Formation Ansible

15 septembre 2016

Ref 20150213-ObjectifLibre-FormationAnsible
Objectif Libre
http://www.objectif-libre.com
contact@objectif-libre.com

Table des matières

Introduction à Ansible

Introduction à la gestion centralisée

Avant la gestion de configuration

Des bonnes pratiques

- cahiers de déploiement
- méthodes unifiées

Une hétérogénéité à tous les niveaux

- OS
- Organisation sur le système de fichiers

Outils classiques de l'administrateur

- Le tout à la main : avec les inséparables sed et awk
- Une automatisation manuelle :
  - connaissance maison
  - adaptée à un cas précis
- Des outils orientés web (plesk, webmin..)
  - limités en capacité
  - gros effets de bord
- Des outils spécifiques pour du déploiement automatique (FAI, Kickstart)
  - Très bien pour installation
  - Inutiles pour configuration des applications (dès l'installation ou dans le temps)
- clusterssh
  - c'est bien mais un peu fastidieux...

La gestion de configuration : Pourquoi ?

- Le spectre du remplacement d'un serveur dans l'urgence à iso-fonctionnalités rapidement
- Besoin de déployer des changements sur tout ou partie des serveurs
  - Sans duplication / effort
- S'assurer d'une vraie homogénéité dans vos serveurs (qui peuvent être eux hétérogènes)

Ansible

Qu'est ce que Ansible ?

Projet Récent (2012)

- Écrit entièrement en Python
- Agentless

Capable de piloter des systèmes :

- Linux
- Unix
- Windows
- Réseau (Cisco, Juniper, ...)

Comment marche Ansible ?

- Pas de serveur central, toute machine avec Ansible peut commander les autres
- Les machines à contrôler nécessitent
  - un accès SSH et Python 2...
  - ou des APIs
- Une description des actions proche du langage naturel

- Pas de PKI
- Pour SSH
  - clés
  - mots de passe
  - kerberos
- Aucune modification des switchs/firewalls

Comparaison avec les autres produits

Puppet

L'outil le plus répandu de gestion de la configuration :

- Projet crée en 2005
- Un langage déclaratif pour exprimer des configurations systèmes
- Un modèle client / serveur principalement
- Un projet GPL écrit en Ruby

Capable de piloter des systèmes :

- Linux
- Unix
- Windows

Côté client

Chaque client possède un agent (qui tourne en daemon de manière habituelle).

L'agent a pour but de contacter le master :

- via une API REST cryptée (SSL) et authentifiée
- vérification régulière (1/2h par défaut) de disponibilité de mises à jour

Côté serveur

Le serveur est le lieu d'exécution du Puppet Master (port 8140)

C'est le lieu de :

- la définition et de la réalisation de la configuration
- la compilation de la configuration pour la présenter aux noeuds

Langage de configuration et abstraction des ressources

Puppet utilise un langage déclaratif propre pour définir les éléments de configuration, nommés des "ressources".

Les ressources définissent l'état du système voulu :

- présence d'un utilisateur
- présence d'un paquet
- état d'un service
- contenu d'un fichier
- ...

Chef

Chef a été crée par des anciens de PuppetLabs :

- sur les mêmes principes que Puppet (Client/Serveur)
- sur les mêmes technologies (Ruby)

Il souhaite résoudre des bottlenecks de Puppet (résolus depuis).

Son langage de configuration est du Ruby (et donc besoin de le connaitre).

Dernièrement son coeur a été ré-écrit en Erlang.

SaltStack

C'est un projet de nouvelle génération (2011), comme Ansible et écrit en Python.

Initialement basé sur ZeroMQ, il possède désormais une alternative (RAET) pour permettre une meilleure scalabilité.

Il possède son propre langage de configuration.

Salt est capable de fonctionner en :

- Agentless
- Client / Master

Mise en œuvre de Ansible

Installation

Ansible est installé uniquement sur une machine de management. L'installation peut être faite

- soit via les paquets de la distribution (pour Debian/Ubuntu)
- soit via des dépôts tiers pour les versions plus récentes (EPEL pour RedHat et dérivées, PPA pour Ubuntu)
- soit par `pip`

L'installation par pip permet d'utiliser une version spécifique d'Ansible, indépendamment de la distribution :

```
pip install ansible
pip install 'ansible>=2.0'
```

Il est également possible d'utiliser un virtualenv python :

```
virtualenv ansible-venv
. ansible-venv/bin/activate
pip install ansible
```

Python (> 2.5, < 3) doit être installé sur les machines sur lesquelles Ansible devra agir.

Pour les machines utilisant selinux par défaut, il faudra installer le paquet python-selinux (éventuellement par Ansible).

Mise en place de l'inventaire

L'inventaire définit la liste des machines sur lesquelles Ansible est susceptible d'agir. Ces machines sont organisées en groupes, une machine pouvant appartenir à plusieurs groupes.

Des variables arbitraires peuvent être associées aux machines et aux groupes. Des variables prédéfinies permettent de modifier le comportement d'Ansible, notamment pour définir les informations d'authentification.

Le fichier inventaire est `/etc/ansible/hosts` par défaut. Il peut être modifié dans la configuration Ansible, ou spécifié lors de l'exécution des commandes ansible.

Les inventaires sont des fichiers au format INI.

Chaque section définit un groupe de machines :

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

[dns]
ns1.example.com
ns2.example.com
```

Il est également possible de définir des "groupes de groupes" :

```
[dns_domain1]
dns1.domain1.com
dns2.domain1.com

[dns_domain2]
dns1.domain2.com
dns2.domain2.com

[dns:children]
dns_domain1
dns_domain2
```

Variables

Les variables sont définies

- soit sur la ligne correspondante à la machine
- soit dans une section dédiée

Définition de variables pour chaque hôte :

```
[dns]
ns1.example.com ansible_ssh_user=ansible type=master
ns2.example.com ansible_ssh_user=ansible type=slave
```

La variable `ansible_ssh_user` est utilisée par Ansible pour établir les connexions SSH avec le bon utilisateur.

La variable `type` n'est pas utilisée par défaut par Ansible, mais pourra l'être par les rôles et playbooks utilisateur.

Si des variables sont communes à tous les hôtes, elle peuvent être définies dans une section du fichier inventaire :

