Ansible Ultimate Edition

Release 0.0.1

www.wescale.fr

Feb 17, 2023

CONTENTS

Ansible est aujourd'hui (2022) un outil mûr, extrêmement polyvalent et adopté par une très large communauté.

Que vous soyez néophytes ou confirmés dans son utilisation, vous devriez trouvez ici de quoi approfondir votre maîtrise.

Nous reprendrons ensemble les bases avant de partager avec vous des patterns d'utilisation qui vous permettront d'utiliser Ansible dans la VraieVie©.

Nous n'avons pas la prétention de détenir LaVérité© mais ce qui est présenté ici a subi l'épreuve du terrain avec succès.

Des exercices sont fournis pour vous guider à reproduire les exemples en conditions réelles chez vous.

Ce guide a été assemblé avec amour par WeScale pour Devoxx 2022.

CHAPTER

ONE

LES BASES

Nous ne reprendrons pas ici les détails mais uniquement les concepts de haut niveau nécessaires à la compréhension des sections avancées. Chaque sous-section contient les liens pertinents vers la documentation officielle pour creuser en autonomie.

Le but de ce guide est de vous fournir une aide à la lecture qui vous permettra (nous l'espèrons) de vous y retrouvez plus facilement dans la littérature Ansible disponible.

1.1 Concepts

Ansible est un logiciel libre (GPLv3) d'automatisation à large spectre. Ansible est notamment très adapté pour la configuration et la gestion des ordinateurs. Il combine le déploiement de logiciels multi-hosts et multi-os, l'exécution de tâches *ad hoc* et la gestion de configuration.

Il interagit avec ces différents hosts à travers SSH (par défaut, de nombreuses options sont disponibles) et ne nécessite l'installation d'aucun logiciel supplémentaire sur ceux-ci.

Le but principal du projet Ansible est de rester simple d'accès et d'utilisation. Le projet déploie également de gros efforts sur la sécurité et la fiabilité. Le langage d'expression d'Ansible est notamment pensé pour être auditable par des humains non formés à Ansible.

1.1.1 Ansible...

- est codé en Python et distribué comme un package PyPi ;
- exécute des actions décrites en YAML (écrit par ses utilisateurs) ;
- peut exécuter des actions sur un ou plusieurs systèmes distants ;
- repose sur SSH comme mécanique de connexion par défaut ;
- vise à l'idempotence ;
- s'installe dans l'espace utilisateur ;
- ne nécessite pas l'installation d'agent sur les machines pilotées.

1.1.2 Lexique

Inventaire

Liste de machines avec lesquelles interagir, organisées en groupes. Un inventaire peut être :

- statique : un fichier au format INI, YAML ou JSON
- dynamique : un retour JSON sur le stdout d'un programme à exécuter par Ansible

Exemple d'inventaire statique

```
web_1 ansible_host=10.10.10.101 ansible_ssh_private_key_file=...
web_2 ansible_host=10.10.10.102 ansible_ssh_private_key_file=...

[web_servers]
web_1
web_2

[databases]
db_1 ansible_host=10.10.20.201 ansible_ssh_private_key_file=...
10.10.20.202 ansible_ssh_private_key_file=...

[production:children]
web_servers
databases
```

Exemple d'inventaire dynamique

```
$ ./dyn_inventory --list
{
  "databases" : {
    "hosts" : [ "db_1", "10.10.20.202" ]
  },

  "web_servers" : [ "web_1", "web_2" ],

  "production" : {
    "children": [ "web_servers", "databases" ]
  }
}

$ ./dyn_inventory --host web_1
{
  "ansible_host": "10.10.10.101"
}
```

Les inventaires dynamiques peuvent prendre la forme de scripts et donc forger leur réponse en interrogeant des API qui connaissent les machines auxquelles se connecter (AWS, OpenStack, Cobbler, *etc.*).

Module/Task

C'est la plus petite unité d'action disponible. Le résultat de son exécution par Ansible peut être `changed`, `ok` ou `failed`.

Exemple d'appel au module `apt`

```
- name: un joli titre c'est mieux
  apt:
    pkg: "tmux"
    state: present
```

Handler

Task lancée en fin de play si "notifiée" par une autre task, c'est-à-dire si la task porte un attribut `notify` avec le `name` du handler et que son résultat d'exécution est `changed`. Un handler notifié plusieurs fois ne sera exécuté qu'une fois.

Exemple de handler

```
---
- hosts: webservers
  tasks:
    - template: [...]
      notify: restart my service

  handlers:
    - name: restart my service
      service:
        name: my_service
        state: restarted
```

Rôle

Unité de réutilisation de code Ansible. Peut contenir des tasks, des handlers, des templates, des fichiers et des variables.

Organisation d'un rôle

```
├── defaults
│   └── main.yml --> variables par défaut (aisément redéfinissables)
├── files        --> fichiers statiques
├── handlers
│   └── main.yml --> handlers
├── meta
│   └── main.yml --> fiche d'info et dépendances
├── tasks
│   └── main.yml --> tâches (appels de modules)
├── templates    --> templates Jinja2
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml --> variables redéfinissables uniquement par option en ligne de commande
```

Playbook

Liste d'objets Play. Un Play est une liste de rôles et/ou de tasks à appliquer sur un groupe de machines cibles.

Exemple de fichier playbook contenant 2 plays

```
---
- hosts: web_servers
  become: yes
  tasks:
    - service:
        name: sshd
        state: stopped

- hosts: web_servers
  become: yes
  tasks:
    - service:
        name: sshd
        state: started
```

Collection

Unité de réutilisation de code Ansible. Peut contenir des rôles, des playbooks et des plugins.

Organisation d'une collection

```
├── docs/
├── galaxy.yml
├── meta/
│   └── runtime.yml
├── plugins/
│   ├── modules/
│   │   └── module1.py
│   ├── inventory/
│   └── .../
├── README.md
├── roles/
│   ├── role1/
│   ├── role2/
│   └── .../
├── playbooks/
│   ├── files/
│   ├── vars/
│   ├── templates/
│   └── tasks/
└── tests/
```

Approfondir

- Documentation Ansible - Accueil
- Documentation Ansible - Rôles

- Documentation Ansible - Collection
- Jinja2 Template Designer Documentation
- YAML Specification 1.1
- SSH config manpage

1.2 Cas d'usage

Ansible est un outil d'automatisation très versatile. Vous pouvez en tirer avantage pour tous les cas suivants :

- automatisation d'installation sur tous types de machines distantes (équipements réseau, serveurs…);
- langage de scripting local ;
- gestion de configuration (templating, actions de post-configuration, *etc.*) ;
- orchestration de migrations (base de données, flux réseaux, *etc.*) ;
- tests d'infrastructure.

Perle de sagesse

Ansible vous permet d'écrire des scripts qui s'exécutent à travers tout votre SI sans restriction.

La grande force d'Ansible réside dans les très nombreux modules disponibles sur ansible galaxy qui vous permettront d'enchaîner, au sein d'une même exécution :

- un post RabbitMQ ;
- une modification de flux sur une appliance réseau ;
- une extinction de service sur un serveur ;
- une modification d'enregistrement DNS.

La principale question est de savoir ce que vous avez envie de faire. Tant qu'il s'agit de sujet DevOps/PureOps, entre les modules existants et les capacités d'extension, il y a forcément une façon de tirer parti d'Ansible pour vous faciliter la vie.

Approfondir

- Ansible Galaxy - Accueil

1.3 Installation

Outre le guide officiel, qui est la référence pour toutes les plateformes, voici quelques retours de terrain pour vous garder dans une zone de confort, à l'écart des problèmes imprévus.

La diversité des systèmes est le premier puzzle à résoudre pour permettre à une équipe de travailler efficacement. L'approche préconisée diminue la surface de frottement avec le système au maximum pour s'attaquer très tôt aux éventuels problèmes et ne plus avoir à y penser par la suite.

1.3.1 Méthode

Packages Pip dans un virtualenv

La meilleure façon d'installer Ansible de notre point de vue est de s'appuyer sur un virtualenv pour chacun de vos projets. Cela permet d'isoler les dépendances projet du système, mais également d'un projet à un autre et de pouvoir travailler avec plusieurs versions fixes d'Ansible en parallèle.

Une release Ansible officielle commence par une publication sur PyPi. Le travail des mainteneurs démarre de ce point, donc autant se greffer ici pour être le plus à jour possible.

Il existe de nombreuses manières de réaliser cette gestion de virtualenv. Je vous livre ici une méthode qui est robuste et nécessite peu d'expertise Python et un minimum de prérequis (comprendre : on n'a pas réussi à faire moins jusqu'ici).

Mise en pratique

Installer Ansible

Alternative - Packages Pip au niveau système

L'installation de packages Python au niveau système, que ce soit pour un ou tous les utilisateurs, peut rapidement apporter des problèmes de conflits de version sur des dépendances. Source de problèmes, cette méthode est donc écartée.

Alternative - Packages systèmes

À moins que vous en ressentiez le besoin parce que vous maîtrisez déjà Ansible : déconseillé. Les packages systèmes sont rarement à jour et vous allez rapidement avoir un décalage entre la documentation que vous pourrez consulter et la version que vous avez installé. C'est une source d'erreur facilement évitable.

Alternative - Conteneur de travail

Certains fans de conteneurs voudront enfermer Ansible pour s'assurer de l'isolation totale des dépendances de Dev avec le système. C'est un chemin respectable et vous trouverez en bas de page un guide pour vous aidez.

Nous ne développons pas cette technique ici, dans la mesure où elle implique des connaissances supplémentaires et que nous visons une approche très Lean d'Ansible.

Approfondir

- Documentation Ansible - Installation
- Guide de construction de conteneur de travail Ansible

1.4 Organisation de projet

1.4.1 Fichiers et répertoires

L'organisation des collections Ansible colle parfaitement avec tous les besoins d'un projet. Par conséquent, il n'est pas utile de tenter d'établir un nouveau standard exotique. On s'en tient à la documentation ! Votre code sera beaucoup plus maintenable si vous adoptez les standards de la communauté.

Perle de sagesse

Tout projet Ansible contenant des playbooks doit être une collection.

Mise en pratique

Créer un projet

1.4.2 Configuration

Le plus efficace est d'adopter une méthode de gestion de la configuration qui puisse convenir à Ansible mais également à d'autres outils qui seront impliqués dans notre développement. La méthode la plus tout-terrain est celle qui passe **par les variables d'environnement**. Cela tombe bien c'est aussi une préconisation du manifeste "12-factor app". On peut configurer Ansible de cette façon, mais aussi :

- Terraform (et l'immense majorité des providers Terraform) ;
- AWS CLI ;
- kubectl ;
- helm ;
- et bien d'autres.

C'est une méthode de travail fiable et pratique, pour peu qu'on soit un peu aidé. C'est aussi un des aspects qui nous ont poussés à choisir `direnv` pour la gestion de virtualenv décrite dans la section *Installation*. `Direnv` charge tout fichier `.envrc` se trouvant dans un répertoire dans lequel on se place, mais il remet l'environnement à son état d'origine dès qu'on en sort. Il n'y a donc plus de risque d'oublier une variable d'environnement en changeant de projet et qu'elle vienne impacter un autre projet.

On va donc s'appuyer sur le fichier `.envrc` dédié à `direnv` pour y placer la configuration commune à toute l'équipe.

Quelques clés de configuration Ansible à connaître (et les valeurs conseillées) :

- `ANSIBLE_STDOUT_CALLBACK="ansible.posix.debug"` : formatage de la sortie standard d'Ansible pour la rendre lisible par un humain (interpréter les '\n' notamment).
- `ANSIBLE_INVENTORY="inventory"` : le fichier d'inventaire pris par défaut.
- `ANSIBLE_FORKS="10"` : nombre de connexions SSH simultanées qui vont appliquer les playbooks aux machines cibles.
- `ANSIBLE_ROLES_PATH="roles"` : chemin où Ansible va rechercher les rôles inclus depuis les playbooks.
- `ANSIBLE_CALLBACKS_ENABLED="timer,profile_tasks"` : ajout d'une prise de mesure des performances de chaque tasks et un top 15 des plus chronophages en fin d'exécution de playbook.

Approfondir

- Documentation Ansible - Configuration Settings
- the twelve-factor app - Configuration

1.4.3 Configuration personnelle

Outre les configurations communes destinées au comportement d'Ansible ou de vos autres outils, il émerge un besoin de façon quasi-systématique : redéfinir une valeur de configuration ou y ajouter des valeurs spécifiques à chaque membre d'équipe (comme un chemin absolu incluant le nom de l'utilisateur).

Pour cela, encore une fois, `direnv` nous permet de régler le problème de façon simple. Il suffit d'ajouter une section de chargement d'un autre fichier depuis notre `.envrc`:

```
layout python3

ENV_ADDONS=".env.local .env.personal .env.secrets"
for addon in ${ENV_ADDONS}; do
    if [ -e "${PWD}/${addon}" ]; then
        source "${PWD}/${addon}"
    fi
done
```

Évidemment, on git-ignore les fichiers `.env.local`, `.env.personal` et `.env.secrets` pour s'assurer qu'ils ne soient pas poussés ailleurs que notre machine de travail.

C'est typiquement dans ce fichier git-ignoré que vous pourrez sereinement ranger des variables d'environnement comme `AWS_PROFILE`, `SCW_SECRET_KEY` et autres `GANDI_API_KEY`. Le chargement sera assuré par `direnv` dès que vous entrez dans le répertoire et sera déchargé en le quittant. Cela vous met à l'abri des problèmes de mauvaise clé active quand vous changez de contexte projet.

Cette convention ne coûte pas cher à mettre en place et n'ajoute aucune dépendance d'outil.

Mise en pratique

Configurer un projet

1.5 Gestion des dépendances

1.5.1 Dépendances Python

Autant ne rien inventer et suivre les coutumes locales :

- un fichier `requirements.txt` au format attendu par Pip ;
- une commande `pip install -U requirements.txt`.

Pas besoin de plus pour les dépendances de librairies Python. Les étapes précédentes de *gestion de virtualenv dédié* au projet commencent à payer puisque cette approche va naturellement installer les librairies dans le répertoire du virtualenv, proprement isolé d'un projet à l'autre.

Perle de sagesse

Tous les projets Python qui ont des dépendances devraient avant tout installer les modules `pip`, `wheel` et `setuptools` pour éviter des problèmes communs de build de paquets Pip.

1.5.2 Dépendances Ansible

Les dépendances Ansible sont de 2 natures :

- des rôles ;
- des collections.

