



Samuel ANTUNES
Consultant Ingénieur DevSecOps
OCTO Technology
Email : contact@samuelantunes.fr



1. Concepts de base & Manips

- a. Quésako ?
- b. Au service du DevOps
- c. L'installation
- d. Les inventaires
- e. Les commandes simples

2. Déploiement avec Ansible & Bonnes pratiques

- a. Les modules
- b. Les playbooks
- c. Les plays
- d. Les tâches (tasks)
- e. Écrire et lancer un déploiement

3. Déploiement Dynamique

- a. D'autres modules
- b. Les variables
- c. Les templates et filtres Jinja2

4. Modularisation du Code

- a. Les handlers & Notify
- b. Les rôles réutilisables
- c. Les tags

“Concepts de
base & Manips”

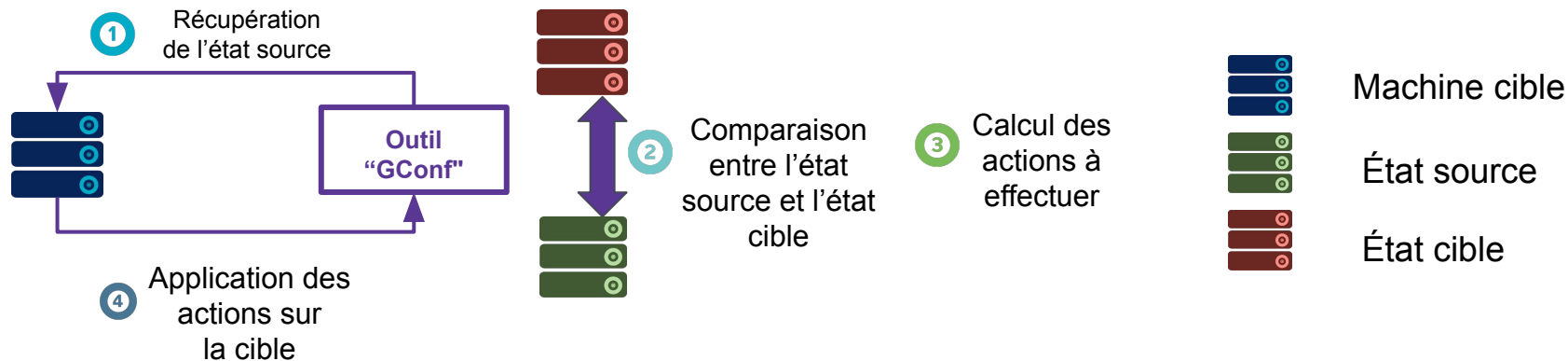
- De nombreuses problématiques liées aux applications
 - La portabilité des applications
 - La distribution des applications
 - Le besoin de décorrélérer applications et infrastructure
 - La rationalisation des infrastructures
- La montée en puissance
 - Des solutions de PaaS
 - De la philosophie DevOps

La gestion de configuration ou "GConf"

La **gestion de configuration** consiste à définir un **état cible** à atteindre. L'outil de "GConf" calcule ce qu'il y a comme actions à effectuer et les applique sur la cible.

Par exemple, tel serveur doit :

- ▷ être accessible avec l'utilisateur ansible
- ▷ avoir l'accès root désactivé
- ▷ avoir un répertoire X accessible uniquement par un utilisateur Y



Le déploiement automatisé

Un outil de déploiement **orchestre** les différentes étapes liées à un déploiement applicatif.

Par exemple, pour un déploiement avec interruption de service :

- ▷ arrêt de l'application
- ▷ mise à jour du schéma de base
- ▷ mise à jour des configurations applicatives
- ▷ recopie du binaire applicatif
- ▷ démarrage de l'application
- ▷ vérification de la disponibilité du service.

Il est **important** d'appliquer les actions dans un ordre **précis**.

Ansible : à la croisée de la GConf et de l'automatisation

Artefact applicatif

Middleware

Système d'exploitation

Machine

*Outil de
déploiement*

*Outil de
configuration*

*Provisioning de
l'infra*

- Déploiement automatisé
 - **Déploiement d'artefacts immuables**
 - Gestion de la configuration applicative
 - Exemple outils : **Ansible**, XL Deploy, Capistrano
- Gestion de configuration (« Gconf »)
 - **Configuration de l'OS et des middlewares**
 - Middleware : « WebLogic », « Oracle DB », ...
 - Exemple d'outils : **Ansible**, Puppet, Chef, Salt
- Infrastructure as a Service (pilotable par **Ansible**)
 - **Création et interconnexion des ressources matériels**
 - Gestion de l'OS
 - Exemple d'outils : VmWare, OpenStack, Terraform

Ansible est un outil à la croisée de la
Gestion de Configuration
et du **Déploiement automatisé**.

Ansible : l'aboutissement d'une histoire

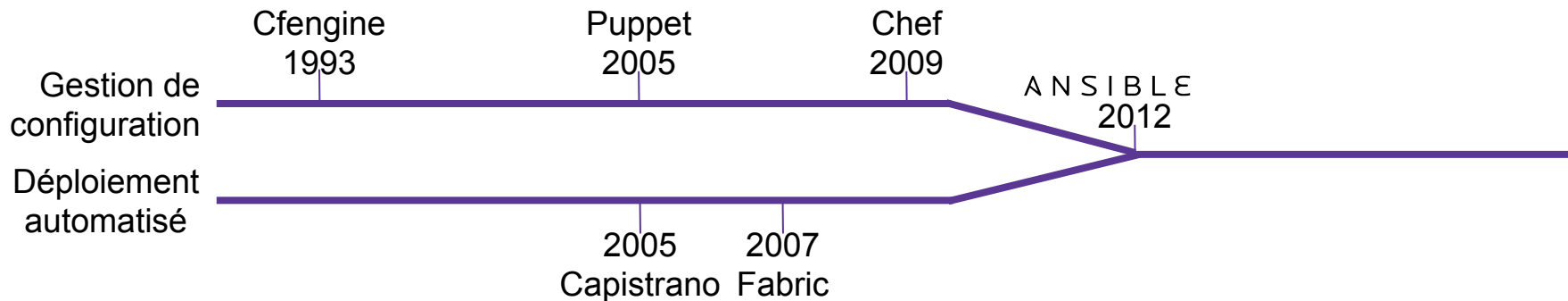
Ansible est né en 2012, bien après ses concurrents

▷ Gestion de configuration

- > Cfengine (1993)
- > Puppet (2005)
- > Chef (2009)

▷ Déploiement automatisé

- > Capistrano (2005)
- > Fabric (2007)

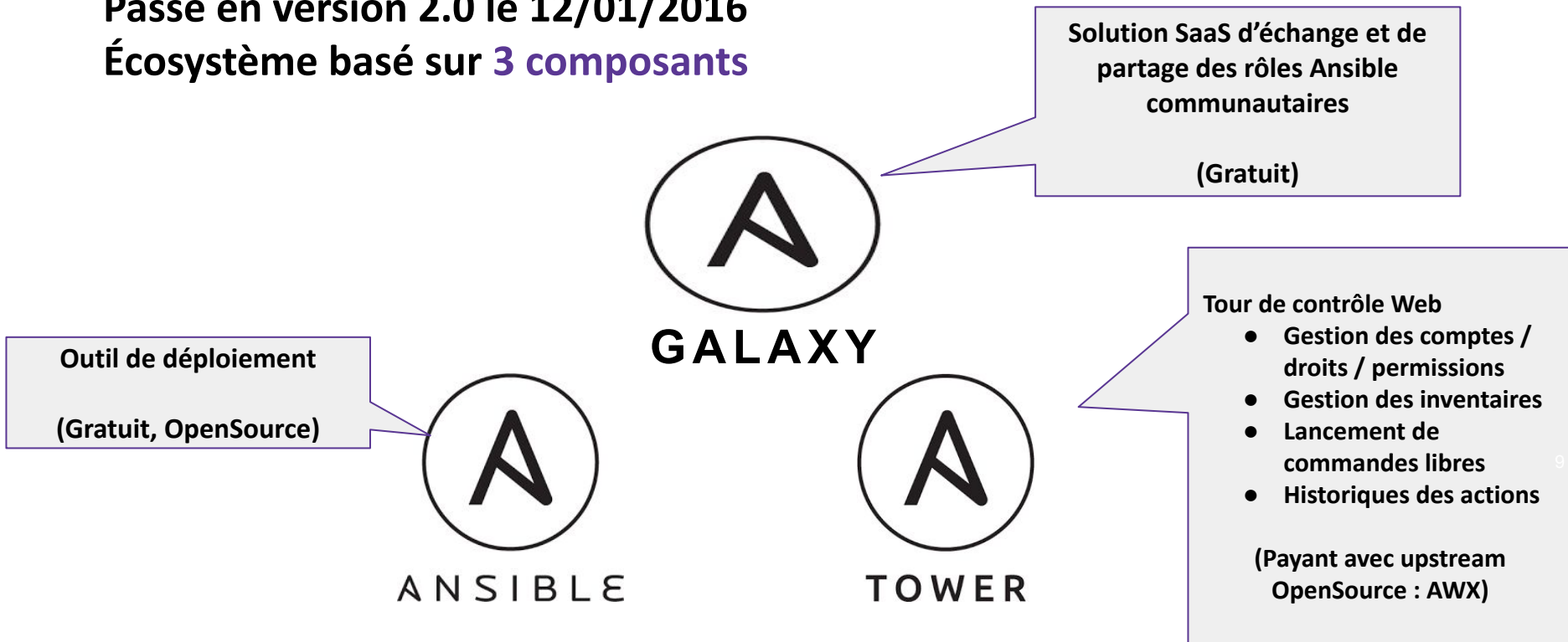


9 La galaxie "Ansible" aujourd'hui

Racheté par RedHat le 16/10/2015

Passé en version 2.0 le 12/01/2016

Écosystème basé sur 3 composants



Ansible Galaxy : une communauté autour d'Ansible

Ansible Galaxy est composé :

- ▷ d'une commande : "ansible-galaxy"
- ▷ d'une IHM Web (<https://galaxy.ansible.com/>) pour chercher des "recettes"

Nom de la recette : nginx

Description : ansible role nginx

Auteur : Author bennojoy

OS sur lesquels la recette a été testée : Platforms EL, Fedora, Ubuntu

Tags : web

Created : 12/19/13 1:23 AM

Last Imported : NA

Nombre de téléchargements : 198

Visibilité : Watch 7

Appréciation : Star 63

Les recettes de la galaxy peuvent servir de base ou d'inspiration.

Tip : regardez l'auteur (entreprise, recettes écrites) et l'appréciation pour choisir la "meilleure"

Ansible, ça permet de :

- ▷ **Lancer des commandes sur plusieurs machines**
 - > Comme un SSH en parallèle
- ▷ **Gérer les configurations des machines**
 - > Installer des logiciels
 - > Gérer des configurations (fichiers, services, ...)
- ▷ **Orchestrer des déploiements d'applications**
 - > Potentiellement complexes
 - > En synchronisant les actions entre plusieurs machines
- ▷ **Provisioner des environnements**
 - > IaaS / Cloud : AWS, Azure, Google Compute Engine, OpenStack, VMWare...
- ▷ **Piloter des appliances**
 - > F5 Big IP, Junos,

Et de manière concrète ?

