" owner=root group=root
mode=644

▶ ansible-playbook site.yml
```

Depuis la version 2.4, on distingue 2 modes d'utilisation du contenu réutilisable :
- statique,
- dynamique.

import* -> statique - les directives sont évaluées avant l'exécution
include* -> dynamique - les directives sont évaluées lors de l'exécution

La directive **include** est désormais (v2.4) obsolète.

Il faut lui préférer la directive **include_tasks** ou l'équivalent statique **import_playbook** / **import_tasks**.

▶ L'exemple suivant ne marche pas.

```
- hosts: all
  tasks:
  - import_tasks: "{{ ansible_distribution }}.yml"
```

En effet, {{ ansible_distribution }} est inconnu au moment où la directive est analysée, pendant la phase de préprocessing.

Il est également possible d'importer ou d'inclure des rôles parmi une liste de tâches avec les directives **import_role** et **include_role**.

Les limites des playbooks

Un playbook, surtout s'il fait référence à des fichiers externes (directives *include*, modules *templates, copy, ...*) est difficile :

- à partager
- à réutiliser
- à intégrer dans un gestionnaire de version
- à documenter (à moins de rajouter un Readme externe)

Les rôles sont une solution à cette problématique.

5.2.2- Les rôles

Ref: https://docs.ansible.com/ansible/latest/playbooks_reuse_roles.html

Les rôles proposent un niveau d'abstraction par rapport aux playbooks.

Si le playbook décrit une liste d'actions à exécuter sur la cible, le rôle va définir l'objectif fonctionnel de la cible.

Les rôles permettent également de charger automatiquement des **variables**, des **tâches**, des **handlers**, etc. au moyen d'une arborescence spécifique qui démarre d'un répertoire ayant pour nom celui du rôle et contenant ceci :

defaults	variables utilisées par le rôle
files	contient les fichiers que ce rôle doit déployer (notamment via le module *copy*)
handlers	définition des handlers associés au rôle
meta	Métadonnées du rôle
tasks	tâches
templates	modèles à déployer, notamment par le module *template*
tests	fichiers utilisés pour tester le rôle
vars	autres variables associées au rôle.
README.md	documentation du rôle

Cette arborescence peut être facilement créée :

```
▶ ansible-galaxy init tco.common
- tco.common was created successfully
▶ /bin/ls -1 tco.common
defaults
files
handlers
meta
README.md
tasks
templates
tests
vars
```

Le nom d'un rôle est communémement préfixé par votre nom/pseudo.

Chacun de ces répertoires, s'il est utilisé, doit contenir un fichier ***main.yml*** contenant les informations et instructions adéquates.

Afin de conserver une certaine modularité, et de permettre par exemple de définir des rôles multi plate-forme, les fichiers d'entrée (***main.yml***) sont généralement assez simples et se contentent d'inclure d'autres fichiers.

Par exemple :

```
▶ cat tasks/main.yml
- name: include platform-dependent tasks
  include_tasks: "{{ ansible_os_platform }}.yml"
```

5.2.3- Utilisation d'un rôle

L'association cible <-> rôle se fait dans un ***play*** :

```
---
- hosts: webservers
  roles:
  - tco.common
```

À partir de là, seront ajoutés au play:

1. les tâches définies dans tco.common/tasks/main.yml,
2. les handlers définis dans tco.common/handlers/main.yml,
3. les variables définies dans : tco.commonecommon/vars/main.yml
   et tco.common/defaults/main.yml
4. tous les modules copy, script, template ou includes pour faire références aux fichiers présents dans tco.common/files, tco.common/templates, tco.common/tasks et tco.common/vars sans avoir besoin d'utiliser des chemins absolus.

- Ansible va rechercher le rôle concerné dans les emplacements suivants:

1. $PWD/roles
2. ~/.ansible/roles , /usr/share/ansible/roles , /etc/ansible/roles # cf directive de configuration *roles_path*
3. $PWD

5.2.4- Variables dans les rôles

5.2.4.1-

On peut associer des variables à un rôle lorsque l'on y fait référence:

```
---
- hosts: webservers
  roles:
    - common
    # YAML inline
    - { role: foo, dir: '/opt/w1', listen_port: 5000 }
    - { role: foo, dir: '/opt/w2', listen_port: 5001 }
    # YAML
    - role: foo
      vars:
        dir: /opt/w3
        listen_port: 5002
```

5.2.4.2- Variables par défaut

On peut définir des variables par défaut (qui auront la plus basse priorité possible) dans le fichier role/defaults/main.yml. Elles sont faites pour être surchargées.

Les variables dans role/vars/main.yml ne peuvent pas être surchargées par l'inventaire.

5.2.4.3- Visibilité des variables dans les rôles

Les variables définies dans un rôle sont accessibles aux autres rôles du play. Par conséquent, il convient d'être vigilant dans le choix du nom des variables.

5.2.5- Dépendances entre rôles

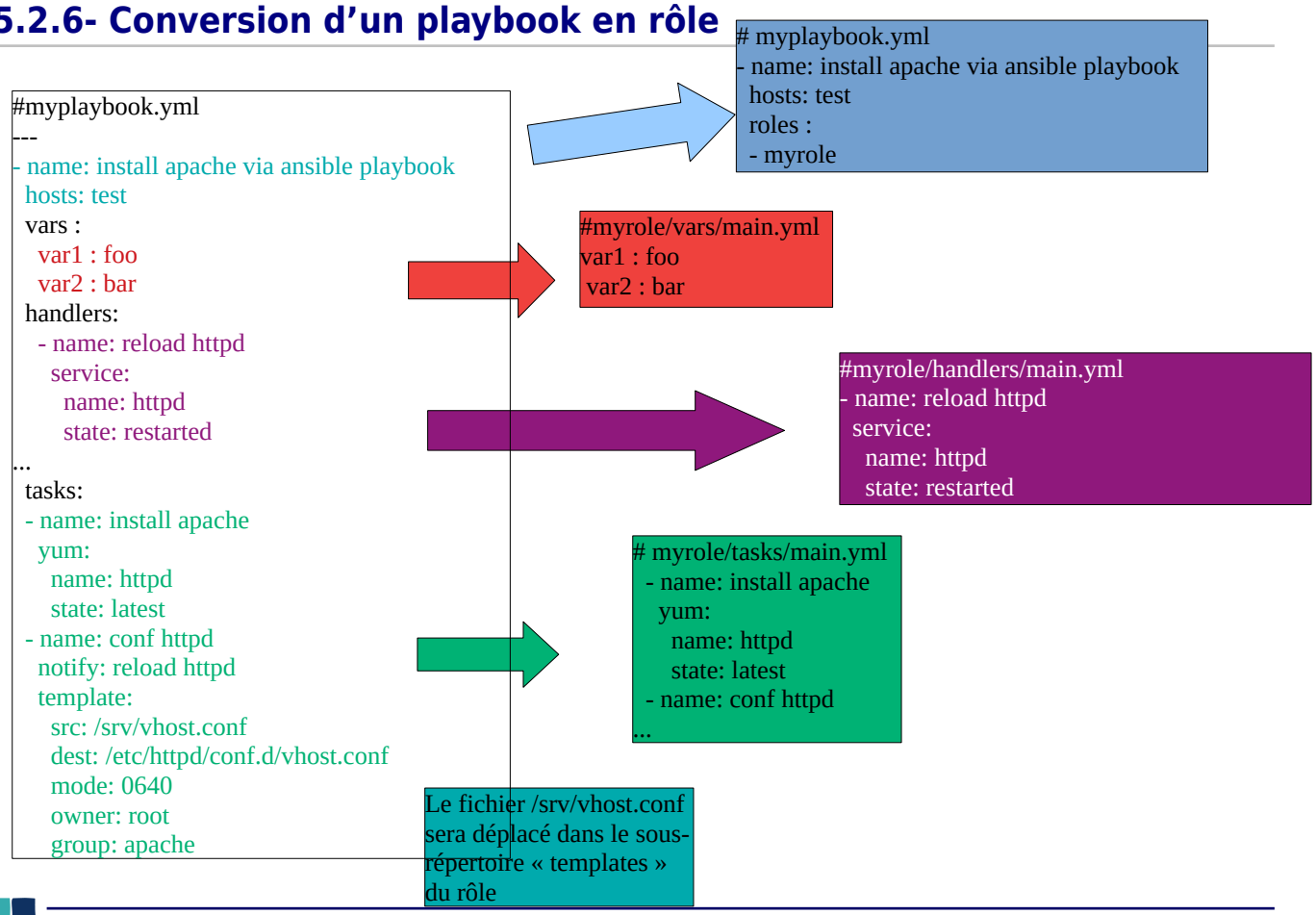
Depuis la version 1.3, on peut, via le fichier meta/main.yml définir des dépendances entre rôles :

```
▶ cat roles/foo/meta/main.yml
---
dependencies:
 - role: common
 - { role: apache, apache_port: 80 }
```

Un playbook qui utilise le rôle ***foo*** exécutera dans cet ordre les rôles :

1. ***common***
2. ***apache***
3. ***foo***

5.2.6- Conversion d'un playbook en rôle



5.3- Les tags

Ref: https://docs.ansible.com/ansible/latest/playbooks_tags.html

5.3.1- Définir des étiquettes

Les tags (étiquettes) sont un moyen de catégoriser des tâches afin d'en simplifier la sélection.

La manière la plus simple d'étiqueter une tâche est d'inclure la directive *tags* :

```
- name: Install mandatory packages
  tags: sudo
  package: name=sudo state=latest
```

On peut également "tagguer" un playbook, ou un rôle. Dans ce cas, toutes les tâches associées au groupe ou au rôle vont hériter des étiquettes ainsi définies :

```
- hosts: all
  tags: common

  tasks:
  - include_tasks: common.yml
    tags: [bootstrap]
```

Les tâches du fichier common.yml seront étiquetées common et bootstrap.

L'option ***list-tags*** permet de lister toutes les étiquettes utilisées dans un playbook :

```
▶ ansible-playbook --list-tags /etc/ansible/Playbooks/0epsi.yml

playbook: /etc/ansible/Playbooks/0epsi.yml

  play #1 (epsi): epsi  TAGS: []
      TASK TAGS: [CentOS, bash, debian, default, epel, reboot, ssh, ssh tmp, syslog, vim]
```

L'option ***list-tasks*** va quant à elle lister toutes les tâches (et leurs étiquettes) composant un playbook :

```
▶  ansible-playbook --list-tasks /etc/ansible/Playbooks/0epsi.yml
playbook: /etc/ansible/Playbooks/0epsi.yml

  play #1 (epsi): epsi  TAGS: []
    tasks:
      vim installation - Debian TAGS: [debian, vim]
      vim installation - CentOS TAGS: [CentOS, vim]
      Vim configuration TAGS: [vim]
      EPEL repo definition      TAGS: [epel]
      include   TAGS: []
      bash installation TAGS: [bash]
      liquidprompt req  TAGS: [bash]
      liquidprompt install      TAGS: [bash]
      liquidprompt config       TAGS: [bash]
```

5.3.2- Utiliser les étiquettes

L'option --***tags=tag1,tag2*** de ansible-playbook permet de n'exécuter que les tâches étiquetées correctement.

On dispose également de l'option opposée "--***skip-tags***".

5.3.2.1- Étiquettes spéciales

On peut utiliser les étiquettes spéciales ***tagged***, ***untagged*** et ***all*** pour sélectionner respectivement les tâches avec ou sans étiquettes, ou l'ensemble des tâches.

L'étiquette ***always*** permet d'exécuter systématiquement une tâche, à moins qu'elle ait été spécifiquement exclue via un –***skip-tags=always***.

L'étiquette ***never*** empêche l'exécution d'une tâche à moins qu'elle ait été explicitement demandée avec l'étiquette ***never*** ou une étiquette de la tâche.

- Par défaut, Ansible exécute les playbooks avec le tag ***all***.

5.4- Les modules de la communauté

La collection de modules standard est suffisante pour couvrir la majorité des besoins.

Et si vous ne trouvez pas votre bonheur, il est fort probable qu'une âme charitable ait développé un module particulier.

Ceux-ci sont généralement hébergés sur **GitHub** [5]

Listes des modules en cours de développement :

https://github.com/ansible/ansible/labels/new_module

5 https://github.com/search?p=7&q=ansible+module&type=Repositories

5.5- Ansible-galaxy : partager son code

Ref:

- http://docs.ansible.com/ansible/latest/galaxy.html
- https://galaxy.ansible.com/docs/

Ansible-galaxy est à la fois un site communautaire[6] de partage de code Ansible (rôles et collections) et une commande permettant d'installer, de créer et de gérer ce contenu à partir de ce site.

Plutôt que de réinventer la roue, il peut être intéressant de s'appuyer sur des rôles et collections existants pour la mise en œuvre d'un projet.

5.5.1- Remarques sur la sécurité

Il convient d'être conscient des implications que l'utilisation d'un code 'externe' peut avoir vis-à-vis des impératifs de sécurité.

En exécutant les plays d'un rôle, on exécute du code sur les machines cibles.

Avant tout chose, dans le cas d'un rôle issu de la communauté, il est nécessaire d'auditer celui-ci. Examiner la popularité, le score et le contenu du gestionnaire de bugs est également un excellent moyen pour juger de la qualité d'un rôle.

6 https://galaxy.ansible.com/

5.5.2- Recherche et information