```
[dns]
ns1.example.com type=master
ns2.example.com type=slave

[dns:vars]
ansible_ssh_user=ansible
```

Il est également possible de définir les variables dans des fichiers placés dans les dossiers `group_vars` et `host_vars`. Ces fichiers utilisent le langage `YAML` pour définir les variables.

L'exemple précédent pourrait être réécrit en séparant les informations dans plusieurs fichiers.

Contenu de `/etc/ansible/hosts` :

```
[dns]
ns1.example.com
ns2.example.com
```

Contenu de `/etc/ansible/group_vars/dns` :

```
ansible_ssh_user: ansible
```

Contenu de `/etc/ansible/host_vars/ns1.example.com` :

```
type: master
```

Contenu de `/etc/ansible/host_vars/ns2.example.com` :

```
type: slave
```

CLI Ansible

Ansible fournit plusieurs outils en ligne de commande :

- `ansible` : exécution d'une commande unique sur un ensemble de serveurs
- `ansible-playbook` : exécution de playbooks (ensemble de tâches à effectuer)
- `ansible-doc` : accès au listing et à la documentation des modules
- `ansible-vault` : gestion de fichiers chiffrés (pour stocker les variables et mots de passe par exemple)
- `ansible-galaxy` : accès au dépôt de rôles Ansible
- `ansible-pull` : exécution d'Ansible directement sur une machine cible (sans machine de management)

Le module `ping` permet de valider l'inventaire, et l'accès aux machines distantes. La commande `ansible` permet son exécution :

```
ansible -i hosts dns -m ping
```

L'option `-i` spécifie le chemin vers l'inventaire (s'il n'est pas placé à l'endroit par défaut attendu par Ansible).

`dns` est le groupe de machines sur lesquelles la commande sera executée. La valeur `all` permet de déclencher une exécution sur toutes les machines définies dans l'inventaire.

L'option `-m` spécifie le module Ansible à utiliser sur les hôtes.

Gérer les accès Ansible avec SSH et sudo

L'accès SSH aux hôtes peut être explicité de plusieurs manières :

- en spécifiant toutes les informations de connexion dans l'inventaire
- en spécifiant ces informations sur la ligne de commande `ansible`
- en configurant la machine de management pour une connexion transparente (`~/.ssh/config`)

Ligne de commande

La commande `ansible` fournit plusieurs arguments pour spécifier les paramètres d'authentification :

- `--ask-pass` : demande du mot de passe SSH
- `--private-key` : chemin vers une clé SSH privée
- `--user` : login utilisateur sur l'hôte distant

Cette méthode d'authentification s'avère vite contraignante.

Configuration de la machine de management

Il est préférable de configurer la machine de management et les hôtes distants pour ne pas utiliser de mots de passe (ni pour SSH, ni pour sudo).

Cela implique :

- la génération d'une paire de clés SSH
- le déploiement de la clé publique sur chacun des serveurs
- l'utilisation d'un agent SSH pour éviter les demandes de passphrase
- une configuration sudo sans mot de passe sur les hôtes distants, ou un login en tant que `root`

La création d'une clé SSH se fait grâce à la commande **`ssh-keygen`** :

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
...
```

La commande **ssh-copy-id** permet la copie de la clé publique sur les hôtes distants :

```
$ ssh-copy-id ansible@dns1.example.com
$ ssh ansible@dns1.example.com  # authentification sans mot de passe
```

Le login utilisateur pour chaque hôte peut être spécifié dans l'inventaire, mais également dans le fichier de configuration SSH local :

```
$ cat ~/.ssh/config
Host dns1.example.com
    User ansible
```

Si un compte utilisateur autre que `root` est utilisé, `sudo` peut être configuré pour ne pas demander de mot de passe lors de son utilisation :

```
$ cat /etc/sudoers.d/ansible
ansible ALL=(ALL) NOPASSWD: ALL
```

Pour consolider la sécurité d'accès aux hôtes distants, il est possible de restreindre les accès à l'utilisateur Ansible

- en désactivant le login utilisateur par mot de passe (accès par clé SSH uniquement)
- en limitant l'accès SSH à l'IP de la machine de management uniquement

Fichier de configuration

Ansible peut fonctionner sans modifier sa configuration par défaut, mais quelques éléments peuvent influer sur son comportement.

La configuration d'Ansible se fait dans un fichier au format INI. Ansible cherche ce fichier à plusieurs endroits du système, dans cet ordre :

1. variable d'environnement `ANSIBLE_CONFIG`
2. `ansible.cfg` dans le dossier courant
3. `~/.ansible.cfg`
4. `/etc/ansible/ansible.cfg`

De nombreuses options de configuration sont disponibles, parmi lesquelles on trouve :

[defaults]/fork
: Nombre de processus à exécuter en parallèle

[defaults]/inventory
: Emplacement du fichier d'inventaire (voir *Mise en place de l'inventaire*)

[defaults]/library
: Emplacement de modules Ansible (dossiers séparés par `:`)

[defaults]/roles_path
: Emplacement de roles Ansible (dossiers séparés par `:`)

[ssh_connection]/pipelining
: A définir à `True` pour limiter le nombre de connexions à un hôte.

[ssh_connection]/host_key_checking
: A définir à `True` pour éviter la validation des clés SSH du serveur.

L'ensemble des directives utilisables dans le fichier de configuration est disponible sur la documentation Ansible.

Utilisation des principaux modules ad hoc

Les modules Ansible

Les modules sont la base des actions exécutées par Ansible. Chaque module traite d'un type d'action spécifique (modification d'un fichier, exécution d'une commande, récupération d'informations, ...).

Les modules acceptent des arguments, généralement sous la forme `clé=valeur`.

Ansible fournit un ensemble de modules (*core*) dont la liste est accessible via la commande `ansible-doc --list`.

La commande **`ansible-doc`** permet également d'afficher la documentation de chaque module : `ansible-doc <MODULE>`. Par exemple :

```
$ ansible-doc apt
```

On peut aussi les retrouver sur le web (parfois plus lisible) :

http://docs.ansible.com/ansible/modules_by_category.html

La majorité des modules peuvent être utilisés via la commande **`ansible`**, mais certains ne fonctionnent qu'au sein de playbooks (par exemple le module template).

La syntaxe de base pour l'utilisation de modules avec la commande **`ansible`** est :

```
ansible (hote/group/all) -m MODULE [-a "arg1=val1 [arg2=val2]"]
```

Par exemple, pour l'installation d'un paquet par le module `apt` :

```
$ ansible (hote/group/all) -m apt -a "name=vim state=present"
```

L'option `-m` spécifie le module à utiliser.

