Tutoriel pour Apprendre à utiliser Ansible

Par Quentin Busuttil

Date de publication : 3 juillet 2017

CONFIRMÉ

Ansible vous donne la possibilité d'automatiser l'installation, le déploiement et la gestion de vos serveurs.

Pour réagir à ce tutoriel, un espace de dialogue vous est proposé sur le forum. **Commentez**

I - Automatiser la gestion des serveurs avec Ansible

Ansible est un outil qui permet - entre autres choses - d'automatiser l'installation, le déploiement et la gestion de vos serveurs. Vous utilisez certainement ssh pour installer les programmes dont vous avez besoin et configurer vos serveurs. Peut-être même avez-vous créé des scripts pour que tout ça aille plus vite. Ansible permet de créer des « Playbooks », qui ne sont autres que des scripts à la sauce Ansible, et permettent de configurer vos serveurs.

Sa grande force est qu'il est *agentless*, autrement dit, rien n'est à placer sur vos serveurs. Vous installez Ansible sur votre laptop par exemple, et le tour est joué. Vous pouvez ensuite lancer l'install de vos 40 serveurs de base de données en une seule commande ! Ça vous émoustille ? Alors allons-y !

I-A - L'install

Vous vous doutez bien qu'il faut avant tout installer Ansible sur votre *control machine*. Rien de bien compliqué, ça tourne sur à peu près tout sauf Windows et il y a plusieurs procédures au choix : du `git clone` au `apt-get`. Elles sont pour la plupart détaillées sur la **page d'install de Ansible**.

En ce qui me concerne, j'ai voulu tenter de l'installer sur mon Mac via **Macports**, pour plus de facilité et des mises à jour en toute souplesse. L'install se passe assez simplement via un `sudo port install ansible`. Sur Mac, et c'est valable aussi pour l'install via Homebrew, au lieu d'avoir la config dans `/etc/ansible/` tout se passe dans `/opt/local/etc/ansible/`, ça vous évitera de chercher.

Dans le cas où vous suivriez les instructions d'installation de la doc Ansible - laquelle suggère de procéder à l'installation via PIP -, il n'y a pas de fichier de config par défaut. Pourtant, Ansible le cherchera dans `/etc/ansible`, de la même manière que le fichier `hosts`.

Deux solutions sont possibles. Soit vous créez un fichier `.ansible.cfg` dans votre `HOME`, et vous précisez bien dans celui-ci où se trouve le fichier `hosts`, soit vous créez `/etc/ansible` et vous y placez le fichier `hosts`.

I-B - Configuration des hôtes

Avant tout, il faut bien comprendre que Ansible repose sur le protocole ssh. Ainsi, c'est via ce protocole qu'il se connectera à vos serveurs. Par défaut sur le port 22 évidemment. Il tentera aussi par défaut de se connecter via une clef ssh de `~/.ssh/`. De plus, l'utilisation d'OpenSSH permet de lire le fichier de configuration `~/.ssh/conf`. Néanmoins, l'usage d'OpenSSH est conditionné par une version récente de ce dernier. Dans le cas contraire, Ansible fallback sur une **bibliothèque en Phyton**.

Les hôtes se configurent dans `/etc/ansible/hosts` :

```
1. # les crochets permettent de définir des groupes
2. # on pourra ainsi appliquer la même conf à tous les serveurs de backup
3. # un serveur peut appartenir à plusieurs groupes
4. [backup]
5. # on fait ici référence à un serveur présent dans le fichier conf de .ssh
6. jarjar
7.
8. # définir un port non standard
9. www.buzut.fr:5309
10.
11. # il est possible d'adresser plusieurs serveurs qui suivent un nommage spécifique
12. # en utilisant la notation intervalle des REGEX
13. www[01:50].buzut.fr
```

Beaucoup d'autres options sont disponibles, vous trouverez toutes les possibilités dans l'**Inventory**.

Une fois les hôtes configurés, vous pouvez tester une première commande afin de voir que la configuration fonctionne. Le module ping affiche simplement « pong », il n'a rien à voir avec le ping ICMP :

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

# il se peut que vous ayez un problème
buzut | UNREACHABLE! => {
    "changed": false,
    "msg": "SSH encountered an unknown error during the connection. We recommend you re-run the command using -vvvv, which will enable SSH debugging output to help diagnose the issue",
    "unreachable": true
}

# après le -vvv et quelques recherches, Ansible crée un dossier .ansible dans votre home
# il se peut que ce répertoire n'appartienne pas au bon utilisateur
# pour remédier à cela
sudo chown -R votre_user ~/.ansible
```

Dernier petit détail, nous avons entraperçu dans les commentaires qu'il y a de puissants moyens de sélectionner les hôtes. On peut en sélectionner un unique ou tout un groupe, appliquer des **REGEX** pour en exclure ou préciser un intervalle. Pour tirer toute la souplesse et la puissance de cette notation, lisez rapidement la page **patterns** de la doc.

I-C - La ligne de commande

Ansible possède une ligne de commande. Pourquoi donc ? Eh bien ! c'est la même chose que pour les scripts. Parfois vous faites un script parce que vous savez que vous aurez à refaire cette manipulation, parfois, c'est juste du one shot - ou c'est super court - donc vous tapez directement ce que vous voulez faire. Par exemple, admettons que vous vouliez redémarrer tous les serveurs de base de données, vous pourriez faire :

