CRÉEZ VOS PROPRES MODULES ANSIBLE

Introduction

Chaque jour, le nombre de modules Ansible continue de croître avec un support supplémentaire ajouté à mesure que l'utilisation d'Ansible continue. On peut donc se poser la question suivante ? "**Pourquoi aurions-nous besoin de créer nos propres modules Ansible** ?" . Eh bien, il y a un certain nombre de raisons, mais je pense que la plus valable reste de personnaliser un module qui n'existe pas encore dans la bibliothèque Ansible, mais aussi de mieux comprendre le fonctionnement des modules Ansible.

Prérequis

Tout ce dont vous avez besoin est un peu de connaissance de Python, combinez-le avec vos compétences Ansible et vous pourrez commencer à **créer vos propres modules Ansible**. Dans ce chapitre tout sera expliqué au fur et à mesure que nous le parcourons ensemble.

Création d'un module personnalisé

Hello world

Vous allez vous rendre compte qu'il est très facile de créer des modules Ansible. Notre but pour le moment est d'**afficher le fameux message "Hello world"**.

Commencez déjà par créer un fichier `test.py` sous un dossier `library` et rajoutez-y le contenu suivant :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *

def main():

 module = AnsibleModule(argument_spec={})
 response = {"result" : "hello world"}
 module.exit_json(changed=False, meta=response)


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

Voici ci-dessous une liste d'explication du code ci-dessus :

- `#!/usr/bin/python` : l'interpréteur python qui sera utilisé par Ansible.
- `# -*- coding: utf-8 -*-` : l'encodage qui sera utilisé par Ansible.
- `from ansible.module_utils.basic import *` : importation de la librairie permettant de créer des modules Ansible.
- `main()` : le point d'entrée dans votre module.
- `AnsibleModule()` : c'est la classe qui nous permet de créer et manipuler notre module Ansible, comme par exemple la gestion des paramètres de notre module.
- `response = {"result" : "hello world"}` : ce sont les métadonnées de notre module sous forme d'un dictionnaire.
- `module.exit_json()` : cette partie désigne la fin d'exécution de notre module, elle permet l'affichage des métadonnées et l'état de votre module.

Sur la racine de votre projet créez votre playbook et appelez-y votre module comme suit :

```
- hosts: localhost
  tasks:
    - name: Test de notre module
      test:
      register: result

    - debug: var=result
```

Si vous avez suivi à la lettre mes instructions, vous devriez avoir l'arborescence suivante :

```
|___ library
|
|___ test.py

|___ main.yml
```

Exécutez ensuite votre playbook et vous obtiendrez le résultat suivant :

```
ansible-playbook main.yml
```

Résultat :

```
TASK [Test de notre module] *****************
ok: [localhost]

TASK [debug] ******************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": {
            "result": "hello world"
        }
    }
}
```

Les paramètres

Pour que notre module soit plus utile, nous aurons besoin de **rajouter quelques entrées**. Dans notre exemple nous allons imaginer un module qui permet de décrire

un personnage de jeux vidéo avec les spécifications et les exigences suivantes :

Nom	Type	Valeur par défaut	Obligatoire
name	string	unknown	Oui
description	string	empty	Non
attack	string avec choix limité	melee	Oui
inventory	list	[] (liste vide)	Non
monster	bool	False	Non

Transformons ce tableau sous forme de module Ansible, ce qui nous donnera le code suivant :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *

def main():


 fields = {
  "name": {"default" : "unknown", "type": "str"},
  "description": {"default": "empty", "required": False, "type": "str"},
        "attack": {
         "default": "melee",
         "choices": ['melee', 'distance'],
         "type": 'str'
        },
  "inventory": {"default": [], "required": False, "type": "list"},
  "monster": {"default": False, "required": False, "type": "bool"},
 }
 module = AnsibleModule(argument_spec=fields)
 module.exit_json(changed=False, meta=module.params)


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

Vous remarquerez les options suivantes :

- `default` : la valeur par défaut de votre paramètre dans le cas où l'utilisateur ne spécifie rien.

- `choices` : la liste des valeurs possibles à proposer à l'utilisateur final.
- `type` : le type de votre paramètre (str, bool, int dict, list, etc ...).
- `required` : un booléen afin de savoir si le paramètre est obligatoire ou non (True ou False).

> **Information**
>
> Les paramètres sont par défaut obligatoires.