Avant tout un outil en **ligne de commande**

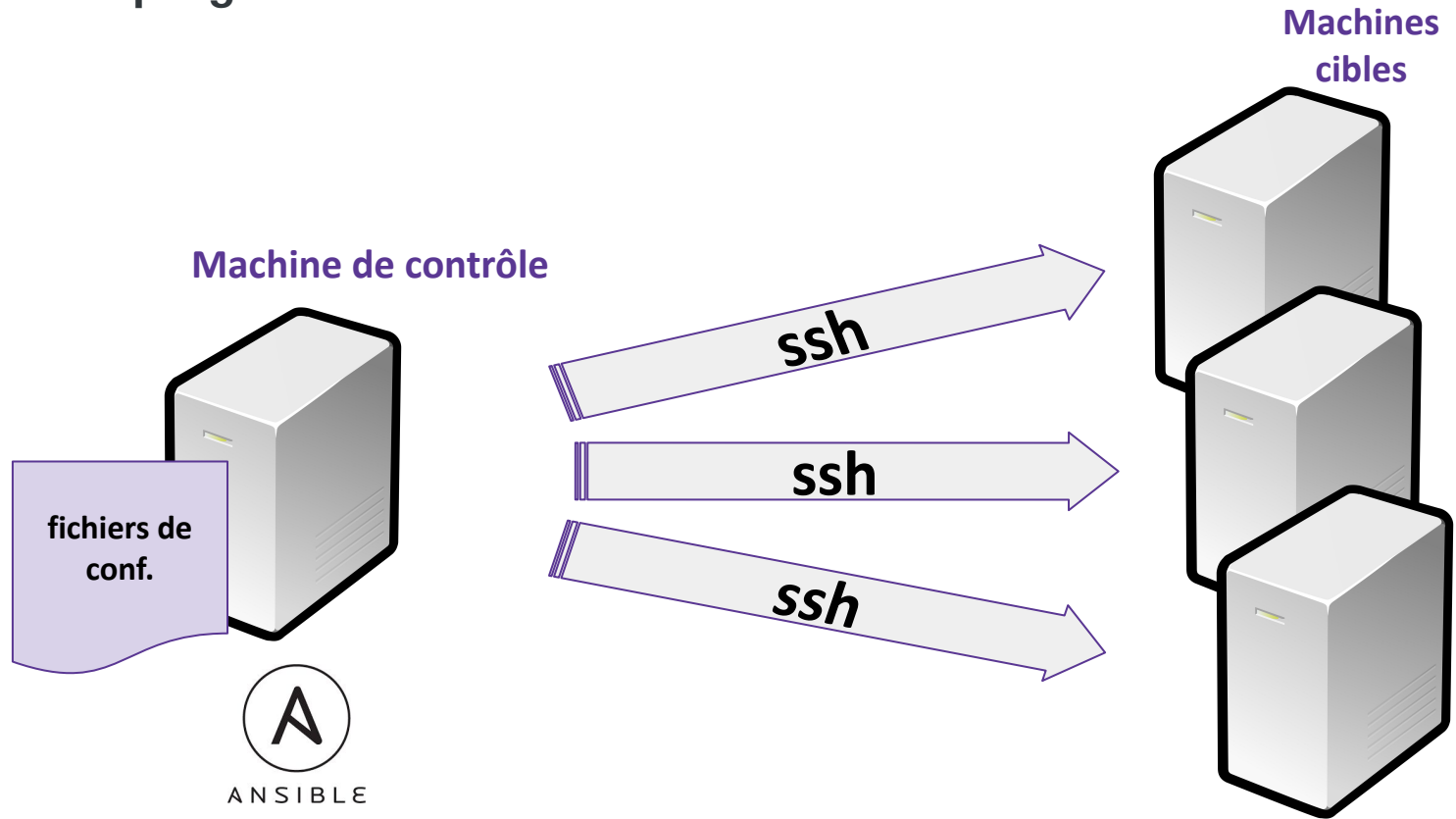
Ansible s'installe sur un serveur UNIX / Linux, appelée la **machine de contrôle**

- ▷ Écrit en **Python**
- ▷ Fonctionne sans serveur persistant
- ▷ En mode «**Push**»

Ansible permet de piloter des machines Linux / UNIX

- ▷ Via SSH par défaut (d'autres protocoles sont disponibles)
- ▷ Les machines contrôlées doivent avoir **Python** (version 3) installé
- ▷ Il n'y a pas d'agent sur les machines pilotées

Ansible lit des **fichiers de configuration** qui contiennent les instructions de déploiement



Connexion SSH aux machines

La configuration de ssh, tant côté client que serveur peut être très poussée sous Linux

- ▷ type d'authentification (mot de passe, clé, agent SSH)
- ▷ rebond (proxycommand)
- ▷ alias des machines

Sur la machine de contrôle, nous recommandons **l'utilisation** d'un fichier **`$HOME/.ssh/config`**

Ansible fait abstraction de toute cette configuration et suppose que la connexion SSH est simplement possible

Les connexions SSH par clés

SSH permet les connexions aux machines à l'aide de clés SSH.

La mécanique utilisée repose sur **une paire de clés** (RSA par exemple) forgée sur la machine de contrôle :

- ▷ la **clé publique** est installée sur la machine cible
 - > dans `$HOME/.ssh/authorized_keys`
- ▷ la **clé privée** est sur la machine de contrôle
 - > dans `$HOME/.ssh/id_rsa`

SSH permet également les **connexions aux machines** à l'aide d'un **agent SSH**

- ▷ ssh-agent
- ▷ gnome-keyring-daemon

L'agent monte en mémoire la clé privée préalablement installée

Le client “ssh” local communique avec l'agent pour faire les opérations de cryptographie

Une fois connecté sur une machine, **SSH peut propager l'accès à l'agent** pour permettre les **rebonds** sans avoir à **installer les clés privées sur la machine** intermédiaire

Et pour Windows ?

Le **niveau de maturité** du produit pour Windows est **moyen**.

La **connexion** pour Windows se fait via **WinRM** avec :

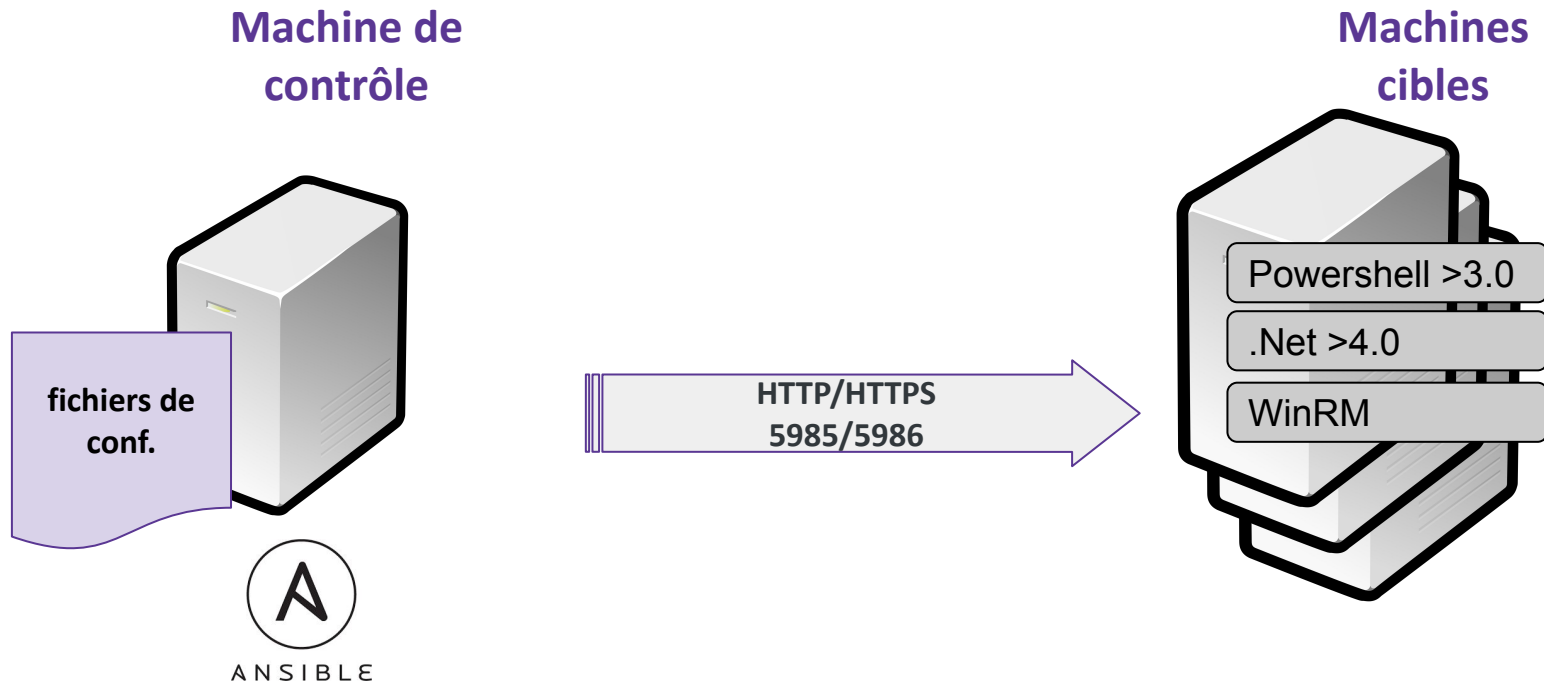
- ▷ Login / Mot de passe
- ▷ Kerberos

Les machines cibles doivent être configurées pour accepter les connexions

- ▷ Service “WinRM” à lancer et configurer (HTTP/HTTPS)
- ▷ PowerShell version 3.0 et .NET Framework 4.0 au minimum
- ▷ Ouverture du firewall

Ansible exécute des scripts **PowerShell** sur les machines cibles

Cinématique générale d'Ansible avec des cibles Windows



Points d'attention avec Windows

Ansible **ne peut pas s'exécuter** sur une **machine de contrôle Windows**.

Cependant, Il est possible de l'exécuter via l'émulation Linux :

- ▷ Windows Subsystem for Linux (WSL)
- ▷ Cygwin

Des fonctions basiques sont disponibles :

- ▷ Configuration OS
 - **chocolatey**, gestionnaire de package windows (mais dépendant de la livraison des packages/versions par la communauté)
 - Installer/désinstaller des features Windows (par ex : IIS)
- ▷ Déploiement applicatif sur IIS

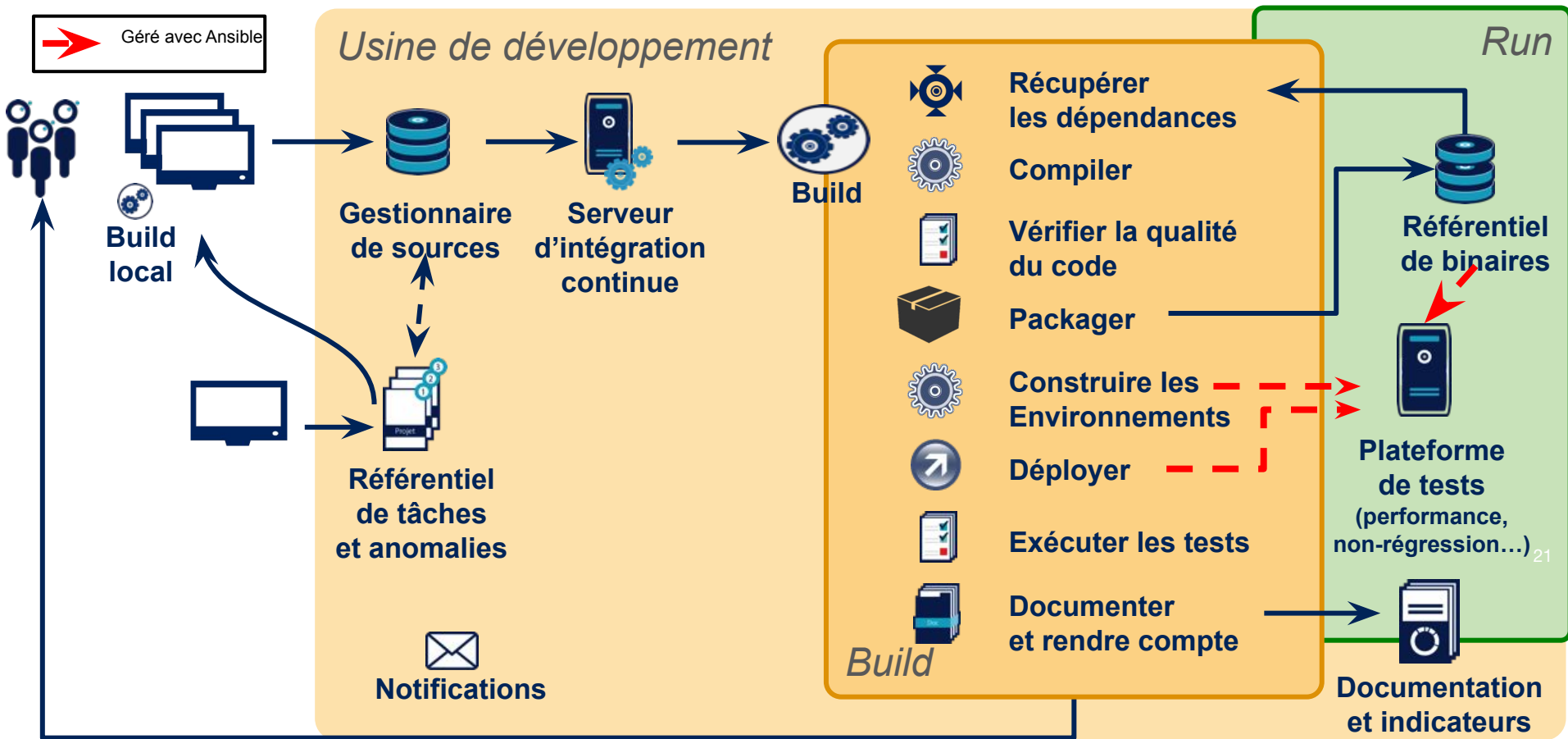
Mais le **nombre d'actions possibles reste limité** comparé à des machines cibles Linux.

