Formats JSON et YAML Ansible

Objectifs de certification

RHCE EX294 (RHEL8)

Si vous poursuivez des objectifs de certification, vous pourriez trouvez ici des bases qui vous aideront pour :

- **2. Maîtrise des composants de base d'Ansible**
  - 2.3. Variables
  - 2.4. Facts
- **6. Création des jeux et des playbooks Ansible**
- **9. Utilisation des fonctions Ansible avancées**
  - 9.1. Créer et utiliser des modèles pour générer des fichiers de configuration personnalisés
  - 9.3. Utilisation de variables et de facts Ansible

1. Formats de présentation des données

On présentera brièvement dans cette section trois formats de présentation de données :

- **XML**, le standard
- **JSON**, le standard en version lisible
- **YAML**, pour présenter des données utilisateur

Le format JSON sera plus amplement étudié dans une section suivante.

1.1. XML

L'*Extensible Markup Language*, généralement appelé XML est un métalangage informatique de balisage générique qui est un sous-ensemble du Standard Generalized Markup Language (SGML). Sa syntaxe est dite “extensible” car elle permet de définir différents langages avec chacun leur vocabulaire et leur grammaire, comme XHTML, XSLT, RSS, SVG… Elle est reconnaissable par son usage des chevrons (`<`, `>`) encadrant les noms des balises. L'objectif initial de XML est de faciliter l'échange automatisé de contenus complexes (arbres, texte enrichi, etc.) entre systèmes d'informations hétérogènes (interopérabilité). Avec ses outils et langages associés, une application XML respecte généralement certains principes :

- la structure d'un document XML est définie et validable par un schéma ;
- un document XML est entièrement transformable dans un autre document XML.

Exemple :

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()"/>
    <menuitem value="Open" onclick="OpenDoc()"/>
    <menuitem value="Close" onclick="CloseDoc()"/>
  </popup>
</menu>
```

1.2. JSON

JavaScript Object Notation (JSON) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.

```
{
   "menu": {
      "id": "file",
      "value": "File",
      "popup": {
         "menuitem": [
            { "value": "New", "onclick": "CreateNewDoc()" },
            { "value": "Open", "onclick": "OpenDoc()" },
            { "value": "Close", "onclick": "CloseDoc()" }
```

```
        ]
      }
    }
}
```

Remarque : JSON ne supporte pas les commentaires au contraire de YAML ou de XML.

1.3. YAML

YAML, acronyme de “Yet Another Markup Language” dans sa version 1.0, il devient l’acronyme récursif de “YAML Ain’t Markup Language” (“YAML n’est pas un langage de balisage”) dans sa version 1.1, est un format de représentation de données par sérialisation Unicode. Il reprend des concepts d’autres langages comme XML.

YAML est facilement à lire et à encoder pour un humain.