L'option `-a` spécifie les arguments à passer au module.

L'exécution d'un module peut retourner 3 états différents :

- l'exécution du module a échoué (mauvais arguments, échec côté hôte distant)
- l'exécution du module a fonctionné, et un changement a été appliqué
- l'exécution du module a fonctionné, mais aucun changement n'a été fait

En plus de cet état, des données au format JSON sont retournées par Ansible :

```
$ ansible target -m apt -a "name=vim state=present"
target | success >> {
    "changed": true,
    "stderr": "",
    "stdout": "Reading package lists..."
}
$ ansible target -m apt -a "name=vim state=present"
```

```
target | success >> {
    "changed": false
}
```

ping

Le module *ping* est utile pour valider la configuration de l'inventaire Ansible. Ce module ne prend pas d'argument.

```
$ ansible all -m ping
dns1.example.com | success >> {
    "changed": false,
    "ping": "pong"
}
dns2.example.com | FAILED => SSH Error: data could not be sent to the remote host.
   Make sure this host can be reached over ssh
```

setup

Le module *setup* n'exécute aucune action sur l'hôte, mais retourne un ensemble de *facts* : des informations sur l'hôte.

Ces facts couvrent un ensemble d'éléments, parmi lesquels des informations sur :

- le système d'exploitation (nom, version)
- la configuration réseau
- le matériel (BIOS, processeur, RAM)
- les disques et l'espace de stockage
- des facts custom
- ...

```
$ ansible dns1.example.com -m setup
dns1.example.com | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.10.10.10"
        ],
        ...
    }
}
```

shell / command

Les modules *shell* et *command* permettent d'exécuter des commandes sur les hôtes.

command n'utilise pas le shell pour exécuter la commande, et ne peut donc pas utiliser les variables d'environnements, les pipes ou les redirections.

shell exécute la commande via un shell, et permet l'utilisation des éléments non permis par le module *command*.

Ces deux modules ont des arguments similaires. On trouve notamment :

- `chdir` : déplacement dans le dossier associé avant l'exécution
- `creates` : si le fichier spécifié existe, la commande n'est pas exécutée
- `removes` : si le fichier n'existe pas, la commande n'est pas exécutée

Enfin la commande à exécuter est également passée sous forme d'argument du module.

Exemples :

```
$ ansible all -m command -a "ping -c 2 google.com"
$ ansible all -m shell -a "echo foo > /etc/bar creates=/etc/bar"
```

user

Le module *user* permet de gérer les utilisateurs (`/etc/passwd`) sur les hôtes, leurs mots de passe (`/etc/shadow`), leur homedir, ...

file

Le module *file* permet de gérer les droits sur les fichiers (propriétaire, groupe, droits d'accès), s'assurer qu'il s'agisse de lien ou répertoire, ...

service

Le module *service* permet de gérer les services systèmes :

- arrêt/démarrage/redémarrage
- activation/désactivation au boot

Exemples :

```
# Redémarrage de bind sur tous les serveurs DNS
$ ansible dns -m service -a "name=bind9 state=restarted"

# Activation au démarrage
$ ansible dns -m service -a "name=bind9 enabled=yes"
```

yum / apt / zypper / dnf

Ces modules permettent de gérer l'installation, la mise à jour et la suppression de paquets :

```
# Pour RedHat et distributions dérivées
$ ansible dns -m yum -a "name=named state=latest"

# Pour Debian et distributions dérivées
$ ansible dns -m apt -a "name=bind9 state=latest"
```

Des modules complémentaires permettent de gérer les dépôts :

- *yum* lui même pour RedHat
- *apt_repository* et *apt_key* pour Debian
- *zypper_repository* pour SuSE

Exercices

1. S'assurer que `openssl` et `bash` sont à jour sur tous les serveurs
2. Créer un utilisateur `ansible` avec un shell `/bin/bash`
3. Installer la clef publique SSH de notre utilisateur sur l'utilisateur ansible
4. S'assurer des bons droits sur `/etc/passwd` (0644) et `/etc/shadow` (0400).

Les playbooks Ansible

Intérêt

Utiliser les modules les uns après les autres via la commande **`ansible`** ne permet pas de profiter des fonctionnalités d'orchestration d'Ansible.

Les *playbooks* sont la description d'un ensemble de tâches à effectuer de manière **séquentielle**, sur tout ou partie de l'inventaire.

Chaque tâche (*task*) d'un playbook utilise un module Ansible. Les playbooks offrent cependant des fonctionnalités supplémentaires :

- émission de notifications permettant d'exécuter des actions dans certaines conditions (par exemple redémarrer un service sur modification d'un fichier de configuration)
- sauvegarde des facts et exécutions conditionnelles selon leurs valeurs (par exemple pour gérer un parc hétérogène RedHat/Debian)
- exécution de tâches par un autre hôte (par exemple pour intégrer l'hôte en cours de configuration dans le système de monitoring)
- utilisation de templates pour créer/modifier des fichiers

Les playbooks se définissent en langage YAML.

Un playbook est constitué au minimum d'une liste d'actions à effectuer sur un ensemble d'hôtes, définis dans l'inventaire.

On trouve au minimum dans chaque playbook :

- une variable `hosts` définissant le groupe d'hôtes sur lequel agir
- une variable `tasks` définissant les tâches à exécuter

On peut également trouver :

- des variables spécifiques au playbook (en plus des variables de l'inventaire)
- des éléments de configuration pour Ansible (par exemple l'utilisation de sudo)
- des *rôles* à utiliser
- des *handlers*, tâches exécutées sur notification

Exemple de playbook simple :

```
- hosts: dns
  become: yes
  tasks:
  - name: Installing vim
    package:
      name: vim
      state: present
    tags:
    - vim
```

La commande ansible-playbook

L'exécution du playbook est déclenchée via la commande **`ansible-playbook`** :

```
$ ansible-playbook mon_playbook.yml
```

Par défaut toutes les règles du playbook s'exécuteront sur tous les hôtes concernés.

Il est possible de limiter les actions :

- en définissant des *tags* dans le playbook et en utilisant l'option `--tags` (ou `--skip-tags`) de **`ansible-playbook`**
- en utilisant l'option `-l` de **`ansible-playbook`**

Par exemple :

```
$ ansible-playbook mon_playbook.yml --tags vim -l dns1.example.com
```

Syntaxe YAML

Ansible utilise le langage YAML pour la plupart des données. YAML est un langage de sérialisation. Il est plus facile à lire et écrire que le JSON, et beaucoup moins verbeux que le XML.