La mise en place d'un certificat pour une connection HTTPS avec winRM peut être complexe à

Nous ne conseillons pas d'utiliser Ansible dans le cas d'un parc majoritairement sous Windows

“ Ansible et
DevOps ”

Utilisation d'Ansible dans une Usine de Développement



“Installation”

Installation d'Ansible

En fonction des distributions et du contexte, il y a plusieurs moyens d'installer Ansible

▷ **Ansible est disponible dans la bonne version sous forme d'un paquet officiel**

> CentOS, RedHat :

```
yum install ansible
```

> Debian, Ubuntu :

```
apt install ansible
```

▷ **Ansible étant écrit en Python, il peut également être installé avec l'utilitaire pip**

```
pip install ansible
```



Dans certains cas, l'installation peut demander plus de travail (permissions, proxy, firewall, pas d'accès à Internet ou aux dépôts publics...)

```
git clone https://5AIW2:d-Gx32tFcduyu-y92ein@gitlab.com/santunes-formations/ansible.git
```


TP #1

“ Les Inventaires ”

Les inventaires

Les **inventaires** constituent une partie des fichiers de **configuration d'Ansible**

Ils permettent de décrire les machines à contrôler

Les fichiers d'inventaire sont au format INI

Ils décrivent :

- ▷ **Les machines**
 - > par leur nom
 - > par leur adresse IP

- ▷ **Les groupes auxquels les machines appartiennent**
 - > une machine peut appartenir à plusieurs groupes
 - > un groupe peut être constitués d'autres groupes

Exemple d'inventaire Ansible



Inventaire

```
[load-balancers]
lb1-srv
lb2-srv

[web-servers]
app1-srv
app2-srv

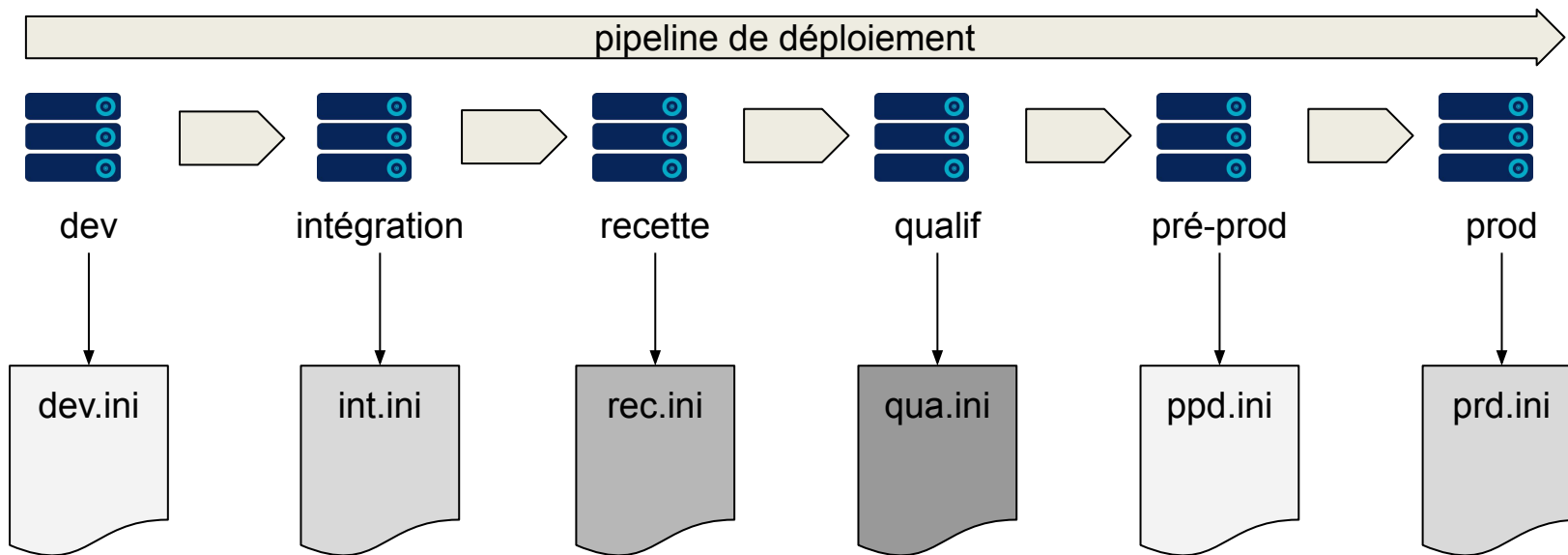
[db-masters]
db1-srv

[db-slaves]
db2-srv

[db-servers:children]
db-masters
db-slaves
```

- Le fichier d'inventaire contient la **topologie** des serveurs et de leur(s) rôle(s) applicatif(s)
- Fichier **simple à parcourir** manuellement ou automatiquement, et à **générer**
- Les machines appartiennent toutes à un groupe «**all**»

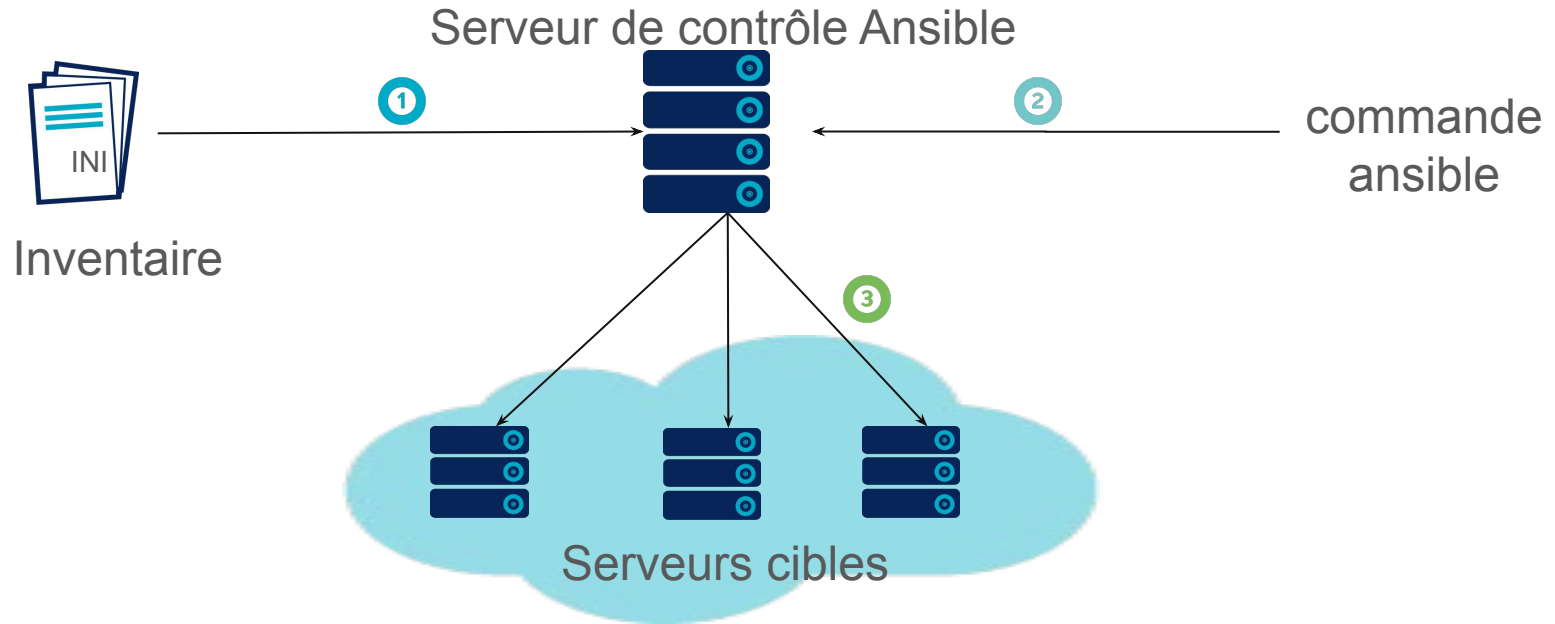
Différents inventaires permettent de représenter chaque environnement que l'on souhaite déployer



“ Les commandes simples ”

Utilisation des commandes Ansible simple

La machine de contrôle lance des commandes sur un inventaire *listant* les serveurs cibles



32 Exemples de commandes simples

Syntaxe générale de la commande ansible pour lancer une commande sur plusieurs machines

```
ansible -i <fichier d'inventaire> <filtre> -a "commande"
```

```
ansible -i inv1.ini all -a "hostname"
```

```
ansible -i inv1.ini all -b -a "whoami"
```

La notion de filtre permet de choisir un groupe de machines concernées. `all` contient toutes les machines d'un inventaire.

Exemples d'autres fonctions

Exemple3 : simplement tester la connexion ssh avec les machines

```
ansible -i inv1.ini all -m ping
```

Exemple4 : Lister les membres d'un groupe (ne se connecte pas aux machines)

```
ansible -i inv1.ini groupe1 --list-hosts
```

L'utilisation de l'option **--list-hosts** permet de s'assurer que le fichier d'inventaire est syntaxiquement valide. Pourquoi ne pas vérifier automatiquement que tous les fichiers d'inventaires sont valides ?

Exemples d'autres fonctions

Exemple5 : collecte de propriétés des machines

```
ansible -i inv1.ini all -m setup
```

La collecte des propriétés des machines porte le nom de collecte de «facts» (facts gathering)

C'est une fonction importante d'Ansible qui permet de remonter des caractéristiques des machines

- ▷ **Nombre de CPUs**
- ▷ **RAM**
- ▷ **Type d'OS / distribution**
- ▷ **Adresse(s) IP de la machine**
- ▷ **...**

La notion de facts d'Ansible est comparable à celle de Chef (ohai) ou Puppet (facter)

Le fichier “ansible.cfg”

“ansible.cfg” est une **facilité** offerte pour l’outil.