Par chance (ou intelligence des designers d'Ansible), les 2 peuvent être décrits dans le même fichier. Ce fichier de dépendances se nomme par convention `requirements.yml` et ressemble à ceci :

```
---
roles:
  # Install a role from Ansible Galaxy.
  - name: geerlingguy.java
    version: 1.9.6

collections:
  # Install a collection from Ansible Galaxy.
  - name: geerlingguy.php_roles
    version: 0.9.3
    source: https://galaxy.ansible.com
```

Approfondir

- Documentation Ansible - Le requirements.yml

1.5.3 Plus haut, plus vite, plus fort

Si vous lisez ce texte, nous pouvons supposer qu'une part de votre travail est d'automatiser des choses pour aller plus vite.

Par conséquent, poussons un petit cran plus loin. L'ajout d'un fichier `Makefile` pour grouper toutes les commandes de rapatriement des dépendances est assez confortable sur le long terme.

Le choix de Makefile a été dicté par sa grande disponibilité sur un ensemble de système (il est même souvent dans les packages par défaut).

Voici ce que pourrait donner un Makefile un peu travaillé pour se faciliter la vie :

```
.PHONY: env
env-desc = "Setup local dev env"
env:
        @echo "==> $(env-desc)"

        @[ -d "${PWD}/.direnv" ] || (echo "Venv not found: ${PWD}/.direnv" && exit 1)
        @pip3 install -U pip wheel setuptools --no-cache-dir && \
```

(continues on next page)

(continued from previous page)

```
        echo "[  OK  ] PIP + WHEEL + SETUPTOOLS" || \
        echo "[FAILED] PIP + WHEEL + SETUPTOOLS"

        @pip3 install -U --no-cache-dir -r "${PWD}/requirements.txt" && \
        echo "[  OK  ] PIP REQUIREMENTS" || \
        echo "[FAILED] PIP REQUIREMENTS"

        @ansible-galaxy install -fr ${PWD}/requirements.yml && \
        echo "[  OK  ] ANSIBLE-GALAXY REQUIREMENTS" || \
        echo "[FAILED] ANSIBLE-GALAXY REQUIREMENTS"
```

Une fois toutes *Les bases* respectées, on obtient une convention où un nouvel arrivant sur le projet à besoin de :

- Direnv
- Git
- Python
- Make

… et devient capable de produire du code après avoir lancé :

- `git clone [...]`
- `direnv allow .`
- `make env`

Cela paraît abordable par le plus grand nombre, même débutants. Ce n'est évidemment pas la seule façon de faire les choses, mais de toutes celles qu'on a essayé, c'est clairement le chemin de l'effort et de l'emmerdement minimal.

Mise en pratique

Gérer les dépendances

1.6 Connexion aux serveurs cibles

1.6.1 Rappels utiles

La méthode de connexion par defaut d'Ansible est SSH. Par conséquent, si vous arrivez à vous connecter avec un client SSH quelconque, Ansible DOIT y arriver aussi.

Les paramètres de connexion SSH utilisés par Ansible peuvent être placés à plusieurs endroits, en fonction de la saison et du sens du vent. Là encore, choisir une convention et s'y tenir vous facilitera la vie.

1.6.2 Configuration SSH

La méthode la plus efficace pour renseigner les informations de connexion est de construire un fichier de configuration SSH pur. Le gros avantage de cette approche est de pouvoir tester la connectivité en sortant Ansible de la boucle. Une fois le fichier de configuration SSH mis au carré, il faut configurer Ansible pour qu'il s'appuie exclusivement sur cette configuration SSH.

Exemple de configuration SSH

```
#
# ssh.cfg
#
Host ultimate-controller
  Hostname 51.15.202.92

Host ultimate-master
  Hostname 192.168.42.1

Host ultimate-minion
  Hostname 192.168.42.11

Host ultimate-master ultimate-minion
  ProxyJump ultimate-controller

Host ultimate-*
  User root
  IdentityFile  group_vars/ultimate_platform/secrets/ultimate.key
  StrictHostKeyChecking no
  ControlMaster   auto
  ControlPath     ~/.ssh/mux-%r@%h:%p
  ControlPersist  15m
  ServerAliveInterval 100
  TCPKeepAlive yes
```

La maîtrise des options de configuration de client SSH vous apportera un grand confort au quotidien. Les options qu'on retrouve dans l'exemple sont des classiques dans la pratique Ansible.

```
Host ultimate-controller
  Hostname 51.15.202.92
```

Vous permet de vous connecter via une commande `ssh -F ssh.cfg ultimate-controller`.

```
Host ultimate-master ultimate-minion
  ProxyJump ultimate-controller
```

Vous permet de rebondir de façon transparente au travers d'une connexion SSH à `ultimate-controller` pour atteindre `ultimate-master` ou `ultimate-minion`, qui peuvent donc être configurés avec des adresses IP sur un réseau privé routable depuis `ultimate-controller`.

```
Host ultimate-*
  [...]
```

Vous permet de grouper les options par patterns de nommage, pour éviter de créer des fichiers de configuration trop verbeux.

```
Host ultimate-*
  [...]
  ControlMaster   auto
  ControlPath     ~/.ssh/mux-%r@%h:%p
  ControlPersist  15m
  ServerAliveInterval 100
  TCPKeepAlive yes
```

Renforce les comportements de maintien de connexion de SSH, afin d'éviter à Ansible de relancer une session à chaque task. Cela améliore grandement les performances globales des playbooks.

Si on reprend, les étapes sont donc :

- construire un fichier de configuration SSH (local au projet), nommé par exemple `ssh.cfg`
- s'assurer qu'on peut se connecter aux machines cibles via une commande `ssh -F ssh.cfg le_serveur`
- ajouter la mention `export ANSIBLE_SSH_ARGS="-F ${PWD}/ssh.cfg"` à la configuration Ansible (le désormais fameux fichier `.envrc`)

Approfondir

- SSH config manpage

1.6.3 Cohérence SSH-Config/Inventaire

Une fois que la connectivité est assurée en SSH pur, il ne nous reste plus qu'à tailler notre inventaire en reprenant les labels attribués aux hosts dans la configuration SSH.

```
#
# ssh.cfg
#
Host ultimate-controller
  Hostname 51.15.202.92

Host ultimate-master
  Hostname 192.168.42.1

Host ultimate-minion
  Hostname 192.168.42.11
[...]
```

On a donc 3 hosts `ultimate-controller`, `ultimate-master` et `ultimate-minion` à répartir dans notre inventaire en taillant les groupes qui nous conviennent.

```
#
# ansible inventory
#
# Pour l'exemple un groupe de host incluant la totalité de nos cibles
[ultimate_platform:children]
ultimate_master_nodes
ultimate_minions_nodes
ultimate_bastions

# un groupe des noeuds masters avec notre host dedans, tel que nommé dans la conf SSH
[ultimate_master_nodes]
ultimate-master

# un groupe des noeuds minions avec notre host dedans, tel que nommé dans la conf SSH
[ultimate_minion_nodes]
ultimate-minion

# un groupe des noeuds de contrôle avec notre host dedans, tel que nommé dans la conf SSH
[ultimate_bastions]
ultimate-controller
```

1.6.4 Validation de la connectivité

Pour valider la connectivité avec les hosts de notre inventaire, il suffit de lancer une commande :

```
$ ansible -m ping all
ultimate-controller | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ultimate-master | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ultimate-minion | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Si par hasard, au fil du projet, Ansible présente une erreur de connexion, commencez directement votre troubleshooting au niveau configuration SSH par un :

```
$ ssh -F ssh.cfg ultimate-master
```

Mise en pratique

Premier host cible

CHAPTER

TWO

CRAFTSMANSHIP

2.1 Ansible-Lint

`ansible-lint` est un outil d'analyse statique de code dédié à Ansible. Il va parcourir les sources Ansible qu'il détecte et générer un rapport des violations par rapport à un ensemble de règles maintenu par la communauté. Il est bien sûr possible de le configurer afin de limiter son périmètre d'analyse ou les règles qu'il doit vérifier.

C'est un must-have sur tout projet Ansible afin d'avoir un rapport objectif sur la qualité du code produit.

2.1.1 Installation

Partant du principe que vous avez suivi les recommandations du chapitre sur *Les bases*, pour installer ansible-lint :

- Dans le fichier `requirements.txt` de votre projet, ajoutez une ligne `ansible-lint[yamllint]`
- Lancez la commande : `make env`

2.1.2 Configuration

Plutôt que de vous attaquer à la maîtrise des options CLI de `ansible-lint`, mieux vaut simplement rajouter un fichier de de configuration `.ansible-lint` à la racine de votre projet. L'exemple donné si dessous peut convenir pour un grand nombre de projets mais pensez à approfondir votre maîtrise de l'outil en allant creuser les possibilités par vous-même.

```
# .ansible-lint
#
# Reference documentation:
#   https://ansible-lint.readthedocs.io/en/latest/configuring.html#configuration-file
#
# Chemins à exclure de l'analyse statique
#
exclude_paths:
  - .cache/
  - .github/
  - .direnv/
  - .git/
  - keys/
  - secrets/
  - tests/
  - requirements.yml
```

(continues on next page)

(continued from previous page)

```
  - molecule.yml
parseable: true     # Utiliser un format parseable de rapport
quiet: false        # Limiter le contenu du rapport à son strict minimum
verbosity: 1        # Niveau de verbosité du rapport
#
# Oblige les variables de boucles à être nommées avec ce préfixe, comme ceci :
#
#   - name: Exemple conforme
#     debug:
#       msg: "Élément courant {{ __current_name }}"
#     loop:
#       - "premier"
#       - "second"
#     loop_control:
#       loop_var: "__current_name"
#
loop_var_prefix: "__current_"
#
# Active le jeu de règles par défaut
#
use_default_rules: true
#
# Ignore toutes les règles listées dans cette 'skip_list'
#
skip_list:
  - fqcn-builtins   # Pas besoin de surcharger le code avec des noms de modules complets␣
↪quand il
                    # s'agit de 'ansible.builtin.*'

  - meta-no-info    # La plupart des projets internes n'ont pas besoin de renseigner un␣
↪fichier
                    # 'meta/main.yml' pour leurs rôles. À retirer si vous comptez publier.
#
# Certaines règles ont un tag 'opt-in', elles ne sont pas activées à moins de les inclure
# dans cette 'enable_list'
#
enable_list:
  - no-log-password # Vérifie tant que possible que des passwords ne sont pas loggés
  - no-same-owner   # Vérifie que les transfert de fichier mentionnent 'owner' et 'group'
  - yaml            # Intègre un rapport YAML-Lint dans le rapport d'Ansible-Lint
#
# Règles à relever comme des Warning et non des Fautes
#
warn_list:
  - git-latest      # Les usages du module 'git' devraient mentionner la gitref ciblée.
  - experimental    # Relève les usages de modules Ansible marqués comme expérimentaux␣
↪(par défaut)
#
# Désactive l'installation du requirements.yml
#
offline: true
```

2.1.3 Exécution

Pour lancer un scan de votre code, lancez simplement :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training

$ ansible-lint
[...]
roles/rproxy/defaults/main.yml:12: yaml no new line character at the end of file (new-
↪line-at-end-of-file)
roles/rproxy/meta/main.yml:6: yaml comment not indented like content (comments-
↪indentation)
roles/rproxy/tasks/install.yml:6: no-loop-var-prefix Role loop_var should use configured␣
↪prefix.: rproxy_
roles/rproxy/tasks/install.yml:30: no-loop-var-prefix Role loop_var should use␣
↪configured prefix.: rproxy_
roles/rproxy/tasks/install.yml:39: no-loop-var-prefix Role loop_var should use␣
↪configured prefix.: rproxy_
roles/rproxy/tasks/install.yml:49: no-loop-var-prefix Role loop_var should use␣
↪configured prefix.: rproxy_
roles/rproxy/tasks/install.yml:58: no-loop-var-prefix Role loop_var should use␣
↪configured prefix.: rproxy_
roles/rproxy/tasks/install.yml:75: unnamed-task All tasks should be named
roles/rproxy/tasks/install.yml:75: yaml no new line character at the end of file (new-
↪line-at-end-of-file)
roles/rproxy/tasks/main.yml:5: yaml missing starting space in comment (comments)
roles/rproxy/tasks/main.yml:20: unnamed-task All tasks should be named
[...]
Finished with 51 failure(s), 27 warning(s) on 128 files.
```

Il ne vous reste plus qu'à en faire l'intégration dans votre chaine de CI préférée pour avoir un contrôle de qualité en continu.

Approfondir

- Configurer Ansible-Lint
- Liste complète des règles Ansible-Lint

2.2 Molecule

Molecule est un outil dédié aux tests de rôles Ansible. Molecule permet de tester aussi bien avec des containers qu'avec de l'infrastructure éphémère. Il apporte un workflow pour :

- définir et gérer les cycles de vie des environnements de test ;
- appliquer des playbooks de setup/teardown ;
- appliquer des rôles Ansible ;
- lancer des suites de tests de validation avec plusieurs frameworks.

Les environnements de tests peuvent être provisionnés avec à peu près n'importe quelle solution du moment qu'elle est pilotable par Ansible. Pour les technologies les plus courues (Docker, Podman, *etc.*), des intégrations existent. Il reste toujours possible de coder ses propres implémentations du workflow en playbook Ansible pour intégrer n'importe quel outil.

Mise en pratique

- *Molecule & Docker*
- *Molecule & Terraform*

2.2.1 Cycle de vie

Le workflow d'un scenario de test Molecule comprend les étapes suivantes :

- `lint` : Analyse statique du code.
- `destroy` : Destruction des éventuels environnements de tests encore existants.
- `dependency` :Rapatriement des dépendances nécessaires listées dans le fichier `meta.yml`.
- `syntax` : Validation de syntaxe avec Ansible.
- `create` : Creation de l'environnement de test du scénario.
- `prepare` : Application d'un playbook sur l'environnement de test pour préparer l'application du rôle en cours de test.
- `converge` : Application du rôle testé, doit s'exécuter sans erreur.
- `idempotence` : Seconde application du rôle testé, qui ne doit généré aucune `task` en état `changed`.
- `side_effect` : Partie obscure et mal documentée du framework, cette phase doit permettre d'introduire des effets de bord à l'environnement (modification contextuelle au rôle, simulation de pannes).
- `verify` : Application du code de test (playbooks, TestInfra, etc.)
- `destroy` : Destruction des environnements de test.

CHAPTER

THREE

BONNES PRATIQUES

3.1 Inventaires

3.1.1 Dynamiques ou statiques ?

Tout va dépendre de votre cas et de la géométrie du parc que vous gérez avec Ansible…

Parmi les bonnes pratiques poussées par Ansible, il est conseillé de privilégier des inventaires dynamiques dès lors que vous êtes dans un milieu très dynamique comme un fournisseur de Cloud. Le contre-argument est que si vous devez gérer un parc de VM dans un milieu Cloud, vous aurez suûrement mieux à faire que de vous connecter en SSH aux machines pour quelque opération que ce soit. Dans ce genre de cas, Ansible a bien plus sa place au niveau d'un pipeline de build d'images serveur dans une approche Golden Image.