```
# database est notre groupe de serveur [database]
# l'option -a permet de préciser l'argument de la commande
ansible database -a "/sbin/reboot"

# par défaut, ansible lance la commande sur cinq serveurs en parallèle en faisant des forks de son processus
# si votre groupe contient beaucoup de serveurs - front[1-500] par exemple - il peut être judicieux de l'augmenter
# vous pouvez gérer cela avec le paramètre -f
ansible database -a "/sbin/reboot" -f 25
```

Vous pouvez utiliser l'option -u username pour exécuter une commande depuis un autre utilisateur, -k pour passer en root et entrer le mot de passe root.

La CLI permet d'appeler des modules pour exécuter les commandes que nous voulons. Dans l'exemple ci-dessus, nous n'avons rien précisé, car le module par défaut est command et c'est ce que nous voulions. Admettons maintenant que nous voulions copier un fichier sur l'ensemble de nos serveurs web :

```
# le module copy permet de transférer un fichier
ansible web -m copy -a "src=/Users/Buzut/Desktop/super-site.conf dest=/etc/apache2/sites-available/super-site.conf"
```

Vous pouvez en savoir plus sur les modules disponibles directement sur **la page de docs dédiée**.

I-D - Les playbooks

J'en ai rapidement parlé en introduction, les playbooks sont des fichiers en YAML qui décrivent les tâches qu'Ansible doit accomplir sur vos serveurs. Les playbooks utilisent une syntaxe très simple : on définit les hôtes, les variables éventuelles, puis on crée des tâches. Chaque tâche possède un nom et appelle des modules (les mêmes qu'en CLI).

Sachez que vous pouvez lancer vos playbooks avec l'option `--syntax-check`, qui comme son nom l'indique, s'assure qu'il n'y a pas d'erreur dans la syntaxe de vos playbooks. Ensuite, vous pouvez également utiliser `--check` afin de simuler un play sans effectuer aucun changement. Dans cette dernière option, Ansible va se connecter au(x) serveur(s) et lancer les modules en leur demandant de ne pas effectuer de modifications (tous les modules ne sont pas compatibles, auquel cas, ils ne retourneront pas les changements potentiels).

```
1. ---
2. # les documents YAML commencent toujours par "---"
3.
4. # le nom de l'hôte ou groupe concerné
5. - hosts: webserver
6.
7.   # on déclare les éventuelles variables
8.   vars:
9.     http_port: 80
10.     domain: buzut.fr
11.
12.   # nom de l'utilisateur du compte (lance celui du .ssh/conf par défaut)
13.   remote_user: root
14.
15.   # ici débute la liste des tâches
16.   tasks:
17.
18.   # nom d'une tâche
19.   - name: ensure server is up to date
20.     # nom du module à utiliser
21.     apt:
22.       update_cache: yes
23.       upgrade: full
24.
25.   # installons apache
26.   - name: install apache2
27.     # il existe une syntaxe alternative, plus condensée
28.     # apt: name=apache2 update_cache=yes state=latest
29.
30.     apt:
31.       name: apache2
32.       update_cache: yes
33.       state: latest
34.
35.   # s'assurer que le mod rewrite est actif (l'activer sinon)
36.   - name: enabled mod_rewrite
37.     apache2_module:
38.       name: rewrite
39.       state: present
40.
41.   # le module template fonctionne de manière similaire à copy (vu plus haut)
42.   # mais il injecte dynamiquement les variables nécessaires
43.   - name: write the apache config file
44.     template:
45.       src: /Users/Buzut/.ansible/templates/vhost.conf
46.       dest: /etc/apache2/sites-available/buzut.conf
47.
48.   - name: enable vhost
49.     command: a2ensite buzut
50.
51.   - name: restart apache
52.     service:
53.       name: httpd
54.       state: restarted
```

Pour que vous compreniez bien le fonctionnement des variables, voici à quoi ressemble le fichier de template vhost.conf :

```
<VirtualHost *:{{ http_port }}>
    ServerAdmin webmaster@{{ domain }}
    ServerName {{ domain }}
    ServerAlias www.{{ domain }}
    DocumentRoot /var/www/{{ domain }}

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

     <Directory /var/www/{{ domain }}/>
        Options -Indexes +FollowSymLinks
        AllowOverride All
     </Directory>
</VirtualHost>
```

Comme expliqué en commentaire, le module template charge le fichier de template et pour chaque {{ variable }}, injecte la valeur correspondante depuis les variables définies au début. Vous rendez-vous compte de la puissance de la chose ?!

I-D-1 - Les variables

Nous avons rapidement pu voir les variables dans la partie précédente. Néanmoins, Ansible fournit de nombreuses variables sur l'environnement serveur. Le module setup nous renseigne sur ces dernières :