```
menu:
    id: file
    value: File
    popup:
        menuitem:
          - value: New
            onclick: CreateNewDoc()
          - value: Open
            onclick: OpenDoc()
          - value: Close
            onclick: CloseDoc()
```

1.4. Valider, convertir et travailler avec les formats de données

Code Beautify

2. JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) est donc un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée. On en trouvera une description dans le RFC 8259.

JSON se base sur deux structures:

1. **Une collection de couples nom/valeur**. Divers langages la réifient par un "objet", un "enregistrement", une "structure", un "dictionnaire", une "table de hachage", une "liste typée" ou un "tableau associatif".
2. **Une liste de valeurs ordonnées**. La plupart des langages la réifient par un "tableau", un "vecteur", une "liste" ou une "suite".

2.1. Types de données JSON

Les types de données de base de JSON sont les suivantes :

1. **Objet** : une collection non ordonnée de paires nom-valeur dont les noms (aussi appelés clés) sont des chaînes de caractères.[1] Les objets sont délimités par des accolades ("`{`" et "`}`") et séparés par des virgules, tandis que dans chaque paire, le caractère deux-points "`:`" sépare la clé ou le nom de sa valeur. La valeur peut être un tableau, un nombre, une chaîne de caractère, ou une valeur boléenne, ou nulle.
2. **Tableau** (*array*) : une liste ordonnée de zéro ou plus de valeurs, dont chacune peut être de n'importe quel type. Les tableaux utilisent la notation par crochets ("`[`" et "`]`") et les éléments sont séparés par des virgules ("`,`").
3. **Nombre** : est déclaré sans protection des guillemets.
4. **Chaîne de caractères** (*string*) : une séquence de zéro ou plus de caractères Unicode. Les chaînes de caractères sont délimitées par des guillemets ("`"`" et "`"`") et supportent une barre oblique inversée comme caractère d'échappement ("`\`").
5. **Booléen** : valeur binaire, l'une ou l'autre des valeurs "`true`" ou "`false`", sans guillemets.
6. **Nulle** : Une valeur vide, en utilisant le mot "`null`", sans guillemets.

Les espaces limités sont autorisés et ignorés autour ou entre les éléments syntaxiques (valeurs et ponctuation, mais pas à l'intérieur d'une valeur de chaîne). Seuls quatre caractères spécifiques sont considérés comme des espaces : espace, tabulation horizontale, saut de ligne et retour chariot.

JSON ne fournit pas de syntaxe pour les commentaires.

2.2. Exemple de format JSON

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Ces données sont constituées de 6 clés :

- “address”,
- “age”,
- “firstName”,
- “gender”,
- “lastName”,
- et “phoneNumber”

L’objet “address” est consitué de 4 clés.

- “city”,
- “postalCode”,

- “state”,
- et “streetAddress”

L’objet “phoneNumber” est constitué d’un tableau à deux entrées contenant chacune deux clés.

```
{
  "type": "home",
  "number": "212 555-1234"
},
{
  "type": "fax",
  "number": "646 555-4567"
}
```

2.3. Traitement avec jq

Le logiciel `jq` est outil de traitement du texte à la manière de `sed` mais pour traiter des sorties JSON.

```
ansible localhost -b -m package -a "name=jq state=present"
```

Avant tout il faut des données à traiter, par exemple l’exemple qui précède :

```
cat << EOF > data
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
```

```
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
EOF
```

Les données à traiter viennent en entrée de la commande :

```
cat data | jq
```

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
```

```
}
```

Le filtre se place entre trémas pour incorporer des filtres divers :

```
cat data | jq '.'
```

Par exemple le filtre keys affiche les clés d’un objet :

```
cat data | jq '. | keys'
```

```
[
  "address",
  "age",
  "firstName",
  "gender",
  "lastName",
  "phoneNumber"
]
```

Par exemple le filtre length compte les objets :

```
cat data | jq '. | length'
```

```
6
```

On peut afficher la valeur d’une clé :

```
cat data | jq '.firstName'
```

```
"John"
```

```
cat data | jq '.address'
```

```
{
  "streetAddress": "21 2nd Street",
  "city": "New York",
  "state": "NY",
  "postalCode": "10021"
}
```

```
cat data | jq '.address.city'
```

```
"New York"
```

Cela fonctionne tant que l’on a à faire avec des objets. Mais cela ne va plus avec des dictionnaires :

```
cat data | jq '.phoneNumber'
```

```
[
  {
    "type": "home",
    "number": "212 555-1234"
  },
  {
    "type": "fax",
    "number": "646 555-4567"
  }
]
```

```
cat data | jq '.phoneNumber.type'
```

```
jq: error (at <stdin>:24): Cannot index array with string "type"
```

Reprenons. L’objet “phoneNumber” est consitué d’un dictionnaire de deux objets :

```
cat data | jq '.phoneNumber'
```

```
[
  {
    "type": "home",
    "number": "212 555-1234"
  },
  {
    "type": "fax",
    "number": "646 555-4567"
  }
]
```