Si vous choisissez de vous porter vers un inventaire dynamique, assurez vous de contrôler son retour avant de lancer un playbook à l'aveugle. En effet, le côté dynamique augmente le risque d'inclure des machines cibles imprévues. Il est très simple de choisir un périmètre trop large et de faire déborder un playbook.

La technique la plus sûre à notre avis reste celle de l'inventaire statique généré. Vous fournissez à Ansible un fichier statique mais au préalable vous le générez, avec un programme d'inventaire dynamique ou votre propre implémentation. Et surtout, entre la génération et la consommation, insérez une phase d'audit/relecture. La confiance n'exclut pas la contrôle.

3.1.2 Géométrie des groupes

Choisissez vos groupes en fonction des playbooks/rôles que vous souhaitez appliquer. Les groupements de machines dans l'inventaire doivent vous faciliter la vie, pas la complexifier.

Intégrez le nom de l'environnement dans le nom des groupes vous évitera de reconfigurer votre production. Il faut éviter à tout prix que des groupes de groupes ne contiennent des machines de différents environnements.

```
#
# NE PAS REPRODUIRE
#
[webservers]
web-01
web-02
web-03
web-04

[staging]
web-01
```

(continues on next page)

(continued from previous page)

```
web-02

[production]
web-03
web-04
```

Dans cet exemple, si je sélectionne le groupe `webservers` sans exclure explicitement les machine appartenant au groupe `production`

Préférez cette version, sans ambigüité :

```
[staging_webservers]
web-01
web-02

[production_webservers]
web-03
web-04
```

Avec cette version vous ne risquez plus de sélectionner des serveurs de plusieurs environnements par mégarde.

Idéalement, une machine doit figurer dans un minimum de groupes différents.

3.1.3 Syntaxe de sélection

Une information qui peut vous aider à structurer vtre inventaire est de connaître l'existence des patterns de sélection.

Un pattern de sélection est typiquement en toute première place dans les playbooks et les exemples de code sur la toile ne s'attardent que rarement dessus. Dans sa forme la plus simple, ça donne :

```
---
- name: Playbook de démo
  hosts: un_groupe_de_mon_inventaire
#        ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
#           pattern de sélection

  tasks:
    [...]
```

Mais la syntaxe de sélection est plus riche que cela.

Description	Pattern(s)	Cibles
tous les hosts	`all`	tous les hosts de l'inventaire (mot réservé)
un host en particulier	`host1`	
plusieurs hosts	`host1:host2`	
un groupe	`group1`	tous les hosts présents dans `group1`
plusieurs groupes	`group1:group2`	tous les hosts de `group1` ET ceux de `group2`
exclusion de groupe	`group1:!group2`	tous les hosts de `group1` ET **absents** de `group2`
intersection de groupes	`group1:&group2`	tous les hosts présents dans `group1` ET `group2`

Approfondir

Documentation Ansible - Sélection de hosts

3.1.4 Keep It Simple

Les groupes que vous façonnez dans votre inventaire doivent simplement refléter vos habitudes opérationnelles :

- quels sont les hosts qui se ressemblent ? Grouper les hosts d'un cluster semble évident.
- quels sont les hosts qui s'assemblent ? Un groupe parent `application` pour englober 2 groupes `backend` et `frontend` peut être intéressant

Il n'y a pas de véritable règle de design dans la mesure où il n'existe aucune norme sur l'organisation de machines au sein de réseaux. Garder en tête que si vous avez besoin régulièrement d'utiliser les options de limitations de `ansible-playbook`, c'est que votre design d'inventaire n'est pas en adéquation avec vos usages.

```
# Si vous voyez ce genre de commande souvent, retournez affiner votre inventaire.
$ ansible-playbooks playbooks/exemple.yml --limit='web_servers:!centos_servers'
```

Perle de sagesse

If something is too complex to understand, it must be wrong.

Arjen Poutsma, The Poutsma Principle, Xebia Essential Cards

3.2 Workspaces

Perle de sagesse

L'erreur est humaine. Répéter l'erreur sur 50 serveurs d'un coup est DevOps.

@DEVOPS_BORAT, Twitter, traduction libre.

Plus nous poussons l'automatisation, plus il est crucial de se forcer à donner un périmètre à nos lancements de procédures automatisées. Personne n'a envie de laisser une infrastructure de production se détruire en profondeur via un playbook, tout en allant prendre un café… L'excès de confiance crée des pannes.

Les conventions, de code ou de conduite, peuvent être facilement oubliées dans un instant d'égarement. Par conséquent il est plus prudent de se poser des pièges à soi-même (et à son équipe) pour empêcher certains comportements à risque.

C'est là que le concept de **Workspace** entre en scène. L'idée de Workspace Ansible consiste à se **forcer** à définir intentionnellement le périmètre d'impact de nos playbooks. On offre ainsi à notre cerveau une chance de reconnaître une erreur au moment où l'on agit.

3.2.1 Inventaires multiples

Si vous ne définissez PAS d'inventaire par défaut (clé de configuration `ANSIBLE_INVENTORY`), et que vous maintenez un inventaire statique par environnement, vous serez forcé de fournir une option d'inventaire au lancement de tout playbook :

```
$ ansible-playbook playbooks/danger_zone.yml -i inventories/dev_env.inventory
```

Cela peut être une réponse satisfaisante à notre problème de dinition de périmètre.

3.2.2 Variabiliser les patterns de sélection

Vous avez la possibilité d'intégrer des variables dans votre sélection de hosts au sein d'un playbook.

```
---
- name: Exemple de pattern variabilisé
  hosts: "{{ env_name }}_webservers"
  tasks:
    [...]
```

Dans le flot d'exécution d'Ansible, le playbook est évalué bien avant le chargement des variables. Par conséquent si vous définissez `env_name` dans un fichier de variables de groupe ou de host, vous aurez tout de même une erreur disant qu'elle n'a pas de valeur.

Vous serez obligé de fournir la valeur de `env_name` via une option de ligne de commande, comme ceci :

```
$ ansible-playbook playbooks/danger_zone.yml -e env_name=production
```

De cette façon, la variable sera disponible pour le playbook au bon moment et le playbook s'appliquera sur le groupe `production_webservers` (selon notre exemple).

Les playbooks Ansible peuvent servir à automatiser d'autres outils d'Infra-as-Code qui ont déjà ce comportement de workspace, comme Terraform, pensez à propager votre variable de workspace à ces outils pour un rendu plus homogène.

3.2.3 S'appuyer sur l'environnement

Une autre façon d'implémenter la variabilisation des patterns de hosts est de se reposer sur une variable d'environnement qui, par convention avec votre équipe, servira de clé de workspace. Cela peut se faire en passant par le lookup `env` d'Ansible :

```
---
- name: Exemple de pattern variabilisé via env
  hosts: "{{ lookup('env', 'PROJECT_X_WORKSPACE') }}_webservers"
  tasks:
    [...]
```

Évidemment, vous éviterez d'inclure une valeur par défaut de cette variable à votre configuration projet pour forcer chacun à démarrer sa session de travail par un :

```
$ export PROJECT_X_WORKSPACE="production"
```

C'est une méthode que nous utilisons fréquement et qui satisfait les équipes.

3.3 Ansible Galaxy Release

La ligne de commande `ansible-galaxy` vous permet de publier votre collection ou votre rôle sur la plateforme communautaire [galaxy.ansible.com] et ainsi de permettre à d'autres d'utiliser votre code.

Cependant, `ansible-galaxy` ne permet que de publier le répertoire courant au moment de l'exécution et ne contient pas de mécanique de templating pour le fichier de meta `galaxy.yml`. Voici un playbook pour compenser ce léger manque.

Nous vous conseillons de placer ce playbook `release_collection.yml` dans un répertoire `build/` à la racine du projet.

```
---
- hosts: localhost
  connection: local
  become: false
  gather_facts: false

  vars:
    project_repository: "project_git url"
    galaxy_namespace: your_namespace
    galaxy_name: your_collection

    project_dir: "{{ (playbook_dir + '/../') | realpath }}"
    build_dir: "{{ project_dir }}/build"
    clone_dir: "{{ build_dir }}/clone"

    dot_ansible_dir: "{{ lookup('env', 'HOME') }}/.ansible"
    galaxy_token_file: "{{ dot_ansible_dir }}/galaxy_token"
    galaxy_token: "{{ lookup('env','ANSIBLE_GALAXY_TOKEN') }}"
    src_galaxy_template: "{{ clone_dir }}/build/templates/galaxy.yml.j2"
    galaxy_version: "{{ gitref | regex_replace('v','') }}"
    galaxy_archive_name: "{{ galaxy_namespace }}-{{ galaxy_name }}-{{ galaxy_version }}.
↪tar.gz"
    galaxy_archive_file: "{{ clone_dir }}/{{ galaxy_archive_name }}"
    galaxy_meta_file: "{{ clone_dir }}/galaxy.yml"

  pre_tasks:
    - name: Ensure the ANSIBLE_GALAXY_TOKEN environment variable is set.
      assert:
        that:
          - (galaxy_token | length) > 0
        msg: Env variable 'ANSIBLE_GALAXY_TOKEN' is not set.

    - name: Ensure $HOME/.ansible directory exists
      file:
        path: "{{ dot_ansible_dir }}"
        state: directory
        mode: 0700

    - name: Write the galaxy token to $HOME/.ansible/galaxy_token
      copy:
        content: |
          token: {{ lookup('env','ANSIBLE_GALAXY_TOKEN') }}
```

(continues on next page)

(continued from previous page)

```
      dest: "{{ galaxy_token_file }}"
      mode: 0600

tasks:
  - name: Delete old clone
    file:
      path: "{{ clone_dir }}"
      state: absent

  - name: Clone project at desired gitref
    git:
      repo: "{{ project_repository }}"
      dest: "{{ clone_dir }}"
      version: "{{ gitref }}"

  - name: Render galaxy.yml
    template:
      src: "{{ src_galaxy_template }}"
      dest: "{{ galaxy_meta_file }}"
      mode: 0644
    register: galaxy_yml_rendering

  - include_vars:
      file: "{{ galaxy_meta_file }}"
      name: galaxy_meta

  - shell: >-
      rm -rf {{ galaxy_meta.build_ignore | join (' ') }}
    args:
      chdir: "{{ clone_dir }}"

  - name: Build the collection
    command: >-
      ansible-galaxy collection build
    args:
      chdir: "{{ clone_dir }}"
    when: galaxy_yml_rendering is changed

  - name: Publish the collection
    command: >-
      ansible-galaxy collection publish {{ galaxy_archive_file }}
    args:
      chdir: "{{ clone_dir }}"
    when:
      - galaxy_yml_rendering is changed
```

Vous aurez noté qu'il s'appuie sur une template. Vous pouvez prendre exemple et adapter à votre cas le template suivant, à placer dans un fichier `build/templates/galaxy.yml.j2`.

À savoir

La partie la plus importante est l'attribut `build_ignore`, vous être attentif à :

- ne jamais livrer des secrets.
- ignorer tout ce qui n'est pas strictement nécessaire, de façon à alléger l'archive qui sera téléchargée par vos utilisateurs.

```
---
namespace: "{{ galaxy_namespace }}"
name: "{{ galaxy_name }}"
version: "{{ galaxy_version }}"
readme: README.md

# A list of the collection's content authors. Can be just the name or in the format 'Full␣
↪Name <email> (url)
# @nicks:irc/im.site#channel'
authors:
  - Aurélien Maury <aurelien.maury@wescale.fr>

description: >-
  Demo collection

license:
  - MIT

# A list of tags you want to associate with the collection for indexing/searching. A tag␣
↪name has the same character
# requirements as 'namespace' and 'name'
tags: []

# Collections that this collection requires to be installed for it to be usable. The key␣
↪of the dict is the
# collection label 'namespace.name'. The value is a version range
# L(specifiers,https://python-semanticversion.readthedocs.io/en/latest/#requirement-
↪specification). Multiple version
# range specifiers can be set and are separated by ','
dependencies:
  ansible.netcommon: ">=2.5.0"
  ansible.posix: ">=1.3.0"

# The URL of the originating SCM repository
repository: >-
  https://github.com/wescale/my_project

# The URL to any online docs
documentation: >-
  https://my_project.rtfd.io

# The URL to the homepage of the collection/project
homepage: >-
  https://my_project.rtfd.io

# The URL to the collection issue tracker
issues: >-
  https://github.com/wescale/my_project/issues
```

(continues on next page)

(continued from previous page)

```
# Since: ansible>=2.10
# Backported with love in build/release_collection.yml
#
build_ignore:
  - "*.local"
  - "*.secrets"
  - "*.tar.gz"
  - ".ansible-lint"
  - ".direnv"
  - ".env*"
  - ".envrc"
  - ".gitignore"
  - ".gitignore"
  - "Makefile"
  - "ansible.cfg"
  - "build"
  - "docs"
  - "documentation/Makefile"
  - "documentation/conf.py"
  - "documentation/requirements.txt"
  - "group_vars"
  - "host_vars"
  - "hosts"
  - "hosts.sample"
  - "keys"
  - "mkdocs.yml"
  - "ops_*.yml"
  - "requirements.*"
  - "requirements.txt"
  - "secrets"
  - "snippets"
  - "ssh.cfg"
  - "tests"
```

3.4 Rôles

Quelques observations et réflexions sur les méthodes et conventions qui facilitent le développement et la maintenance des rôles Ansible.