```
ansible buzut -m setup
buzut | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "37.59.21.45"
        ],
        "ansible_all_ipv6_addresses": [
            "2001:41d0:8:852d::",
            "fe80::225:90ff:fe7c:fa36"
         ],
         "ansible_architecture": "x86_64",
         "ansible_bios_date": "01/03/2014",
         "ansible_bios_version": "3.0a",
         […]
         "ansible_lsb": {
             "codename": "trusty",
             "description": "Ubuntu 14.04.4 LTS",
             "id": "Ubuntu",
             "major_release": "14",
             "release": "14.04"
         },
        […]
```

Toutes ces variables peuvent être utilisées dans vos playbooks. On y accède de la manière suivante :

```
# variable simple
{{ ansible_architecture }}

# pour accéder à une propriété
{{ ansible_lsb.major_release }}

# pour accéder à un tableau (première propriété)
{{ ansible_all_ipv4_addresses[0] }}
```

Vous commencez à percevoir la puissance des variables. Mieux, on peut alimenter ces dernières directement depuis une commande exécutée sur le serveur ! Voyons donc cela :

```
1. ---
2. - hosts: buzut
3.   tasks:
4.     - name: determine vhosts
5.       command: /bin/ls /etc/apache2/sites-enabled/
6.       register: vhosts
```

vhosts contient maintenant la liste des éléments présents dans /etc/apache2/sites-enabled/ telle que l'affiche la commande ls.

Il y a un type de variable un peu particulier qui peut s'avérer très utile : ce sont les variables *prompt*. Au moment de l'exécution, l'utilisateur qui lance le script se voit demander la valeur qu'il veut attribuer à la variable. Par exemple dans le cas d'un playbook permettant d'installer un nouveau serveur, inutile - peut-être même dangereux - de l'appliquer à all s'il n'y a qu'un seul nouveau serveur à paramétrer. Par ailleurs, c'est fastidieux d'éditer le playbook avant exécution. Il suffit donc de procéder comme ceci :

```
1.     ---
2.     - hosts: "{{ servernames }}"
3.       vars_prompt:
4.         - name: "servernames"
5.           prompt: "Which hosts would you like to setup?"
6.           private: no
7.       tasks:
8.         […]
```

Vous noterez que je précise ici private: no. En effet, par défaut, Ansible considère le prompt comme *password-sensitive* et n'affiche donc pas les caractères.

Enfin, sachez qu'il est également possible de passer des variables au playbook directement depuis la CLI au moment de l'invocation de ce dernier. Reprenons le même exemple que ci-dessus en enlevant le vars_prompt :

```
1.     ---
2.     - hosts: "{{ servernames }}"
3.       tasks:
4.         […]
```

Il suffit d'appeler le playbook de cette manière : ansible-playbook baseServer.yml --extra-vars "servernames=db6". Sachez également que les variables prennent des guillemets lorsqu'elles débutent une ligne. Petite illustration :

```
1.     # ici la variable doit être entourée de guillemets
2.     - hosts: "{{ servernames }}"
3.       tasks:
4.         […]
5.
6.     # mais pas dans ce cas là
7.     - name: Enable VirtualHost
8.       file:
9.         src: /etc/nginx/sites-available/{{ domain }}
10.         dest: /etc/nginx/sites-enabled/{{ domain }}
11.         state: link
```

I-D-1-a - Debug, le var_dump d'Ansible

Récupérer des variables c'est bien, savoir ce qu'il y a dedans, c'est mieux. C'est justement l'objet de debug. Reprenons l'exemple précédent :

```
1. ---
2. - hosts: buzut
3.   tasks:
4.     - name: determine vhosts
5.       command: /bin/ls /etc/apache2/sites-enabled/
6.       register: vhosts
```

```
7.      - debug: msg="{{ vhosts }}"
```

```
1. ---
2. # on lance le playbook
3. ansible-playbook test.yml
4.
5. PLAY ***************************************************************************
6.
7. TASK [setup] *******************************************************************
8. ok: [buzut]
9.
10. TASK [determine vhosts] ********************************************************
11. changed: [buzut]
12.
13. TASK [debug] *******************************************************************
14. ok: [buzut] => {
15.     "msg": {
16.         "changed": true,
17.         "cmd": [
18.             "/bin/ls",
19.             "/etc/apache2/sites-enabled"
20.         ],
21.         "delta": "0:00:00.014652",
22.         "end": "2016-03-24 23:21:32.612755",
23.         "rc": 0,
24.         "start": "2016-03-24 23:21:32.598103",
25.         "stderr": "",
26.         "stdout": "buzut.conf\default.conf",
27.         "stdout_lines": [
28.             "buzut.conf",
29.             "default.conf"
30.         ],
31.         "warnings": []
32.     }
33. }
34.
35. PLAY RECAP ***********************************************************************
36. cloud                      : ok=3    changed=1    unreachable=0    failed=0
```

Nous voyons ici que si nous voulons accéder à la valeur du premier vhost, il faut faire {{ vhosts.stdout_lines[0] }}, tout simplement.

Les variables présentent bien d'autres possibilités, on peut par exemple récupérer des informations sur un autre hôte. Je vous laisse découvrir toute cette magie directement **dans la doc**.

I-D-2 - Les boucles

Vous avez pu constater que les variables peuvent avoir des propriétés et contenir des tableaux. Qui dit tableau dit boucle. Nous allons reprendre notre exemple précédent et effacer tous les vhosts déjà présents avant d'ajouter le nôtre. C'est parti :