Affichons les objets séparément :

```
cat data | jq '.phoneNumber[]'
```

```
{
  "type": "home",
  "number": "212 555-1234"
}
{
  "type": "fax",
  "number": "646 555-4567"
}
```

Filtrons par la clé “type” :

```
cat data | jq '.phoneNumber[] | .type'
```

```
"home"
"fax"
```

Filtrons sur le second objet :

```
cat data | jq '.phoneNumber[1]'
```

```
{
  "type": "fax",
  "number": "646 555-4567"
}
```

Quelle est la valeur de la seconde clé du champ “number” ?

```
cat data | jq '.phoneNumber[1].number'
```

```
"646 555-4567"
```

2.4. Module setup

Le module setup exécuté en mode ad-hoc permet de récupérer des méta-données sur les hôtes en format JSON.

```
ansible localhost -m setup
```

Ce traitement traite la sortie standard en seule ligne moins la chaîne de caractère `| SUCCESS =>`

```
ansible localhost -m setup -o | sed 's/.*> {/{/g'
```

Pour traitement ultérieur, on valorise une variable `$FACTS` :

```
FACTS=$(ansible localhost -m setup -o | sed 's/.*> {/{/g')
```

Voyez vous-même :

```
echo $FACTS
```

2.5. Sorties JSON

Formatage JSON

Début des donées :

```
echo $FACTS | jq . | head -20
```


```
{
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "10.6.19.161",
      "172.17.0.1",
      "192.168.122.1"
    ],
    "ansible_all_ipv6_addresses": [
      "fe80::207:cbff:fe0b:1b57"
    ],
    "ansible_apparmor": {
      "status": "enabled"
    },
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "01/17/2018",
    "ansible_bios_version": "00.00.00.0012",
```

```
    "ansible_cmdline": {
      "boot": "local",
      "console": "ttyS1,9600n8",
      "nbd.max_part": "16",
```

Fin des données :

```
echo $FACTS | jq . | tail -20
```

```
      "hw_timestamp_filters": [],
      "macaddress": "52:54:00:c9:73:7e",
      "mtu": 1500,
      "promisc": true,
      "timestamping": [
        "tx_software",
        "rx_software",
        "software"
      ],
      "type": "ether"
    },
    "ansible_virtualization_role": "host",
    "ansible_virtualization_type": "kvm",
    "gather_subset": [
      "all"
    ],
    "module_setup": true
  },
  "changed": false
}
```

Quel est le nombre d’objets ?

```
echo $FACTS | jq '. | length'
2
```

Quels sont ces deux objets ?

```
echo $FACTS | jq '. | keys'
```

```
[
  "ansible_facts",
  "changed"
]
```

Quels sont les objets imbriqués dans “ansible_facts” ?

```
echo $FACTS | jq '.ansible_facts | keys'
```

```
[
  "ansible_all_ipv4_addresses",
  "ansible_all_ipv6_addresses",
  "ansible_apparmor",
  "ansible_architecture",
  "ansible_bios_date",
  "ansible_bios_version",
  "ansible_cmdline",
  "ansible_date_time",
  "ansible_default_ipv4",
  "ansible_default_ipv6",
  "ansible_device_links",
  "ansible_devices",
  "ansible_distribution",
  "ansible_distribution_file_parsed",
  "ansible_distribution_file_path",
  "ansible_distribution_file_variety",
  "ansible_distribution_major_version",
  "ansible_distribution_release",
  "ansible_distribution_version",
  "ansible_dns",
  "ansible_docker0",
  "ansible_domain",
  "ansible_effective_group_id",
  "ansible_effective_user_id",
  "ansible_env",
  "ansible_eth0",
  "ansible_fips",
  "ansible_form_factor",
  "ansible_fqdn",
```