YAML permet de manipuler quelques types de données :

- du *texte*
- des *listes*
- des *dictionnaires* (ou *hash*)

Listes

Les éléments d'une liste sont précédés du caractère `-` :

```
beatles:
- john
- paul
- george
- ringo
```

La syntaxe suivante est moins utilisée mais équivalente :

```
beatles: [ john, paul, george, ringo ]
```

Dictionnaires

Les dictionnaires sont des couples clé/valeur, séparées par le caractère `:` :

```
instruments:
  john: piano
  paul: basse
  george: guitare
  ringo: batterie
```

La syntaxe suivante est également possible :

```
instruments: { john: piano, paul: basse }
```

Trucs et astuces

L'indentation lors de l'imbrication d'éléments est structurante. L'utilisation des espaces est obligatoire (les tabulations ne sont pas supportées).

Les chaînes de caractères ne nécessitent pas de formatage particulier, sauf en cas d'ambiguïté. Dans ce cas les chaînes sont entourées de caractères " (double quote) :

```
# Invalide
message: error: blablabla
text: {{ hello world }}

# Valide
message: "error: blablabla"
text: "{{ hello world }}"
```

Définir les tâches (tasks)

Chaque tâche définie dans un playbook fait appel à un unique module Ansible.

Une tâche est un dictionnaire qui comporte au minimum ces couples de clés/valeurs :

- un nom (`name`), optionnel, mais vivement recommandé pour la documentation du playbook
- le module à appeler (la clé est le nom du module), avec ses arguments

Des attributs additionnels peuvent être rencontrés, parmi lesquels :

- `notify` : définit un *handler* à appeler si la tâche a généré une modification
- `delegate_to` : délègue l'action à un autre hôte
- `register` : sauve le résultat de l'exécution dans une variable
- `when` : exécute l'action sous certaines conditions
- `include` : inclusion d'un fichier externe
- `ignore_errors` : arrête ou non l'exécution du playbook en cas d'erreur

Notifications et handlers

La section `handlers` permet de définir des tâches qui ne seront exécutées que sur notification (mot clé `notify` d'une tâche). Les handlers possèdent un nom et un appel à un module. Les `handlers` ne sont appelés qu'une fois toutes les tâches réalisées (et dans leur ordre d'apparition, et non pas dans l'ordre des notifications).

L'exemple suivant décrit comment redémarrer le service `apache2` lorsque sa configuration a changé :

```
- hosts: webservers
  tasks:
  - name: Configure apache
    template:
      src: httpd.conf
      dest: /etc/apache2/apache2.conf
    when: ansible_os_family == 'Debian'
    notify: Restart apache

  handlers:
  - name: Restart apache
    service:
      name: apache2
      state: restarted
```

Le notify peut être une liste :

```
- hosts: localhost
  become: no
  tasks:
  - name: Action qui fait une modif 1
    file:
      path: /tmp/toto1
      state: touch
      mode: 0644
    notify:
    - Handler 1
    - Handler 2
```

Avertissement : L'argument de `notify` et le nom du handler doivent être rigoureusement identiques.

Exercices

1. Écrire un playbook permettant de :
   - créer un utilisateur `ansible`
   - importer une clé SSH publique pour cet utilisateur
   - configurer `sudo` pour cet utilisateur (sans mot de passe)
2. Écrire un playbook de déploiement de apache avec support de PHP fonctionnant sur Debian et un autre pour RedHat.

Structures de contrôle

Les facts

La première action effectuée par Ansible lors de l'exécution d'un playbook est la récupération des facts de chacun des hôtes (via le module `setup`).

Cette action permet d'obtenir des variables (préfixées par `ansible_`) utilisables dans les filtres, tests et boucles du playbook.

Des variables liées à l'inventaire sont aussi disponibles :

- `inventory_hostname` est le nom de l'hôte en cours de traitement, tel que trouvé dans l'inventaire
- `groups` permet l'accès aux noms d'hôtes et groupes de l'inventaire. Par exemple : `groups.dns[0]` (premier hôte du groupe `dns`)
- `hostvars` permet l'accès aux variables d'un hôte. Par exemple : `hostvars[groups.dns[0]].type`

Note : On peut désactiver la récupération des facts dans un playbook en utilisant la directive `gather_facts: no`.

Il est possible de définir ses propres facts sur une machine.

Cela se fait en les déclarant dans un fichier `/etc/ansible/facts.d/mon_fact.fact`

Les facts sont définis au format JSON ou INI, ou peuvent être un exécutable qui retourne du JSON.

Les conditions

Certaines tâches du playbook peuvent ne pas toujours devoir être exécutées. La directive `when` permet de définir un test à effectuer. Si celui-ci retourne `false`, la tâche n'est pas exécutée.

Exemple d'utilisation de `when` pour gérer des distributions Linux différentes :

```
- name: Installing apache (Debian/Ubuntu)
  apt:
    name: apache2
    state: present
  when: ansible_os_family == 'Debian'

- name: Installing apache (RHEL)
  yum:
    name: httpd
    state: present
  when: ansible_os_family == 'RedHat'
```

Exécution d'une commande dont le succès détermine la suite de l'exécution du playbook :

```
- name: Is the device up?
  shell: "grep -q '^ 0:.*UpToDate' /proc/drbd"
  ignore_errors: true
  register: device_is_up

- name: Activate the device if not up
  shell: "drbdadm up mon"
  when: device_is_up.rc != 0
```

Depuis la version 2.0 d'Ansible, les conditions peuvent s'appliquer à des blocs entiers :

```
- block:
  - debug: msg="Step 1"
  - debug: msg="Step 2"
  - debug: msg="Step 3"
  when: ansible_os_family == 'Debian'
```

Les boucles

Les boucles permettent d'itérer sur les variables de types liste et dictionnaire. Cette technique permet d'exécuter plusieurs fois la même action sur un nombre d'éléments indéfini lors de l'écriture du rôle ou playbook.

La liste complète des boucles est disponible sur la documentation Ansible.

Les exemples suivant illustrent les boucles les plus utilisées.

with_items

Le mot clé `with_items` permet de boucler sur une liste. A chaque itération une variable `item` prend la valeur suivante de la liste.