```
▶ ansible-galaxy search apache
Found 288 roles matching your search:

 Name                                          Description
 ----                                          -----------
 Shashikanth-Komandoor.apache
 shashikanth.komandoor@gmail.com.apache
 Shashikanth-Komandoor2.apache
 Shashikanth-Komandoor.apache2
 jpnewman.apache                               Apache2
 adarnimrod.apache                             Apache
...
▶ ansible-galaxy info geerlingguy.apache

Role: geerlingguy.apache
        description: Apache 2.x for Linux.
        active: True
        commit: 88b469048a937ea0c7c5a55665d239ebf0418c5a
        commit_message: Merge pull request #134 from ArgonQQ/patch-1

Removed inconsistent ssl vhost example
        commit_url:
https://github.com/geerlingguy/ansible-role-apache/commit/88b469048a937ea0c7c5a55665d239ebf0418c5a
        company: Midwestern Mac, LLC
        created: 2014-02-28T22:24:11.514Z
        download_count: 67313
        ...
```

Pour juger de la qualité d'un contenu externe: examiner la date de dernière mise à jour, le nombre de téléchargements, le nombre de bugs ouverts, ...

5.5.3- Installation d'un rôle

▶ `ansible-galaxy install geerlingguy.apache`

Suivant le compte utilisé pour exécuter cette commande, le rôle sera téléchargé et installé de manière locale (compte non privilégié) ou globale (si la commande est exécutée en tant que root).

Voir la variable d'environnement ***ANSIBLE_ROLES_PATH*** et l'option ***roles-path***.

5.5.4- Créer un rôle

▶ `ansible-galaxy init role_name`
Permet de créer l'arborescence nécessaire à la mise au point d'un rôle.

5.5.5- Contribuer

Avec un compte GitHub, vous avez la possibilité d'importer vos rôles sur la plate-forme pour en faire bénéficier la communauté. Cela se fait au moyen des commandes **login**, **import** et **setup** d'**ansible-galaxy**.

5.5.6- Installation d'une collection

```
ansible-galaxy collection install debops.debops
```

La collection sera téléchargée et installée dans l'emplacement défini par le paramètre de configuration ***COLLECTIONS_PATH***.

5.5.7- Créer une collection

```
ansible-galaxy collection init nqb.mycollection
```

Permet de créer une collection vierge, appelée **mycollection**, qui pourra ensuite être importée dans l'espace de nom **nqb** sur Ansible Galaxy.

5.5.8- Contribuer

Les pré-requis pour contribuer sont les mêmes que pour un rôle. Cependant la publication sur Ansible Galaxy se fait à l'aide des commandes suivantes :

- **ansible-galaxy collection build**
- **ansible-galaxy collection publish**

6- Compléments sur les playbooks

6.1- Les variables

Ref: https://docs.ansible.com/ansible/latest/playbooks_variables.html

Les variables permettent de gérer les petites différences pouvant exister entre différentes machines qui ne justifient pas l'écriture d'une ou plusieurs tâches sensiblement différentes.

Il est important :
- de les nommer correctement
- de les documenter
- de leur donner une valeur par défaut (si possible)
- de ne pas en abuser

6.1.1- Nommer les variables

Le nom d'une variable est composé de lettres, de nombres et d'underscore.

Il est recommandé de les préfixer (par ex par le nom du service concerné) pour éviter les confusions : *apache_listen_port* > *listen_port*

On peut, en YAML, définir des tableaux:

```
foo:
 field1: valueF1
 field2: valueF2
```

On peut obtenir la valeur d'un élément du tableau via

```
foo['field1']   # différent de foo[field1]
foo.field1
```

- Les simples quotes annulent l'interpolation de `field1` en tant que variable.

6.1.2- Variables de play

```
- hosts: webservers
  vars:
    http_port: 80
    https_port: 443
  tasks:
 ...
```

6.1.3- Variables de machines et de groupes

Il est déconseillé de définir des variables directement dans le fichier d'inventaire.

Il est préférable d'utiliser des fichiers YAML que l'on stockera dans les répertoires situés dans group_vars[7] (pour les variables de groupes) et host_vars[1] (pour les variables de machines) et ayant pour nom celui des groupes ou des machines pour lesquelles on souhaite définir des variables spécifiques.

Ainsi si j'ai 2 groupes web*servers* et db*servers* et des machines *lamp[1:3],* j'aurai l'arborescence suivante :

```
group_vars
├── webservers.yml
└── dbservers.yml
host_vars
├── lamp1
├── lamp2
└── lamp3
```

7 **ce répertoire doit être dans le même répertoire que le fichier d'inventaire utilisé**

Pour les groupes, il est également possible de créer un dossier au nom du groupe qui contient plusieurs fichiers YAML :

```
group_vars
├── dbservers
│   ├── vars.yml
│   └── vault.yml
└── webservers
    ├── vars.yml
    └── vault.yml
```

6.1.4- Variables de rôle

Les variables de rôles peuvent être définis dans vars/main.yml et defaults/main.yml

Les variables définies dans defaults/main.yml ont la priorité la plus basse. C'est le contexte idéal pour définir des valeurs qui seront appliquées par défaut.

6.1.5- Variables définies par le résultat d'une commande

On peut enregistrer le résultat d'une commande comme valeur d'une variable via la directive *register* :

Dans cet exemple, le résultat de la première tâche sera sauvé dans la variable *foo_status*.

On choisit d'ignorer les erreurs (par exemple fichier illisible) avec la directive *ignore_errors* .

La 2ème tâche ne sera exécutée que si le contenu du fichier (la sortie standard (stdout) de la commande cat /var/run/foo.status) est égal à "success".