"ansible_hostname",
"ansible_interfaces",
"ansible_is_chroot",
"ansible_iscsi_iqn",
"ansible_kernel",
"ansible_lo",
"ansible_local",
"ansible_lsb",
"ansible_lvm",
"ansible_machine",
"ansible_machine_id",
"ansible_memfree_mb",
"ansible_memory_mb",
"ansible_memtotal_mb",
"ansible_mounts",
"ansible_nodename",
"ansible_os_family",
"ansible_pkg_mgr",
"ansible_processor",
"ansible_processor_cores",
"ansible_processor_count",
"ansible_processor_threads_per_core",
"ansible_processor_vcpus",
"ansible_product_name",
"ansible_product_serial",
"ansible_product_uuid",
"ansible_product_version",
"ansible_python",
"ansible_python_version",
"ansible_real_group_id",
"ansible_real_user_id",
"ansible_selinux",
"ansible_selinux_python_present",
"ansible_service_mgr",
"ansible_ssh_host_key_dsa_public",
"ansible_ssh_host_key_ecdsa_public",
"ansible_ssh_host_key_ed25519_public",
"ansible_ssh_host_key_rsa_public",
"ansible_swapfree_mb",
"ansible_swaptotal_mb",

```
  "ansible_system",
  "ansible_system_capabilities",
  "ansible_system_capabilities_enforced",
  "ansible_system_vendor",
  "ansible_uptime_seconds",
  "ansible_user_dir",
  "ansible_user_gecos",
  "ansible_user_gid",
  "ansible_user_id",
  "ansible_user_shell",
  "ansible_user_uid",
  "ansible_userspace_architecture",
  "ansible_userspace_bits",
  "ansible_virbr0",
  "ansible_virbr0_nic",
  "ansible_virtualization_role",
  "ansible_virtualization_type",
  "gather_subset",
  "module_setup"
]
```

De quoi est composé l’objet “ansible_facts.ansible_all_ipv4_addresses” ?

```
echo $FACTS | jq '.ansible_facts.ansible_all_ipv4_addresses | keys'
```

```
[
  "10.6.19.161",
  "172.17.0.1",
  "192.168.122.1"
]
```

C’est un tableau à trois entrées. Quelle est la seconde entrée ?

```
echo $FACTS | jq '.ansible_facts.ansible_all_ipv4_addresses[1]'
```

3. Format YAML pour Ansible

Pour Ansible, presque tous les fichiers YAML commencent par une liste. Chaque élément de la liste est une liste de paires clé / valeur, communément appelée “hash” ou “dictionary”. Il est donc nécessaire de savoir écrire des listes et des dictionnaires en YAML.[2]

On ici trouvera le minimum à connaître pour utiliser Ansible avec des fichiers de variables de données.

3.1. Début du fichier

Tous les fichiers YAML (indépendamment de leur association avec Ansible ou non) peuvent éventuellement commencer par `---` et se terminer par `...`. Cela fait partie du format YAML et indique le début et la fin d’un document.

3.2. Listes

Tous les membres d’une liste sont des lignes commençant au même niveau d’indentation avec un tiret et un espace : `-`

```
---
## A list of tasty fruits
- Apple
- Orange
- Strawberry
- Mango
...
```

3.3. Dictionnaires

Un dictionnaire est représenté sous une forme simple (les deux points doivent être suivis d’un espace): `key: value`, `clé: valeur`

```
## An employee record
martin:
  name: Martin D'vloper
  job: Developer
  skill: Elite
```

La relation entre les données se fait via indentation (2 ou 4 espaces par niveau).

Les valeurs de type “string” (texte) peuvent être encadrées par des guillemets `"` ou des trémas `'`. La valeurs numérique ne doivent pas être encadrées.