3.4.1 Initialisation

Squelette standard

Partant du principe que vous avez suivi les recommandations du chapitre sur *Les bases*, pour créer un rôle vide, lancez :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training
```

(continues on next page)

(continued from previous page)

```
$ ansible-galaxy role init a_tester --init-path=roles
- Role a_tester was created successfully

$ tree -a roles/a_tester/
roles/a_tester/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
├── .travis.yml
└── vars
    └── main.yml

8 directories, 9 files
```

Squelette sur mesure

Le squelette de base de la commande `ansible-galaxy` n'est pas forcément adapté à vos usages. Aucune directive claire n'est fournie dans la documentation officielle pour l'utilisation du contenu du répertoire `tests/` et tout le monde ne base pas sa CI sur Travis…

3.4.2 Variables

Perle de sagesse

Les variables de rôle, de `defaults/main.yml` et `vars/main.yml` n'ont que leur propre rôle comme portée de variable et partagent le même espace de variable.

Defaults ou Vars ?

Une variable de pilotage appartenant à un rôle doit-elle trouver sa place :

- dans `defaults/main.yml` ?
- dans `vars/main.yml` ?

La réponse à cette question est assez simple, pour peu qu'on modélise correctement le problème. L'un des principes de la programmation objet est l'encapsulation. Si vous n'êtes pas familier avec le concept, il s'agit de masquer le fonctionnement interne d'un objet. Les échanges entre un objet et un autre se font au travers d'une interface clairement définie.

Si l'on considère notre objet "rôle" sous cet angle, alors toutes les variables définies dans `defaults/main.yml` sont notre interface avec l'extérieur. C'est au travers de ces variables qu'un utilisateur va pouvoir influer sur le comportement de notre rôle.

Le fichier `vars/main.yml` est à réserver à la construction de variables internes, potentiellement des compositions des variables issues de `defaults/main.yml`. Un bon exemple pourrait être :

```
#
# defaults/main.yml
#
---
nginx_server_name: "alpha"
```

```
#
# vars/main.yml
#
---
__nginx_site_config_path: "/etc/nginx/sites-availables/{{ nginx_server_name }}.conf"
__nginx_site_link_path: "/etc/nginx/sites-enabled/{{ nginx_server_name }}.conf"
```

On offre à l'utilisateur la possibilité de redéfinir le nom du serveur que nous allons configurer, mais les chemins des fichiers de configuration sont soumis aux conventions de nginx et par conséquent on forge des variables dans `vars/main.yml` pour respecter cela.

Convention de nommage

Pour faire extrêmement simple :

- Toute variable de rôle de `defaults/main.yml` doit être préfixée par le nom du rôle.
- Toute variable de rôle de `vars/main.yml` doit être préfixée par un double underscore (__) suivi par le nom du rôle.

Ainsi, vous vous assurez que vos variables ne viendront pas écraser des variables existantes externes au rôle, et vous saurez d'un coup d'oeil où une variable a été définie.

Sorties

Si d'aventure vous éprouvez le besoin qu'un rôle exporte des variables à destination d'autres tasks ou à la suite du flot d'exécution de votre playbook :

- Définissez ces variables depuis des tasks `set_fact` dans votre rôle
- Préfixez ces variables par le nom du rôle et la mention `fact` (soit `rolename_fact_*`).

Là encore le but est de pouvoir identifier l'origine d'une variable au premier coup d'oeil. En période de debug, vous vous remercierez d'avoir adopté cette convention.

En résumé

Les variables nommées :

- `rolename_*` : paramètres d'entrée définis dans `defaults/main.yml`
- `rolename_fact_*` : valeurs de sortie définies via le module `set_fact`
- `__rolename_*` : variables internes définies dans `vars/main.yml`

3.4.3 Handlers

Observation

Parmi les exemples de code que vous trouverez immanquablement sur la toile, le déclenchement de tasks en fonction du résultat d'une autre task revient souvent sous ces formes :

```
#
# Ne pas reproduire chez vous
#
- name: Templating de conf
  template:
    src: ...
    dest ...
  register: _templating

- name: Restart de service si la conf change
  service:
    state: restarted
  when: _templating is changed
```

Cette approche complexifie inutilement le code de vos rôles/playbooks en réimplémentant la mécanique interne à Ansible qu'est le handler. Un des arguments est que les handlers se déclenchent en fin d'exécution de playbook et donc diffèrent parfois trop le lancement de ces comportements conditionnels.

Proposition

En couplant la définition classique de handler avec l'usage du module **meta**, moins connu, on obtient l'effet désiré. Si on prend l'exemple d'un micro-playbook pour illustrer, cela donne :

```
---
- hosts: localhost
  become: yes

  tasks:
    - name: Templating de conf
      template:
        src: ...
        dest: ...
      notify: Restart service

    - name: Application de tous les handlers notifiés jusqu'ici
      meta: flush_handlers

    - debug:
        msg: "Si la config a été modifiée, à ce stade, le service est déjà redémarré."

  handlers:
    - name: Restart service
      service:
        name: ...
        state: restarted
```

Pour compléter le pattern, ajoutez **systématiquement** un appel au `meta: flush_handlers` en fin de rôle :

```
#
# un_role/tasks/main.yml
#

[...]

- name: Toujours flush les handlers en dernière task de rôle
  meta: flush_handlers
#
# End-Of-File
#
```

Si on oublie ce flush de fin de rôle, il est possible qu'un autre rôle plante et le re-jeu de notre rôle ne détectera pas de modifications sur son périmètre, donc ne lancera pas de `notify` et ne déclenchera pas les handlers.

En appliquant cette technique on s'assure que chaque rôle ne déborde pas de son périmètre et a bien fini ses actions avant de passer la main à un autre rôle.

Intégration

Cette approche est aussi bien applicable dans un contexte de playbooks comme vu au-dessus, que dans un contexte de rôle. Dans un rôle vous rangerez vos handlers dans le fichier `handlers/main.yml` et vous utiliserez la combo attribut `notify` + module `meta: flush_handlers` là où cela sera utile.

3.4.4 Supporter plusieurs versions de systèmes

Observation

Il peut arriver de vouloir supporter :

- différents systèmes
- différentes versions d'un même système

Ansible collecte des **facts** à la connexion au système pour nous permettre de différencier un OS d'un autre. C'est classiquement sur ces variables collectées qu'il convient de s'appuyer pour savoir dans quel cas l'exécution se déroule.

Pour cela, vous trouverez couramment des approches basées sur l'exécution conditionnelle de tasks.

```
#
# Ne pas reproduire chez vous
#
- name: Debian-only templating
  template:
    src: ...
    dest: ...
  when: ansible_distribution == 'Debian'
```

Le problème latent est la duplication de code, qui arrive rapidement avec la multiplication des cas particuliers.

Le niveau supérieur est de charger un fichier de tasks dédié en s'appuyant sur la même variable :

```
#
# Ne pas reproduire chez vous
#
- name: Debian-only tasks
  include_tasks: debian.yml
  when: ansible_distribution == 'Debian'
```

...ou encore :

```
#
# Ne pas reproduire chez vous
#
- name: Debian-only tasks
  include_tasks: "{{ ansible_distribution | lower }}.yml
```

Mais là encore on va rapidement se retrouver avec des cas et des sous-cas au fil de la vie du code.

Proposition

L'organisation de code pour pouvoir supporter des tasks spécifiques par système est un problème récurrent et chaque projet y va de son implémentation. Voici une méthode tout-terrain.

L'idée est de créer un fois pour toutes une chaine de chargement préférentiel qui s'appuie sur nos facts. L'inclusion de facts dans le nommage des fichiers recherchés permet de ne charger que les cas les plus spécifiques, et cela pour des fichiers de variable ET pour des fichiers de tasks.

```
- name: OS markers
  debug:
    msg: >-
      Distrib: {{ ansible_distribution | lower }} - Version: {{ ansible_distribution_
↪major_version }} - Arch: {{ ansible_architecture | lower }}
    verbosity: 1

- name: Load os-specific vars
  include_vars: "{{ _current_os_vars }}"
  with_first_found:
    - skip: true
      files:
        - "{{ ansible_distribution | lower }}_{{ ansible_distribution_major_version }}_{
↪{ ansible_architecture | lower }}.yml"
        - "{{ ansible_distribution | lower }}_{{ ansible_architecture | lower }}.yml"
        - "{{ ansible_distribution | lower }}_{{ ansible_distribution_major_version }}.
↪yml"
        - "{{ ansible_distribution | lower }}.yml"
        - "{{ ansible_os_family | lower }}.yml"
  loop_control:
    loop_var: _current_os_vars

- name: Execute os-specific tasks
  include_tasks: "{{ _current_os_tasks }}"
  with_first_found:
    - skip: true
      files:
        - "{{ ansible_distribution | lower }}_{{ ansible_distribution_major_version }}_{
↪{ ansible_architecture | lower }}/main.yml"
        - "{{ ansible_distribution | lower }}_{{ ansible_architecture | lower }}/main.yml
↪"
        - "{{ ansible_distribution | lower }}_{{ ansible_distribution_major_version }}/
↪main.yml"
        - "{{ ansible_distribution | lower }}/main.yml"
  loop_control:
    loop_var: _current_os_tasks
```

Concrètement, ce code exécuté depuis un rôle sur une machine Debian Bullseye d'architecture x86 va :

- tenter de charger le fichier de variables `vars/debian_11_x86_64.yml` ;
- si le fichier est absent, tenter `vars/debian_x86_64.yml` ;
- si le fichier est absent, tenter `vars/debian_11.yml` ;
- si le fichier est absent, tenter `vars/debian.yml` ;
- si le fichier est absent, passer à la suite sans erreur ;

- tenter d'exécuter le fichier de tasks `tasks/debian_11_x86_64/main.yml` ;
- si le fichier est absent, tenter `tasks/debian_x86_64/main.yml` ;
- si le fichier est absent, tenter `tasks/debian_11/main.yml` ;
- si le fichier est absent, tenter `tasks/debian/main.yml` ;
- si le fichier est absent, passer à la suite sans erreur.

Intégration

Une fois ce code en place dans le fichier `tasks/main.yml` d'un rôle, vous n'aurez plus besoin d'y toucher. Implémenter des variables et comportements spécifiques reviendra à ajouter les fichiers à charger en suivant la convention de nommage des fichiers.

À vous de trouvez la priorisation qui convient le mieux à vos usages et de poser une convention avec vos équipes.

CHAPTER

FOUR

OUTILLAGE

4.1 Packer

Hashicorp Packer est un outil qui vous permet de créer des images de machines identiques pour différentes plateformes, à partir d'un modèle unique.

Packer permet d'adopter le principe de "golden image", c'est-à-dire où l'on va construire une image avec des composants testés et validés pour la mise en production. Cette image sera la source pour déployer toutes les machines virtuelles ou installer un ensemble de serveurs physiques.

La force de Packer réside dans le fait qu'il est relativement simple à utiliser et propose de nombreuses extensions (dont un provisionner Ansible).

Approfondir

- Documentation Packer
- Documentation Packer - Installation

4.1.1 Provisioner

Un provisioner utilise des logiciels intégrés et tiers pour installer et configurer l'image de la machine après le démarrage. Un provisionner prépare le système pour l'utilisation, avec par exemple :

- installation de paquets
- compilation de modules du kernel
- création d'utilisateurs
- récupération de code source ou exécutables

Provisioner Ansible

Le provisioner Ansible utilise le mode push (le mode de fonctionnement natif) pour exécuter des playbooks Ansible. Le mode de fonctionnement implique que le code Ansible doit être sur votre machine de travail ou sur la machine de contrôle. Des connexions SSH seront alors établies vers les machines cibles via SSH afin de recueillir et appliquer les changements.

Ce provisionner est très utile dans le cas où vous devez construire des images système pour vos instances de machines virtuelles.

Prenons l'exemple de la création d'une image ISO pour une machine virtuelle compatible Virtualbox :

On crée le fichier ansible-remote.json :

```
{
  "builders": [
    {
      "type": "docker",
      "image": "stelar/debian:11-systemd",
      "export_path": "image-builded-by-packer.tar",
      "run_command": ["-d", "-i", "-t", "--entrypoint=/bin/bash", "{{.Image}}"]
    }
  ],
  "provisioners": [
    {
      "type": "ansible",
      "playbook_file": "./playbook.yml"
     }
  ]
}
```

On crée le playbook :

```
---
# playbook.yml
- name: 'Provision Image'
  hosts: default
  become: true

  tasks:
    - name: install Apache
      package:
        name: 'apache2'
        state: present
```

On valide que la configuration de Packer est correcte :

```
$ packer validate virtualbox-iso.json
The configuration is valid.
```

On lance la création de l'image Docker :

```
packer build ansible-remote.json
```

Packer va utiliser les APIs de Docker afin de créer une image.

Le résultat de ce build sera une image Docker Debian 11 avec Apache installé, exporté dans le fichier `image-builded-by-packer.tar`.

Provisioner Ansible Local

Le provisioner Ansible Local utilise le mode push. A l'inverse du mode pull, ce sont directement les machines cibles qui exécutent le code Ansible et appliquent les changements en local, sans passer par une connexion SSH.

Cela implique qu'Ansible soit pré-installé sur les machines cibles.

```
{
  "builders": [
    {
      "type": "docker",
      "image": "williamyeh/ansible:ubuntu18.04",
      "export_path": "packer_ansible_ultimate",
      "run_command": ["-d", "-i", "-t", "--entrypoint=/bin/bash", "{{.Image}}"]
    }
  ],
  "variables": {
    "topping": "mushroom"
  },
  "provisioners": [
    {
      "type": "ansible-local",
      "playbook_file": "./playbook.yml",
      "extra_arguments": [
        "--extra-vars",
        "\"pizza_toppings={{ user `topping`}}\""
      ]
    }
  ]
}
```

On valide que la configuration de Packer est correcte :

```
$ packer ansible-local.json
The configuration is valid.
```

On lance la création de l'image Docker :

```
packer build ansible-local.json
```

Packer va utiliser les APIs de Docker afin de créer une image.

Le résultat de ce build sera une image Docker Debian 11 avec l'affichage d'un message "mushroom", exporté dans le fichier `image-builded-by-packer.tar`

Approfondir

- Documentation Packer - Plugins
- Documentation Packer - Provisionner
- Documentation Packer - Provisionner Ansible
- Documentation Packer - Provisionner Ansible Local

4.2 AWX

AWX permet de gérer l'execution de vos playbooks de façon avancée. En effet il est possible de plannifier leur exécution dans le temps, ainsi que de centraliser la gestion des inventaires.

En plus de son interface graphique, AWX propose une API ! Et c'est là que ça deviens intéressant. Cela vous permet d'intégrer des projets entiers à n'importe quelle application ou sein de votre SI.

AWX gère tout le nécessaire à l'exécution et la gestion de vos rôles, collections ou playbooks Ansible : clés SSH, identités externes, synchronisation projets GIT, configuration des modules Ansible, etc.

Approfondir

- Introduction AWX - Blog Wescale
- Utilisation AWX - Blog Wescale

4.2.1 Installation

```
git clone -b 20.1.0 https://github.com/ansible/awx.git
cd awx
```

On construit l'image docker :

```
make docker-compose-build
```

On démarre les conteneurs AWX, Postgres et Redis :

```
make docker-compose
```

Attendez que les processus d'initialisation se terminent, notamment les migrations de l'ORM d'AWX(Django)

Pour suivre l'initialisation :

```
docker-compose -f tools/docker-compose/_sources/docker-compose.yml logs -f
```

On construit la web ui :

```
docker exec tools_awx_1 make clean-ui ui-devel
```

On créer un utilisateur administrateur :

```
docker exec -ti tools_awx_1 awx-manage createsuperuser
```

Chargeons les données example de démonstration

```
docker exec tools_awx_1 awx-manage create_preload_data
```

L'interface web est disponible à cette adresse : https://localhost:8043/#/home

L'API est disponible à cette adresse : https://localhost:8043/api/v2

Approfondir

- Documentation AWX - Installation via docker-compose

- Pour aller plus loin - Intégration avec ARA

4.2.2 La CLI

Le client en ligne de commande officiel s'installe très simplement d'un coup de Pip :

```
pip install awxkit
# On vérifie que le client est bien installé :
awx --help
```

On récupère le token de connexion :