```
- ansible.builtin.shell: cat /var/run/foo.status
  register: foo_status
  ignore_errors: True

- module: module_arg=value
  when: foo_status.stdout == "success"
```

6.1.6- Variables d'une autre machine

La directive *hostvars['host_name']['variable_name']* permet de récupérer une variable (y compris un *fact*) d'une autre machine :

```
---
- hosts: test
  become: true
  remote_user: tom
  handlers:
  - name: echo handler
    ansible.builtin.command: echo "i have been called"


  tasks:
  - ansible.builtin.debug: msg="la machine gelatine est une {{ hostvars['gelatine']
['ansible_distribution'] }}"
```

Attention : le fait de positionner des simples quotes autour d'un mot entre crochets supprime l'évaluation de ce mot en tant que variable.

6.1.7- Variables issues de la ligne de commande

Il est également possible de (re)définir des variables en les passant en ligne de commande via l'option --extra-vars

```
---
- hosts: test
  become: false
  remote_user: tom

  tasks:
  - ansible.builtin.debug:
      var: myvariable
```

```
▶ ansible-playbook -i inventaire -k test.yml
✔ PLAY [test]
✔
✔ TASK [debug]
✔ ok: [192.168.10.183] => {
✔     "myvariable": "VARIABLE IS NOT DEFINED!"
✔ }
▶ ansible-playbook -i inventaire -k test.yml  --extra-vars='myvariable="foo bar"'
PLAY [test]

TASK [debug]
ok: [192.168.10.183] => {
    "myvariable": "foo bar"
}
```

On peut également utiliser la syntaxe --extra-vars @fichier.yml avec fichier.yml contenant la valeur de la ou des variables.

6.1.8- Jinja2 : Utiliser les variables dans un playbook

Les variables peuvent être utilisées via le système de template Jinja2 utilisé par Ansible.

On fait référence à la valeur d'une variable via la construction {{ nom_variable }}

Dans cet exemple, on utilise la variable *ansible_distribution* pour distribuer un fichier de configuration avec des réglages spécifiques aux différentes distributions utilisées.

Cette variable est une des variables *fact*, récupérées par Ansible et contenant diverses informations sur la machine cible.

```
ansible.builtin.template:
   src: "/etc/ansible/Srv/zabbix_agentd_{{ ansible_distribution }}.conf"
   dest: "/etc/zabbix/conf/zabbix_agentd.conf"
   owner: "root"
   group: "zabbix"
   mode="0640"
```

6.1.9- Utiliser des variables dans des tests

Nous avons vu plus haut l'utilisation de la directive *when* qui contrôle l'exécution d'une tâche en fonction d'un test.

On peut par exemple exécuter une tâche différente suivant la distribution:

```
---
- name: vim installation - Debian
  ansible.builtin.apt: name=vim state=present
  when: ansible_distribution=='Debian'

- name: vim installation - CentOS
  ansible.builtin.yum: name=vim-enhanced state=present
  when: ansible_distribution=='CentOS'
```

6.1.10- Synthèse sur les variables et leur priorité

Ref : https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html

Ce tableau est une version simplifiée de la documentation.

rang	Origine	Remarque
0	Variable par défaut du rôle	RoleName/defaults/main.yml
1	Variables d'inventaire	Hosts_var > group_var > Groupe
2	Facts	
3	Variables de play	
4	Variables de rôle	RoleName/vars/main.yml
5	Variables de tâche	
6	Include_vars	
7	Ligne de commande	--extra-vars

(par ordre de priorité croissante)

6.2- Les templates Jinja2 et les filtres

Ref: https://docs.ansible.com/ansible/latest/playbooks_templating.html

https://docs.ansible.com/ansible/latest/playbooks_filters.html

http://jinja.pocoo.org/docs/dev/

6.2.1- Principes du templating Jinja2

Ansible utilise le moteur de template *Jinja2*.

Le principe d'un template (modèle) est simple :

1. on insère des marqueurs dans un fichier template (extensions .jj ou .j2)
2. ils seront remplacés par d'autres éléments lors de l'interprétation du template.

On utilise ce mécanisme quand on référence une {{ variable }} par exemple.

On va pouvoir utiliser cette variable telle quelle ou lui appliquer un **filtre**.

Définition d'une variable

```
{{ var }}
{{ var | default( 5 ) }}
```

Filtre de validation

```
{{ myvar | ipaddr }}
```

Filtre de hashage et mot de passe

```
{{ 'passwordsaresecret' | password_hash('sha512', 65534 | random( seed=inventory_hostname ) | string) }}
```

Filtre de manipulation de chaînes de caractères.

```
{{ 'ansible' | regex_search('(foobar)') }}
{{ 'localhost:80' | regex_replace(':80') }}
```

Filtres divers

```
{{ path | basename }}
{{ path | dirname }}
{{ '%Y-%m-%d' | strftime }}
"{{ 59 | random( seed=inventory_hostname ) }} * * * * root /script/from/cron"
```

6.2.2- Le module template : processeur Jinja2

Ce module prend en source un template, et va appliquer les transformations demandées avant d'envoyer le fichier vers la machine distante.

▶ cat defaults/main.yml:
```
---
mysql_root_password: 123Soleil!!
mysql_root_username: root
mysql_root_home: /root
```

▶ cat templates/root-my.cnf.j2 :
```
[client]
host=localhost
user={{ mysql_root_username }}
password={{ mysql_root_password }}
```

▶ cat tasks/main.yml
```
...
ansible.builtin.template:
  src: root-my.cnf.j2
  dest: "{{ mysql_root_home }}/.my.cnf"
  mode: 0600
  owner: root
  group: root
...
```

▶ L'interprétation du template a lieu sur le contrôleur Ansible et non sur les cibles.

6.3- Structures de contrôle : Conditions, Boucles et Blocks

Ref : https://docs.ansible.com/ansible/latest/playbooks_loops.html

6.3.1- Conditions

La directive when permet de conditionner l'exécution d'une tâche :

```
- name: vim installation - Debian
  apt: name=vim state=present
  when: ansible_distribution=='Debian'
```

On peut également conditionner l'exécution d'une tâche au résultat d'une tâche précédente :