Le fichier “ansible.cfg” :

- ▷ est recherché dans le répertoire courant
- ▷ contient différentes options passables en ligne de commande
- ▷ évite d’avoir à préciser systématiquement toutes les options

La documentation d’Ansible détaille avec précision les nombreuses options possibles.

```
ansible.cfg
```

```
[defaults]
```

```
force_color = 1
```

```
inventory = inventories/inv.ini
```

```
git clone https://5AIW2:d-Gx32tFcduyu-y92ein@gitlab.com/santunes-formations/ansible.git
```

TP #2

Pour aller plus loin que les simples commandes...

Nous n'avons pour l'instant que très peu exploité les capacités des **modules** (ping, setup, shell et bien d'autres).

Tout l'intérêt d'Ansible est de faire travailler ensemble ces modules afin d'atteindre un **état cible**.

Nous allons donc regrouper les appels à ces modules dans des **playbooks**.

Un playbook est constitué d'une liste de **plays**, chaque **play** regroupant un ensemble de **tâches** à effectuer sur un groupe de machine.

L'unité de base, la **tâche**, correspond à l'appel à un **module** Ansible.

Les tâches, plays et les fichiers playbooks sont écrits au format YAML

Quelques mots sur YAML...

YAML est un format de sérialisation de données compatible avec JSON

Il se veut « plus lisible par des humains »

Il permet de représenter des structures de données

- ▷ Simples / scalaires (entiers, flottants, chaînes, booléens)
- ▷ Complexes (tableaux, dictionnaires)

```
# un tableau
- valeur1
- valeur2
- valeur3
```

```
# un dict
cle1: val1
cle2: val2
cle3: val3
```

```
# composition de ces structures
fr:
- cle11: valeur 1
  cle12: true
  cle13: 666
- cle21: 3.12159
  cle22: 666
eng:
- un mot
- 3.14
```

“ Les modules et les tâches ”

Les modules

- Il y a des centaines de modules disponibles avec l'installation standard d'Ansible
- Ils sont regroupés en familles
- Certains nécessitent des programmes ou bibliothèques Python spécifiques sur les machines cibles
- Certains modules ne concernent que certains OS
- Vous pouvez écrire vos propres modules
- Les modules sont invoqués dans des tâches (au format YAML)



- All Modules
- Cloud Modules
- Clustering Modules
- Commands Modules
- Crypto Modules
- Database Modules
- Files Modules
- Identity Modules
- Inventory Modules
- Messaging Modules
- Monitoring Modules
- Net Tools Modules
- Network Modules
- Notification Modules
- Packaging Modules
- Remote Management Modules
- Source Control Modules
- Storage Modules
- System Modules
- Utilities Modules
- Web Infrastructure Modules
- Windows Modules

Exemple pour la section Database

Misc

- `mongodb_user` (E) - Adds or removes a user from a MongoDB database.
- `redis` (E) - Various redis commands, slave and flush
- `riak` (E) - This module handles some common Riak operations

Mysql

- `mysql_db` - Add or remove MySQL databases from a remote host.
- `mysql_replication` (E) - Manage MySQL replication
- `mysql_user` - Adds or removes a user from a MySQL database.
- `mysql_variables` - Manage MySQL global variables

Postgresql

- `postgresql_db` - Add or remove PostgreSQL databases from a remote host.
- `postgresql_ext` (E) - Add or remove PostgreSQL extensions from a database.
- `postgresql_lang` (E) - Adds, removes or changes procedural languages with a PostgreSQL database.
- `postgresql_privs` - Grant or revoke privileges on PostgreSQL database objects.
- `postgresql_user` - Adds or removes a users (roles) from a PostgreSQL database.

Vertica

- `vertica_configuration` (E) - Updates Vertica configuration parameters.
- `vertica_facts` (E) - Gathers Vertica database facts.
- `vertica_role` (E) - Adds or removes Vertica database roles and assigns roles to them.
- `vertica_schema` (E) - Adds or removes Vertica database schema and roles.
- `vertica_user` (E) - Adds or removes Vertica database users and assigns roles.

Note

- (D): This marks a module as deprecated, which means a module is kept for backwards compatibility but usage is discouraged. The module documentation details page may explain more about this rationale.
- (E): This marks a module as 'extras', which means it ships with ansible but may be a newer module and possibly (but not necessarily) less actively maintained than 'core' modules.
- Tickets filed on modules are filed to different repos than those on the main open source project. Core module tickets should be filed at [ansible/ansible-modules-core](https://github.com/ansible/ansible-modules-core) on GitHub, extras tickets to [ansible/ansible-modules-extras](https://github.com/ansible/ansible-modules-extras) on GitHub

Anatomie d'une tâche

```
- name: Fait un truc  
  nom_du_module:  
    argument1: valeur1  
    argument2: "valeur 2"
```

Nom / description de la tâche
(texte libre)

Le nom du module à invoquer

Les paramètres à transmettre
au module

Les modules décrivent un état désiré (Desired State Configuration)

- Je veux que le paquet soit dans un état «installé»
- Je veux que le fichier soit dans un état «absent»

Une tâche, plusieurs syntaxes

```
# Tâche
- name: Description appel au module
  nom_du_module:
    argument1: truc
    argument2: tric
```

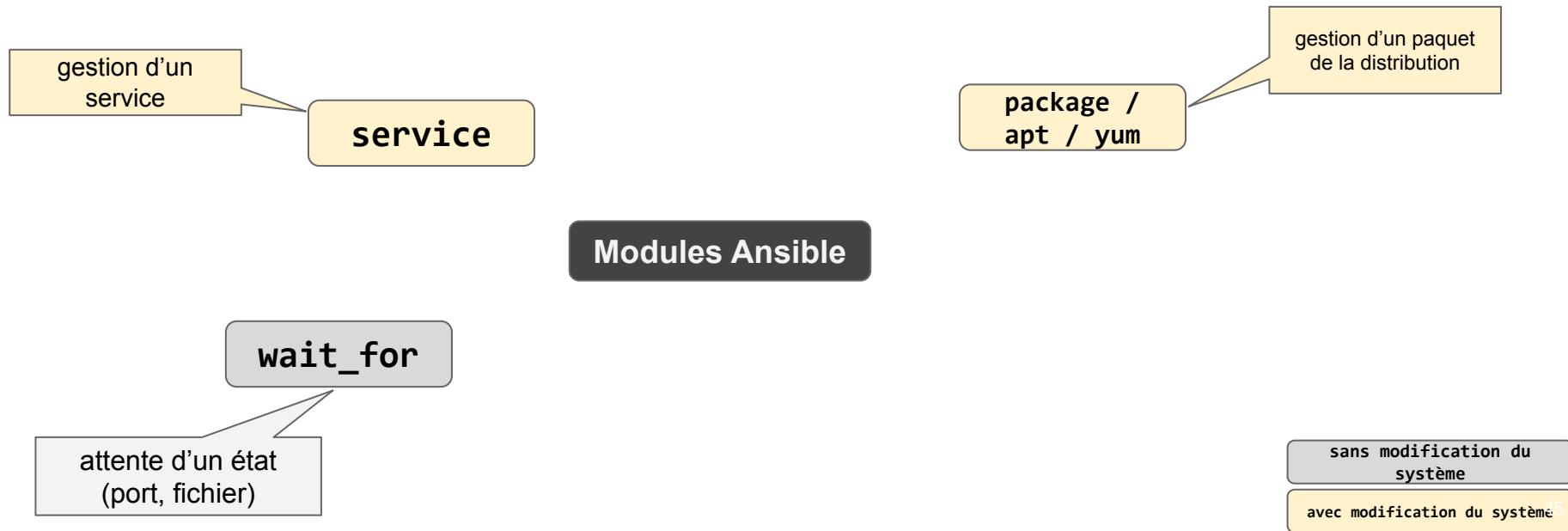
Variante à
utiliser

```
# Tâche
- name: Description appel au module
  nom_du_module: argument1=truc argument2=tric
```

```
# Tâche
- name: Description appel au module
  action: nom_du_module argument1=truc argument2=tric
```

Le «name» d'une tâche est théoriquement facultatif, mais nous recommandons de **toujours en préciser un** : il permet de documenter la tâche, et de la retrouver plus facilement.

Quelques exemples de modules



Chaque module est documenté sur le site d'Ansible à l'adresse
[http://docs.ansible.com/ansible/\\${nom du module} module.html](http://docs.ansible.com/ansible/${nom du module} module.html)
(ex : http://docs.ansible.com/ansible/service_module.html)

Exemple de documentation : service (1/2)

Synopsis

Controls services on remote hosts. Supported init systems include BSD init, OpenRC, SysV, Solaris SMF, systemd, upstart.

Options

parameter	required	default	choices	comments
arguments	no			Additional arguments provided on the command line aliases: args
enabled	no		<ul style="list-style-type: none"> yes no 	Whether the service should start on boot. At least one of state and enabled are required.
name	yes			Name of the service.
pattern	no			If the service does not respond to the status command, name a substring to look for as would be found in the output of the <code>ps</code> command as a stand-in for a status result. If the string is found, the service will be assumed to be running.
runlevel	no	default		For OpenRC init scripts (ex: Gentoo) only. The runlevel that this service belongs to.
sleep (added in 1.3)	no			If the service is being <code>restarted</code> then sleep this many seconds between the stop and start command. This helps to workaround badly behaving init scripts that exit immediately after signaling a process to stop.
state	no		<ul style="list-style-type: none"> started stopped restarted reloaded 	<code>started</code> / <code>stopped</code> are idempotent actions that will not run commands unless necessary. <code>restarted</code> will always bounce the service. <code>reloaded</code> will always reload. At least one of state and enabled are required.

Exemple de documentation : service (2/2)

Examples

Example action to start service httpd, if not running

- service: name=httpd state=started

Example action to stop service httpd, if running

- service: name=httpd state=stopped

Example action to restart service httpd, in all cases

- service: name=httpd state=restarted

Example action to reload service httpd, in all cases

- service: name=httpd state=reloaded

Example action to enable service httpd, and not touch the running state

- service: name=httpd enabled=yes

Example action to start service foo, based on running process /usr/bin/foo

- service: name=foo pattern=/usr/bin/foo state=started

Example action to restart network service for interface eth0

- service: name=network state=restarted args=eth0

Formalisme recommandé :

```
- name: Ensure httpd is started
service:
  name: httpd
  state: started
```

This is a Core Module

For more information on what this means please read [Core Modules](#)

Les informations de retour d'un module

L'invocation d'un module fournit des informations de retour qui peuvent être exploitées dans la suite du traitement Ansible

Informations retournées

- ▷ **changed** (booléen) : retournée par défaut
 - > Est-ce que l'exécution du module a donné lieu à un changement ?
 - Non (état désiré, pas d'action) => false
 - Oui (non-conformité, action) => true

- ▷ **failed** (booléen) : retournée en cas d'erreur uniquement
 - > Est-ce que l'exécution du module a retourné une erreur ?
 - Oui => true

Par défaut, un module qui échoue sur une machine la met dans un état d'erreur. Ansible s'arrête lorsqu'il atteint une tâche dont toutes les cibles sont en erreur.

Pour plus tard : il est possible de modifier les retours (**changed** et **failed**) des tâches pour affiner la gestion de l'idempotence.