```
awx login --conf.host https://localhost:8043 --conf.insecure --conf.username <votre_
↪username> --conf.password <votre_password>
```

Récupérez le token, et allons configurer quelques variables d'environnement pour l'exercice :

```
export CONTROLLER_OAUTH_TOKEN=<votre_token>
export CONTROLLER_HOST=https://localhost:8043
export CONTROLLER_VERIFY_SSL=false
$(CONTROLLER_USERNAME=alice CONTROLLER_PASSWORD=secret awx login -f human)
```

Votre client AWX est prêt à l'emploi ! Cependant ce n'est une configuration à utiliser sur des environnements réels, ici nous vous proposons la configuration la plus rapide à mettre en oeuvre pour tester et apprendre. Nous irons plus loin, dans un prochain chapitre, où nous parlerons de l'intégration SSO et plus globalement comment mettre en place un serveur AWX de façon sécurisé prêt pour la production.

Quelques commandes pour la route :

```
# Lister les rôles AWX
awx roles list

# Lister les jobs, filtrer sur le nom 'Example Job Template'
awx jobs list --all --name 'Example Job Template' \
    -f human --filter 'name,created,status'

# Lister les instances AWX
awx instance list

# Lister les projets AWX
awx project list

# Lister les métriques exposées
awx metrics

# Lister un inventaire
awx inventory get 1
```

Création et lancement d'un job template :

```
awx projects create --wait \
    --organization 1 --name='Example Project' \
    --scm_type git --scm_url 'https://github.com/ansible/ansible-tower-samples' \
```

(continues on next page)

(continued from previous page)

```
    -f human
awx job_templates create \
    --name='Example Job Template' --project 'Example Project' \
    --playbook hello_world.yml --inventory 'Demo Inventory' \
    -f human
awx job_templates launch 'Example Job Template' --monitor -f human
```

Approfondir

- Documentation AWX CLI
- Documentation AWX CLI - Examples

4.3 ARA

Ara est un callback module qui enregistre toute la sortie d'Ansible. Cela en fait un outil assez pratique pour investiguer et conserver une traçabilité des exécutions de vos playbooks.

Pour cela vous avez à votre disposition une interface en ligne de commande et une interface web.

4.3.1 Installation

Pour installer Ara rien de plus simple ! Clonez ce projet GIT : https://github.com/gloterman/ansible-ara-quickstart ou suivez le guide de départ de la documentation officielle

Une fois Ara activé, et votre premier playbook exécuté, toutes les informations seront stockées dans une base Sqlite (par défaut). Voici quelques commandes pour afficher ces informations :

```
# lister les playbooks
$ ara playbook list

# Voir le détail du playbook dont l'id est 1
$ ara playbook show 1

# Voir les métriques des playbooks
$ ara playbook metrics

# Voir la list des tasks
$ ara task list

# Voir le détail d'une task
$ ara task show 1
```

Pour démarrer l'interface web :

```
$ ara-manage runserver
```

Approfondir

- Documentation Ara - CLI

4.3.2 Démo

Une instance de démonstration est disponible à cette adresse : https://demo.recordsansible.org

4.3.3 Pour aller plus loin

Félicitations ! Vous avez Ara qui est configuré pour enregistrer votre utilisation d'Anible. Cependant ce n'est pas suffisant en situation de production.

On préfèrera utiliser Ara en mode "api server", où toutes les données stockées resteront sur un serveur prévue à cet effet.

Pour l'exemple :

```
# Create a directory for a volume to store settings and a sqlite database
$ mkdir -p ~/.ara/server

# or with docker from the image on quay.io:
$ docker run --name api-server --detach --tty \
    --volume ~/.ara/server:/opt/ara:z -p 8000:8000 \
    quay.io/recordsansible/ara-api:latest
```

Cette fois ci, on va configurer Ara pour contacter l'API server. Éditez le fichier d'environnment et ajoutez ceci:

```
# fichier .envrc
export ARA_API_CLIENT="http"
export ARA_API_SERVER="http://127.0.0.1:8000"
```

Approfondir

- Documentation Ara
- Documentation Ara - Github
- Démonstration Ara interface web

CHAPTER

FIVE

DÉVELOPPEMENT

5.1 Module

Tous les modules que vous développez doivent se trouver au sein d'une collection, cela facilitera la diffusion, que ce soit en au sein de la communauté ou pour vos projets d'entreprise.

Les modules custom que vous développez doivent se trouver dans le répertoire `plugins/modules` pour être retrouvés à l'installation de la collection en dépendances d'un projet.

Le guide officiel de développement de module est très fourni sur le sujet.

5.1.1 Initialisation

En phase de développement, vous devez indiquer à Ansible où trouvez vos modules en cours de développement.

- Ajoutez ceci à votre `.envrc`

```
export ANSIBLE_LIBRARY="${PWD}/plugins/modules:${ANSIBLE_LIBRARY}"
```

5.1.2 Exemple fonctionnel

Collez ceci dans le fichier `plugins/modules/hello.py`

```
#!/usr/bin/python

from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = r'''
---
module: hello

short_description: Module de demonstration

# If this is part of a collection, you need to use semantic versioning,
# i.e. the version is of the form "2.5.0" and not "2.4".
version_added: "1.0.0"

description: This is my longer description explaining my test module.

```

(continues on next page)

(continued from previous page)

```
options:
    name:
        description: This is the message to send to the test module.
        required: true
        type: str
    new:
        description:
            - Control to demo if the result of this module is changed or not.
            - Parameter description can be a list as well.
        required: false
        type: bool
# Specify this value according to your collection
# in format of namespace.collection.doc_fragment_name
extends_documentation_fragment:
    - my_namespace.my_collection.my_doc_fragment_name

author:
    - Your Name (@yourGitHubHandle)
'''

EXAMPLES = r'''
# Pass in a message
- name: Test with a message
  my_namespace.my_collection.my_test:
    name: hello world

# pass in a message and have changed true
- name: Test with a message and changed output
  my_namespace.my_collection.my_test:
    name: hello world
    new: true

# fail the module
- name: Test failure of the module
  my_namespace.my_collection.my_test:
    name: fail me
'''

RETURN = r'''
# These are examples of possible return values, and in general should use other names␣
↪for return values.
message:
    description: The output message that the test module generates.
    type: str
    returned: always
    sample: 'goodbye'
'''

from ansible.module_utils.basic import AnsibleModule

def run_module():
    # define available arguments/parameters a user can pass to the module
```

(continues on next page)

(continued from previous page)

```
    module_args =dict(
        world=dict(type='str', required=True)
    )

    # seed the result dict in the object
    # we primarily care about changed and state
    # changed is if this module effectively modified the target
    # state will include any data that you want your module to pass back
    # for consumption, for example, in a subsequent task
    result = dict(
        changed=False,
        ansible_facts=dict(),
    )

    # the AnsibleModule object will be our abstraction working with Ansible
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # if the user is working with this module in only check mode we do not
    # want to make any changes to the environment, just return the current
    # state with no modifications
    if module.check_mode:
        module.exit_json(**result)

    # manipulate or modify the state as needed (this is going to be the
    # part where your module will do what it needs to do)
    result['message'] = "Hello {}".format(module.params['world'])
    result['changed'] = True

    # in the event of a successful module execution, you will want to
    # simple AnsibleModule.exit_json(), passing the key/value results
    module.exit_json(**result)


def main():
    run_module()


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

- Collez ceci dans un playbook en `playbooks/hello.yml`

```
---
- hosts: localhost
  become: no
  gather_facts: false
```

(continues on next page)

(continued from previous page)

```
  tasks:
    - hello:
        world: Arrakis
      register: hello_res

    - debug:
        var: hello_res
```

Vous pouvez maintenant utiliser votre module de démo via la commande :

```
$ ansible-playbook playbooks/hello.yml

PLAY [localhost]␣
↪*********************************************************************************************************************

TASK [hello]␣
↪*********************************************************************************************************************
mardi 19 avril 2022  01:27:13 +0200 (0:00:00.006)       0:00:00.006 ***********
changed: [localhost]

TASK [debug]␣
↪*********************************************************************************************************************
mardi 19 avril 2022  01:27:14 +0200 (0:00:00.161)       0:00:00.167 ***********
ok: [localhost] => {
    "hello_res": {
        "ansible_facts": {},
        "changed": true,
        "failed": false,
        "message": "Hello Arrakis"
    }
}

PLAY RECAP␣
↪*********************************************************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0    skipped=0␣
↪   rescued=0    ignored=0

Playbook run took 0 days, 0 hours, 0 minutes, 0 seconds
mardi 19 avril 2022  01:27:14 +0200 (0:00:00.010)       0:00:00.177 ***********
===============================================================================
hello ------------------------------------------------------------------------------------------
↪--------------------------------------------------------------------------- 0.16s
debug ------------------------------------------------------------------------------------------
↪--------------------------------------------------------------------------- 0.01s
```

5.2 Filter plugins

Tous les filters que vous développez doivent se trouver au sein d'une collection, cela facilitera la diffusion, que ce soit en au sein de la communauté ou pour vos projets d'entreprise.

Les modules custom que vous développez doivent se trouver dans le répertoire `plugins/filter_plugins` pour être retrouvés à l'installation de la collection en dépendances d'un projet.

Le guide officiel de développement de plugins est très fourni sur le sujet.

5.2.1 Initialisation

En phase de développement, vous devez indiquer à Ansible où trouvez vos plugins filters en cours de développement.

- Ajoutez ceci à votre `.envrc`

```
export ANSIBLE_FILTER_PLUGINS="${PWD}/plugins/filter_plugins:${ANSIBLE_FILTER_PLUGINS}"
```

5.2.2 Exemple fonctionnel

Coller ceci dans le fichier `plugins/filter_plugins/rev.py` :

```
#!/usr/bin/python

class FilterModule(object):
    ''' Nested dict filter '''

    def filters(self):
        return {
            'rev': self.rev
        }

    def rev(self, input):

        return input[::-1]
```

Vous pouvez le tester en créant le fichier `playbooks/rev.yml` :

```
---
- hosts: localhost
  become: false
  gather_facts: false

  tasks:
    - debug:
        msg: "{{ item | rev }}"
      loop:
        - palindrome
        - kayak
```

Et en lançant la commande :

```
$ ansible-playbook playbooks/rev.yml
PLAY [localhost]␣
↪*********************************************************************************************************************

TASK [debug]␣
↪*********************************************************************************************************************
mardi 19 avril 2022  01:38:22 +0200 (0:00:00.006)       0:00:00.006 ***********
ok: [localhost] => (item=palindrome) => {}

MSG:

emordnilap
ok: [localhost] => (item=kayak) => {}

MSG:

kayak

PLAY RECAP␣
↪*********************************************************************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped=0␣
↪   rescued=0    ignored=0

Playbook run took 0 days, 0 hours, 0 minutes, 0 seconds
mardi 19 avril 2022  01:38:22 +0200 (0:00:00.021)       0:00:00.028 ***********
===============================================================================
debug ------------------------------------------------------------------------
↪------------------------------------------------------------------------- 0.02s
```

CHAPTER

SIX

HOWTO

6.1 Préparer un système OSX

Vous pouvez également réaliser les exercises sur OSX. Cependant, quelques pré-requis sont à mettre en œuvre avant de commencer.

Note

Ce guide peut être réalisé sur les Mac avec processeurs Intel. Les Mac avec processeurs M1 ont une architecture ARM, ce qui les rend incompatibles avec les conteneurs Docker construits sur des machines sous architecture Intel x86 64 bits.

6.1.1 Gestion des paquets

Évidemment, on n'utilisera pas APT, mais HomeBrew. N'oubliez pas d'adapter les commandes d'installation de paquets le cas échéant dans les exercices.

Nous vous recommandons d'utiliser HomeBrew sur MacOSX, afin de gérer l'installation de vos logiciels open-source.

6.1.2 Installation des pré-requis

```
$ brew install coreutils direnv wget curl

$ brew cask install docker
```

Vous devriez être paré pour attaquer les *Exercices*

Approfondir

- Documentation Docker - Installation sur Mac

6.2 Installer un outil annexe au niveau projet

En suivant la méthode poussée dans ce guide jusqu'au bout, l'installation d'un outil annexe, comme Terraform, devrait se faire :

- avec une version fixe, car le code projet sera lié à la version de cet outil
- via `direnv` ou `make` qui sont les racines de manipulation de l'environnement de travail

La méthode présentée ici a l'avantage de rester dans la logique initiale et donc de demander un minimum de connaissances supplémentaires.

Cela peut être vu comme restrictif par certains qui maîtrisent d'autres techniques mais cela reste un avantage lorsqu'on a une promotion de débutants à former.

L'idée est de réaliser une installation directement dans le répertoire `.direnv` de notre projet, en shell, via la mécanique native de `direnv`.

Prenons ici comme exemple l'installation d'une version de Terraform pour un projet.

- Ajoutez le bloc suivant à votre fichier `.envrc` de projet pour construire un répertoire local à votre projet qui contiendra les binaires à ajouter au `PATH` :

```
#
# .envrc
#   - Manipulation du PATH pour privilégier le répertoire .direnv/bin
#
export DIRENV_TMP_DIR="${PWD}/.direnv"
export DIRENV_BIN_DIR="${DIRENV_TMP_DIR}/bin"
if [ ! -e "${DIRENV_BIN_DIR}" ]; then
    mkdir -p "${DIRENV_BIN_DIR}"
fi
export PATH="${DIRENV_BIN_DIR}:${PATH}"
```

6.2.1 Installer Terraform

- Ajoutez à votre `.envrc` :

```
#
# .envrc
#   - Installation automatique de terraform
#
TF_VERSION="1.1.8"
TF_ARCH="linux_amd64"
TF_PKG_NAME="terraform_${TF_VERSION}_${TF_ARCH}.zip"
TF_PKG_URL="https://releases.hashicorp.com/terraform/${TF_VERSION}/${TF_PKG_NAME}"
TF_PKG_PATH="${DIRENV_TMP_DIR}/${TF_PKG_NAME}"
if [ ! -e "${DIRENV_BIN_DIR}/terraform" ]; then
    echo "===> Getting terraform:${TF_VERSION}:${TF_ARCH} (can take a while to execute)"
    curl -s -L "${TF_PKG_URL}" -o "${TF_PKG_PATH}"
    unzip "${TF_PKG_PATH}" -d "${DIRENV_BIN_DIR}"
    chmod 700 "${DIRENV_BIN_DIR}/terraform"
    rm -f "${TF_PKG_PATH}"
fi
```

- Suivi de la commande :

```
> direnv allow .
direnv: loading ~/ansible-workspaces/ultimate/training/.envrc
===> Getting terraform:1.1.8:linux_amd64 (can take a while to execute)
Archive:  /home/amaury/ansible-workspaces/ultimate/training/.direnv/terraform_1.1.8_
↪linux_amd64.zip
  inflating: /home/amaury/ansible-workspaces/ultimate/training/.direnv/bin/terraform

> which terraform
/home/user/ansible-workspaces/ultimate/training/.direnv/bin/terraform
```

6.2.2 Installer Goss

- Ajoutez à votre `.envrc` :

```
#
# .envrc
#   - Installation automatique de goss
#
GOSS_VERSION="0.3.16"
GOSS_ARCH="linux-amd64"
GOSS_PKG_NAME="goss-${GOSS_VERSION}-${GOSS_ARCH}"
GOSS_PKG_URL="https://github.com/aelsabbahy/goss/releases/download/v${GOSS_VERSION}/$
↪{GOSS_PKG_NAME}"
GOSS_PKG_PATH="${DIRENV_TMP_DIR}/${GOSS_PKG_NAME}"
if [ ! -e "${DIRENV_BIN_DIR}/goss" ]; then
    echo "===> Getting goss:${GOSS_VERSION}:${GOSS_ARCH} (can take a while to execute)"
    curl -s -L "${GOSS_PKG_URL}" -o "${DIRENV_BIN_DIR}/goss"
    chmod 700 "${DIRENV_BIN_DIR}/goss"
fi
```

- Suivi de la commande :

```
> direnv allow
direnv: loading ~/ansible-workspaces/ultimate/training/.envrc
===> Getting goss:0.3.16:linux-amd64 (can take a while to execute)

> which goss
/home/user/ansible-workspaces/ultimate/training/.direnv/bin/goss
```

CHAPTER

SEVEN

EXERCICES

Perle de sagesse

La connaissance s'acquiert par l'expérience, tout le reste n'est que de l'information.

