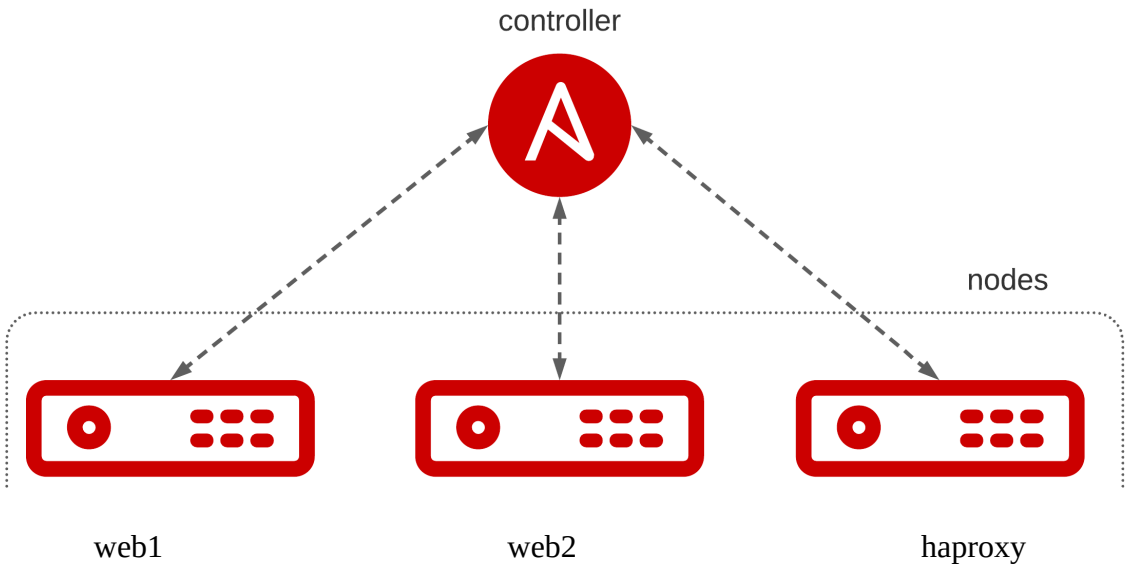
Lab3 _ Les playbooks

Topologie de lab

On trouvera ici la topologie du lab à monter, constituée de quatre noeuds Linux dont trois à gérer à partir d'un contrôleur avec Ansible.



Une solution avec Docker et un script Bash lance le lab. Cette solution est facile à monter et assez robuste.

Cette solution a le très grand avantage de fonctionner partout, très vite et au coût le plus réduit. Cette solution convient pour des exercices de gestion multi-noeuds au début de l'apprentissage.

Toutefois, les conteneurs avec Docker connaissent des limites tenant à leur nature notamment concernant le support du lancement des services à chaud. Cela signifie que toute perspective qui touche aux services (démarrage/arrêt) sera difficile à éprouver, de même que les redémarrages du conteneur lui-même à partir de l'espace du conteneur.

Prérequis

Le seul prérequis est d'installer **Docker**, ici sous Linux

https://docs.docker.com/engine/install/ubuntu/

Se procurer le script

```
curl -s https://raw.githubusercontent.com/goffinet/docker-ansible-lab/master/startlab.sh > ./startlab.sh
chmod +x startlab.sh
```

Lancer le lab

./deploy.sh

Options :

- --create : lancer des conteneurs
- --drop : supprimer les conteneurs créer par le deploy.sh
- --infos : caractéristiques des conteneurs (ip, nom, user...)
- --start : redémarrage des conteneurs
- --ansible : déploiement arborescence ansible

les conteneurs basés sur Debian 10 qui agissent comme des noeuds exploitables. Ces noeuds ont déjà été approvisionnés avec la clé ssh du conteneur.