```
1. ---
2. - hosts: webserver
3.   vars:
4.     http_port: 80
5.     domain: buzut.fr
6.   remote_user: root
7.   tasks:
8.     […]
9.     - name: determine vhosts
10.       command: /bin/ls /etc/apache2/sites-enabled/
11.       register: vhosts
12.
13.     # on enlève le (ou les) vhosts par défaut
14.     - name: deregister default vhosts
15.       command: a2dissite {{ item }}
16.       with_items:
```

```
17.         - "{{ vhosts.stdout_lines }}"
18.
19.     # maintenant on peut rajouter notre vhost
20.     - name: enable vhost
21.       command: a2ensite buzut
22.     […]
```

Bien entendu, les boucles recèlent encore bien d'autres secrets, et comme à mon habitude, je vous laisse avec **la doc** si vous souhaitez creuser le sujet.

I-D-3 - Le notify pattern

Qu'est-ce donc que cela me direz-vous ? On peut dire que le *notify pattern* est la programmation événementielle Ansible *flavoured* en quelque sorte. OK, je m'explique. Plutôt que de lancer une action plusieurs fois , comme de redémarrer Apache - après l'activation d'un module ou la modification d'un fichier de config, les *handlers* ne sont lancés que si le fichier de conf change réellement. En outre, bien que plusieurs tâches puissent nécessiter une même action, l'action en question ne sera lancée qu'après l'exécution de tous les blocs tâches.

Illustrons ce comportement en reprenant notre exemple :

```
1. ---
2. - hosts: webserver
3.
4.   vars:
5.     http_port: 80
6.     domain: buzut.fr
7.   remote_user: root
8.
9.   tasks:
10.     […]
11.     - name: determine vhosts
12.       command: /bin/ls /etc/apache2/sites-enabled/
13.       register: vhosts
14.
15.     - name: deregister default vhosts
16.       command: a2dissite {{ item }}
17.       with_items:
18.         - "{{ vhosts.stdout_lines }}"
19.       notify:
20.         - restart apache2
21.
22.     - name: enabled mod_rewrite
23.       apache2_module:
24.         name: rewrite
25.         state: present
26.       notify:
27.         - restart apache2
28.
29.     - name: enable vhost
30.       command: a2ensite buzut
31.       notify:
32.         - restart apache2
33.     […]
34.
35.     handlers:
36.     - name: restart apache2
37.       service:
38.         name: apache2
39.         state: restarted
```

Ainsi, dès lors que nos anciens `vhosts` sont effacés, que le nouveau est installé et que le `mod_rewrite` est activé, Apache sera redémarré. Si nous avions référencé directement le module `apache2` avec l'instruction de redémarrer dans chacun des blocs de tâche, nous aurions redémarré Apache trois fois…

À ce stade, comme tout notre code est dans le même playbook, il est vrai que l'intérêt de notify s'avère plutôt limité. Dans un cas comme celui-ci, il nous suffit en effet de redémarrer Apache une fois en fin de fichier et le tour est joué. Cependant, lorsque nos playbooks sont plus complexes, on les sépare en plusieurs morceaux logiques, qui peuvent ainsi être réutilisables dans d'autres playbooks. Là, tout de suite vous percevez sans doute bien mieux l'intérêt de *notify*.

Puisque l'on parle de séparer nos playbooks en différentes parties, il est temps d'introduire les *includes* !

I-E - Meilleure organisation avec include

L'*include* dans Ansible, c'est exactement comme le include de PHP. Cela vous permet de scinder vos tâches en différents fichiers et de les importer au besoin dans vos playbooks. Imaginez par exemple que vous ayez une tâche qui se charge d'installer un dæmon de monitoring, vous voudrez certainement qu'elle s'exécute aussi bien sur vos serveurs de base de données, que sur vos serveurs front, etc.

Sans l'*include*, vous devriez répéter ça dans tous vos playbooks, tandis qu'avec cette petite magie, une simple référence suffit.

On définit donc nos tâches dans un fichier dédié, on l'appelle pour l'exemple setupMonitoring.yml.

```
1. ---
2. - name: Install monitoring agent
3.   apt:
4.     name: blabla
5.
6. # on ajoute toutes les tâches que l'on veut
```

Comme ce fichier sera intégré directement dans un playbook, nous n'avons pas à référencer tasks:, nous plaçons directement notre liste de tâches.

Une fois ce fichier créé et enregistré, admettons pour l'exemple que nous enregistrions toutes nos tâches dans tasks et que les playbooks soient à la racine, voici à quoi ressemblerait notre playbook :