Albert Einstein, Mathématicien, Physicien, Scientifique (1879 - 1955)

7.1 Préparation

Bienvenue, nous allons passer de bons moments ensemble, autant se préparer à ce que tout se déroule au mieux. Cette préparation trace le chemin d'emmerdement minimum pour te faciliter la vie pour la suite.

Il y a trop de configurations de systèmes pour pouvoir couvrir tous les cas ici. Notre cas de référence sur Debian 11 n'est pas tiré de nulle part : cette configuration permet à de nombreux professionnels de gagner leur vie en étant productif au quotidien (Debian sur laptop, aujourd'hui ce n'est pas un problème).

7.1.1 Figures libres

Pour réaliser les exercices qui vont suivre, il faut se procurer :

- un système Debian stable (11 à l'heure de la dernière modification)
- un terminal branché dessus avec un shell Bash
- un environnement de développement intégré (IDE) capable d'éditer les fichiers de votre système Debian (LunarVim, VSCodium)

…tous les coups sont permis pour atteindre ces premiers critères.

Nota Bene

- Si vous êtes sous Windows, un WSL bien configuré doit vous permettre de faire ça (n'hésitez pas à contribuer un guide en PR sur le projet).
- Si vous êtes sous OSX, vous pourriez être intéressé par le guide *Préparer un système OSX*

7.1.2 Figures imposées

Une fois votre système prêt, il faut y ajouter plusieurs paquets pour pouvoir travailler :

```
> sudo apt update
> sudo apt-get install python3 python3-dev python3-venv python3-pip git direnv make bash␣
↪curl lsb-release
> grep -q 'eval "$(direnv hook bash)"' ~/.bashrc || echo 'eval "$(direnv hook bash)"' >>␣
↪~/.bashrc
> source ~/.bashrc
```

Note

Il vous faudra également un démon Docker pour pouvoir exécuter les exercices qui concernent les tests automatisés de code Ansible :

- Guide officiel d'installation de Docker.

7.2 Fondamentaux

Premier chapitre des exercices où l'on vient configurer le poste de travail pour se placer dans les meilleures conditions pour travailler. Ces exercices s'enchaînent et se complètent pour vous apprendre une méthodologie de travail avec Ansible.

7.2.1 Prérequis

- *Préparation*

7.2.2 Exercices

Installer Ansible

Objectif

Installation d'Ansible sur un poste de travail.

Prérequis

- *Préparation*

Création du virtualenv

Pour isoler l'installation de nos dépendances du reste du système, nous allons créer un virtualenv global à tous nos futurs projets. Chaque projet pourra ensuite venir surcharger ce virtualenv avec le sien propre pour isoler ses dépendances au fil des besoins.

- Création de notre espace de travail Ansible

```
> mkdir ~/ansible-workspaces
.lineos mkdir: création du répertoire '/home/user/ansible-workspaces'

> cd ~/ansible-workspaces
/home/user/ansible-workspaces
```

- Création d'un fichier `.envrc` pour indiquer à direnv de créer un virtualenv

```
#
# .envrc
#
layout python3
```

- Référencement du `.envrc` auprès de direnv. Lancer la commande:

```
> direnv allow .
```

Activation automatique du virtualenv

Si vous lancez la commande `which python3` depuis le répertoire `~/ansible-workspaces` ou en dehors, vous n'obtenez pas le même chemin.

Direnv active automatiquement le virtualenv qu'il crée lorsque votre shell se trouve dans un sous-répertoire comparé à l'emplacement du fichier `.envrc`.

Installation d'Ansible

Maintenant que notre virtualenv est prêt, une simple commande nous permet d'y installer Ansible :

```
> pip install ansible-core
```

Installation locale

Si vous lancez la commande `which ansible` depuis le répertoire `~/ansible-workspaces`, vous pourrez observer que le binaire qui répond est situé dans le virtualenv créé juste avant.

Ligne d'arrivée

Félicitations, vous venez d'installer Ansible dans un virtualenv. Si vous lancez la commande `which ansible` depuis le répertoire `~/ansible-workspaces`, vous pourrez observer que le binaire qui répond est situé dans le virtualenv créé juste avant.

```
> which ansible
/home/user/ansible-workspaces/.direnv/python-3.9.2/bin/ansible

> ansible --version
ansible [core 2.14.2]
  config file = None
  configured module search path = ['/home/user/.ansible/plugins/modules', '/usr/share/
↪ansible/plugins/modules']
  ansible python module location = /home/user/ansible-workspaces/.direnv/python-3.9.2/
↪lib/python3.9/site-packages/ansible
  ansible collection location = /home/user/.ansible/collections:/usr/share/ansible/
↪collections
  executable location = /home/user/ansible-workspaces/.direnv/python-3.9.2/bin/ansible
  python version = 3.9.2 (default, Feb 28 2021, 17:03:44) [GCC 10.2.1 20210110] (/home/
↪user/ansible-workspaces/.direnv/python-3.9.2/bin/python3)
  jinja version = 3.1.2
  libyaml = True
```

Créer un projet

Objectif

Création d'un projet Ansible vierge.

Prérequis

- *Préparation*
- *Installer Ansible*

Création de la structure

Pour disposer de notre installation locale d'Ansible, nous devons nous placer dans le répertoire contenant le virtualenv géré par `direnv` :

```
> cd ~/ansible-workspaces
```

Puis, nous pouvons nous appuyer sur la commande `ansible-galaxy` pour initier la structure de notre projet :

```
> ansible-galaxy collection init ultimate.training
- Collection ultimate.training was created successfully
```

```
> cd ultimate/training
> tree -a
.
├── docs
├── galaxy.yml
├── meta
│   └── runtime.yml
├── plugins
│   └── README.md
├── README.md
└── roles

4 directories, 4 files
```

On peut voir qu'`ansible-galaxy` a créé le minimum. À nous de remplir le reste pour obtenir un espace de travail complet.

```
> mkdir -p playbooks/group_vars inventories/
> touch playbooks/group_vars/all.yml inventories/.gitkeep roles/.gitkeep docs/.gitkeep
> tree -a
.
├── docs
│   └── .gitkeep
├── galaxy.yml
├── inventories
│   └── .gitkeep
├── meta
│   └── runtime.yml
├── playbooks
│   └── group_vars
│       └── all.yml
├── plugins
│   └── README.md
├── README.md
└── roles
    └── .gitkeep

7 directories, 8 files
```

Nous avons maintenant un premier cadre de travail utile, nous pouvons l'encadrer par une gestion de version :

```
> git init
> git add .
> git commit -m "ultimate init"
```

Ligne d'arrivée

Félicitations, vous avez créé votre premier cadre de projet Ansible normé. Libre à vous de lancer quelques commandes de plus pour venir lier votre dépôt local git avec une plateforme centrale comme GitHub ou GitLab.

Configurer un projet

Objectif

Configurer un projet pour pouvoir travailler proprement avec.

Prérequis

- *Préparation*
- *Installer Ansible*
- *Créer un projet*

Création d'un virtualenv dédié

Afin d'être certain de fixer la version d'Ansible avec laquelle nous travaillons et pour préparer l'isolation des dépendances Python du projet, nous créons un virtualenv dédié. C'est la même méthode que pour l'installation initiale, reproduite au niveau de notre projet.

```
> cd ~/ansible-workspaces/ultimate/training
> echo "layout python3" > .envrc
> direnv allow
> which python
> git add .envrc && git commit -m "adding .envrc"
```

On peut voir que le virtual env est activé à partir de l'activation de direnv.

Il nous reste à faire ignorer par git le répertoire `.direnv` local au projet pour éviter de l'inclure dans un commit.

```
> echo ".direnv" >> .gitignore
> git add .gitignore && git commit -m "ignore local virtualenv"
```

Configuration d'Ansible

Nous allons maintenant configurer quelques comportements basiques d'Ansible en remplissant le fichier `.envrc`:

```
#
# .envrc
#
export DIRENV_TMP_DIR="${PWD}/.direnv"
export ANSIBLE_STDOUT_CALLBACK="ansible.posix.debug"
export ANSIBLE_FORKS="10"
export ANSIBLE_INVENTORY="inventory"
```

(continues on next page)

(continued from previous page)

```
export ANSIBLE_SSH_ARGS="-F ssh.cfg"
export ANSIBLE_ROLES_PATH="${DIRENV_TMP_DIR}/ansible_roles:${PWD}/roles"
export ANSIBLE_COLLECTIONS_PATHS="${DIRENV_TMP_DIR}"
export ANSIBLE_CALLBACKS_ENABLED="timer,profile_tasks"
```

Puis on valide ces changements auprès de direnv et on commit nos modifications.

```
> direnv allow .
> git commit -am "Added ansible configuration"
```

Configuration personnelle

Sur cette base, nous ajoutons notre configuration personnelle: des valeurs qui ne doivent pas être partagées même au travers de git.

Pour l'exemple, nous prenons la variable sensibles permettant de configurer le chemin local vers le fichier contenant notre mot de passe `ansible-vault`.

Tout d'abord nous rajoutons un fichier dans le `.gitignore` afin de ne pas faire d'erreur de manipulation :

```
> echo '.env.local' >> .gitignore
```

Ensuite nous pouvons remplir ce fichier avec nos exports de variables :

```
#
# .env.local
#
export ANSIBLE_VAULT_PASSWORD_FILE="~/.ansible-vault-password"
```

Enfin, pour assurer le chargement transparent de ce nouveau fichier, on intègre un bout de shell dans notre `.envrc` qui sert de point d'accroche à direnv :

```
#
# .envrc
#
LOCAL_CONFIG="${PWD}/.env.local"
if [ -e "${LOCAL_CONFIG}" ]; then
  source "${LOCAL_CONFIG}"
fi
```

Une fois modifié le fichier `.envrc`, direnv nécessite que l'on revalide son chargement. Lancer :

```
> direnv allow .
> git commit -am "Added personal configuration loading"
```

Ligne d'arrivée

Félicitations, vous venez d'ajouter de la configuration Ansible à votre projet. Toutes les configurations communes à tous les intervenants sur le code peuvent être gérées de cette façon.

Gérer les dépendances

Objectif

Ajouter des dépendances Python et Ansible à un projet.

Prérequis

- *Préparation*
- *Installer Ansible*
- *Créer un projet*
- *Configurer un projet*

Échauffement

Pour éviter des problèmes communs de construction de package Pip, commencez par lancer :

```
> pip3 install -U pip wheel setuptools --no-cache-dir
```

Création d'un fichier de requirements Pip

Créez un fichier `requirements.txt` avec notre version préférée d'Ansible pour le projet.

```
#
# requirements.txt
#
ansible-core==2.14.2
```

Une fois ceci fait, nous pouvons rapatrier les dépendances listées avec la commande :

```
> pip3 install -U --no-cache-dir -r requirements.txt
```

Création d'un fichier de requirements Ansible Galaxy

Créez un fichier `requirements.yml` avec un rôle et une collection tirés de la plateforme centrale Ansible Galaxy :

```
#
# requirements.yml
#
---
# Install roles from Ansible Galaxy.
roles:
  - name: geerlingguy.java
    version: 1.9.6

# Install collections from Ansible Galaxy.
collections:
  - name: community.general
    version: 4.5.0
  - name: ansible.posix
```

Une fois ceci fait, nous pouvons rapatrier les dépendances listées avec la commande :

```
> ansible-galaxy install -fr requirements.yml
```

Vous pouvez observer que la collection et le rôle sont installés dans un sous-répertoire de `.direnv`.

Création d'un Makefile basique pour simplifier la mise à jour des dépendances

Comme nous sommes des gens d'automatisation, la complexité des commandes précédentes sera mieux placée dans un fichier `Makefile` :

```
.PHONY: prepare
prepare-desc = "Prepare local workspace"
prepare:
        @echo "==> $(env-desc)"

        @[ -d "${PWD}/.direnv" ] || (echo "Venv not found: ${PWD}/.direnv" && exit 1)
        @pip3 install -U pip wheel setuptools --no-cache-dir && \
        echo "[  OK  ] PIP + WHEEL + SETUPTOOLS" || \
        echo "[FAILED] PIP + WHEEL + SETUPTOOLS"

        @pip3 install -U --no-cache-dir -r "${PWD}/requirements.txt" && \
        echo "[  OK  ] PIP REQUIREMENTS" || \
        echo "[FAILED] PIP REQUIREMENTS"

        @ansible-galaxy install -fr "${PWD}/requirements.yml" && \
        echo "[  OK  ] ANSIBLE-GALAXY REQUIREMENTS" || \
        echo "[FAILED] ANSIBLE-GALAXY REQUIREMENTS"
```

Vous pouvez relancer la procédure complète de rapatriement des dépendances avec la commande :

```
> make prepare
```

Ligne d'arrivée

Félicitations, vous avez maintenant l'expérience de la gestion des dépendances Python et Ansible, ce qui vous servira à coup sûr dans vos futurs projets. Vous avez également complété les exercices qui vous permettent de poser des bases saines pour tout projet Ansible.