“ Les playbooks et les plays ”

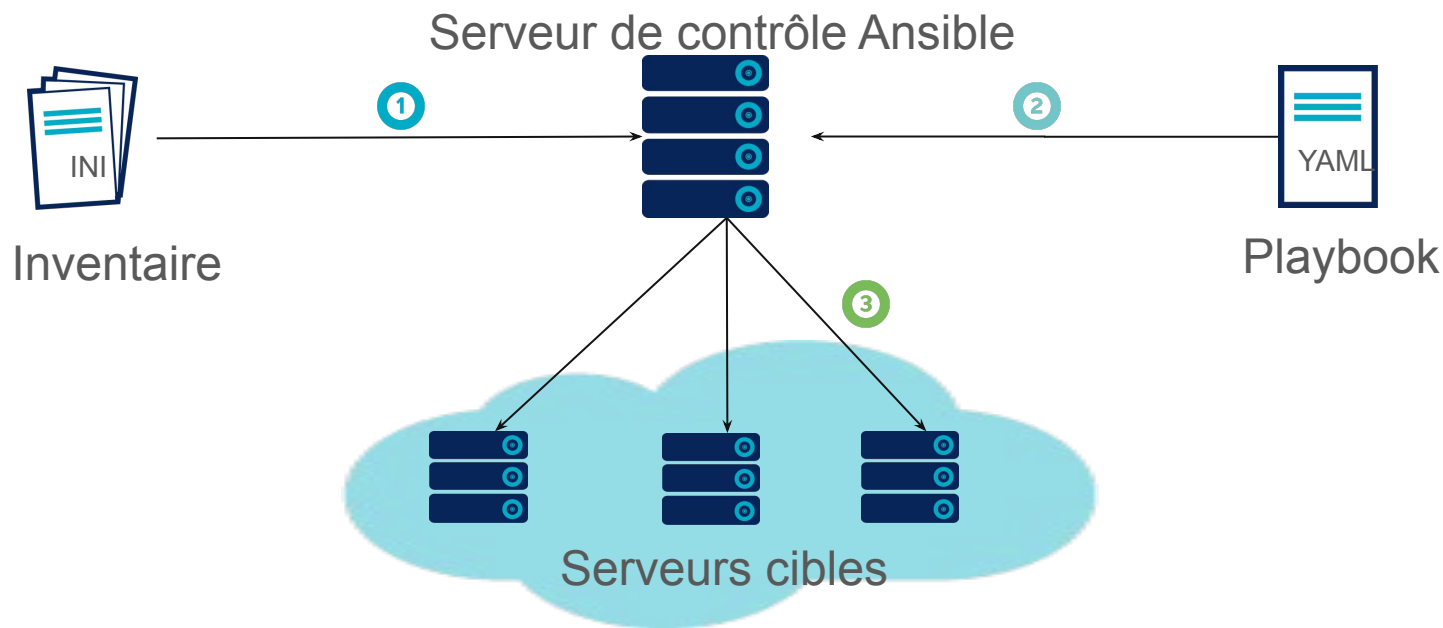
Objectif d'un playbook

Les objectifs d'un playbook sont :

- ▷ **De décrire un enchaînement d'appels à des modules Ansible sous forme d'un fichier rejouable**
- ▷ **D'attribuer les commandes à jouer suivant l'appartenance des machines à un ou des groupes**
- ▷ **D'éviter de taper des commandes à la main**
 - > et donc de faire des erreurs

Un playbook est un fichier de `code d'infrastructure` et devra être traité comme tel et placé dans un gestionnaire de sources dès que possible.

Un serveur exécute un playbook sur un inventaire contenant les serveurs cibles



Plays et Playbooks

```
# playbook install.yml
```

```
---
```

```
# Mon super play
```

```
- hosts: all
```

```
  tasks:
```

```
    - name: Fait un truc
```

```
      module1:
```

```
        argument: valeur44
```

```
    - name: Fait un autre truc
```

```
      module2:
```

```
        argument: valeur66
```

```
# Un autre play
```

```
- hosts: groupe2:groupe3
```

```
  tasks:
```

```
    - name: Fait un machin
```

```
      module3:
```

```
        argument: valeur99
```

Les machines ciblées par
le play

une tâche

une autre tâche

un play

un autre play

Remarques sur les tâches et plays

Une **tâche** doit être intégralement exécutée sur toutes les machines concernées avant de passer à la tâche suivante.

Une **tâche** s'exécute en **parallèle** sur **N hosts**. N est défini via la ligne de commande (option -f) ou via le fichier ansible.cfg (défaut à 5).

Un **play** doit être intégralement exécuté sur toutes les machines concernées avant de passer au play suivant.

Il est possible de forcer un play à être joué :

- ▷ En série

```
- hosts: groupe666
  serial: 1
  tasks:
    - ...
```

- ▷ Par itération sur un pourcentage des machines concernées

```
- hosts: groupe666
  serial: 67%
  tasks:
    - ...
```

Play et utilisateurs

Un play peut être lancé en tant que root (similaire à sudo en shell) avec l'option "become".

L'option "become" est l'équivalent de l'option « -b » lorsque l'on utilise la commande `ansible`

```
- hosts: all
  become: true
  tasks:
    - ...
```

Play et facts

Un play peut collecter les facts sur les machines.

Cela va lancer le module **setup** sur toutes les machines

```
- hosts: all
  gather_facts: true
  tasks:
  - ...
```

Cela permet de collecter des variables par machine et de les rendre disponibles dans les tâches.

Mais nous verrons cela un peu plus tard...

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

Toutes les machines

```
- ...
```

```
- hosts: all:!groupe1  
tasks:
```

Toutes les machines, sauf
celles du groupe1

```
- ...
```

```
- hosts: groupe1:groupe2:!groupe3  
tasks:
```

Les machines du groupe1 +
les machines du groupe2,
sauf celles qui sont dans le
groupe3

```
- ...
```

```
- hosts: groupe4[0]  
tasks:
```

La première machine du groupe4.
*Il faut faire attention à l'ordre des membres dans
les groupes*

```
- ...
```


Un playbook digne de ce nom !

Sur toutes les machines en tant que root :

- S'assurer que le paquet ntp est installé et activé
- S'assurer que le service ntp est démarré

Sur toutes les machines du groupe **web-servers** en tant que root :

- S'assurer que le paquet nginx est installé
- S'assurer que le service nginx est démarré et activé
- S'assurer qu'un processus écoute sur le port 80

```
# playbook install.yml
---
- hosts: all
  become: true
  gather_facts: true
  tasks:
    - name: Installation NTP
      package:
        name: ntp
    - name: Activation NTP
      service:
        name: ntp
        state: started
        enabled: true

- hosts: web-servers
  become: true
  tasks:
    - name: Installation NGinx
      package:
        name: nginx
    - name: Activation NGinx
      service:
        name: nginx
        state: started
        enabled: true
    - name: Attente port 80
      wait_for:
        port: 80
        host: 0.0.0.0
```

“Lancement d'un
playbook”

La commande ansible-playbook

```
$ ansible-playbook [options] <playbook>
```

Options fréquemment utilisées :

- ▷ **-b**
 - > Pour passer root :-) (ou tout autre utilisateur définit par --become-user)
- ▷ **-i <fichier d'inventaire>, sauf si ansible.cfg en définit un par défaut.**
 - > Pour préciser l'inventaire à utiliser
- ▷ **--diff**
 - > Affiche les changements qui vont être opérés sur les fichiers, sous forme de diff UNIX
- ▷ **--check**
 - > Ne fait rien, mais affiche ce qu'il faudrait faire (dry-run)
- ▷ **-v**
 - > Pour être verbeux (jusqu'à -vvvvv)

Lancer un playbook en mode --check --diff dans le doute...

La sortie de la commande ansible-playbook

1. Ansible se connecte à l'hôte, et récupère des informations sur l'hôte : des « facts »
2. Il traduit les tâches en commandes, qu'il transmet aux machines via SSH
3. Selon si l'action est exécutée et son résultat, Ansible retourne :

ok,
 changed,
 skipped,
 failed...

```

arno@arthur ~/tmp>ansible-playbook -i inv play.yml

PLAY [all] *****

GATHERING FACTS *****
ok: [lb2-srv]
ok: [lb1-srv]

TASK: [common | Install repo] *****
ok: [lb2-srv]
ok: [lb1-srv]

TASK: [common | Install NTP configuration file] *****
ok: [lb1-srv]
ok: [lb2-srv]

PLAY [load-balancers] *****

TASK: [haproxy | Install haproxy] *****
ok: [lb1-srv]
ok: [lb2-srv]

TASK: [haproxy | Install configuration file] *****
changed: [lb1-srv]
changed: [lb2-srv]

NOTIFIED: [haproxy | Reload haproxy] *****
changed: [lb1-srv]
changed: [lb2-srv]

PLAY RECAP *****
lb1-srv          : ok=6    changed=2    unreachable=0    failed=0
lb2-srv          : ok=6    changed=2    unreachable=0    failed=0
  
```

Un mot sur l'idempotence...

Les lignes marquées **changed** ou **ok** mettent en évidence le travail de convergence vers un état attendu et d'idempotence effectué par Ansible pour chaque tâche décrite dans un playbook

▷ **changed**

- > L'état constaté n'est pas l'état attendu de cette tâche, Ansible a dû effectuer des actions pour converger vers cet état

▷ **ok**

- > L'état constaté est l'état attendu, Ansible n'a pas eu à effectuer d'action pour cette tâche
- > idempotence de cette tâche

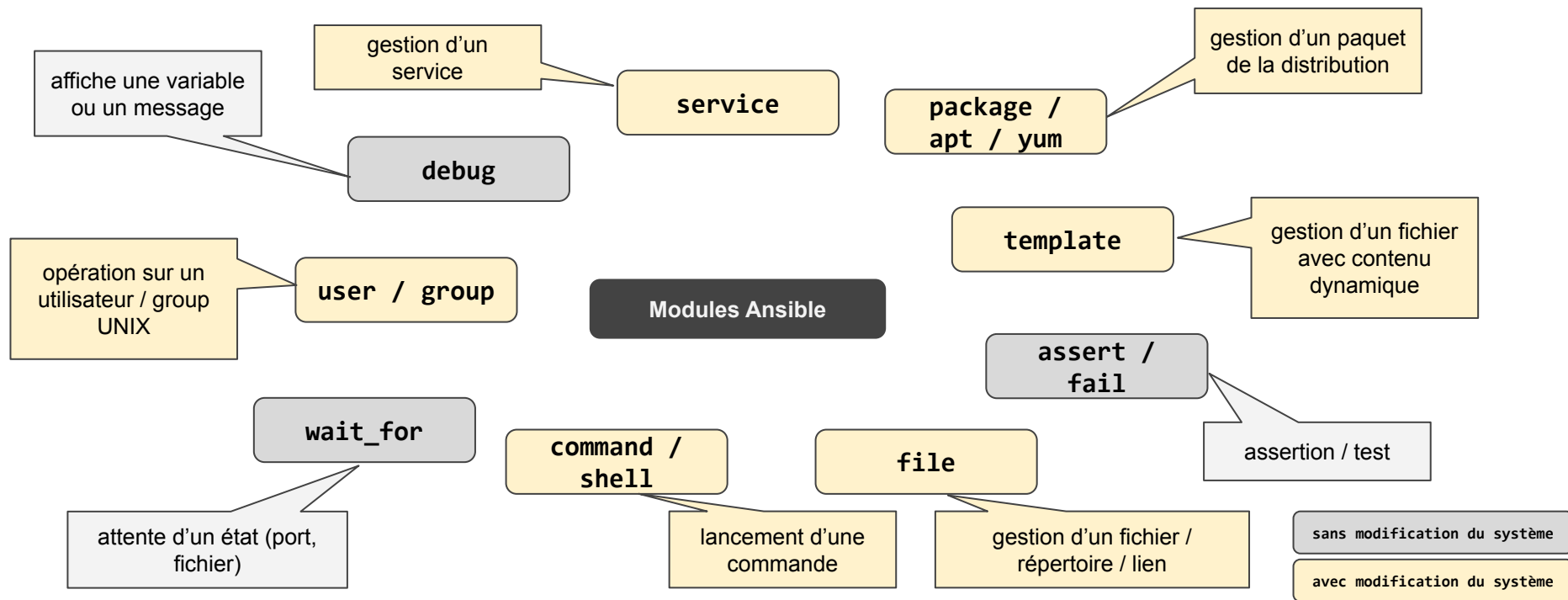
Du code Ansible bien écrit devrait toujours retourner aucun élément changé lors du second lancement successif

```
git clone https://5AIW2:d-Gx32tFcduyu-y92ein@gitlab.com/santunes-formations/ansible.git
```