Playbook :

```
- name: Installing editors packages
  apt:
    name: "{{ item }}"
  with_items:
  - vim
  - emacs
  - nano
```

with_dict

Le mot clé `with_dict` permet de boucler sur un dictionnaire. A chaque itération la variable `item` définit deux attributs : `key` et `value` :

```
- name: Creating users
  user:
    name: "{{ item.key }}"
    uid: "{{ item.value }}"
  with_dict:
    user1: 1000
    user2: 1001
    user3: 1002
```

Utilisation des inclusions

Directive `include`

La directive `include` permet d'importer une liste de tâches ou de handlers. Cette technique permet de rendre génériques certaines actions.

Exemple :

```
- hosts: all
  tasks:
  - include: common-setup.yml
  - name: something else
    ...
```

Il est possible de définir des variables au moment de l'inclusion d'un fichier :

```
- include: dns-master.yml domain=foo.com
- include: dns-master.yml domain=bar.net
```

Exercice

1. Ecrire un script qui servira de `fact` (sortie en JSON) et qui indiquera :
   - l'uptime de la machine
   - le nombre d'utilisateurs connectés
   - les charges système moyennes des 1, 5 et 15 dernières minutes

   => uptime.fact
2. Ecrire un playbook qui permettra de déployer ce fact sur les noeuds
3. vérifier que le nouveau fact est bien obtenu via la commande :

```
$ ansible hote -m setup -a "filter=ansible_local"
```

4. Ecrire un playbook avec une tache qui n'est exécutée que si la charge de la dernière minute écoulée est inférieure à X (à adapter en fonction de la charge de la machine).

Exercice 2

Reprendre les 2 playbooks apache du TP précédent, et n'en faire qu'un seul qui fonctionne pour les distributions RedHat et Debian.

Templates (Jinja2)

Ansible offre un module *template* permettant la création/modification de fichiers sur les hôtes via un langage de template (Jinja2).

Les templates peuvent utiliser les variables de l'inventaire et des playbooks, et offrent des structures de contrôles telles que les tests et les boucles.

Utilisation du module

Les paramètres `src` et `dest` sont les seuls obligatoires. Définir le propriétaire, le groupe et les droits associés au fichier à créer ou modifier est vivement recommandé.

Il est possible de créer une sauvegarde du fichier existant avant modification grâce au paramètre `backup`.

Exemple d'utilisation :

```
- template:
    src: /templates/appli.conf.j2
    dest: /etc/appli/appli.conf
    owner: root
    group: appli
    mode: 0640
    backup: yes
```

Syntaxe de base d'une template

Dans sa forme la plus simple, une template contient les données finales du fichier à installer sur l'hôte distant. Le même résultat peut être obtenu avec le module *copy*.

Le premier intérêt du module template est la possibilité d'utiliser des variables. Les variables utilisent le format `{{ NOM_VARIABLE }}`.

Exemple d'utilisation de variables dans une template :

```
$db_name = {{ db.name }};
$db_user = {{ db.user }};
$db_password = {{ db.password }};
$db_host = {{ db.host }};
```

Les filtres utilisables dans les playbooks sont valides dans les templates (voir la section *Filtres*).

Structures de contrôle

Les structures de contrôle Jinja permettent d'inclure un minimum d'intelligence dans les templates.

Les structures de contrôle sont définies dans des blocs. Les débuts et fins de blocs sont délimités par les caractères `{%` et `%}`, et contiennent le type de traitement à effectuer.

Tests

Les blocs `if`, `elif`, `else` et `endif` permettent d'appliquer des tests.

Exemple :

```
zone "{{ dns.domain }}" {
    {% if dns.type == 'master' %}
    type master;
    {% else %}
    type slave;
    masters {{ dns.masters | join(";") }};
    {% endif %}

    file "db.{{ dns.domain }}";
};
```

Boucles

Il est possible de boucler sur les listes, avec une syntaxe similaire à celle du langage python :

```
<ul>
{% for fruit in fruits %}
  <li>{{ fruit }}</li>
{% endfor %}
</ul>
```

Il en va de même pour boucler sur les éléments d'un dictionnaire :

```
<dl>
{% for name, inst in beatles.items() %}
  <dt>{{ name }}</dt>
  <dd>{{ inst }}</dd>
{% endfor %}
</dl>
```

Documentation complète

L'ensemble des fonctionnalités de Jinja2 est disponible sur la documentation en ligne.

Développer du code réutilisable

Ansible offre un ensemble d'outils permettant de facilement réutiliser des playbooks (ou des parties de playbooks).

Les playbooks peuvent utiliser des *variables*, les rendant adaptables à de multiples environnements.

Les tâches peuvent être regroupées dans des *rôles*, réutilisables au sein de différents playbooks.

Définir et utiliser des variables

Définir les variables

Les variables peuvent être définies à plusieurs endroits :

- dans l'inventaire, et les dossiers associés (`group_vars` et `host_vars`)
- dans un playbook
- dans un fichier externe, utilisé via la directive `include_vars` d'un playbook.

Cela offre donc de nombreuses possibilités, de la souplesse, mais cela peut vite amener à une certaine difficulté pour s'y retrouver si une certaine organisation n'est pas mise en place.

Il est recommandé de ne placer dans l'inventaire (`hosts`) que les variables propres à ansible (ansible_ssh_user, ansible_become, etc.).

Les variables en rapport avec les playbooks ou roles seront plus logiquement placées dans ces rôles et playbooks, ou dans des fichiers d'inclusions localisés à côté.

Les variables sont de la forme clé/valeur. Les valeurs peuvent être des chaînes de caractères ou des structures plus complexes (listes ou dictionnaires) :

Exemple de fichier de variables :

```
username: bill
groups:
- lp
- adm
packages:
- vim
- bash-completion
db:
  sql_user: linus
  sql_password: linus_passwd
```

Accéder aux variables

Ansible utilise la syntaxe de *jinja2* (moteur de template python) pour l'accès aux variables. Les variables peuvent être utilisées à n'importe quel endroit d'un playbook.

La syntaxe pour utiliser une variable est `{{ CLE_VARIABLE }}`.