Premier host cible

Objectif

- Créer un répertoire de pilotage pour un host
- Configurer le dépôt local pour gérer la machine avec Ansible

---

Prérequis

- *Préparation*
- *Installer Ansible*
- *Créer un projet*
- *Configurer un projet*
- *Gérer les dépendances*
- un serveur (autre que celui qui contient le projet) auquel vous connecter en SSH
- les accès de base de connexion au serveur (login et clé privée de connexion)

Nota Bene

Pour l'exercice, nous prendrons un host fictif que nous nommerons `ultimate-server`, charge à vous de créer un serveur de test.

Configuration SSH

- Créer un répertoire pour les variables dédiées au serveur :

```
mkdir -p inventories/ultimate/host_vars/ultimate-server/secrets
```

- Deposer la clé privée de connexion dans votre répertoire de travail : `inventories/ultimate/host_vars/ultimate-server/secrets/private.key`
- Remplissez un fichier `inventories/ultimate/ssh.cfg` avec les paramètres de connexion au serveur.

```
#
# inventories/ultimate/ssh.cfg
#
Host ultimate-server
  Hostname          <IP ADDRESS>
  User              <DEFAULT USER>
```

(continues on next page)

(continued from previous page)

```
  IdentityFile      host_vars/ultimate-server/secrets/private.key

  ControlMaster         auto
  ControlPath           ~/.ssh/mux-%r@%h:%p
  ControlPersist        15m
  ServerAliveInterval   100
  TCPKeepAlive          yes
```

L'essentiel pour savoir si vous avez atteint la fin de cette étape est de valider la connexion en lançant :

```
> cd inventories/ultimate
> ssh -F ssh.cfg ultimate-server
```

Ajout à l'inventaire

Une fois votre connexion SSH validée, ajoutez le label de votre serveur à l'inventaire (le fichier `inventory`) :

```
#
# inventories/ultimate/inventory
#
ultimate-server
```

Vérifiez que votre configuration Ansible pointe bien sur cet inventaire :

```
> env | grep ANSIBLE_INVENTORY
ANSIBLE_INVENTORY=inventory
```

Vérifiez que votre configuration Ansible prend en compte votre fichier de configuration SSH :

```
> env | grep ANSIBLE_SSH_ARGS
ANSIBLE_SSH_ARGS=-F ssh.cfg
```

Si ce n'est pas le cas, ajustez votre fichier `.envrc` sans oublier de lancer un `direnv allow .` après vos modifications.

Premier contact

- Lancer la commande :

```
> ansible -m ping ultimate-server
ultimate-server | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Ce `ping` Ansible permet de valider que vous avez bien une connexion SSH valide et qu'il y a au moins une version de Python sur le serveur auquel on se connecte.

Ligne d'arrivée

Félicitations, vous venez d'effectuer votre premier ping Ansible. Si vous avez bien suivi tous les exercices de la section, vous l'avez fait depuis un projet au propre qui plus est.

Vous avez maintenant un projet de travail sain avec lequel nous allons continuer à jouer.

7.3 Testing

Dans ce chapitre d'exercices, vous aurez une première expérience de la mise en place de tests afin de pouvoir valider la pertinence de votre code en continu.

7.3.1 Installation de Molecule

Prérequis

Pour que tout se passe comme prévu, vous aurez besoin que votre projet se conforme aux pratiques vues dans les *Fondamentaux*.

Installation

Partant du principe que vous avez suivi les recommandations du chapitre sur *Les bases*, pour installer Molecule :

- Dans le fichier `requirements.txt` de votre projet, ajoutez une ligne :

```
molecule==3.5.2
molecule-docker
molecule-goss
```

Note

Cette dépendances est fixée sur 3.5.2 pour des problèmes de bugs non résolus au moment de la rédaction. Cela est destiné à changer.

- Dans le fichier `requirements.yml` de votre projet, ajoutez la collection `community.docker` :

```
collections:
  - name: community.docker
```

- Lancez la commande : `make env`

Validation

Molecule, ainsi que le driver Docker sont maintenant installés. Vous pouvez le vérifier en lançant :

```
$ molecule --version
molecule 3.6.1 using python 3.9
    ansible:2.12.3
    delegated:3.6.1 from molecule
    docker:1.1.0 from molecule_docker requiring collections: community.docker>=1.9.1
```

Vous pouvez également constater la présence du répertoire `.direnv/ansible_collections/community/docker` qui contient la collection `community.docker` et ses modules indispensables au fonctionnement du driver Docker de Molecule.

7.3.2 Molecule & Docker

À savoir

Pour réaliser vos tests de rôles Ansible avec Molecule et Docker, vous allez devoir commettre une hérésie.

Vous trouverez dans tous les bon tutoriels sur Docker qu'il est contre-nature de faire démarrer un conteneur avec systemd ou un autre service manager comme processus principal. C'est ce que vous allez devoir faire si vous testez des rôles qui mettent en place des services systèmes (spoiler : c'est le cas 90% du temps).

Prérequis

- *Installation de Molecule*
- Avoir un démon Docker installé sur votre machine de travail.
- Que votre utilisateur de travail ait la permission de gérer Docker (pour éviter de devoir lancer vos tests en `root`).
- Avoir un role à tester, pour l'exemple nous prendrons un rôle fictif placé dans le répertoire `roles/molecule_docker_demo`.

Initialisation

Pour démarrer avec Molecule :

- Placez vous à la racine du rôle que vous souhaitez tester :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training

$ cd roles

$ molecule init role molecule_docker_demo --driver-name docker

$ cd molecule_docker_demo

$ tree -a molecule/
molecule/
```

(continues on next page)

(continued from previous page)

```
└── default
    ├── converge.yml
    ├── molecule.yml
    └── verify.yml

1 directory, 3 files
```

On peut voir que la commande d'init a créé un répertoire `molecule/default/` et plusieurs fichiers :

`molecule.yml`	fichier de configuration des environnements de test et des options molecule pour ce scénario.
`converge.yml`	playbook qui sera appliqué aux environnements de test
`verify.yml`	playbook qui sera joué pour vérifier que `converge.yml` a bien effectué les modifications attendues.

Configuration

Allez modifier le fichier `molecule/default/molecule.yml` pour qu'il ressemble à ceci :

```
---
driver:
  name: docker
platforms:
  - name: debian
    image: "ultimate:11"
    pre_build_image: false
    pull: false
    privileged: true
provisioner:
  name: ansible
verifier:
  name: ansible
```

Approfondir

- Spécification complète des options du driver Docker pour Molecule

Construire son conteneur cible

Pour que Molecule prenne en charge la construction du conteneur qui sert de machine cible, créez un template de Dockerfile au chemin par défaut pour le scénario, `molecule/default/Dockerfile.j2`, avec ce contenu :

```
#
# TEST-ONLY Dockerfile, NE PAS DEPLOYER
#
ARG DEBIAN_TAG=11-slim
FROM debian:$DEBIAN_TAG
ARG DEBIAN_FRONTEND=noninteractive
RUN set -eux; \
  apt-get update && apt-get upgrade && apt-get dist-upgrade; \
  apt-get install --no-install-recommends -y apt-utils \
  curl ca-certificates sudo \
  python python3 python3-apt locales \
  systemd systemd-sysv libpam-systemd dbus dbus-user-session; \
  localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias en_US.UTF-8; \
  localedef -i fr_FR -c -f UTF-8 -A /usr/share/locale/locale.alias fr_FR.UTF-8
ENV LANG fr_FR.utf8
RUN rm -f /lib/systemd/system/multi-user.target.wants/* \
  /etc/systemd/system/*.wants/* \
  /lib/systemd/system/local-fs.target.wants/* \
  /lib/systemd/system/sockets.target.wants/*udev* \
  /lib/systemd/system/sockets.target.wants/*initctl* \
  /lib/systemd/system/sysinit.target.wants/systemd-tmpfiles-setup* \
  /lib/systemd/system/systemd-update-utmp*
ENTRYPOINT ["/lib/systemd/systemd"]
```

Nous construisons **volontairement** un conteneur qui démarre avec `systemd` pour pouvoir tester la mise en place de services système avec Ansible.

Coder le comportement du rôle

Nous allons maintenant remplir les tasks de notre rôle de test dans `tasks/main.yml` :

```
---
#
# roles/molecule_docker_demo/tasks/main.yml
#
- name: Installation de sshd
  apt:
    name: openssh-server
    update_cache: yes

- name: Activation de sshd
  service:
    name: sshd
    state: started
    enabled: true
```

Coder le playbook de test

Enfin nous remplissons le fichier `molecule/default/verify.yml` avec des tasks qui devront valider l'état de notre machine de test :

```
---
- name: Verify
  hosts: all
  gather_facts: false
  tasks:
    - name: Activation de sshd
      service:
        name: sshd
        state: started
        enabled: true
      register: sshd_service_status

    - name: Sshd est actif et démarré
      assert:
        that:
          - sshd_service_status is not changed
```

Lancement

Depuis le répertoire de notre rôle à tester, lancez la commande :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training/roles/molecule_docker_demo

$ molecule test
INFO     default scenario test matrix: dependency, lint, cleanup, destroy, syntax,␣
↪create, prepare, converge, idempotence, side_effect, verify, cleanup, destroy
INFO     Performing prerun...
[...]
Wait for instance(s) deletion to complete ------------------------------- 5.34s
Destroy molecule instance(s) -------------------------------------------- 0.32s
Delete docker networks(s) ----------------------------------------------- 0.02s
INFO     Pruning extra files from scenario ephemeral directory
```

Le workflow complet peut prendre un peu de temps à tourner pour le premier lancement (construction du conteneur oblige).

Si `molecule test` s'exécute sans erreur, notre test est valide.

Ligne d'arrivée

Félicitations, vous venez d'effectuer votre premier test de rôle Ansible avec Molecule et le driver Docker. Vous pouvez extrapoler vos propres suites de tests à partir ce premier exemple basique.

7.3.3 Molecule & Terraform

Nous allons voir ici comment utiliser le driver `delegated` de Molecule en faisant une implémentation basée sur Terraform.

Pour diminuer la phase de prérequis, l'exemple s'appuie également sur Docker, plutôt que sur un Cloud Provider.

Prérequis

- Avoir effectué l'*Installation de Molecule*
- Avoir un démon Docker installé sur votre machine de travail.
- Que votre utilisateur de travail ait la permission de gérer Docker (pour éviter de devoir lancer vos tests en `root`).
- Avoir installé Terraform (voir la documentation officielle ou la *proposition Ultimate*).

Initialisation

- Créez un rôle pour notre exercice

```
$ pwd
/home/user/ansible-workspaces/ultimate-training

$ cd roles

$ molecule init role molecule_terraform_demo --driver-name delegated

$ cd molecule_terraform_demo

$ tree -a molecule/
molecule/
└── default
    ├── converge.yml
    ├── create.yml
    ├── destroy.yml
    ├── INSTALL.rst
    ├── molecule.yml
    └── verify.yml

1 directory, 6 files
```

On peut voir que la commande d'init a créé un répertoire `molecule/default/` et plusieurs fichiers :

`molecule.yml`	fichier de configuration des environnements de test et des options molecule pour ce scénario.
`converge.yml`	playbook qui sera appliqué aux environnements de test
`verify.yml`	playbook qui sera joué pour vérifier que `converge.yml` a bien effectué les modifications attendues.
`create.yml`	playbook qui sera lancé pour la création des environnements de test
`destroy.yml`	playbook qui sera lancé pour la destruction des environnements de test

Terraforming

Nous allons maintenant ajouter le Terraform nécessaire pour simuler un serveur acessible en SSH avec un conteneur Docker local. Il s'agit de techniques dédiées aux tests, à proscrire dans des contexte de déploiements quels qu'ils soient.

Créez un répertoire pour hébergé le code Terraform :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training/roles/molecule_terraform_demo

$ mkdir -p molecule/default/terraform
```

Créez et remplissez les fichiers suivants (les chemins attendus sont en en-tête de chaque bloc) :

- Un Dockerfile pour notre instance de serveur.

```
#
# roles/molecule_terraform_demo/molecule/default/terraform/Dockerfile
#
ARG DEBIAN_TAG=11-slim
FROM debian:$DEBIAN_TAG
ARG DEBIAN_FRONTEND=noninteractive
ARG ROOT_PUBLIC_KEY=to-be-defined
RUN set -eux; \
  apt-get update && apt-get upgrade && apt-get dist-upgrade; \
  apt-get install --no-install-recommends -y apt-utils \
  curl ca-certificates sudo \
  python python3 python3-apt locales \
  systemd systemd-sysv libpam-systemd dbus dbus-user-session openssh-server; \
  localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias en_US.UTF-8; \
  localedef -i fr_FR -c -f UTF-8 -A /usr/share/locale/locale.alias fr_FR.UTF-8
ENV LANG fr_FR.utf8
RUN rm -f /lib/systemd/system/multi-user.target.wants/* \
  /etc/systemd/system/*.wants/* \
  /lib/systemd/system/local-fs.target.wants/* \
  /lib/systemd/system/sockets.target.wants/*udev* \
  /lib/systemd/system/sockets.target.wants/*initctl* \
  /lib/systemd/system/sysinit.target.wants/systemd-tmpfiles-setup* \
  /lib/systemd/system/systemd-update-utmp*; \
  mkdir -p /var/run/sshd /root/.ssh
RUN systemctl enable ssh
RUN echo $ROOT_PUBLIC_KEY > /root/.ssh/authorized_keys
```

(continues on next page)

(continued from previous page)

```
EXPOSE 22
ENTRYPOINT ["/lib/systemd/systemd"]
```

- Le strict minimum de code Terraform pour construire le conteneur, le lancer et récupérer les informations de connexion.

```
#
# roles/molecule_terraform_demo/molecule/default/terraform/main.tf
#
terraform {
  required_providers {
    docker = { source = "kreuzwerker/docker", version = "2.16.0" }
    tls    = { source = "hashicorp/tls", version = "3.3.0" }
  }
}

locals {
  base_name          = "molecule"
  container_name     = "${terraform.workspace}"
  root_key_algorithm = "ED25519"
  identity_file      = "${abspath(path.module)}/${terraform.workspace}.key"
}

resource "tls_private_key" "root" {
  algorithm = local.root_key_algorithm
}

resource "docker_image" "fake_server" {
  name = local.base_name
  build {
    path      = "."
    tag       = ["${local.base_name}:${local.container_name}"]
    build_arg = { ROOT_PUBLIC_KEY : tls_private_key.root.public_key_openssh }
  }
}

resource "null_resource" "private_key" {
  provisioner "local-exec" {
    command = "cat > ${local.identity_file} <<EOF\n${tls_private_key.root.private_key_
↪openssh}\nEOF"
  }
  provisioner "local-exec" {
    command = "chmod 600 ${local.identity_file}"
  }
}

resource "docker_container" "fake_server" {
  name       = local.container_name
  image      = docker_image.fake_server.latest
  privileged = true
}
```

(continues on next page)

(continued from previous page)

```
output "address" { value = docker_container.fake_server.ip_address }
output "user" { value = "root" }
output "identity_file" { value = local.identity_file }
output "instance" { value = terraform.workspace }
output "port" { value = 22 }
```

Intégration Molecule

Maintenant que nous avons de quoi démarrer un serveur local accessible en SSH, il faut l'intégrer dans le cycle de gestion de Molecule.