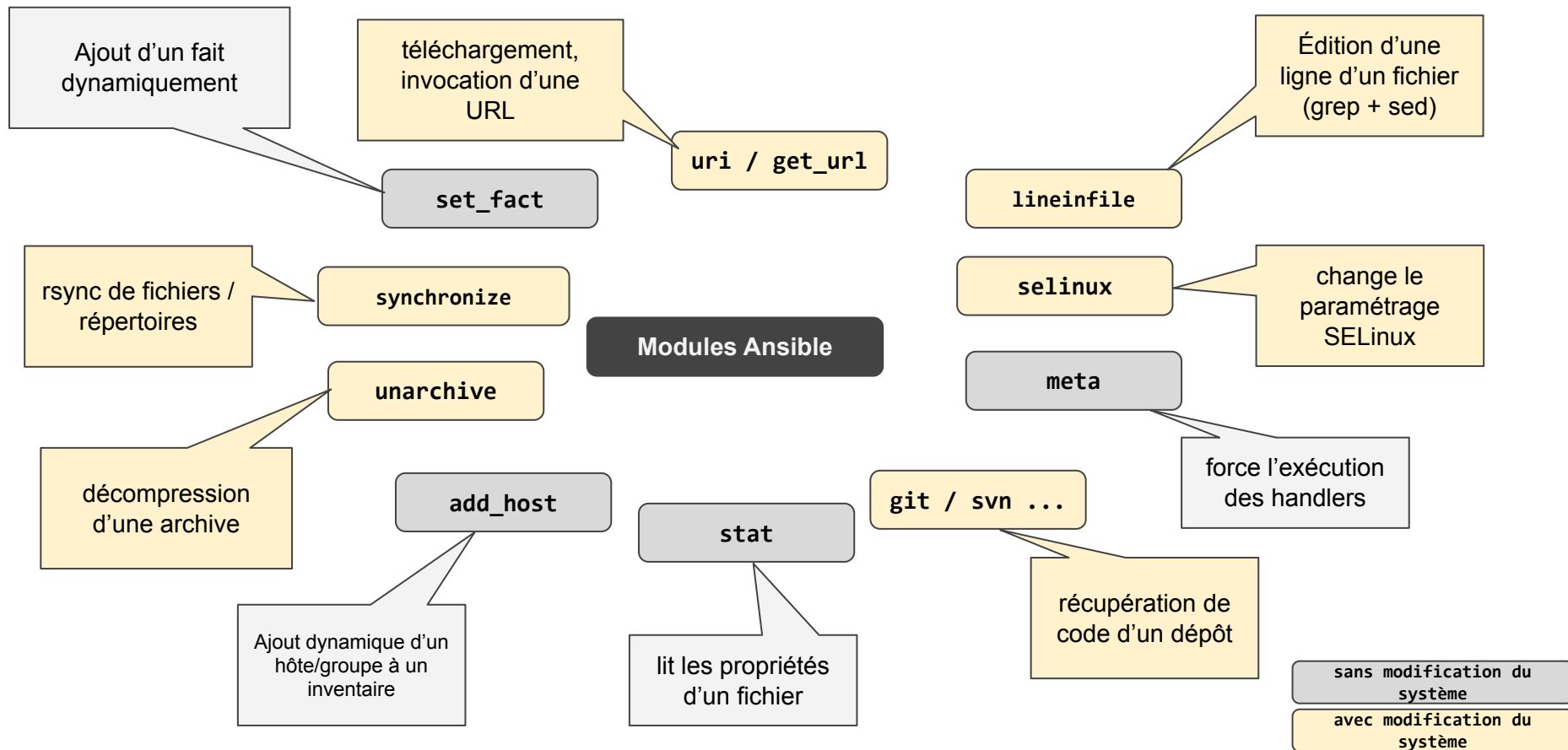
TP #3

“ D'autres
modules ”

Les modules de base utilisés



D'autres modules utilisables



“ Les variables dans Ansible ”

Les variables : premier contact

Ansible permet d'écrire des tâches sans connaître *a priori* certaines valeurs.

Ces tâches exploitent des variables disponibles dans le contexte de l'hôte cible et provenant de sources variées afin de construire leurs arguments.

```
- hosts: all
  tasks:
  - name: Récupération d'une archive
    get_url:
      url: http://{{ repo_ip }}/{{ archive_name }}
      dest: /tmp/archive.tar
```

Les variables `repo_ip` et `archive_name` peuvent varier d'un hôte/groupe/inventaire à l'autre

Les sources possibles de variables

▷ Au niveau d'un play :

```
- hosts: all
  vars:
    repo_ip: 127.0.0.1
    archive_name: archive_2016.tar
  tasks:
    ...
```

▷ Au niveau de la ligne de commande (-e/--extra-vars) :

```
ansible-playbook site.yml -e "repo_ip=127.0.0.1" -e "archive_name=archive_2016.tar"
```

▷ Au niveau de l'inventaire (par hôte et par groupe) :

```
[group1]
server1 archive_name=archive_2016.tar

[group1:vars]
repo_ip=127.0.0.1
```

Des “host vars” et “group vars” hors des inventaires !

Il existe une **meilleure façon** de gérer les variables d'hôtes et de groupes :
les fichiers “group_vars” et “host_vars”

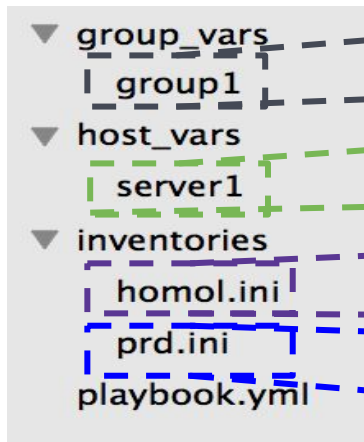
```
[group1]
server1 repo_ip=127.0.0.1

[group1:vars]
archive_name=archive_2016.tar
```

```
[group1]
server2

[group1:vars]
archive_name=archive_2016.tar
```

Devient



```
---
archive name: archive_2016.tar
```

```
---
repo_ip: 127.0.0.1
```

```
[group1]
server1
```

```
[group1]
server2
```

Les facts, des variables comme les autres

Comme évoqué en partie 1.2 une fois que les facts d'un hôte ont été collectés ils sont disponibles sous forme de variables dans le contexte de ce serveur.

Exemple :

Les facts sont automatiquement collectés pour les hôtes concernés par le play

```
- hosts: all
  tasks:
    - name: compilation de l'application
      command: make -j {{ ansible_processor_cores }}
```

Ils peuvent ensuite être exploités comme n'importe quelles variables

Liste exhaustive des sources de variable

Par priorité croissante (tiré de http://docs.ansible.com/ansible/playbooks_variables.html) :

role defaults (moins prioritaire)

inventory vars

inventory group_vars

inventory host_vars

playbook group_vars

playbook host_vars

host facts

registered vars

set_facts

play vars

play vars_prompt

play vars_files

role and include vars

block vars

task vars

extra vars (plus prioritaire)

Dans un projet Ansible, il est bon de limiter les sources de variables possibles (notre recommandation en **vert**).

Cette limitation améliore grandement la prédictibilité et la compréhension du code.

Variables conseillées : use cases

- ▷ **Role default :**
 - > variables positionnées au niveau d'un rôle
 - > exemple : numéro de version de Java par défaut
- ▷ **Playbook Group Vars**
 - > variables associées à un groupe de machine
 - > exemple : numéro de version de Java à déployer sur ces machines
- ▷ **Host facts**
 - > variables associées à une machine
 - > exemple : type de l'OS
- ▷ **Set facts et registered vars**
 - > variables définies pendant l'exécution d'un playbook
 - > exemple : valeur de retour d'une tâche
- ▷ **Extra Vars**
 - > variables passées via la ligne de commande "-e <nom_var>=<valeur_var>"
 - > exemple : numéro de version d'une application

“ Les templates et filtres Jinja2 ”

Les templates dans Ansible

Les templates permettent de produire des fichiers de configuration dont le contenu est dynamiquement évalué lors du passage d'Ansible

La syntaxe pour écrire des templates est celle de [Jinja2](#) (bibliothèque Python)

Elle permet :

- ▷ **D'utiliser des [variables](#) (comme les facts par exemple) de la machine courante**
- ▷ **D'utiliser des [variables des autres machines](#)**
- ▷ **D'appliquer des transformations de ces variables à l'aide de [filtres](#)**
 - > Standards dans Jinja2
 - > Spécifiques à Ansible
 - > Que vous pouvez écrire vous-même
- ▷ **D'écrire des [structures de contrôle](#)**
 - > Conditions
 - > Boucles

Les templates dans Ansible

Les blocs Jinja2 sont reconnaissables à leurs balises

- ▷ **{{ }}** pour afficher une variable
 - > `{{ ma_var }}`
- ▷ **{% %}** pour identifier les structures de contrôle
 - > `{% if ma_var == true %}...{% endif %}`

Exemple

```
name {{ inventory_hostname }}  
weight {{ 1 + 2 + 3.1415 }}
```

```
{% for i in ntp_servers %}  
server {{ i }}  
{% endfor %}
```

La syntaxe de Jinja2 rappelle celle de Twig (PHP) ou Go (Templates)

Les templates dans Ansible

Ansible fournit des variables très intéressantes pour consulter l'inventaire et les facts des (autres) machines

- ▶ **groups[]** => contient les membres à partir du nom du groupe. Retourne la liste des `inventory_hostname` correspondant.
- ▶ **hostvars[]** => contient les variables (notamment les *facts*) par machine, indexées par leur `inventory_hostname`

Les *facts* ne sont disponibles que si un **gather_facts** a eu lieu plus tôt dans le playbook

Exemples de templates Ansible

Faire une boucle sur tous les membres d'un groupe et récupérer un *fact* pour chaque machine :

```
# Exemple de /etc/hosts
# {{ ansible_managed }}
127.0.0.1 localhost
{% for i in groups['all'] %}
{{ hostvars[i].ansible_default_ipv4.address }} {{ i }}
{% endfor %}
```

```
; inventaire inv.ini
machine1.trololo.com
machine2.trololo.com
machine3.trololo.com
```



```
# Exemple de /etc/hosts
# ANSIBLE MANAGED, do not edit
127.0.0.1 localhost
10.0.3.1 machine1.trololo.com
10.0.3.2 machine2.trololo.com
10.0.3.3 machine3.trololo.com
```

Les filtres Jinja2 et Ansible

Les filtres permettent d'opérer des transformations sur des valeurs avant affichage

La syntaxe est semblable à celle des *pipes* sur la ligne de commande :

```
{{ ma_var | filtre1 | filtre2 | filtre3 }}
```

Les filtres principaux

- ▷ **default()** => pour affecter une valeur par défaut si variable absente
- ▷ **join()** => pour joindre une liste avec un séparateur
- ▷ **dirname / basename** => pour extraire des portions d'un nom de fichier
- ▷ **bool, int, float** => pour forcer le *cast* d'un objet
- ▷ **replace()** => pour faire une substitution dans une chaîne
- ▷ **sort** => pour trier une liste
- ▷ **upper / lower** => pour changer la casse
- ▷ **union / intersect / difference** => pour manipuler des listes
- ▷ **regex_replace** => remplacements à base d'expressions régulières
- ▷ **match** => vérification qu'une chaîne respecte un *pattern* donné
- ▷ **password_hash('sha256', 'mysecretsalt')** => hachage de mot de passe

```
git clone https://5AIW2:d-Gx32tFcduyu-y92ein@gitlab.com/santunes-formations/ansible.git
```


TP #4

“Les “Handlers” et “Notify””