3.4. Dictionnaires et listes

Des structures de données plus complexes sont possibles, telles que des listes de dictionnaires, c’est-à-dire des dictionnaires dont les valeurs sont des listes ou un mélange des deux:

```
## Employee records
- martin:
  name: Martin D'vloper
  job: Developer
  skills:
    - python
    - perl
    - pascal
- tabitha:
  name: Tabitha Bitumen
  job: Developer
  skills:
    - lisp
    - fortran
    - erlang
```

3.5. Forme abrégée

Les dictionnaires et les listes peuvent également être représentés sous une forme abrégée si vous voulez vraiment:

```
---
martin: {name: Martin D'vloper, job: Developer, skill: Elite}
fruits: ['Apple', 'Orange', 'Strawberry', 'Mango']
```

Celles-ci sont appelées “collections de flux”.

3.6. Valeur booléenne

On peut également spécifier une valeur booléenne (true / false) sous plusieurs formes :

```
---
create_key: yes
needs_agent: no
knows_oop: True
likes_emacs: TRUE
uses_cvs: false
```

3.7. Valeurs sur plusieurs lignes

Les valeurs peuvent couvrir plusieurs lignes en utilisant `|` ou `>`. Le fait de couvrir plusieurs lignes en utilisant un “Literal Block Scalar” `|` inclura les nouvelles lignes et tous les espaces de fin. L’utilisation d’un “Folded Block Scalar” `>` pliera les lignes nouvelles dans les espaces; il est utilisé pour rendre ce qui serait autrement une très longue ligne plus facile à lire et à éditer. Dans les deux cas, l’indentation sera ignorée. Les exemples sont:

```
---
include_newlines: |
        exactly as you see
        will appear these three
        lines of poetry

fold_newlines: >
        this is really a
        single line of text
        despite appearances
```

Alors que dans l’exemple ci-dessus toutes les nouvelles lignes sont repliées dans des espaces, il existe deux manières de faire respecter une nouvelle ligne à conserver :

```
---
fold_some_newlines: >
  a
  b

  c
  d
```

```
    e
  f
same_as: "a b\nc d\n  e\nf\n"
```

3.8. Synthèse du format YAML

Combinons ce que nous avons appris jusqu'ici dans un exemple YAML arbitraire. Cela n'a vraiment rien à voir avec Ansible, mais cela vous donnera une idée du format de présentation de données à maîtriser.

```
---
## An employee record
name: Martin D'vloper
job: Developer
skill: Elite
employed: True
foods:
  - Apple
  - Orange
  - Strawberry
  - Mango
languages:
  perl: Elite
  python: Elite
  pascal: Lame
education: |
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
```

3.9. Appel aux variables

Les variables sont appelées dans les livres de jeu Ansible écrit en YAML en étant encadré par des doubles accolades avec espace (Jinja2).

```
- name: "print ansible_hostname variable play"
  hosts: all
  gather_facts: True
  tasks:
    - name: "print ansible_hostname variable task"
      debug:
        msg: "{{ ansible_hostname }}"
```

...

4. Modélisation Jinja2

Jinja2

Exemple[3].

```
apt update && apt install npm
npm install -g json2yaml
```

```
cat << EOF > template.j2
{{ ansible_facts.ansible_hostname }}

{% for capability in ansible_facts.ansible_system_capabilities %}
{{ capability }}
{% endfor %}
EOF
```

```
ansible localhost -m setup -o | sed 's/.*> {/{/g' | json2yaml > facts.yaml
```

```
ansible localhost -m template -a "src=template.j2 dest=/tmp/file.tmp" -e "@facts.yaml"
```

```
cat /tmp/file.tmp
```

1. Puisque les objets sont destinés à représenter des tableaux associatifs, il est recommandé, bien que non requis, que chaque clé soit unique dans un objet. ↩
2. YAML Syntax ↩
3. Voir Ansible snippets - manipulating JSON data et Json parsing in Ansible. ↩

Revision #1
Created 3 October 2021 21:10:06 by garfi
Updated 3 October 2021 21:10:51 by garfi