- Remplacez le contenu du fichier `roles/molecule_terraform_demo/molecule/default/create.yml` par :

```
---
- name: Create
  hosts: localhost
  connection: local
  gather_facts: false
  no_log: "{{ molecule_no_log }}"
  tasks:

    - name: Create the test environment
      terraform:
        project_path: "{{ playbook_dir }}/terraform"
        force_init: true
        workspace: "{{ item.name }}"
        state: present
      register: server
      loop: "{{ molecule_yml.platforms }}"

    - when: server.changed | default(false) | bool
      block:
        - name: Populate instance config dict
          set_fact:
            instance_conf_dict:
              instance: "{{ item.outputs.instance.value }}"
              address: "{{ item.outputs.address.value }}"
              user: "{{ item.outputs.user.value }}"
              port: "{{ item.outputs.port.value }}"
              identity_file: "{{ item.outputs.identity_file.value }}"
          with_items: "{{ server.results }}"
          register: instance_config_dict

        - debug:
            var: instance_conf_dict
        - name: Convert instance config dict to a list
          set_fact:
            instance_conf: "{{ instance_config_dict.results | map(attribute='ansible_
↪facts.instance_conf_dict') | list }}"
```

(continues on next page)

(continued from previous page)

```
        - name: Dump instance config
          copy:
            content: |
              # Molecule managed

              {{ instance_conf | to_json | from_json | to_yaml }}
            dest: "{{ molecule_instance_config }}"
            mode: 0600
```

- Remplacez le contenu du fichier `roles/molecule_terraform_demo/molecule/default/destroy.yml` par :

```
---
- name: Destroy
  hosts: localhost
  connection: local
  gather_facts: false
  no_log: "{{ molecule_no_log }}"
  tasks:
    - name: Destroy the test environment
      terraform:
        project_path: "{{ playbook_dir }}/terraform"
        force_init: true
        workspace: "{{ item.name }}"
        state: absent
      register: server
      loop: "{{ molecule_yml.platforms }}"

    - name: Populate instance config
      set_fact:
        instance_conf: {}

    - name: Dump instance config
      copy:
        content: |
          # Molecule managed

          {{ instance_conf | to_json | from_json | to_yaml }}
        dest: "{{ molecule_instance_config }}"
        mode: 0600
      when: server.changed | default(false) | bool
```

Code du rôle

- Afin d'avoir quelque chose à tester, remplissez le fichier de tasks du rôles :

```
---
#
# roles/molecule_terraform_demo/tasks/main.yml
#
- name: Installation de nginx
  apt:
    name: nginx
    update_cache: yes

- name: Activation de nginx
  service:
    name: nginx
    state: started
    enabled: true
```

Code des tests

- Enfin, remplissez le fichier de vérification Molecule :

```
---
#
# roles/molecule_terraform_demo/molecule/default/verify.yml
#
- name: Verify
  hosts: all
  gather_facts: false
  tasks:
  - name: Installation de nginx
    apt:
      name: nginx
    register: nginx_install

  - name: Activation de nginx
    service:
      name: nginx
      state: started
      enabled: true
    register: nginx_enable

  - assert:
      that:
        - nginx_install is not changed
        - nginx_enable is not changed
```

Test complet

Tout est en place, vous pouvez maintenant lancer un test bout en bout avec les commandes suivantes :

```
$ pwd
/home/user/ansible-workspaces/ultimate/training/roles/molecule_terraform_demo

$ molecule test
INFO     default scenario test matrix: dependency, lint, cleanup, destroy, syntax,␣
↪create, prepare, converge, idempotence, side_effect, verify, cleanup, destroy
INFO     Performing prerun...
INFO     Set ANSIBLE_LIBRARY=/home/user/.cache/ansible-compat/9c82a6/modules:/home/user/.
↪ansible/plugins/modules:/usr/share/ansible/plugins/modules
INFO     Set ANSIBLE_COLLECTIONS_PATHS=/home/user/.cache/ansible-compat/9c82a6/
↪collections:/home/user/ansible-workspaces/ultimate/training/.direnv:/home/user/ansible-
↪workspaces/ultimate/training/.direnv
INFO     Set ANSIBLE_ROLES_PATH=/home/user/.cache/ansible-compat/9c82a6/roles:/home/user/
↪ansible-workspaces/ultimate/training/roles/molecule_terraform_demo/roles:roles
INFO     Running default > dependency
WARNING  Skipping, missing the requirements file.
WARNING  Skipping, missing the requirements file.
INFO     Running default > lint
INFO     Lint is disabled.
INFO     Running default > cleanup
WARNING  Skipping, cleanup playbook not configured.
INFO     Running default > Destroy
[...]
=============================================================================
Destroy the test environment ------------------------------------------------ 2.74s
Dump instance config -------------------------------------------------------- 0.34s
Populate instance config ---------------------------------------------------- 0.02s
INFO     Pruning extra files from scenario ephemeral directory
```

Ligne d'arrivée

Vous avez maintenant un workflow complet de Molecule qui intègre Terraform comme implémentation du driver `delegated`. Libre à vous d'adapter le code Terraform pour pouvoir lancer vos tests Molecule directement sur AWS, GCP ou tout autre founisseur d'infrastructure.

7.3.4 Goss Verifier

Prérequis

- Avoir effectué l'*Installation de Molecule*
- Avoir un démon Docker installé sur votre machine de travail.
- Que votre utilisateur de travail ait la permission de gérer Docker (pour éviter de devoir lancer vos tests en `root`).

Initialisation

- Créez un rôle pour notre exercice

```
$ pwd
/home/user/ansible-workspaces/ultimate-training

$ cd roles

$ molecule init role molecule_goss_demo --driver-name docker --verifier-name goss

$ cd molecule_goss_demo
```

- Créez un fichier `molecule/default/Dockerfile.j2` pour service de modèle à notre host de test :

```
#
# TEST-ONLY Dockerfile, NE PAS DEPLOYER
#
ARG DEBIAN_TAG=11-slim
FROM debian:$DEBIAN_TAG
ARG DEBIAN_FRONTEND=noninteractive
RUN set -eux; \
  apt-get update && apt-get upgrade && apt-get dist-upgrade; \
  apt-get install --no-install-recommends -y apt-utils \
  curl ca-certificates sudo \
  python python3 python3-apt locales \
  systemd systemd-sysv libpam-systemd dbus dbus-user-session; \
  localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias en_US.UTF-8; \
  localedef -i fr_FR -c -f UTF-8 -A /usr/share/locale/locale.alias fr_FR.UTF-8
ENV LANG fr_FR.utf8
RUN rm -f /lib/systemd/system/multi-user.target.wants/* \
  /etc/systemd/system/*.wants/* \
  /lib/systemd/system/local-fs.target.wants/* \
  /lib/systemd/system/sockets.target.wants/*udev* \
  /lib/systemd/system/sockets.target.wants/*initctl* \
  /lib/systemd/system/sysinit.target.wants/systemd-tmpfiles-setup* \
  /lib/systemd/system/systemd-update-utmp*
ENTRYPOINT ["/lib/systemd/systemd"]
```

- Modifiez le fichier `molecule/default/molecule.yml` comme ceci :

```
---
dependency:
  name: galaxy
driver:
  name: docker
platforms:
  - name: debian
    image: "ultimate:11"
    pre_build_image: false
    pull: false
    privileged: true
provisioner:
  name: ansible
```

(continues on next page)

(continued from previous page)

```
verifier:
  name: goss
```

À ce stade nous pouvons déjà lancer un `molecule test` qui s'appuie sur notre Dockerfile. Cependant nous testerions un rôle vide, donc peu de gloire.

Code du rôle

- Modifiez le fichier `tasks/main.yml` pour que notre rôle installe et active le démon SSH :

```
---
- name: Installation de sshd
  apt:
    name: openssh-server
    update_cache: yes

- name: Activation de sshd
  service:
    name: sshd
    state: started
    enabled: true
```

Code des tests

- Modifiez l'état attendu par goss en éditant le fichier `molecule/default/tests/test_default.yml` :

```
---
port:
  tcp:22:
    listening: true
    ip:
    - 0.0.0.0
service:
  sshd:
    enabled: true
    running: true
process:
  sshd:
    running: true
```

Lancement

Tout est prêt vous pouvez maintenant lancer les tests :

```
$ pwd
/home/user/ansible-workspaces/ultimate-training/roles/molecule_goss_demo

$ molecule test
[...]
```

Ligne d'arrivée

Vous avez là la combinaison la plus confortable pour tester des rôles Ansible au moment de la rédaction.

7.4 Pipelines

Dans ce chapitre d'exercices, vous aurez une première expérience dans l'intégration d'Ansible avec Packer, dans une chaîne d'intégration et de déploiement continus.

7.4.1 Intégration Gitlab-CI

Objectif

Industrialiser la création d'images de machines virtuelles EC2.

Note

Cet exercice a été validé sur gitlab.com, le service SaaS de GitLab. Il vous faut créer un compte gratuit sur sur gitlab.com en pré-requis, ou utiliser votre compte personnel si vous en possédez déjà un.

Notre image doit comporter tout le nécessaire pour construire des AMI AWS.

Créez un projet GitLab vierge. Ce projet comportera un fichier Packer, un playbook Ansible et son `requirements.yml` pour la gestion des dépendances.

Fichier `quick-start.pkr.hcl`

VPC par défaut

Le fichier de configuration Packer suivant fonctionne dans le cas où le VPC par défaut existe dans le compte AWS cible. Si vous ne disposez pas de VPC par défaut, utilisez le fichier de configuration n°2.

- Exemple n°1 (VPC par défaut requis) :

```
packer {
  required_plugins {
    amazon = {
      version = ">= 1.0.8"
      source  = "github.com/hashicorp/amazon"
    }
  }
}

locals { timestamp = regex_replace(timestamp(), "[- TZ:]", "") }

source "amazon-ebs" "quick-start" {
  ami_name      = "packer-example ${local.timestamp}"
  instance_type = "t2.micro"
```

(continues on next page)

(continued from previous page)

```
  region        = "eu-west-1"
  source_ami    = "ami-0a2616929f1e63d91"
  ssh_username  = "ubuntu"
}

build {
  sources = ["source.amazon-ebs.quick-start"]

  provisioner "ansible" {
    playbook_file   = "ansible/playbook.yml"
  }
}
```

Choix du VPC

Le fichier de configuration Packer suivant fonctionne dans le cas où le VPC par défaut n'existe pas. Dans ce cas, vous devez spécifier à Packer le VPC et le sous-réseau à utiliser.

- Exemple n°2

```
source "amazon-ebs" "quick-start" {
  ami_name      = "packer-example ${local.timestamp}"
  instance_type = "t2.micro"
  region        = "eu-west-1"
  source_ami    = "ami-0a2616929f1e63d91"
  ssh_username  = "ubuntu"

  subnet_filter {
    filters = {
      "tag:Name" = "<Valeur du tag:Name du sous-réseau à utiliser>"
    }
    most_free = true
    random    = false
  }

  vpc_filter {
    filters = {
      cidr       = "<CIDR du VPC à utiliser>"
      isDefault  = "false"
      "tag:Name" = "<Valeur du tag:Name du VPC à utiliser"
    }
  }
}

build {
  sources = ["source.amazon-ebs.quick-start"]
  provisioner "ansible" {

    playbook_file   = "ansible/playbook.yml"
  }
}
```

Fichiers Ansible

Organisez les fichiers Ansible en suivant les bonnes pratiques présentées ici.

Ajoutons le playbook `ansible/playbook.yml` suivant :

```
- hosts: "{{ target|default('all') }}"
  become: true
  roles:
    - ansible-role-security
  tags:
    - security
```

Le rôle `ansible-role-security` est un rôle public qui permet de configurer et ajouter des paramètres de sécurité basiques. Ajoutons le rôle dans le fichier `ansible/requirements.yml` :

```
- src: https://github.com/geerlingguy/ansible-role-security.git
  scm: git
  version: "2.1.0"
```

À ce stade vous devriez avoir l'arborescence suivante dans votre projet GitLab :

```
.
├── README.md
├── ansible
│   ├── playbook.yml
│   └── requirements.yml
└── quick-start.pkr.hcl

1 directory, 4 files
```

Configuration Gitlab-CI

Dans le menu *Settings –> CI/CD* déroulez la section *Variables* afin d'ajouter les variables d'environnement suivantes :

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_DEFAULT_REGION`

Authentification AWS

Pour plus d'informations au sujet des déploiements sur AWS depuis Gitlab

---

Maintenant que les variables sont ajoutées, il nous reste à définir les actions dans la pipeline. Pour cela il suffit d'ajouter un fichier `.gitlab-ci.yml` comme suit :

```
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Packer.
↪gitlab-ci.yml

image:
```

(continues on next page)

(continued from previous page)

```
  name: stelar/packer_ansible_build:latest

before_script:
  - packer --version
  - find . -maxdepth 1 -name '*.pkr.hcl' -print0 | xargs -t0n1 packer init

stages:
  - validate
  - build
  - test
  - deploy

validate:
  stage: validate
  script:
    - find . -maxdepth 1 -name '*.pkr.hcl' -print0 | xargs -t0n1 packer validate

build:
  stage: deploy
  environment: production
  before_script:
    - ansible-galaxy install -r ansible/requirements.yml -p ./ansible/roles --force
  script:
    - find . -maxdepth 1 -name '*.pkr.hcl' -print0 | xargs -t0n1 packer build
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual
```

Ce fichier de configuration de CI/CD est largement basé sur le modèle disponible en lien en haut du fichier.

On utilise une image Docker qui contient tout le nécessaire dont Packer, Ansible et Awscli.

Ligne d'arrivée

Féliciations, vous venez de configurer votre première chaîne d'intégration et de déploiement continus pour la construction des AMI AWS via Packer et Ansible !

Mais attention, il reste encore du chemin avant d'aller en production ! En effet, nous avons posé les bases, nous irons plus loin dans une prochaine section.

CHAPTER

EIGHT

CRÉDITS

8.1 Auteurs

- Aurélien Maury
- Gautier Loterman

8.2 Dispensateurs de savoir-faire

- Jérôme Ornech

8.3 Grands Testeurs

- Super Doudou Ninja

Mention spéciale

Un grand merci à tous les gens avec qui nous avons interagi au fil de nos missions, meetups, conférences, etc. C'est grâce à tous ces échanges que nous progressons.

Enfin, aucun mot ne saurait remercier à leur juste valeur tous les contributeurs, petits ou grands, qui rêve le monde en Open Source chaque jour.