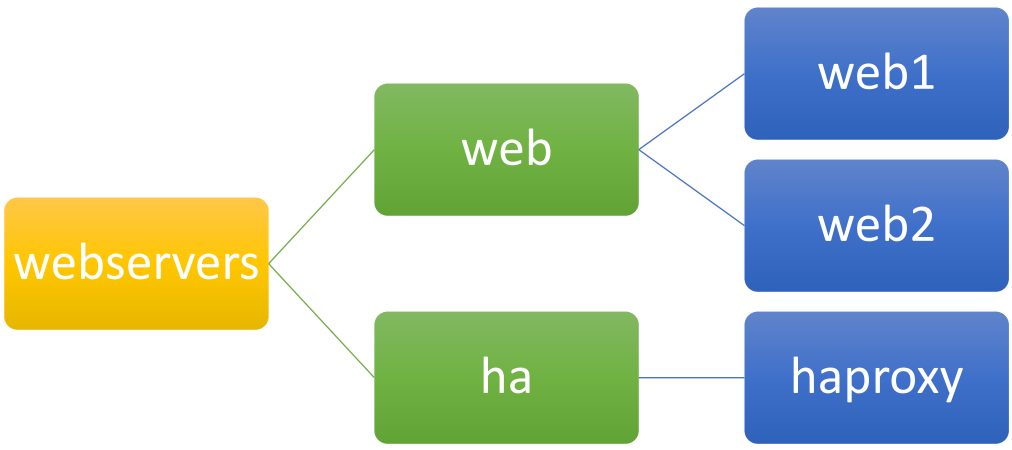
Récupérer les adresses IP des machines

Récupérer les adresses IP des nœuds :

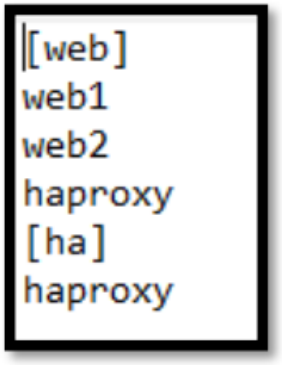
./deploy.sh –infos

Créer un fichier d'inventaire :

Dans le fichier /etc/ansible/hosts, ajouter node0 et web2 au groupe **web** et node2 au groupe **haproxy**. Les deux groupes(web et haproxy) appartiennent au groupe **webservers**



/etc/ansible/hosts



Tester les commandes ad-hoc suivantes :

ansible all -m ping

ansible web1:web2 -m ping

ansible web:ha -m ping

ansible 'web*' -m ping

ansible all :!web1 -m ping

ansible 'web:&ha -m ping

Les playbooks Ansible

Les commandes ad-hoc sont des appels directs de modules Ansible qui fonctionnent de façon idempote.

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

Syntaxe yaml

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

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

A quoi ça ressemble ?

Une liste

```
- 1
- Poire
- "Message à caractère informatif"
```

Un dictionnaire

```
clé1: valeur1
clé2: valeur2
clé3: 3
```

Un exemple imbriqué plus complexe

```
marché: # debut du dictionnaire global "marché"
  lieu: Crimée Curial
  jour: dimanche
  horaire:
    unité: "heure"
    min: 9


    max: 14 # entier
  fruits: #liste de dictionnaires décrivant chaque fruit
    - nom: pomme
      couleur: "verte"
      pesticide: avec #les chaines sont avec ou sans " ou '
            # on peut sauter des lignes dans interrompre la liste ou le
dictionnaire en court
    - nom: poires
      couleur: jaune
      pesticide: sans
  légumes: #Liste de 3 éléments
    - courgettes
    - salade

    - potiron
#fin du dictionnaire global
```

Structure d'un playbook

```
---
- name: premier play # une liste de play (chaque play commence par un
tiret)
  hosts: serveur_web # un premier play
  become: yes
  gather_facts: false # récupérer le dictionnaires d'informations (facts)
relatives aux machines

  vars:
    logfile_name: "auth.log"

  var_files:
    - mesvariables.yml

  pre_tasks:
    - name: dynamic variable
      set_fact:
        mavariable: "{{ inventory_hostname + 'prod' }}" #guillemets
obligatoires
```

```
  roles:
    - flaskapp

  tasks:
    - name: installer le serveur nginx
      apt: name=nginx state=present # syntaxe concise proche des commandes
ad hoc mais moins lisible

    - name: créer un fichier de log
      file: # syntaxe yaml extensive : conseillée
        path: /var/log/{{ logfile_name }} #guillemets facultatifs
        mode: 755

    - import_tasks: mestaches.yml

  handlers:
    - systemd:
        name: nginx
        state: "reloaded"

- name: un autre play
  hosts: dbservers
  tasks:
    ...
```