```
1. ---
2. - hosts: dbservers
3.   vars:
4.     email: mon@email.fr
5.
6.   vars_prompt:
7.     - name: "dbrootpasswd"
8.       prompt: "Database root password"
9.
10.   tasks:
11.     - include: tasks/commonSetup.yml
12.     - include: tasks/setupMonitoring.yml
13.     - include: tasks/installMySQL.yml
14.
15.     # on peut bien entendu mélanger des includes et des tâches classiques
16.     - name: Install htop
17.       apt:
18.         name: htop
```

Il y a quelque chose que je ne vous ai pas dit concernant les *includes*. Nous avons vu comment inclure des tâches, mais l'on peut aussi inclure des playbooks. Tout dépend de l'endroit où l'*include* est utilisé.

Dans l'exemple précédent, nous l'avons inséré après tasks:, il est donc évident qu'il ne peut inclure que des tâches. Cependant, s'il est inséré au premier niveau du playbook, il insèrera un playbook. Il est ainsi possible de créer des meta-playbooks. Attention cependant, car la syntaxe de ces fichiers devra être celle d'un playbook !

C'est simple, mais redoutable de puissance ! Depuis sa version 1.2, Ansible a mis en place un mécanisme qui pousse encore plus loin cette logique afin de rendre les playbooks plus organisés, plus clairs et plus réutilisables, j'ai nommé : les rôles.

Les rôles constituent un moyen d'automatiser le chargement des variables, des tâches et des *handlers* grâce à une convention d'arborescence de fichiers. C'est une automatisation des includes qui permet une grande souplesse et une très bonne organisation des tâches complexes. J'y consacre un **article entier** !

I-F - Les objets de valeur au coffre

C'est une bonne pratique de versionner ses scripts Ansible. Cependant, qui dit versionning, dit souvent dépôt distant. Il va sans dire que certains éléments de configurations ne sont pas à laisser en clair sur n'importe quel dépôt : clef ssh, fichiers de conf avec mot de passe, etc.

D'une part, tous les collaborateurs n'ont pas forcement à y avoir accès, d'autre part, un Gitlab ou Github, même avec un dépôt privé et même s'il est installé sur vos serveurs, peut toujours être compromis.

Bien entendu, séparer les fichiers sensibles est envisageable. Mais vous n'aurez pas le confort d'avoir l'intégralité de vos éléments d'install ou deploy d'un simple petit coup de `git clone`, ou équivalent avec SNV ou Mercurial.

Pour répondre à cette problématique, Ansible propose un coffre - *Vault* dans la langue de Shakespeare. Vous spécifiez un mot de passe et Ansible chiffre le fichier (par défaut en AES). Lors de l'édition de fichiers, Ansible ouvrira votre l'éditeur défini dans la variable `$EDITOR`. Veillez à en définir un si ce n'est pas fait. Le cas échéant, votre fichier sera ouvert avec *vi*.

L'usage du vault est très simple. Vous n'aurez que quatre commandes à retenir :

```
1. # chiffrer des fichiers
2. ansible-vault encrypt fichierA [fichierB …]
3.
4. # afficher un fichier
5. ansible-vault view fichierA [fichierB …]
6.
7. # éditer un fichier déjà chiffré
8. ansible-vault edit fichierA
9.
10. # si jamais vous avez envie de déchiffrer un fichier précédemment chiffré
11. ansible-vault decrypt ficherA [fichierB …]
```

Lorsque vous désirez lancer un playbook qui nécessitera d'utiliser des fichiers présents dans le vault, il faudra passer l'option `--ask-vault-pass`.

Enfin, il est possible d'utiliser des mots de passe différents pour différents fichiers. Cependant, tous les fichiers utilisés au sein d'un même Playbook doivent partager le même mot de passe.

I-G - Performances

Négligeable quand vous n'avez que quelques serveurs à traiter, les réglages qui influent sur les performances peuvent avoir un impact important en termes de temps d'exécution si vous gérez un parc de serveurs important. Passons en revue les optimisations qui permettent de doper les performances d'Ansible. Tout va se passer dans le fichier de config `/etc/ansible/ansible.cfg` ou `/opt/local/etc/ansible/ansible.cfg.default` (qu'il faudra d'ailleurs renommer pour lui enlever `.default`) sur OS X.

I-G-1 - Forks

Nous en avons déjà rapidement parlé, Ansible gère par défaut les hôtes cinq par cinq. Ce qui veut dire qu'il attendra que l'exécution de vos instructions soit terminée sur vos cinq serveurs avant de poursuivre sur d'autres. Vous pouvez évidemment, c'est ce que nous avons vu, préciser autre chose avec l'option -f. Néanmoins, si vous souhaitez toujours vous adresser à plus de serveurs en parallèle, autant modifier ce paramètre dans le fichier de configuration et ne plus avoir à la spécifier manuellement à chaque fois. Nous sommes là pour automatiser nos tâches après tout ! Il n'y a pas de règle spécifique, les deux principaux facteurs limitants seront la charge CPU et la charge réseau engendrées. Le réglage de cinq par défaut est extrêmement conservateur, si votre machine est décemment récente et que vous avez autre chose qu'un modem 56K, vous pouvez tout à fait passer cette variable à 20 et ajuster après avoir testé.