L'extrait de playbook suivant utilise le fichier de variables défini dans la section précédente :

```
tasks:
- include_vars: variables.yml

- name: Create a default user
  user:
    name: "{{ username }}"
```

L'accès aux éléments d'une liste se fait par leur indice :

```
vars:
  users: [ 'roy', 'moss', 'jen' ]

tasks:
- user:
    name: "{{ users[0] }}"
```

L'accès aux attributs d'un dictionnaire peut se faire via 2 syntaxes :

```
vars:
  user:
    name: roy
    id: 1000

tasks:
- user:
    name: "{{ user.name }}"
    uid: "{{ user['id'] }}"
```

Filtres

Ansible offre la possibilité d'appliquer des filtres aux valeurs des variables. Ces filtres permettent de transformer les variables.

Les filtres sont des fonctions, et la plupart acceptent un ou plusieurs arguments.

La liste des filtres utilisables est disponible sur la documentation Ansible.

Il est possible de créer ses propres filtres (en python).

Les filtres s'appliquent avec le caractère `|` placé après le nom de la variable : `{{ VARIABLE | filtre(arg1, arg2) }}`

Exemples de filtres

Variables obligatoires et optionnelles :

```
{{ var1 | mandatory }}
{{ var2 | default("value") }}
```

Travail sur les listes :

```
# Valeur minimum d'une liste
{{ [4, 8, 9] | min }}

# Formatage
{{ groups | join(",") }}
{{ groups | to_nice_json }}
```

Travail sur les chemins de fichiers :

```
{{ path | dirname }}
{{ path | basename }}
```

Expressions régulières :

```
# Retourne un booléen (true/false)
{{ path | match("/home/toto/.*/public_html") }}

# Remplacements
{{ path | regex_replace("(.*) (.*)", "\\2 \\1" }}
```

Résumé / Précédence des variables

- les variables extras (-e en ligne de commande) l'emportent toujours
- puis viennent les variables de connexion définies dans l'inventaire (ansible_ssh_user, etc.)
- puis le "tout venant" (options de la ligne de commande, variables dans le playbook, fichiers d'inclusion, variables de rôles)
- puis les autres variables dans l'inventaire (autres que de connexion donc)
- puis les facts obtenus du système
- puis les valeurs par défaut définies dans les rôles

La notion de Rôle

Les *rôles* sont des playbooks génériques, qui peuvent être intégrés dans d'autres playbooks.

Un rôle définit un ensemble d'éléments :

- un nom (qui correspond à un dossier stockant les éléments du rôle)
- des variables non modifiables depuis un playbook (dans un dossier `vars`)
- des valeurs par défaut pour des variables modifiables (dans un dossier `defaults`)
- des tâches (dans un dossier `tasks`)
- des handlers (optionnels, dans un dossier `handlers`)
- des metadata (optionnels, dans un dossier `meta`)
- des fichiers et templates, si les modules associés sont utilisés

Pour chaque élément d'un rôle (tâches, handlers, ...), un fichier nommé `main.yml` sert de point d'entrée.

Exemple d'arborescence :

```
$ tree roles/dns-dhcp/
roles/dns-dhcp/
|-- handlers
|   `-- main.yml
|-- meta
|   `-- main.yml
|-- tasks
|   `-- main.yml
|-- templates
|   |-- db.192.168.1.j2
|   |-- db.local.net.j2
|   `-- dhcpd.conf.j2
`-- vars
    `-- main.yml
```

Il ne sera pas nécessaire d'indiquer les directives *vars*, *tasks*, ou *handlers* dans les fichiers *vars/main.yml*, *tasks/main.yml*, *handlers/main.yml*, cette information étant obtenue à partir du nom du répertoire.

Par exemple un fichier *tasks/main.yml* ne contiendra qu'une liste de tâches :

```
- name: Install Apache
  apt: name=apache2 state=present
- name: Ensure service is registered and running
  service: name=apache2 enabled=true state=started
```

De la même façon, les modules *copy* et *template* iront chercher les fichiers sources directement dans les répertoires *files* et *templates*, sans besoin de les préciser dans le path.

```
- name: Install apache conf
  copy: src=apache2.conf dest=/etc/apache2/apache2.conf
- name: Installation d'un vhost
  template: src=vhost.j2 dest=/etc/apache2/sites-available/...
```

La commande **`ansible-galaxy`** permet d'initialiser l'arborescence d'un rôle :

```
$ ansible-galaxy init --offline role
$ ansible-galaxy init --offline -p /path/to/roles role
```

La directive `roles` permet d'utiliser un ensemble de rôles dans un playbook :

```
- hosts: dns

  tasks:
  - name: une tache
    debug: msg="Une tache..."

  roles:
  - dns-dhcp
```

Si le playbook définit des rôles et des tâches, les tâches des rôles seront exécutées avant les tâches du playbook.

Les tâches pre et post

Ces tâches permettent d'exécuter des actions avant de faire le traitement sur un hôte, et après le traitement. Elles peuvent être utiles pour passer une machine en mode maintenance, éventuellement en agissant sur un autre serveur.

L'exemple suivant illustre comment ces tâches pourraient être utilisées pour sortir un hôte d'un load balancer :

```
- hosts: webserver
  pre_tasks:
  - script: remove-server.sh {{ inventory_hostname }}
    delegate_to: groups.lb[0]

  roles:
  - apache2
  - { role: apache2_vhost, vhost_name: 'test1.example.com', document_root: '/var/www/test1' }
  - { role: apache2_vhost, vhost_name: 'test2.example.com', admin_mail: 'root@example.com' }
  - { role: apache2_vhost, vhost_name: 'test3.example.com' }

    ...

  post_tasks:
  - script: add-server.sh {{ inventory_hostname }}
    delegate_to: groups.lb[0]
```

Note : La variable *inventory_hostname* permet d'obtenir la machine en cours de traitement.

Obtenir des rôles : Galaxy

Ansible fournit une bibliothèque de rôles en ligne : Galaxy.

Les rôles disponibles sur Galaxy sont soumis par la communauté Ansible, et leur qualité varie. Un système de notation permet de trouver les meilleurs rôles.

Installation d'un rôle

La commande **`ansible-galaxy`** permet l'installation d'un rôle. Les noms des rôles sont toujours de la forme `<utilisateur>.<rôle>` :

```
$ ansible-galaxy install geerlingguy.apache
```

Suppression d'un rôle

```
$ ansible-galaxy remove geerlingguy.apache
```

Exercices

1. Créer un rôle permettant de déployer apache
2. Créer un rôle permettant de configurer un vhost apache
3. Créer un rôle permettant de déployer une application PHP depuis un dépôt git

Pour aller plus loin

Développer ses propres modules

Il est possible d'écrire de nouveaux modules pour Ansible. Ces modules peuvent être écrits en différents langages, mais python est le langage recommandé.