Faisons appel à notre beau module depuis notre playbook, comme suit :

```
- hosts: localhost
  tasks:
    - name: Test de notre module
      test:
        name: "MageDarkX"
        attack: distance
        inventory:
          - powder
          - stick
          - potion
      register: result

    - debug: var=result
```

Puis exécutons notre playbook :

```
ansible-playbook main.yml
```

Résultat :

```
TASK [debug] ***************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": {
            "attack": "distance",
            "description": "empty",
```

```
            "inventory": [
                "powder",
                "stick",
                "potion"
            ],
            "monster": false,
            "name": "MageDarkX"
        }
    }
}
```

Vous pouvez aussi **récupérer chaque entrée dans une variable unique**. Pour ce faire, créons une fonction nommée `presentation()` afin de mieux présenter notre personnage selon les entrées récupérées de notre utilisateur :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *

def presentation(module):
 name = module.params['name']
 attack = module.params['attack']
 inventory = module.params['inventory']
 return {"Presentation" : "My name is {} and my type of attack is {}, here is what yo

def main():
 fields = {
  "name": {"default" : "unknown", "type": "str"},
        "attack": {
         "default": "melee",
         "choices": ['melee', 'distance'],
         "type": 'str'
        },
  "inventory": {"default": [], "required": False, "type": "list"},
 }
 module = AnsibleModule(argument_spec=fields)
 module.exit_json(changed=False, meta=presentation(module))


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

Résultat :

```
TASK [debug] ***************
ok: [localhost] => {
    "result": {
```

```
        "changed": false,
        "failed": false,
        "meta": {
            "Presentation": "My name is MageDarkX and my type of attack is distance,
            here is what you will find in my inventory : ['powder', 'stick', 'potion
        }
    }
}
```

Attributs et méthodes proposées par la librairie Ansible

Si jamais vous souhaitez **afficher tous les attributs et méthodes de la class** `AnsibleModule`, vous pouvez utiliser la fonction native de python nommé `def()`, comme suit :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *

def main():
 module = AnsibleModule(argument_spec={})
 module.exit_json(changed=False, meta=dir(module))

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

Pour tester plus rapidement notre module, utilisons l'option `-M` de la commande `ansible` qui permet de préciser le chemin de la librairie d'un module :

```
ansible all -M library -m test
```

Résultat :

```
TASK [debug] *******************************************************************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": [
            "fail_json"
            "__setattr__",
```

```
            "__sizeof__",
            ...
            ...
            "do_cleanup_files",
            ...
            "warn"
        ]
    }
}
```

Dans la prochaine partie, nous nous intéresserons à deux méthodes bien utiles, à savoir la méthode `fail_json()` et `warn()`.

Afficher un message d'avertissement et déclencher une erreur

La fonction pour **afficher un message d'avertissement** est la méthode `warn()`, elle reste très simple à utiliser et permet par exemple d'indiquer un message de recommandation sans interrompre l'exécution du module, ci-dessous un exemple d'utilisation :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *

def verifyAge(module):
 age = module.params['age']
 if age
```

Exécutons notre script avec un âge inférieur à 18 :

```
ansible all -M library -m test -a "age=17"
```

Résultat :

```
[WARNING]: Attention vous êtes mineur, un accord parental est requis.

localhost | SUCCESS => {
    "changed": false,
```

```
    "meta": 17
}
```

Si vous souhaitez à la place **déclencher une erreur**, vous utiliserez alors la méthode `fail_json()` à la place de la méthode `warn()`, voici un exemple d'utilisation :

```
# ...

def verifyAge(module):
 age = module.params['age']
 if age
```

Exécutons une nouvelle fois notre ancienne commande :

```
ansible all -M library -m test -a "age=17"
```

Résultat :

```
localhost | FAILED! => {
    "changed": false,
    "msg": "Attention vous êtes mineur, un accord parental est requis."
}
```

Exercice

But

Les notions vues précédemment répondront à la plupart de vos besoins. Vous êtes donc désormais capable de rédiger vos propres modules, et si vous le permettez, je vais vous demander de vous exercer en créant un module qui permet de vérifier si un dossier est capable de stocker des données d'une taille entrée par utilisateur.

L'utilisateur final qui utilisera votre playbook devra spécifier deux paramètres :

- `path` : chemin du dossier qui sera analysé (exemple: *'/home/hatim/'*)

- `size` : la taille de comparaison (exemple: *'200g'* pour 200 Gigaoctets, *'200m'* pour 200 Mégaoctets, etc ..)

Un exemple sera plus parlant, voici par exemple à quoi doit ressembler notre playbook :