```
- name: tache 1
  module: args...
  register: result
  ignore_errors: True

- name: tache 2 - ne s'exécute que si tache 1 a renvoyé un code de retour 42
  module: args ...
  when: result.rc == 42
```

ou encore :

```
---
[..]
    - name: run /bin/true command
      ansible.builtin.command: /bin/true
      register: result_true
      ignore_errors: True

    - name: fail
      ansible.builtin.debug:
        msg: "echo failed"
      when: result_true is failed

    - name: ok
      ansible.builtin.debug:
        msg: "echo succeeded because result_true contains failed: false"
      when: result_true is succeeded
```

La directive when s'applique également pour conditionner l'importation d'objets et l'affectation de rôle :

```
- hosts: webservers
  roles:
  - { role: debian_stock_config, when: ansible_os_family == 'Debian' }
```

Dans cet exemple, la condition sera appliquée à *toutes* les tâches du rôle.

6.3.2- Boucles

La directive loop permet d'éviter la répétition d'une même tâche avec un argument différent (gestion des utilisateurs par exemple).

Avant la version 2.5, Ansible utilisait la construction with_<lookup> pour créer des boucles. L'instruction loop est maintenant à privilégier, elle équivaut à with_list. Des exemples de l'ancienne syntaxe sont données en annexe.

```
- name: add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
   - testuser1
   - testuser2
```

Il n'est pas nécessaire d'utiliser cette directive avec les modules apt, yum ou packages, qui acceptent directement des listes :

```
- name: install mandatory packages
  ansible.builtin.yum:
    name: "{{ packages }}"
    state: latest
  vars:
    packages:
    - git
    - tree
```

Il est aussi possible d'itérer sur des dictionnaires :

```
- name: add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

▶ Si on utilise l'instruction when avec une boucle, la condition est évaluée pour chaque élément de la boucle.

6.3.3- Boucles imbriquées

La directive loop et le filtre product permettent d'implémenter des boucles imbriquées (équivalent à with_nested) :

```
---
- hosts: localhost
  vars:
    meubles: ["canapé", "table", "chaise" ]
    couleurs: ["vert", "rouge", "bleu" ]
  tasks:
  - name: with_nested -> loop
    ansible.builtin.debug:
      msg: "un·e {{ item.0 }} de couleur {{ item.1 }}"
    loop: "{{ meubles|product(couleurs)|list }}"
```

retourne :

```
ok: [localhost] => (item=['canapé', 'vert']) => {
    "msg": "un·e canapé de couleur vert"
}
ok: [localhost] => (item=['canapé', 'rouge']) => {
    "msg": "un·e canapé de couleur rouge"
}
ok: [localhost] => (item=['canapé', 'bleu']) => {
    "msg": "un·e canapé de couleur bleu"
}
ok: [localhost] => (item=['table', 'vert']) => {
    "msg": "un·e table de couleur vert"
}
ok: [localhost] => (item=['table', 'rouge']) => {
    "msg": "un·e table de couleur rouge"
}
ok: [localhost] => (item=['table', 'bleu']) => {
    "msg": "un·e table de couleur bleu"
}
```

```
ok: [localhost] => (item=['chaise', 'vert']) => {
    "msg": "un·e chaise de couleur vert"
}
```

6.3.4- Boucle sur fichier

La directive with_fileglob permet de copier un ensemble de fichiers :

```
- name: Copy each file over that matches the given pattern
  ansible.builtin.copy:
    src: "{{ item }}"
    dest: "/etc/fooapp/"
    owner: "root"
    mode: 0600
  with_fileglob:
    - "/playbooks/files/fooapp/*.cfg"
```

La directive with_filetree permet de recréer une arborescence complète.

Pour chaque élément il met à notre disposition différentes information sur l'élément (taille, type, permissions, ...)

```
---
- name: Create directories
  ansible.builtin.file:
    path: {{ item.path }}
    state: directory
    mode: '{{ item.mode }}'
  with_filetree: web/
  when: item.state == 'directory'

- name: Template files
  ansible.builtin.template:
    src: '{{ item.path }}'
    dest: /web/{{ item.path }}
    mode: '{{ item.mode }}'
  with_filetree: web/
  when: item.state == 'file'
```

6.3.5- Divers

Boucler sur des séquences : directive loop avec fonction range et filtre format (équivaut à with_sequence) :

```
- name: with_sequence -> loop
  ansible.builtin.debug:
    msg: "{{ 'testuser%02x' | format(item) }}"
  # range is exclusive of the end point
  loop: "{{ range(0, 4 + 1, 2)|list }}"
```

Boucles do ... until : directive until

```
- ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

Boucler sur (une partie de) l'inventaire :

```
- ansible.builtin.debug:
    msg: "{{ item }}"
  loop:
    - "{{ groups['all'] }}"
```

Les structures de contrôles dans les templates

Les fichiers déployés avec le module *template* peuvent également comporter des structures de contrôle. Voir aussi https://jinja.palletsprojects.com/en/3.0.x/templates/#list-of-control-structures

Boucles:

Variable
```
vars:
   domains:
        - d1
        - d2
        - d3
```

Template
```
'trusted_domains' =>
  array (:
{% for d in domains %}
  {{ loop.index0 }} => {{ d }}
{% endfor %}
),
```

Résultat
```
'trusted_domains' =>
  array (:
    0 => d1
    1 => d2
    2 => d3
),
```

Tests:

```
{% if postfix_use_ldap %}
# cette directive ne sera présente dans le fichier destination que si la variable postfix_use_ldap est définie à True
   ldap:/etc/postfix/ldap_alias.cf
{% endif %}
```

6.4- Les prompts

Ref : https://docs.ansible.com/ansible/latest/playbooks_prompts.html

Cette fonctionnalité rajoute un peu d'interactivité dans l'exécution d'un playbook en offrant la possibilité de demander certaines informations à l'utilisateur. Ces informations seront stockées dans des variables.

La directive ***vars_prompt*** accepte les arguments suivants :

- ***name***: le nom de la variable dans laquelle sera stockée la réponse
- ***prompt***: la question qui sera affichée
- ***private***: booléen contrôlant l'affichage (ou pas) de la réponse.
- ***default***: permet de spécifier une valeur par défaut