La documentation Ansible détaille l'écriture d'un module.

Les modules additionnels sont installés dans un dossier `library` dans l'espace de travail.

Créer des filtres Jinja2

Il est également possible d'écrire de nouveaux filtres pour Jinja2.

L'exemple suivant est un filtre permettant la transformation d'un dictionnaire en une chaîne de caractères :

```
def dict_to_str(d, item_sep, kv_sep):
    items = []
    for k, v in d.items():
        items.append('%s%s%s' % (k, kv_sep, v))
    return item_sep.join(items)

class FilterModule(object):
    def filters(self):
        return {'dict_to_str': dict_to_str}
```

Les filtres additionnels sont installés dans un dossier `filter_plugins` dans l'espace de travail.

Autres plugins

D'autres plugins peuvent être écrits pour compléter les fonctionnalités d'Ansible :

- plugins de *connexion* (SSH ou autre)
- plugins de *lookup* (`with_item`, `with_dict`, ...)
- plugins de *variables* (`group_vars`, `host_vars`)
- plugins de *callback*

Inventaire dynamique

Ansible offre des mécanismes de gestion dynamique de l'inventaire. Cette fonctionnalité s'avère très utile sur des parcs hébergeant de nombreux serveurs, ou des parcs évoluant beaucoup (par exemple un cloud).

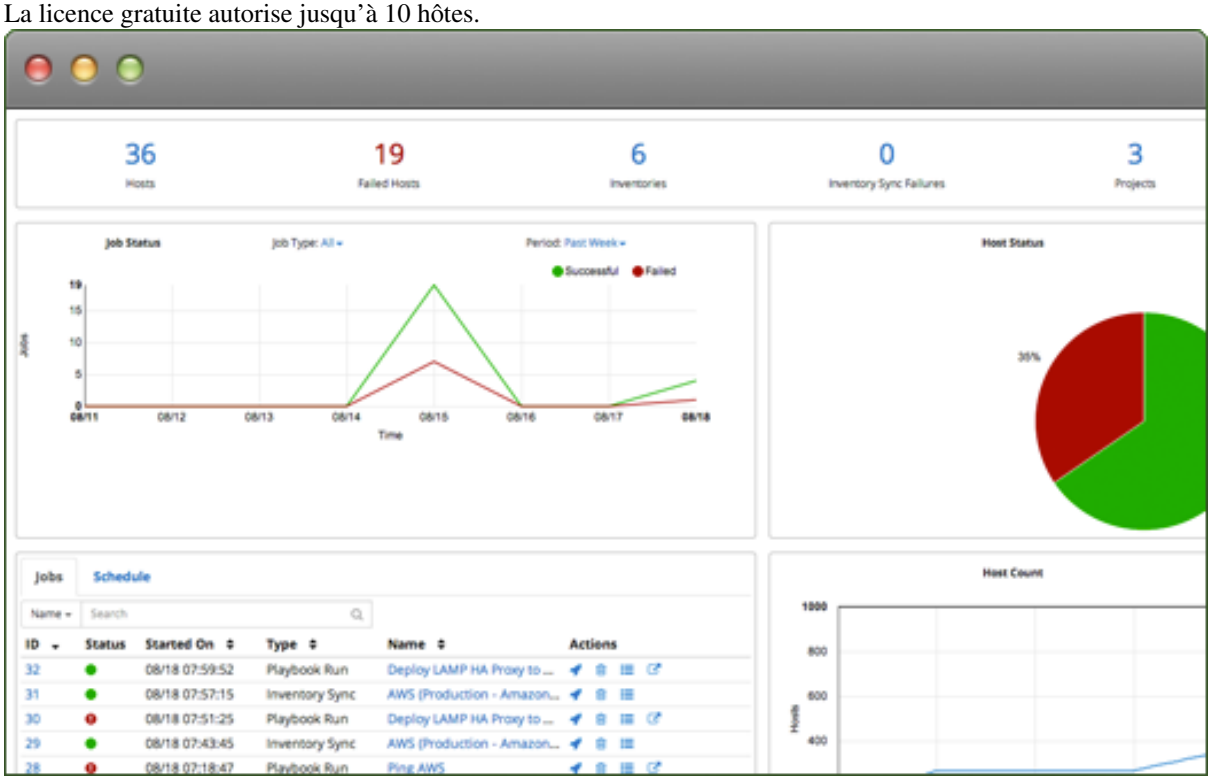
Si le fichier d'inventaire est exécutable, Ansible l'exécute et récupère la liste des groupes et hôtes sur la sortie standard, au format JSON.

Ansible Tower : l'interface graphique

Ansible fournit une interface web permettant de gérer plus facilement les playbooks Ansible. Cet outil permet également d'ordonnancer l'exécution de différents playbooks.

Il permet aussi de récupérer des informations sur les modifications effectuées, ou encore de gérer des droits/roles par utilisateurs.

La licence gratuite autorise jusqu'à 10 hôtes.



Bonnes pratiques, Tips

Debugguer son playbook

Quelques conseils pour pouvoir plus facilement débugguer ses playbooks :

- Toujours positionner un attribut `name` à vos `task` de façon à tracer l'exécution.
- Utiliser le module `debug` pour afficher le contenu de vos variables
- Utiliser le `dry-mode` de **`ansible-playbook`** (à noter que certaines tâches peuvent tout de même être exécutées si on le souhaite en positionnant leur attribut `always_run` à `true`

```
$ ansible-playbook --syntax-check ...
$ ansible-playbook --check ...
```

Bien organiser son inventaire

En plus de la notion de groupes, on peut très bien utiliser des inventaires différents, avec des noms explicites, par exemple `production`, `development`. Il suffira ensuite d'utiliser la commande :

```
ansible-playbook -i production mon_playbook.yml
```

Utiliser les tags

Ne pas hésiter à définir des tags qui permettent de sélectionner (`--tags`) ou d'exclure (`--skip-tag`) un sous-ensemble des tâches.

OS hétérogènes

Dans le cas d'un environnement avec des OS hétérogènes, on peut utiliser les facts pour savoir sur quel type d'OS on s'éxécute (`when`), et on peut inclure des fichiers de variables (ou templates, tasks, etc.) en fonction de ce critère.