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

Ordre d'execution

1. `pre_tasks`
2. `roles`
3. `tasks`
4. `post_tasks`
5. `handlers`

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

bonnes pratiques de syntaxe

- Indentation de deux espaces.
- Toujours mettre un `name:` qui décrit lors de l'execution la tache en court

- Utiliser les arguments au format yaml (sur plusieurs lignes) pour la lisibilité, sauf s'il y a peu d'arguments

Imports et includes

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

- `import_tasks`: importe une liste de tâches (atomiques)
- `import_playbook`: importe une liste de play contenus dans un playbook.

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

Au contraire, `include_tasks` permet d'intégrer une liste de tâche **dynamiquement** pendant l'exécution

Par exemple:

```yaml
vars:
  apps:
    - app1
    - app2
    - app3

tasks:
  - include_tasks: install_app.yml
    loop: "{{ apps }}"
```

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

Élévation de privilège

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

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

Variables Ansible

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

Pour s'en rendre compte on peut lancer : `ansible <hote_ou_groupe> -m debug -a "msg={{ hostvars }}"`

Ce dictionnaire contient en particulier:

- des variables de configuration ansible (`ansible_user` par exemple)
- des facts c'est à dire des variables dynamiques caractérisant les systèmes cible (par exemple `ansible_os_family`) et récupéré au lancement d'un playbook.
- des variables personnalisées (de l'utilisateur) que vous définissez avec vos propre nom

Comprendre et utiliser les balises

Ecrire ce playbook et le lancer avec différents tags

```
---
- name: play1
  hosts: all
  tasks:
  - name: create
    file:
      dest: /home/ansible/fic2
      state: touch
    tags:
      - create

- name: play2
  hosts: all:!haproxy
  tags:
    - delete
  tasks:
  - name: delete
    file:
      dest: /home/ansible/fic2
      state: absent

- name: play3
  hosts: haproxy
  tasks:
  - name: delete
    file:
      dest: /home/ansible/fic2
      state: absent
    tags:
      - delete
```

```
ansible-playbook -i hosts play1.yml

ansible-playbook -i hosts play1.yml –tags create

ansible-playbook -i hosts play1.yml –tags delete

ansible-playbook -i hosts play1.yml –skip-tags delete

ansible-playbook -i hosts play1.yml –skip-tags delete –list-tags
```

Exécuter les tâches en local

La plupart du temps, nous exécutons du code Ansible sur un ensemble de hôtes distants ; mais parfois, il est nécessaire de créer quelque chose en local, c'est-à-dire sur la machine où vous exécutez le code Ansible. Pour créer des clés SSH ou des certificats et vouloir les distribuer à toutes les machines distantes.

Il y a deux façons d'exécuter les commandes Ansible en local :

- Premièrement, on peut traiter notre machine locale comme s'il s'agissait d'une machine distante, faudra la rajouter dans l'inventaire et passer par le mécanisme SSH ou alors, utiliser ce qu'on appelle une « connexion locale ».
- Du fait qu'on n'a pas créé les clés SSH pour une machine, on peut par contre, passer par la notion de localhost**.**

Tester ces deux playbooks :

```
---
- name: play1
  hosts: control
  tasks:
  - name: creation via connexion locale
    file:
      dest: /home/ansible/Chapitre_02/fic3
      state: touch
```

```
---
- name: play1
  hosts: localhost
  connection: local
  tasks:
  - name: creation via connexion locale
    file:
      dest: /home/ansible/Chapitre_02/fic3
      state: touch
```

Utiliser les variables dans l'inventaire

On va voir comment définir des variables dans un fichier inventaire statique. Rappelez-vous, on a ici un inventaire et avec des variables de « host » ou des variables de groupe, comme all-file dans le groupe « all ».