La problématique du 1st deploy vs. upgrade

Ansible fonctionnant sur le principe d'état attendu, il va avoir un comportement particulier qu'il est important de comprendre

Partons du play suivant

```
---  
- hosts: webservers  
  tasks:  
    - name: Install NGinx  
      package:  
        name: nginx  
    - name: template conf file  
      template:  
        src: nginx.conf.j2  
        dest: /etc/nginx/nginx.conf  
    - name: enable / start NGinx  
      service:  
        name: nginx  
        state: started  
        enabled: true
```

La problématique du 1st deploy vs. upgrade

Lors d'un premier lancement d'Ansible, toutes les opérations vont être effectuées car le système n'est pas dans l'état attendu

playbook

```
---  
- hosts: webservers  
  tasks:  
    - name: Install NGinx  
      package:  
        name: nginx  
    - name: template conf file  
      template:  
        src: nginx.conf.j2  
        dest: /etc/nginx/nginx.conf  
    - name: enable / start NGinx  
      service:  
        name: nginx  
        state: started  
        enabled: true
```

1er lancement
sur machine
vierge

actions effectuées

apt-get install -y nginx

changed

cp /tmp/file1 /etc/nginx/nginx.conf

changed

update-rc.d nginx enable
service nginx start

changed

84

État correct

La problématique du 1st deploy vs. upgrade

Lors d'un nouveau lancement d'Ansible sur une machine déjà déployée, aucune opération ne va être effectuée

playbook

```
---  
- hosts: webservers  
  tasks:  
    - name: Install NGinx  
      package:  
        name: nginx  
    - name: template conf file  
      template:  
        src: nginx.conf.j2  
        dest: /etc/nginx/nginx.conf  
    - name: enable / start NGinx  
      service:  
        name: nginx  
        state: started  
        enabled: true
```

nouveau
lancement sur
machine
installée

actions effectuées

noop

ok

noop

ok

noop

ok

État correct

La problématique du 1st deploy vs. upgrade

Si l'on modifie le template et que l'on relance Ansible sur une machine déjà configurée

playbook

```
---
- hosts: webservers
  tasks:
    - name: Install NGinx
      package:
        name: nginx
    - name: template conf file
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
    - name: enable / start NGinx
      service:
        name: nginx
        state: started
        enabled: true
```

3ème
lancement sur
machine
installée

actions effectuées

noop

ok

cp /tmp/file1 /etc/nginx/nginx

changed

noop

ok

86

État incorrect : nginx n'a pas été redémarré

La problématique du 1st deploy vs. upgrade

Solution : forcer le restart systématique

playbook

```
---  
- hosts: webservers  
  tasks:  
    - name: Install NGinx  
      package:  
        name: nginx  
    - name: template conf file  
      template:  
        src: nginx.conf.j2  
        dest: /etc/nginx/nginx.conf  
    - name: enable / start NGinx  
      service:  
        name: nginx  
        state: restarted  
        enabled: true
```

3ème
lancement sur
machine
installée

actions effectuées

noop

ok

cp /tmp/file1 /etc/nginx/nginx.conf

changed

service nginx restart

changed

État correct

La problématique du 1st deploy vs. upgrade

Solution : forcer le restart systématique

Lors d'un nouveau lancement d'Ansible sur une machine déjà déployée / conforme, un restart inutile va avoir lieu

playbook

```
---
- hosts: webservers
  tasks:
    - name: Install NGinx
      package:
        name: nginx
    - name: template conf file
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
    - name: enable / start NGinx
      service:
        name: nginx
        state: restarted
        enabled: true
```

nouveau
lancement sur
machine
installée

actions effectuées

noop

ok

noop

ok

service nginx restart

changed

État correct, mais changement inutile

Handlers et notify

Pour résoudre cette problématique, Ansible dispose de la notion de handler et notify

Les handlers sont des tâches particulières qui n'ont vocation à être exécutées que si une autre tâche a provoqué un changement

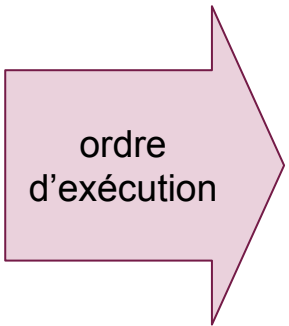
```
# install.yml
---
- hosts: all
  tasks:
  ...
  - name: install conf file
    template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify:
      - restart nginx
  ...
  handlers:
  - name: restart nginx
    service:
      name: nginx
      state: restarted
```

Ordre d'exécution des handlers

Les handlers ne sont pas exécutés juste après la tâche qui les a déclenchés

Ils sont lancés à la fin d'un play.

```
---  
- hosts: all  
  tasks:  
    ...  
    - name: do stuff a  
      module1:  
        notify:  
          - handler1  
    - name: do stuff b  
      module2:  
    - name: do stuff c  
      module3:  
  handlers:  
    - name: handler1  
      module66:
```



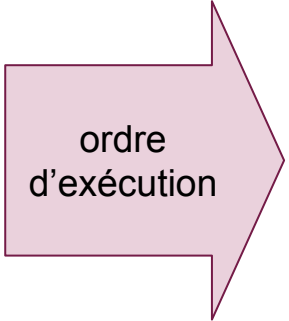
ordre
d'exécution

```
1.  do stuff a  
2.  do stuff b  
3.  do stuff c  
4.  handler1
```

Ordre d'exécution des handlers

Le pseudo module **meta** permet de forcer l'exécution d'un handler à un moment précis de l'exécution d'un playbook

```
---
- hosts: all
  tasks:
  ...
  - name: do stuff a
    module1:
      notify:
        - handler1
  - name: force handlers
    meta: flush_handlers
  - name: do stuff b
    module2:
  - name: do stuff c
    module3:
  handlers:
  - name: handler1
    module66:
```



ordre
d'exécution

1. do stuff a
2. **handler1**
3. do stuff b
4. do stuff c

“ Les rôles ”

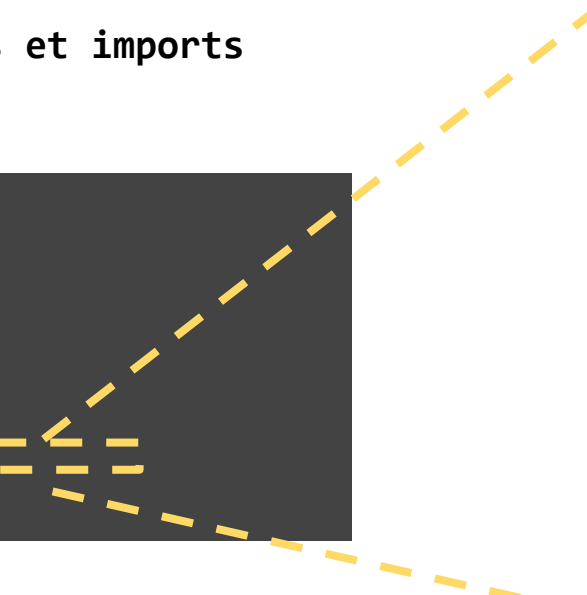
Les limites des playbooks

La seule utilisation des playbooks pose des limites en cas de réutilisation du code

Même si les `includes` et `imports` existent

```
# install.yml
---
- hosts: all
  tasks:
    ...

- name: install nginx stuff
  import_tasks: nginx.yml
```



```
# nginx.yml
---

- hosts: webserver
  tasks:
    - name: Install NGinx
      package:
        name: nginx
    - name: template conf file
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
    - name: enable / start server
      service:
        name: nginx
        stated: started
        enabled: true
```

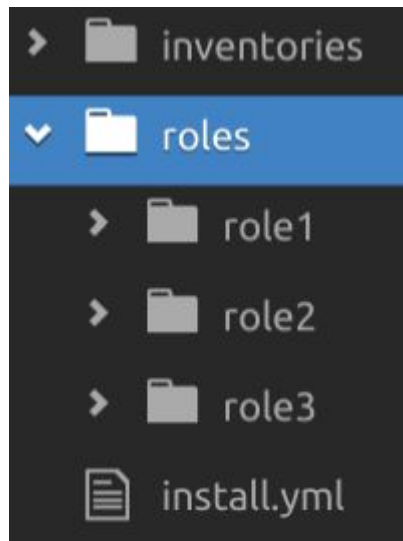
Ils ne permettent pas de masquer toutes les subtilités (nommer templates par exemple,

Les objectifs des rôles

- ▷ **Regrouper des éléments relatifs à**
 - > Un besoin fonctionnel
 - > Une technologie
- ▷ **Masquer une (relative) complexité d'implémentation**
- ▷ **Sous forme d'un artefact**
 - > Packagé
 - > Cohérent
 - > Documenté
- ▷ **Pour favoriser sa réutilisation**

Localisation des rôles

Par défaut, les rôles sont recherchés dans un sous répertoire **roles** à côté des playbooks



Anatomie d'un rôle

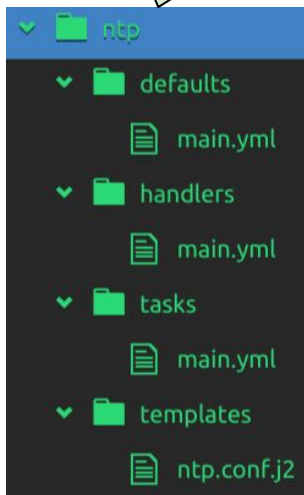
Un rôle est un répertoire qui contient des (au moins un) sous-répertoires

- ▷ **defaults**
 - > Variables par défaut positionnées dans le rôle. Peuvent être surchargées
- ▷ **files**
 - > Fichiers spécifiques au rôle qui doivent être utilisés (copiés) vers les machines cibles
- ▷ **handlers**
 - > Tâches à exécuter en cas de notify
- ▷ **meta**
 - > Description du rôle, notamment de ses dépendances
- ▷ **tasks**
 - > Tâches
- ▷ **templates**
 - > Fichiers templates Jinja2 utilisés dans ce rôle
- ▷ **vars**
 - > Variables utilisées dans le rôle qui ne peuvent pas être surchargées

En général, on évite d'utiliser les **files et les **vars**. À ne réserver qu'à des cas très spécifiques.**

Exemple : le rôle NTP

Rôle
réutilisable



Premier exemple minimaliste d'un rôle réutilisable

Objectif du rôle : synchroniser la machine cible avec des serveurs NTP de référence

Des serveurs NTP par défaut (publics, sur Internet) sont fournis, mais peuvent être surchargés

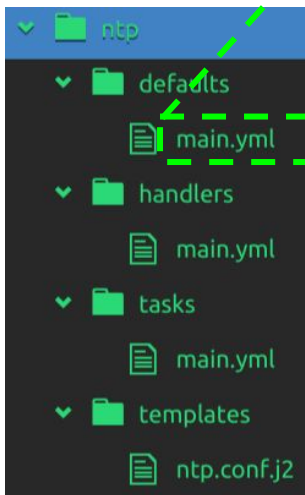
```
# playbook install.yml
---
- hosts: all
  become: true
  tasks:
    - import_role:
      name: ntp
```

Invocation
dans un
playbook

Exemple : le rôle NTP, variables

```
# roles/ntp/defaults/main.yml
---
ntp_servers:
  - 0.pool.ntp.org
  - 1.pool.ntp.org
  - 2.pool.ntp.org
  - 3.pool.ntp.org
```

Par convention, il est recommandé de préfixer le nom des variables par le nom du rôle



- La variable `ntp_servers` est utilisée par le rôle (dans les tâches, les templates)
- Elle prend ici une valeur par défaut qui peut être surchargée lors de l'invocation du rôle

Exemple : le rôle NTP, invocation avec surcharge

```
# playbook install.yml
---
- hosts: all
  become: true
  tasks:
    - import_role:
        name: ntp
```