Exemple :

```
- name: Récupération des variables propres à l'OS.
  include_vars: "{{ ansible_os_family }}.yml"

- include: setup-RedHat.yml
  when: ansible_os_family == 'RedHat'

- include: setup-Debian.yml
  when: ansible_os_family == 'Debian'
```

Plusieurs includes

Numéroter ses fichiers inclus pour connaître leur ordre d'apparition sans avoir à consulter le playbook :

```
- include: 01_install.yml
- include: 02_config.yml
```

Commandes et Shell

Les modules `command` et `shell`, bien que pratiques quand un module n'existe pas pour effectuer certaines actions, doivent être utilisés avec parcimonie, et il convient de s'assurer que leur utilisation sera idempotente (via par exemple l'utilisation des arguments *creates* ou *removes*).

Faire référence à la machine en cours de traitement

ansible_hostname permet d'utiliser le nom renvoyé par la machine (fact).

inventory_hostname permet d'utiliser le nom utilisé via ansible. (*inventory_hostname_short* permet de l'avoir sous sa forme courte, non fqdn).

play_hosts permet de récupérer la liste des machines sur lesquelles on est en train d'exécuter notre playbook.

```
- debug: msg="Execution sur {{ ansible_hostname }}
- debug: msg="Execution sur {{ inventory_hostname }}
- debug: msg="Execution planifie sur {{ item }}"
  with_items: "{{ play_hosts }}"
```

Modules courants

Les modules présentés ici donnent un aperçu des éléments disponibles et de leur utilisation, mais ne constituent pas une liste exhaustive.

La liste complète des modules core et extra disponibles pour Ansible est accessible en ligne.

Utilitaires

debug

Ce module permet l'affichage de messages et variables :

```
- debug: msg="Démarrage de {{ foo }} sur {{ ansible_hostname }}
- debug: var=ma_variable
```

pause

Le module `pause` permet de mettre en attente l'exécution d'un playbook. Ce module offre deux possibilités pour interrompre la pause :

- soit après un certain laps de temps
- soit sur action utilisateur

Attente de 10 secondes :

```
pause:
  seconds: 10
```

Prompt utilisateur :

```
pause:
  prompt: "Valider le démarrage du service X"
```

wait_for

Ce module permet de mettre l'exécution du playbook en pause, jusqu'à ce qu'une condition soit vérifiée. Les conditions peuvent être de plusieurs types :

- présence/absence d'un fichier
- présence/absence d'une chaîne dans un fichier
- accessibilité d'un port réseau

```
- name: attente du port 80
  wait_for:
    port: 80
    delay: 10
```

Packaging

cpanm

Cpan permet l'installation de modules perl :

```
cpanm:
  name: LaTeXML
```

gem

Gem permet l'installation de dépendances ruby :

```
gem:
  name: vagrant
  version: 1.0
  state: present
```

npm

Npm permet l'installation de paquets node.js :

```
npm:
  name: postgresql
  state: present
  production: yes
```

pip

Pip permet l'installation de logiciels python :

```
pip:
  name: ansible
  state: present
  virtualenv: ~/venvs/tools
```

Gestion des fichiers

copy

`copy` permet de copier un fichier local (ou un répertoire de façon récursive) vers l'hôte distant :

```
copy:
  src: files/appli.conf
  dest: /etc/appli.conf
  owner: root
  group: admin
  mode: 0640
```

Note : Pour des copies de très nombreux fichiers, il faudra plutôt se tourner vers le module `synchronize` (rsync) ou `unarchive`.

fetch

`fetch` exécute l'opération inverse de `copy`, il télécharge un fichier depuis un hôte distant. Si l'option `flat` n'est pas utilisée ce module crée un dossier du même nom que l'hôte, dans lequel l'arborescence complète du fichier source est recréée :

```
fetch:
  src: /var/log/syslog
  dest: fetched/
```

get_url

`get_url` permet de récupérer un fichier (ou un répertoire de façon récursive) au travers d'un serveur HTTP, HTTPS ou FTP.

On peut aussi forcer la récupération d'un fichier si la destination n'existe pas déjà.

```
get_url:
  url: http://example.com/path/file.conf
  dest: /etc/foo.conf
  sha256sum: b5bb9d8014a0f9b1d61e21e796d78dccdf1352f23cd32812f4850b878ae4944c
```

file

`file` permet de modifier les attributs d'un fichier (propriétaire, groupe...). Ce module permet également de créer des dossiers, liens symboliques, etc. :

```
- file:
    path: /etc/shadow
    owner: root
    group: root
    mode: 0400
- file:
    path: /etc/config_dir
    state: directory
```

lineinfile

`linefile` permet d'insérer/modifier/supprimer une ligne dans un fichier. L'option `regexp` permet de modifier facilement une ligne existante :

```
lineinfile:
  dest: /etc/apache2/ports.conf
  regexp: "^Listen"
  line: "Listen 8080"
```

Note : Le module `ini_file` est mieux adapté pour gérer les fichiers au format INI.

stat

`stat` ne modifie rien mais récupère des informations sur l'état d'un fichier. Ce module permet par exemple de valider la présence d'un fichier pour déclencher des actions conditionnelles :

```
- stat:
    path: /etc/already-configured
  register: st

- include: configure.yml
  when: st.stat.exists == false
```

Gestion de sources

Les modules `bzr`, `git` et `subversion` permettent de cloner ou mettre à jour des dépôts de code.

Exemple d'utilisation :

```
- git:
    repo: https://github.com/ObjectifLibre/ansible-modules-core.git
    dest: /usr/local/lib/ansible
    version: 1.8.0
  register: git_result

- shell: myscript.sh
  when: git_result|changed
```

Bases de données

MySQL

Plusieurs modules permettent d'administrer des serveurs MySQL :

- `mysql_db` crée des bases de données
- `mysql_user` gère les utilisateurs et droits sur les bases

PostgreSQL

Plusieurs modules permettent d'administrer des serveurs PostgreSQL :

- `postegresql_db` crée des bases de données
- `postegresql_user` gère les utilisateurs et droits sur les bases

Autres bases de données

Des modules additionnels permettent de gérer des bases de données NoSQL :

- `mongodb_user` pour administrer les utilisateurs et permissions de MongoDB
- `redis`

Cloud / Virtualisation

Ansible dispose de modules pour la gestion d'hyperviseurs et plateformes de cloud computing :

- Docker
- Libvirt
- VMware
- OpenStack
- Amazon
- Azure
- Google