```
- hosts: localhost
  tasks:
    - name: 'check folder size of /home/hatim'
      folder_space:
        path: '/home/hatim'
        size: '20g'
```

Si nous ne pouvons pas stocker 20Go de données dans le dossier `/home/hatim`, nous devrions avoir un résultat similaire à celui-ci :

```
TASK [check folder size of /home/hatim] *********
fatal: [localhost]: FAILED! => {"changed": false, "msg": "Not enough space available
```

Dans le cas où le dossier `/home/hatim` possède les 20Go d'espace de stockage nécessaires, alors nous obtiendrons un résultat similaire à celui-ci :

```
TASK [check folder size of /home/hatim] *********
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": {
            "result": "You have enough space :)"
        }
    }
}
```

Aide (sans solution)

Voici quelques fonctions qui vous permettront d'accomplir certaines tâches. La première astuce est une fonction qui vous permettra de récupérer la taille en octet que peut stocker votre dossier :

```
import os

    def getAvailableSpace(path):
        try:
            statvfs = os.statvfs(path)
            return int(statvfs.f_bavail * statvfs.f_frsize)
        except OSError as e:
            print(e)


    path='/home/hatim'
    print("byte of {} : {}b".format(path, getAvailableSpace(path)))
```

Maintenant je vais vous montrer comment convertir la taille d'un octet (byte) en Mégaoctets, Gigaoctets, etc .. et inversement :

```
size_conversion = { "k": 1024, "m": pow(1024,2), "g": pow(1024,3), "t": pow(1024,4),

def convertToByte(space_unit, space):
    global size_conversion
    return int(space * size_conversion[space_unit])


def convertByteToOriginal(space_unit, space):
    global size_conversion
    return round(space / float(size_conversion[space_unit]), 2)

mb_to_byte = convertToByte('m', 1000)
gb_to_byte = convertToByte('g', 1)
print("1000 Mb to byte : {}".format(mb_to_byte))
print("1 Gb to byte : {}".format(gb_to_byte))

b_to_gb = convertByteToOriginal('g', 10000000)
print("10000000 byte to Gb : {}".format(b_to_gb))
```

Voilà, maintenant c'est à votre tour !

Solution

Je vais vous montrer ma solution, sachez juste qu'il existe différentes façons de faire ce type de module, donc n'hésitez pas à le modifier selon votre guise et de partager votre code dans l'espace commentaire ;).

Voici donc à quoi ressemble le code de mon module :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from ansible.module_utils.basic import *
import os

size_conversion = { "k": 1024, "m": pow(1024,2), "g": pow(1024,3), "t": pow(1024,4),


def checkSizeAndUnit(module, space_unit, space):
    if space_unit not in 'kmgtp':
        module.fail_json(msg="Bad size specification for unit {}".format(space_unit)
    if not space.isdigit():
        module.fail_json(msg="Bad value for {}, must be an integer".format(space))


def getSizeAndUnit(module, size):
    space_unit, space = size[-1].lower(), size[0:-1]
    checkSizeAndUnit(module, space_unit, space)
    return space_unit, int(space)


def convertToByte(space_unit, space):
    global size_conversion
    return int(space * size_conversion[space_unit])


def convertByteToOriginal(space_unit, space):
    global size_conversion
    return round(space / float(size_conversion[space_unit]), 2)


def getWantedSpace(module, size):
    space_unit, space_wanted = getSizeAndUnit(module, size)
    return convertToByte(space_unit, space_wanted)


def getAvailableSpace(module, path):
    try:
        statvfs = os.statvfs(path)
        return int(statvfs.f_bavail * statvfs.f_frsize)
    except OSError as e:
        module.fail_json(msg="{}".format(e))


def CheckSizeAvailability(module, path, size):
    space_available = getAvailableSpace(module, path)
    space_wanted = getWantedSpace(module, size)
    space_unit, _ = getSizeAndUnit(module, size)
    space_available_converted = convertByteToOriginal(space_unit, space_available)

    if space_available
```

Conclusion

Vous pouvez dorénavant personnaliser vos modules pour qu'ils répondent exactement à vos besoins. Tentez tout de même de privilégier les modules préconçus par Ansible avant de songer à créer vos propres modules.

Je tiens à créditer cet article qui m'a aidé à la rédaction de cet exercice. J'en ai d'ailleurs profité pour améliorer la structure de son code comme vous pouvez le remarquer :).