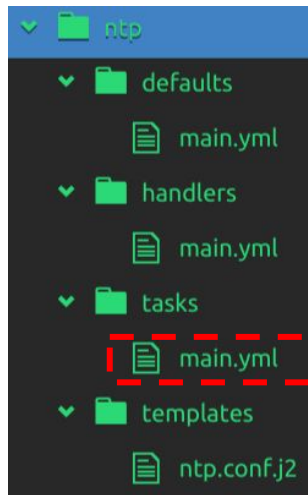
Invocation en utilisant les variables par défaut

```
# playbook install.yml
---
- hosts: all
  become: true
  tasks:
    - import_role:
        name: ntp
      vars:
        ntp_servers:
          - ntp1.internal.local
          - ntp2.internal.local
```

Invocation en surchargeant la variable ntp_servers

Exemple : le rôle NTP, tâches

Les tâches du rôle

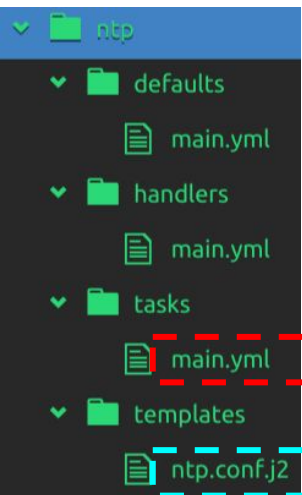


```
# roles/ntp/tasks/main.yml
---
- name: Install NTP packages
  package:
    name: ntp

- name: Configure ntp service
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    mode: 0644
  notify: restart ntp

- name: Start ntp
  service:
    name: ntp
    state: started
    enabled: yes
```

Exemple : le rôle NTP, templates



Les templates sont recherchés dans le sous-répertoire templates du rôle

Utilisation d'une variable ansible

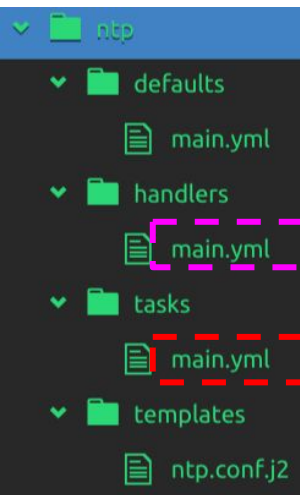
Utilisation d'une variable du rôle

```
# {{ ansible_managed }}
{% for ntp_server in ntp_servers %}
server {{ ntp_server }}
{% endfor %}
```

```
# roles/ntp/tasks/main.yml
---
- name: Install NTP packages
  package:
    name: ntp

- name: Configure ntp service
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    mode: 0644
  notify: restart ntp

- name: Start ntp
  service:
    name: ntp
    state: started
    enabled: yes
```



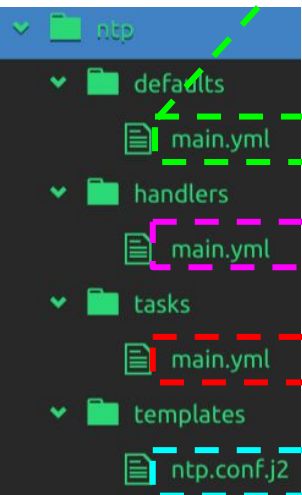
```
# roles/ntp/handlers/main.yml
---
- name: restart ntp
  service:
    name: ntp
    state: restarted
```

```
# roles/ntp/tasks/main.yml
---
- name: Install NTP packages
  package:
    name: ntp

- name: Configure ntp service
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    mode: 0644
  notify: restart ntp

- name: Start ntp
  service:
    name: ntp
    state: started
    enabled: yes
```

Exemple : le rôle NTP complet



```

# roles/ntp/defaults/main.yml
---
ntp_servers:
  - 0.pool.ntp.org
  - 1.pool.ntp.org
  - 2.pool.ntp.org
  - 3.pool.ntp.org
  
```

```

# roles/ntp/handlers/main.yml
---
- name: restart ntp
  service:
    name: ntp
    state: restarted
  
```

```

# {{ ansible_managed }}
{% for ntp_server in ntp_servers %}
  server {{ ntp_server }}
{% endfor %}
  
```

```

# roles/ntp/tasks/main.yml
---
- name: Install NTP packages
  package:
    name: ntp

- name: Configure ntp service
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    mode: 0644
  notify: restart ntp

- name: Start ntp
  service:
    name: ntp
    state: started
    enabled: yes
  
```

Les dépendances entre rôles

Les rôles peuvent tirer des dépendances vers d'autres rôles pour encore améliorer la modularité.

Les rôles en dépendance sont invoqués **avant** les rôles dépendants.

Le fichier “**meta/main.yml**” permet de décrire les dépendances.

Les dépendances peuvent être :

- ▷ **labélisées**
- ▷ **conditionnelles**
- ▷ **paramétrées**
- ▷ **multiples (il est possible d'appeler plusieurs fois le même rôle)**

Les dépendances entre rôles, exemples

▷ Syntaxe générale

```
# roles/rolea/meta/main.yml
---
```

dependencies:

- **role: roleb**
 tags: roleb
- **role: rolec**
 rolec_var_foo: bar
 tags: rolec

Ordre
d'invocation

1. **roleb**
2. **rolec (rolec_var_foo=bar)**
3. **rolea**

Les dépendances entre rôles, exemples

▷ Dépendance conditionnelle

```
# roles/rolea/defaults/main.yml
---
rolea_needs_roleb: false
```

```
# roles/rolea/meta/main.yml
---
dependencies:
- role: roleb
  tags: roleb
  when: rolea_needs_roleb
```

rôle

invocation

```
# install.yml
---
- hosts: all
  tasks:
  - import_role:
      name: rolea
    vars:
      rolea_needs_roleb: true
    tags: rolea
```

Les variables dans les rôles

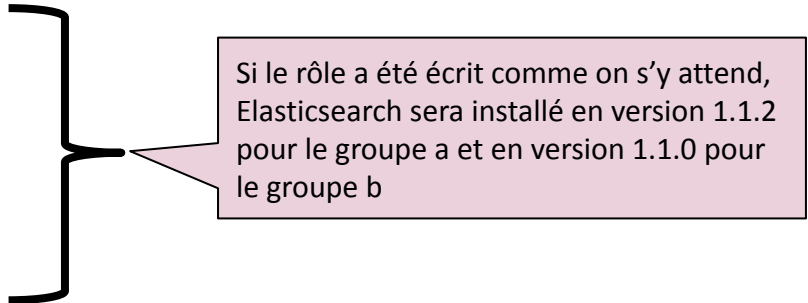
Les rôles permettent d'introduire deux nouvelles sources de variables qui seront disponibles dans le contexte du rôle et pour le reste du play en cours.

- ▷ **Les variables par défaut définies dans defaults/main.yml**
 - > priorité la moins forte (perd face aux group_vars notamment)

```
# roles/elasticsearch/defaults/main.yml
---
elasticsearch_version: 1.1.0
```

```
# group_vars/groupea
---
elasticsearch_version: 1.1.2
```

```
# group_vars/groupeb
---
```



Si le rôle a été écrit comme on s'y attend, Elasticsearch sera installé en version 1.1.2 pour le groupe a et en version 1.1.0 pour le groupe b

- ▷ **Les variables fortes définies dans vars/main.yml**
 - > priorité parmi les plus fortes (à n'utiliser que dans des cas particuliers)

Ansible Galaxy et les rôles

Ansible Galaxy est un dépôt de **rôles**

On peut récupérer un rôle en local avec la commande :

```
ansible-galaxy install <auteur_role.nom_role>
```

On peut aussi aller consulter le code du rôle directement sur GitHub.

Pour choisir un rôle, on regarde :

- ▷ **l'auteur** : qui est-il ? (société, contribution, ...)
- ▷ **la popularité du rôle** : nombre d'étoile, téléchargement, ...
- ▷ **les OS cibles** pour lesquelles le rôle a été conçu : RHEL, Debian, ...
- ▷ **la documentation et les tests**

Les rôles de la Galaxy peuvent être **source d'inspiration**.

Nous vous conseillons de les **adapter** à vos besoins en les **simplifiant**.

Note importante sur la syntaxe d'appel des rôles

Il existe deux syntaxes d'appel à un rôle:

```
# playbook install.yml
---
- hosts: all
  become: true
  tasks:
    - import_role:
      name: ntp
```

VS

```
# playbook install.yml
---
- hosts: all
  become: true
  roles:
    - role: ntp
```

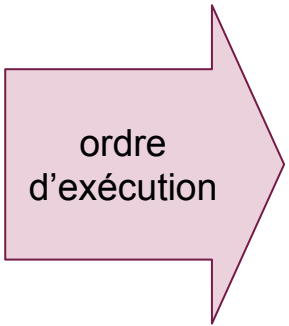
Les deux méthodes fonctionnent mais l'utilisation de la section "roles" implique un ordre d'exécution qui n'est pas intuitif dans certaines conditions.

Ordre d'exécution

Les différentes sections d'un play sont exécutées dans un ordre précis

- ▷ **Premièrement les pré-tâches (pre_tasks)**
 - > nouveau concept pour exécuter des choses avant les rôles !!
- ▷ **Puis les rôles**
- ▷ **Puis les tâches (tasks)**
- ▷ **Enfin les post-tâches (post_tasks)**

```
- hosts: all
pre_tasks:
- name: do stuff x
- name: do stuff y
tasks:
- name: do stuff a
- name: do stuff b
post_tasks:
- name: final stuff
roles:
- role: role1
- role: role2
- role: role3
```



ordre
d'exécution

```
1. do stuff x
2. do stuff y
3. role1
4. role2
5. role3
6. do stuff a
7. do stuff b
8. final stuff
```

“Les tags”

Les tags dans Ansible

Les tags sont un moyen de labéliser différents types d'objets

- ▷ **Rôles**
- ▷ **Tâches**
- ▷ **Inclusions**

L'utilisation des tags permet de filtrer le périmètre d'exécution d'Ansible

- ▷ **Par inclusion**
 - > Ne jouer que les objets qui ont un ou des labels donnés
- ▷ **Par exclusion**
 - > Jouer tout sauf les objets qui ont un ou des labels donnés

Les tags sont un bon moyen pour gagner du temps lors du développement de code Ansible pour ne se focaliser que sur la portion en cours de mise au point

Les tags dans Ansible : tâches

```
# roles/role1/tasks/main.yml
---
- name: do some stuff
  debug:
    msg: "This is some text"
  tags: verbose
- name: do something else
  debug:
    msg: "This is some text"
```

Pour ne pas lancer les tâches avec le tag verbose

```
$ ansible-playbook install.yml --skip-tags=verbose
```

```
$ ansible-playbook install.yml -t verbose
```

Les tags dans Ansible : imports

```
# install.yml
---
- name: import some stuff
  import_tasks: security.yml
  tags: security
```

Pour ne pas lancer l'inclusion :

```
$ ansible-playbook install.yml --skip-tags=security
```

```
$ ansible-playbook install.yml -t security
```

Les tags dans Ansible : rôles

```
# install.yml
---
- hosts: all
  tasks:
    - import_role:
        name: nginx
        tags: [webserver, nginx]
```

Pour ne pas lancer nginx

```
$ ansible-playbook install.yml --skip-tags=nginx
```

```
$ ansible-playbook install.yml -t nginx
```

Nous recommandons de systématiquement tagger les utilisations de rôles

Les tags dans Ansible : le tag always

```
# install.yml
---

- hosts: all
  tasks:
    - import_role:
        name: common
        tags: [ always ]

    - import_role:
        name: nginx
        tags: [ webserver, nginx ]
```

Le tag magique **always** permet de matcher dans tous les cas. Ici, **common** sera toujours exécuté, quelles que soient les options **--tags**, **-t** précisés sur la ligne de commande

```
git clone https://5AIW2:d-Gx32tFcduyu-y92ein@gitlab.com/santunes-formations/ansible.git
```

TP #5