Dans le cas d'un inventaire de type dynamique, on va créer des répertoires host_vars et group_vars dans le dossier de travail, et qui vont contenir les fichiers cibles.

hosts

```
[web]
web1        ansible_connection=ssh  ansible_user=ansible
web2        ansible_connection=ssh  ansible_user=ansible
[ha]
haproxy
[all:vars]
all_file=/home/ansible/all_file
[web:vars]
web_file=/home/ansible/index.html
[ha:vars]
ha_config=/home/ansible/haproxy.cfg
```

play3.yml

```
---
  - hosts: web
    tasks:
    - name: create a web file
      file:
        dest: '{{web_file}}'
        state: '{{file_state}}'

  - hosts: ha
    tasks:
    - file:
        dest: '{{ha_config}}'
        state: '{{file_state}}'
      when: ha_config is defined

  - hosts: all
    tasks:
    - file:
        dest: '{{all_file}}'
        state: '{{file_state}}'
```

ansible-playbook -i hosts play3.yml -e file_state=touch

Créer un inventaire dynamique

Si vous utilisez des outils de développement de VM, ou d'instance de cloud, vous ne pouvez pas en fait, tout gérer avec un inventaire statique. Donc, il faut passer à un inventaire dynamique.

Il existe deux méthodes :

- la première, c'est d'utiliser des scripts qui vont avoir une sortie, dans un format spécifique, JSON. On peut utiliser tout type de scripts, du Python, du Shell, etc.
- Deuxième méthode, c'est utiliser des plug-ins d'inventaire, qui sont dédiés et fournis par les éditeurs, comme Amazon avec AWS EC2, VirtualBox, et d'autres outils provenant du cloud. On va commencer par VirtualBox, et on va d'abord configurer un fichier ansible.cfg, qui va en fait, définir certaines choses, par exemple, la partie inventaire.

Finalement, n'importe quelle source de données peut devenir un inventaire qu'Ansible comprend grâce aux **plugins d'inventaire**. Par défaut, les plugins d'inventaire suivants sont activés :

- `host_list` : une simple liste d'hôte,
- `script` : un script qui construit un inventaire dynamique contre une source de données,
- `yaml` : l'inventaire en format de fichier de données YAML,
- `ini` : l'inventaire en format de fichier de données INI.

Lister les informations de l'inventaire en format JSON ou YAML ou sous forme de graphe.

Avec l'action –list, l'inventaire sort en format JSON :

```
ansible-inventory -i inventory --list
```

Mais il pourrait sortir en format YAML :

```
ansible-inventory -i inventory --list -y
```

Ou sous forme de graphe :

```
ansible-inventory -i inventory --graph
```

Structure d'un inventaire : Hôtes et groupes d'hôtes

L'inventaire Ansible est constitué d'hôtes (`hosts:`), de groupes qui comprennent les hôtes et des variables (`vars:`) attachées à ces hôtes et à ces groupes. un groupe peut imbriqué d'autres groupes (`children:`).

Un hôte appartient toujours au moins à deux groupes, car deux groupes d'hôtes sont toujours présents par défaut :

- `all` : qui contient **tous les hôtes**.

- `ungrouped` : qui contient tous les hôtes qui ne sont pas dans un autre groupe que `all`.

Liste d'hôtes

L'inventaire le plus simple est une liste d'hôte en argument de l'option `-i`. Par exemple, pour désigner un inventaire directement comme une liste d'hôtes avec les binaires `ansible-playbook` ou `ansible` :

```
ansible -i '192.168.122.10,192.168.122.11,192.168.122.12,' -m ping all
```

ou encore par leur nom :

```
ansible -i 'node0,node1,node2,' -m ping all
```

ou encore avec une liste d'un seul hôte :

```
ansible -i '192.168.122.10,' -m ping all
```

Utiliser les variables avec un inventaire dynamique

On a vu comment définir des variables dans un fichier inventaire statique. Rappelez-vous, on a ici un inventaire et avec des variables de « host » ou des variables de groupe, comme all-file dans le groupe « all ». Dans le cas d'un inventaire de type dynamique, on va créer des répertoires host_vars et group_vars dans le dossier de travail, et qui vont contenir les fichiers cibles. Les fichiers portent les noms de cibles.