```
forks=20
```

I-G-2 - Pipelining

Le *pipelining* permet de réduire le nombre de connexions ssh nécessaires. Par conséquent, la vitesse d'exécution des playbooks s'en trouvera grandement améliorée. Il est désactivé par défaut, car certaines configurations nécessitant requiretty ne sont pas compatibles, auquel cas il est possible d'utiliser le **mode accéléré**. Pour activer le pipelining :

```
pipelining=True
```

I-G-3 - Exécution asynchrone et polling

Ansible se connecte à vos serveurs en ssh et ne rend la connexion qu'une fois toutes les actions effectuées. Ainsi, le comportement par défaut est bloquant, et maintenir de nombreuses connections ssh ouvertes « pour rien » impacte négativement les performances. Il est cependant possible de lancer des opérations et de faire du *polling* afin de contrôler à intervalles réguliers l'état des processus ainsi lancés. De cette manière, vous pourrez lancer vos tâches sur plus de serveurs en parallèle.

```
1. # compilation du codec video x264, qui peut être assez longue
2. - name: compile x264
3.   environment: ffmpeg_env
4.   command: "{{ item }}"
5.   args:
6.     chdir: "{{ ffmpeg_source_dir }}/x264"
7.     creates: "{{ ffmpeg_bin_dir }}/x264"
8.   with_items:
9.     - ./configure --prefix={{ ffmpeg_build_dir }} --bindir={{
 ffmpeg_bin_dir }} --enable-static
10.     - make
11.     - make install
12.     - make distclean
13.   # on déclare le temps maximum d'exécution
14.   async: 120
15.   # et intervalle de temps auquel vérifier l'état de l'opération
16.   poll: 10
```

Il est également possible de spécifier poll à 0. Auquel cas on lance l'opération sans en vérifier le statut, en présupposant que le résultat sera celui auquel on s'attend.

Avant de conclure, puisque nous visons à automatiser au maximum la gestion de nos systèmes, j'ai écrit un petit script *pre-ansible*, **dispo sur Github**. Ce dernier permet de configurer l'ajout d'un nouveau serveur au système : ajout dans la config ssh, ajout au fichier hosts d'Ansible et paramétrage du ssh du serveur pour une connexion automatique par clef ssh.

En guise de conclusion, je vous laisse avec un exemple de playbook assez fourni. Il s'agit d'un article de Digital Ocean sur la **configuration d'un serveur Apache** avec Ansible.

Alors, est-ce qu'Ansible va révolutionner votre vie ? Quel est selon vous son plus gros atout ?

II - Tirer toute la puissance d'Ansible avec les rôles

Ansible est absolument génial. Il permet d'automatiser l'installation et la maintenance de machines et d'infrastructures complètes. J'ai déjà consacré un article à la **prise en main d'Ansible**, nous nous concentrerons ici sur les rôles.

Les rôles représentent une manière d'abstraire les directives *includes*. C'est en quelque sorte une couche d'abstraction. Grâce aux rôles, il n'est plus utile de préciser les divers *includes* dans le playbook, ni les paths des fichiers de variables, etc. Le playbook n'a qu'à lister les différents rôles à appliquer et le tour est joué !

En outre, depuis les *tasks* du rôle, l'ensemble des chemins sont relatifs. Inutile donc de préciser l'intégralité du path lors d'un `copy`, `template` ou d'une tâche. Le nom du fichier suffit, Ansible s'occupe du reste.

On peut faire beaucoup avec les playbooks et les *includes*, cependant, lorsque nous commençons à gérer des infrastructures complexes, on a beaucoup de tâches et les rôles s'avèrent salvateurs dans l'organisation et l'abstraction qu'ils apportent.

Par ailleurs, Ansible met à disposition une plate-forme permettant de télécharger et de partager des rôles utilisateurs : **Ansible galaxy**. Un bon moyen de ne pas réinventer la roue.

Pour illustrer cet article, nous allons préparer différents rôles et les assembler dans un playbook afin d'installer un simple serveur LAMP (testé sur Ubuntu 16.04 uniquement, mais l'idée est là). Pour faciliter le suivi, j'ai mis l'ensemble sur **un repo Github**. C'est parti !

En CLI, on peut utiliser ansible-galaxy afin de préparer l'arborescence d'un rôle vide. Nos playbooks seront à la racine de notre dossier tandis que les rôles seront dans `roles`. Ainsi, lorsque nous appellerons un rôle depuis un playbook, sans avoir besoin de préciser autre chose que son nom, Ansible saura où chercher.