```
- hosts: mysql
  become: true
  become_method: sudo
  vars_prompt:
    - name: mysql_root_password
      prompt: "Enter mysql root password"
      private: true
  roles:
  - mysql
```

Chiffrement :

En utilisant ***passlib***, on peut également chiffrer la réponse d'un utilisateur.

- ***encrypt***: permet de spécifier l'algorithme utilisé

    ⇒ voir la documentation de référence pour la liste

- ***salt*** et ***salt_size*** permettent de spécifier le salt utilisé,

- ***confirm***: booléen contrôlant la confirmation (double saisie) d'une réponse.

6.5- Facts

Ref: https://docs.ansible.com/ansible/latest/playbooks_variables.html#variables-discovered-from-systems-facts

Les facts sont des informations obtenues depuis les machines cibles, par exemple, l'adresse ip, le nombre de périphériques de stockage, la distribution, etc.

Les facts sont obtenus au début de l'exécution d'un playbook

⇒ à l'étape G*athering Facts*.

Cette étape peut être désactivée en positionnant la directive *gather_facts: False* dans la définition du playbook.

Ce réglage permet d'améliorer les performances de manière significative.

6.5.1- Visualisation des informations récupérées

```
▶ ansible test -m setup
100.0.0.100 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "100.0.0.100"
        ],
        "ansible_all_ipv6_addresses": [],
        "ansible_apparmor": {
...
```

Les informations sont utilisables dans une tâche comme toute autre variable.

Exemple : copier un fichier dépendant de la version majeure de la distribution:

```
- name: copy version dependant config file
  copy:
    src: "/etc/ansible/Files/file_{{ ansible_distribution_major_version }}.cfg"
    ...
  when: ansible_distribution == "RedHat"
```

Cet exemple implique bien sûr la présence d'autant de fichiers /etc/ansible/Files/file_{5,6,7}.cfg qu'on a de systèmes différents à gérer.

6.5.2- Mise en cache

Les facts peuvent être mis en cache. Cela permet également d'améliorer la vitesse d'exécution des playbooks. C'est utile dans le cas où une tâche utilise des informations d'une autre machine :

```
hostvars['100.0.0.100']['ansible_distribution']
```

Les informations peuvent être mises en cache[8] dans une base Redis, ou dans un fichier JSON.

Exemple de configuration pour l'utilisation d'un fichier JSON :

```
▶ cat /etc/ansible/ansible.cfg
[defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /etc/ansible/cache
fact_caching_timeout = 86400
```

Le cache peut être vidé avant l'expiration du timeout via :

```
▶ ansible-playbook --flush-cache
```

8 https://docs.ansible.com/ansible/latest/playbooks_variables.html#fact-caching

6.5.3- Définir ses propres facts

Sur la machine cible, créer un fichier */etc/ansible/facts.d/myfacts.fact*

```
#/etc/ansible/facts.d/plop.fact
[section]
f1=val1
f2=val2
```

Ce fichier doit être au format ini. Il va permettre l'association des variables définies dans le fichier avec la machine, via le fact suivant :

ansible_local.*nom_du_fichier*.nom_de_la_section.f1, f2, etc

```
▶ rm -f /etc/ansible/facts/centos1.formation.opendoor.fr ; ansible -m setup
centos1.formation.opendoor.fr -a "filter=ansible_local"
Thursday 26 January 2023  12:17:50 +0100 (0:00:00.065)       0:00:00.065 ******
semimarine | SUCCESS => {
    "ansible_facts": {
        "ansible_local": {
            "plop": {
                "section": {
                    "f1": "val1",
                    "f2": "val2"
                }
...
```

6.5.4- Définir des facts lors de l'exécution

Le module *ansible.builtin.set_fact* permet d'associer des variables à la machine en cours de traitement.

```
- name : set some variables
  ansible.builtin.set_fact :
       ssh_key_path : /home/{{ ssh_user }}/.ssh/id_rsa.pub
```

L'argument *cacheable* (False par défaut) du module permet de contrôle la possibilité de mise en cache de cette variable.

6.6- Bonnes pratiques pour la rédaction de playbooks

Ref: https://www.ansible.com/blog/ansible-best-practices-essentials
http://docs.ansible.com/ansible/latest/playbooks_best_practices.html
http://ansible.github.io/lightbulb/decks/ansible-best-practices.html#/
https://ansible.github.io/workshops/decks/ansible_best_practices.pdf

Respectez le principe Keep It Simple, Stupid !

Donnez des noms à vos groupes de machines, vos tâches, vos plays.

Définissez une convention de nommage et respectez-là (notamment pour les noms de variables). Ansible recommande d'utiliser un préfixe en rapport avec l'utilisation de la variable. Par exemple préférez *apache_port* à *port* pour définir le port d'écoute de votre apache.

Préférez la notation YAML par rapport à la notation key=valeur.

N'abusez pas des modules shell, command, ... Il y a probablement un module plus

adapté à votre besoin.

Utilisez les rôles pour organiser vos playbooks, plays et tâches et les rendre plus facilement maintenables. Vous avez également tout intérêt à les intégrer dans un VCS.

Utilisez les répertoires group_vars et host_vars pour associer des variables à des (ensembles de) machines.

Paramétrisez vos tâches et vos playbooks par l'utilisation de variables.

Si vous devez modifier un playbook ou une tâche pour l'adapter à un groupe de machines en particulier, c'est que vous devriez utiliser une variable.

7- Notions avancées

7.1- Vault : chiffrement de données

Ref: https://docs.ansible.com/ansible/latest/vault.html

Vault (coffre-fort) est un mécanisme permettant de stocker des informations sensibles (clés, mots de passe) dans un fichier chiffré, plutôt que dans un playbook ouvert à tous.

Ce type de fichier est manipulé à l'aide de la commande **ansible-vault**.

Le mot de passe permettant de déchiffrer le coffre peut être demandé à l'utilisateur via l'option --***ask-vault-pass*** ou stockée dans un fichier que l'on spécifiera via l'option --***vault-password-file*** *chemin*.

ansible-vault est principalement utilisé pour protéger des variables sensibles. Mais on peut également l'utiliser pour manipuler des fichiers complets.

Il est utilisable via les modules template, copy, ...

7.1.1- Création d'un coffre

```
ansible-vault create myvault.yml
```

Notre *EDITOR* favori sera lancé, nous permettant de rentrer nos premières informations sensibles.

7.1.2- Édition