On peut maintenant lancer notre playbook : ansible-playbook avec, comme extra paramètre, -e file_state=touch.

```
---
- hosts: web
  remote_user: ansible
  tasks:
  - name: create a web file
    file:
      dest: '{{web_file}}'
      state: '{{file_state}}'

- hosts: haproxy
  remote_user: ansible
  tasks:
  - name: create a haproxy file
    file:
      dest: '{{ha_config}}'
      state: '{{file_state}}'

- hosts: all
  remote_user: ansible
  tasks:
  - name: all systems should have a file
    file:
      dest: '{{all_file}}'
      state: '{{file_state}}'
```

Aborder le rôle du playbook :

```
---
- name: Installation des serveurs web
  hosts: web
  vars:
    apache_port: 80
    tomcat_port: 8080
    apache_max_keepalive: 25
  remote_user: ansible
  become: yes
  tasks:
  - name: Install du package apache
    yum:
      name: httpd
      state: latest
  - name: Start du service apache
    service:
      name: httpd
      state: started
  - name: enable selinux
    command: /sbin/setenforce 1
  - name: debug
    debug:
      msg: "Ceci n'apparait qu'avec l'option -vv+"
      verbosity: 2
    tags:
    - debug
```

Exécuter des tâches conditionnelles

Lorsqu'on souhaite exécuter une action sur un hôte, sous condition d'une information, d'un état ou de la valeur d'une variable sur un autre hôte, il nous faut récupérer des informations grâce à une variable hostvars. Hostvars est en réalité un tableau associatif qui fait référence à toutes les variables de toutes les machines.

Le moyen le plus simple de voir un petit peu le contenu du tableau, c'est d'utiliser non pas un « playbook », mais la commande ansible avec un mode de « debug »

ansible web1 -m debug -a "var=hostvars[‘web2’] "

```
---
- hosts: web
  remote_user: ansible
  tasks:
  - name: first create
    file:
      dest: /home/ansible/fic1
      state: '{{file_state}}'
    when: hostvars[inventory_hostname]['inventory_hostname'] == 'web1'

  - name: second create
    file:
      dest: /home/ansible/fic2
      state: '{{file_state}}'
    when: inventory_hostname == 'web2'
```

Tirer parti des boucles

Nous allons voir comment créer une boucle pour faire varier une variable parmi une liste de valeurs. Pour créer les listes, il y a deux formats : le premier étant celui-ci avec la variable « files » avec trois valeurs entre crochets et séparées par des virgules. Ou bien, un format YML standard avec la variable dirs et chaque valeur est sur une ligne précédée par un tiret.

```
---
- hosts: all
  remote_user: ansible
  vars:
    files: [fic1,fic2,fic3]
    dirs:
      - rep1
      - rep2
      - rep3
  tasks:
  - name: create files for Redhat
    file:
      dest: '/home/ansible/{{item}}'
      state: '{{file_state}}'
    with_items: '{{files}}'
    when: ansible_os_family == "RedHat"

  - name: create directories for Redhat
    file:
      dest: '/home/ansible/{{item}}'
      state: '{{dir_state}}'
    with_items: '{{dirs}}'
    when: ansible_os_family == "RedHat"
```

Employer les boucles avec des dictionnaires

De nos jours, avec l'avènement des web services, du DevOps ou du cloud, de manière générale, les données sont souvent sous forme de tableaux associatifs avec la notion de clé valeur.

```
---
- hosts: all
  remote_user: ansible
  vars:
    servers:
      windows:
        win10:
          function: ad
          environment: prod
        win7:
          function: sql
          environment: test
      linux:
        redhat:
          function: web
          environment: test
        debian:
          function: db
          environment: prod
  tasks:
  - name: iterate over windows array
    file:
      name: '/home/ansible/{{item.key}}-{{item.value.function}}'
      state: '{{file_state}}'
    with_dict: '{{servers.windows}}'

  - name: iterate over linux array
    file:
      name: '/home/ansible/{{item.key}}-{{item.value.function}}'
      state: '{{file_state}}'
    with_dict: '{{servers.linux}}'
    when: 'item.value.environment == "prod"'
```