```
1. # on initialise un role vide depuis le dossier roles
2. ansible-galaxy init common
3.
4. cd common && ls -R
5. ls -R
6. README.md    handlers    tasks        vars
7. defaults    meta        tests
8.
9. ./defaults:
10. main.yml
11.
12. ./handlers:
13. main.yml
14.
15. ./meta:
16. main.yml
17.
18. ./tasks:
19. main.yml
20.
21. ./tests:
22. inventory    test.yml
23.
24. ./vars:
25. main.yml
```

On voit ici qu'Ansible nous a créé plusieurs dossiers avec un fichier `main.yml` dans chacun d'eux. Par défaut, Ansible ira chercher les informations dans les `main.yml` de chaque répertoire. Cependant, vous pouvez créer d'autres fichiers et les référencer dans vos instructions. Par exemple, il est tout à fait possible de créer une autre tâche dans `tasks` et de l'inclure depuis `tasks/main.yml`. Passons rapidement en revue la fonction de chaque répertoire.

La commande ansible-galaxy est normalement destinée à être utilisée avec **Galaxy**, nous en tirons cependant profit pour facilement obtenir un boilerplate de rôle vide. Par défaut, elle ne créée pas les répertoires files et templates dont nous allons tout de même parler.

defaults

Ce sont ici les variables par défaut qui seront à disposition du rôle.

vars

De la même manière que defaults, il s'agit ici de variables qui seront à disposition du rôle, cependant, celles-ci ont en général vocation à être modifiées par l'utilisateur et elles prennent le dessus sur celles de defaults si elles sont renseignées.

tasks

Sans grande surprise, c'est ici que vous référencerez vos tâches.

files

Tous les fichiers étant destinés à être traités par le module copy seront placés ici.

templates

Idem que copy, mais cela concerne les fichiers du module template.

meta

Il y a ici plusieurs usages, notamment dans le cas de rôles publiés sur Galaxy. Dans notre cas, on référencera ici les dépendances à d'autres rôles.

tests

Pas utile pour nous, c'est seulement pour Galaxy et la doc n'est pas très loquace à ce propos…

En plus des répertoires susmentionnés, il y a le README qu'il est de bon ton de renseigner afin d'expliquer comment utiliser le rôle, quelles sont les variables à définir, etc.

Aucun des répertoires n'est impératif - quoique sans tasks, notre rôle ne sert pas à grand-chose. On effacera donc les répertoires dont on n'a pas l'utilité.

On est parti pour la création des différents rôles utiles à notre serveur LAMP. On commence avec le rôle common (créé précédemment), qui sert de base à l'ensemble de nos serveurs. Cette tâche est assez fournie, ça permet d'avoir un exemple de diverses choses qu'on peut faire, vous pouvez néanmoins la lire en diagonale, le but n'est pas de configurer un serveur, c'est de découvrir les rôles Ansible !

```
1. # tasks/main.yml
2. ---
3. # maj du système qui est fraichement installé
4. - name: Update & upgrade system
5.   apt:
6.     update_cache: yes
7.     upgrade: dist
8.
9. # souvent livré avec un mdp root envoyé par mail
10. # on en génère un aléatoire avant de le remplacer
11. # on va configurer ssh pour n'accepter que les clefs
```

```
12. - name: Generate password
13.   command: openssl rand -base64 32
14.   register: randomPass
15.
16. - name: Change root passwd
17.   shell: echo "root:{{ randomPass }}" | chpasswd
18.
19. # diverses choses pouvant servir sur tous les serveurs
20. - name: Install base soft (transport-https, fail2ban)
21.   apt:
22.     name: "{{ item }}"
23.   with_items:
24.     - apt-transport-https
25.     - software-properties-common
26.     - fail2ban
27.     - exim4
28.     - htop
29.     - glances
30.     - iotop
31.     - unattended-upgrades
32.
33.
34. # on configure les maj auto de secu
35. # on pourrait ici très bien utiliser un fichier de template au lieu de remplacer à coup de
 regexp
36. - name: Configure unattended-upgrades
37.   replace:
38.     dest: /etc/apt/apt.conf.d/50unattended-upgrades
39.     regexp: "{{ item.regexp }}"
40.     replace: "{{ item.replace }}"
41.   with_items:
42.     - { regexp: '//Unattended-Upgrade::Mail "root";', replace: 'Unattended-Upgrade::Mail
 "{{ email }}";' }
43.     - { regexp: '//Unattended-Upgrade::MailOnlyOnError "true";', replace: 'Unattended-
Upgrade::MailOnlyOnError "true";' }
44.     - { regexp: '//Unattended-Upgrade::Remove-Unused-Dependencies
 "false";', replace: 'Unattended-Upgrade::Remove-Unused-Dependencies "true";' }
45.
46. # ici, on upload directement le fichier de config qui va bien
47. - name: Enable unattended-upgrades
48.   copy:
49.     src: 20auto-upgrades
50.     dest: /etc/apt/apt.conf.d/20auto-upgrades
51.
52. # on remplace tout bonnement la config ssh par défaut
53. - name: Upload ssh server config file
54.   copy:
55.     src: sshServerConfig
56.     dest: /etc/ssh/sshd_config
57.     mode: 0644
58.
59. # on ajoute un script d'init iptables
60. - name: Install iptables.sh
61.   copy:
62.     src: iptables.sh
63.     dest: /usr/local/sbin/iptables.sh
64.     mode: 0744
65.
66. # qui se lancera au boot
67. - name: Make iptable start on boot
68.   copy:
69.     src: rc.local
70.     dest: /etc/rc.local
71.     mode: 0755
72.
73. # fail2ban c'est bien aussi, mangez-en
74. - name: Upload fail2ban config file
75.   copy:
76.     src: fail2ban.conf
77.     dest: /etc/fail2ban/jail.d/defaults-debian.conf
78.     mode: 0644
79.
```