```
ansible-vault edit myvault.yml
```

7.1.3- Visualisation

```
ansible-vault view myvault.yml
```

7.1.4- Utilisation

Il suffit de créer une tâche qui va inclure notre « vault »

```
name: include sensitive data
include_vars: myvault.yml
```

et rajouter l'option --ask-vault-pass à **ansible-playbook** :

```
ansible-playbook --ask-vault-pass playbook.yml
```

7.1.5- Chiffrer une chaîne de caractères

La commande **ansible-vault encrypt-string** permet de chiffrer une chaîne de caractère pour l'intégrer directement dans un fichier YAML non chiffré :

```
▶ ansible-vault encrypt_string  'MonMotDePasseVerySecret!' --name 'user_password'
```

donne :

```
user_password: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          62633231616664623262353864306432303630323539646562356135383262316631316438333739
          62646139616666323734333062613161656236386333363330a3632356632373434303230643338 61
          37373661636237363937396266643562303037316138643266386534353331316331303962336562
          36636563623763663 80a373663363230303631323863386135623134636131326563306532663730
          3231
Encryption successful
```

Ce résultat peut ensuite être copié dans un fichier YAML comme n'importe quelle autre définition de variable.

7.2- Modules et plugins

7.2.1- Modules

Les *modules* sont des programmes utilisables par l'API Ansible, la commande **ansible** et la commande **ansible-playbook**.

Ils fonctionnent de façon autonome et répondent à une structure précise (fonctions, arguments et code de retour).

7.2.2- Plugins

Les *plugins* sont des programmes utilisables par n'importe quel module.

Ils interagissent directement avec le cœur d'Ansible.

7.2.3- Développement

Voir la section en annexe pour le développement des modules et plugins.

7.3- Les lookups

Ref: https://docs.ansible.com/ansible/latest/playbooks_lookups.html#examples

Les plugins **lookups** offrent la possibilité de récupérer des données extérieures à ansible, depuis des fichiers ou d'autres sources de données.

Ces plugins sont en relation avec le système de template utilisé, et sont évalués au niveau du contrôleur. On peut utiliser les lookups partout où les variables Jinja2 sont utilisées.

Le lookup le plus simple (***file***) est celui permettant de lire le contenu d'un fichier

```
- name: ssh key
  tags: ssh
  authorized_key: user=root key="{{ lookup('file', '/home/tom/.ssh/id_rsa.pub') }}"
```

Le lookup ***password*** permet de générer un fichier contenant un mot de passe chiffré (ou de récupérer ce mot de passe s'il existe déjà). Utile lorsque l'on souhaite créer un mot de passe aléatoire.

Le lookup ***csv*** permet de récupérer des informations stockées dans un fichier csv

```
lookup('csvfile', 'key arg1=val1 arg2=val2 ...')
```

Les arguments acceptés sont :

- **key** : utilisé pour sélectionner la ligne à interroger.
- **col** : le n° de la colonne à renvoyer
- **delimiter** : le caractère utilisé pour séparer les champs
- **default** : la valeur renvoyé si rien n'est trouvé dans le fichier
- **encoding** : le codage de caractère utilisé.

Le lookup ***ini*** permet de récupérer des informations depuis un fichier ini.

```
lookup('ini', 'user section=integration file=users.ini')
```

Le fonctionnement est similaire au plugin précédent.

Le lookup ***dig***, quant à lui permet d'interroger les dns. Son utilisation nécessite la présence du module python *dnspython* (à priori le package *python-dns*)

```
lookup('dig', 'example.org.', 'qtype=TXT')
```

D'autres lookups permettent d'interroger un nombre de sources de données encore plus important.

7.4- Utilisation avancée d'Ansible : le projet DebOps

- Code source : https://github.com/debops/debops/
- Documentation : https://docs.debops.org

Dès que l'on commence à administrer plusieurs briques logicielles sur son infrastructure serveurs avec Ansible, il devient nécessaire de représenter les interactions entre ces logiciels.

Pour assurer une cohérence sur des serveurs Debian et Ubuntu en utilisant Ansible, on peut utiliser le projet DebOps.

Exemple de fonctionnalités :

- dépendances entre les rôles
- interconnexion des services de façon sécurisée en utilisant des certificats générés par une PKI
- organisation du code Ansible selon les bonnes pratiques
- gestion des secrets

7.5- Ansible Tower

Ref: http://docs.ansible.com/ansible-tower/
https://www.jeffgeerling.com/blog/2017/get-started-using-ansible-awx-open-source-tower-version-one-minute

7.5.1- Présentation

Ansible Tower est une interface graphique à ansible.

Il s'agit d'une offre commerciale. RedHat, propriétaire d'ansible depuis peu, a récemment libéré le code du produit sous le nom d'AWX, qui est librement installable (https://github.com/ansible/awx).

AWX constitue la version de développement communautaire et est à Tower ce que Fedora est à RHEL.

Si la ligne de commande peut convenir à de petites équipes, Tower propose une vue d'ensemble des processus d'automatisation, lisible aussi bien par le management que par les équipes techniques.

Cela apporte une réelle plus-value pour une utilisation massive d'ansible.

7.5.2- Avantages et inconvénients

Enregistrer les paramètres avec lesquels les playbooks sont lancés,
Reporting,
Interface graphique,
Contrôle d'accès,
REST API and callback, pour intégration avec des outils externes,
Architecture multi composant (postgresql, rabbit_mq,) plutôt lourde et complexe.

7.5.3- Installation

Installer docker et docker-compose.

Utiliser docker-compose :

- `curl -O https://raw.githubusercontent.com/geerlingguy/awx-container/master/docker-compose.yml`
- `docker-compose up -d`

Patienter jusqu'à la fin de la mise à jour.

Se connecter sur l'interface web en tant que admin / password.

7.5.4- Tour d'horizon

Projet:
Permet de définir le lien avec les playbooks, qui peuvent être hébergés dans un SCM ou localement.

Organization : Fait le lien entre projet, inventaire, utilisateurs et credentials.

Utilisateurs et équipes : Compte utilisateurs.

Credentials:
Sont un moyen de définir des mécanismes d'authentification (typiquement pour se connecter aux cibles) sans les communiquer aux utilisateurs.

Inventaire
Identique aux inventaires texte d'ansible.

Templates
Un Job template permet de définir un ensemble de paramètres pour un job (Inventaire, credentials, playbook, utilisateurs habilités, ...). C'est l'équivalent de la commande ansible-playbook associé à un fichier playbook.

8- Annexes

8.1- Boucles (ancienne syntaxe)

8.1.1- with_items

```
- name: install mandatory packages
  yum: name={{ item }} state=latest
  with_items:
  - rsync
  - net-tools
  - bmon
```

On peut boucler sur des tableaux plus complexes :

```
- name: add several users
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

8.1.2- with_nested : boucles imbriquées

```
- name: give users access to multiple databases
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  with_nested:
    - [ 'alice', 'bob' ]
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

8.1.3- Divers

Boucler sur des séquences : directive **with_sequence**

```
    - user:
        name: "{{ item }}"
        state: present
        groups: "evens"
      with_sequence: start=0 end=32 format=testuser%02x stride=1
```

Boucles do ... until : directive **until**

```
- shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

Boucler sur (une partie de) l'inventaire : **with_items**

```
- debug:
    msg: "{{ item }}"
  with_items:
    - "{{ groups['all'] }}"
```

8.2- Développer ses propres modules / plugins

Ref:
https://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html
https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html

Pensez d'abord à vous assurer de ne pas réinventer la roue.

Vous pouvez utiliser n'importe quel langage.

Il est possible de le faire en suivant les règles de l'art, ou rapidement développer son propre module.

Les modules existants sont dans */usr/lib/python2.7/site-packages/ansible/modules/*

Il est relativement simple de s'en inspirer pour en créer un nouveau.

On peut se constituer une bibliothèque de modules personnels, qui peut être :
- dans le sous-répertoire *library* courant au playbook
- dans celui mentionné par la directive de configuration *library*
- dans celui mentionné par la variable d'environnement *ANSIBLE_LIBRARY*

Il ne faut pas se baser sur un module n'acceptant pas d'argument (par exemple le module *command*) s'il on veut que notre module en accepte.

Un module doit respecter un certain nombre de règles, notamment si vous souhaitez en faire bénéficier la communauté.

Définition de l'environnement de développement

Cloner le dépôt ansible :

```
git clone https://github.com/ansible/ansible.git
```

Se déplacer dans le répertoire correspondant :

```
cd ansible
```

Créer un environnement de dev virtuel :

```
virtualenv venv.
```

Activer l'environnement de dev :

```
source venv/bin/activate
```

Installer les dépendances :

```
pip install -r requirements.txt
```

Adapter le shell à l'environnement de dev :

```
source hacking/env-setup
```

Mise au point du module - explication des principales directives

Créer un nouveau fichier à l'endroit voulu (dans lib/ansible/modules/)

Ce script devra contenir les sections suivantes :

METADATA - information de version, statut

DOCUMENTATION - documentation, telle que présentée par ansible-doc

EXAMPLES - exemples d'utilisations

RETURN - description des valeurs renvoyées.

Fonction principale:

```
def run_module():
    # définition des arguments du module
    module_args = dict(
        name=dict(type='str', required=True),
        new=dict(type='bool', required=False, default=False)
    )

# définition de la variable de retour
    result = dict(
        changed=False,
        original_message='',
        message=''
    )

    # instanciation d'un objet de type AnsibleModule. C'est par son intermédiaire
    que l'on aura accès aux arguments et paramètres d'exécution.
    On définit également s'il support l'option --check
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    Pas de modification de l'environnement en mode de vérification (--check)
    if module.check_mode:
        return result

    # Cœur du module, c'est ici que le travail du module est réalisé et que la
    # variable result est définie
    result['original_message'] = module.params['name']
    result['message'] = 'goodbye'

    #Il s'agit ici de déterminer si les actions du module ont modifié la cible,
    # auquel cas il faut le signaler
    if module.params['new']:
        result['changed'] = True
```

```
    # On quitte en renvoyant la variable résultat
    module.exit_json(**result)
    # on a également à disposition la méthode fail_json, qui permet de sortir en cas d'erreur
d'exécution.

def main():
    run_module()

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

Test direct

Il faut d'abord créer un fichier json contenant les arguments avec lesquels notre module sera appelé :

```
{
    "ANSIBLE_MODULE_ARGS": {
        "name": "hello",
        "new": true
    }
}
```

Puis activation de l'environnement virtuel et exécution :

- **source venv/bin/activate**
- **source hacking/env-setup**
- **python ./my_new_test_module.py /tmp/args.json**

Test via un playbook

Il suffit de mettre au point un playbook utilisant notre module :

```
- name: test my new module
  connection: local
  hosts: localhost
  tasks:
  - name: run the new module
    tom:
      text: 'hello'
    register: testout
  - name: dump test output
    debug:
      msg: '{{ testout }}'
```

Et de le lancer via la commande ansible-playbook

```
▶ ansible-playbook play.yml
PLAY [test my new module]
...

ok: [localhost] => {
    "msg": {
        "changed": false,
        "failed": false,
        "text": "hello"
    }
}

PLAY RECAP
*********************************************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
```

8.3- Créer ses propres filtres

Ref : https://docs.ansible.com/ansible/latest/playbooks_filters.html
https://docs.ansible.com/ansible/latest/dev_guide/developing_plugins.html

Les filtres permettent de transformer des variables au moment de leur utilisation.

Ils sont principalement définis dans le fichier /usr/lib/python2.7/site-packages/ansible/plugins/filter/core.py

Il est possible de définir nos propres filtres dans un fichier localisé dans le répertoire défini par la directive de configuration ansible ***filter_plugins***

```
▶ grep filter_plugin /etc/ansible/ansible.cfg
filter_plugins     = ~tom/.ansible/plugins/filter_plugins

▶ cat ~tom/.ansible/plugins/filter_plugins/myfilters.py
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type


def plopify(a):
    return "plop " + a.upper() + " plop"

class FilterModule(object):
```

```
    def filters(self):
        return {
            'plopify': plopify
        }
```

▶ **cat test_filter.yml**

```
---
- hosts: localhost
  connection: local
  become: true
  remote_user: tom
  handlers:
  - name: echo handler
    command: echo "i have been summoned"

  tasks:
- debug: msg="la machine {{ ansible_hostname }} est une {{ ansible_distribution|plopify }}"
```

▶ **ansible-playbook test_filter.yml**

```
ok: [localhost] => {
    "msg": "la machine cafeine est une plop FEDORA plop"
}
```

8.4- Index lexical