```
80. # les alias pour récupérer les emails
81. - name: Configure /etc/aliases
82.   lineinfile:
83.     dest: /etc/aliases
84.     regexp: ^root:.*
85.     line: "root: {{ email }}"
86.
87. # on configure exim4
88. - name: Configure exim4
89.   replace:
90.     dest: /etc/exim4/update-exim4.conf.conf
91.     regexp: "dc_eximconfig_configtype='local'"
92.     replace: "dc_eximconfig_configtype='internet'"
```

On voit déjà que là, on fait pas mal de choses (j'ai bien allégé par rapport à ce que j'utilise réellement) qui seront communes à absolument **tous** nos serveurs. Un rôle qui sera donc utilisé absolument partout.

Vous notez certainement qu'il y a dans cette tâche des variables et des fichiers importés. Il faut ajouter tout cela, sinon Ansible balancera une erreur.

Concernant les variables, il s'agit simplement de l'email, pas de defaults ici, car il faut que l'email soit renseigné, il ne peut pas être « par défaut ». Nous n'avons donc pas l'utilité de defaults et nous mettons cette variable directement dans vars/main.yml :

```
---
email: mon@email.fr
```

Point de vars:, Ansible sait qu'il s'agit de variable, car c'est dans le dossier vars, *what else?*

Il y aurait pas mal de choses à dire sur cette conf, concernant ssh, on ne pourra se connecter qu'avec une clef ed25519. Pour en savoir plus sur la config ssh, je vous invite à lire **mon article dédié**, idem pour **fail2ban**. Quant à iptables, il ferme tout sauf le web et ssh et quelques services de base (DNS, DHCP, NTP…).

files contient naturellement l'ensemble des fichiers que nous avons besoin d'uploader. Vous trouverez leur contenu sur le git de l'article.

```
# voici quand même la liste
ls files/
20auto-upgrades    fail2ban.conf    iptables.sh    rc.local    sshServerConfig
```

Nous n'allons pas détailler tous les rôles un à un, car vous les retrouverez sur le git, nous allons créer un rôle nommé lampserver, qui se chargera d'installer PHP, Apache2, MySQL et letsencrypt. Comme vous vous en doutez, chacune des différentes briques constitue un rôle en elle-même. Nous pourrions vouloir installer letsencrypt avec Nginx en reverse d'un Node.js. Il serait un peu bête de devoir réécrire des tâches alors que nous avons déjà fait le boulot… Vous voyez la logique ?

On commence donc par créer notre meta/main.yml dans lequel on référence nos dépendances :

```
1. dependencies:
2.   - { role: letsencrypt }
3.   - { role: mariadb }
4.   - { role: apache }
5.
6. # notez qu'il est possible de passer des paramètres directement aux rôles
7.   - { role: letsencrypt, withCerts: true }
8.
9. # très intéressant également, il est possible de lister des rôles depuis git
10. # le tag est optionnel, mais représente une option à connaître également
11. - { role: 'git+https://gitlab.com/user/rolename,v1.0.0'}
```

Eh oui, tout simplement. On précise ici que ce rôle dépend des rôles letsencrypt, mariadb et apache qui devront donc être exécutés avant.

Ces rôles sont bien entendu présents dans roles au même niveau que les autres (sauf s'ils sont référencés via une URL).

Notre tâche est ensuite tout ce qu'il y a de plus classique :

```
# tasks/main.yml
---
# à ce stade, apache est déjà installé, car ce rôle précise ses dépendances
- name: Install php and common php ext
  apt:
    name: "{{ item }}"
  with_items:
    - libapache2-mod-php
    - php
    - php-curl
    - php-gd
    - php-json
    - php-mbstring
    - php-mysql
    - php-xml

- name: Enable required apache modules
  apache2_module:
    name: "{{ item }}"
    state: present
  with_items:
    - expires
    - headers
    - http2
    - rewrite
    - ssl
```

Enfin, il ne nous reste plus qu'à créer notre playbook et à donner les instructions pour l'exécution des rôles.

```
---
- hosts: webserver
  roles:
     - common
     - lampserver

     # il est possible de passer des variables aux rôles ici
     - { role: foo, myvar: 'blabla' }

  # ou d'en définir au niveau global
  # elles seront prises en compte dans les rôles
  # vars:
    # blurp: test

  # ici, on demande directement à l'exécution de renseigner une variable
  vars_prompt:
    - name: "mysqlRootPass"
      prompt: "password for MySQL root"
```

Attention cependant à la **priorité des variables**. Dans le cas d'une variable référencée à la fois dans le playbook et dans vars du rôle avec deux valeurs différentes, c'est vars qui l'emporte. Néanmoins, la référence dans le playbook l'emporte sur defaults. En outre, il peut être très pratique de référencer une variable dans le playbook. Par exemple, l'email sera certainement requis dans de nombreux rôles, autant ne le déclarer qu'une seule fois.

Nous avons à peu près fait le tour des rôles. J'espère que vous y voyez un peu plus clair, et surtout, que vous êtes convaincu d'en user et d'en abuser !

III - Note de la rédaction de Developpez.com

Nous tenons à remercier **Quentin Busuttil** qui nous a aimablement autorisés à publier ses tutoriels : **Automatiser la gestion des serveurs avec Ansible** et **Tirer toute la puissance d'Ansible avec les rôles**. Nous remercions également **Winjerome** pour la mise au gabarit et **Claude Leloup** pour la relecture orthographique.