Installation SaltStack Documentation

Version 1.0.0

Aurelazy

16 July 2015

Table des matières

Contents :

CHAPITRE 1

Introduction

Je me suis complétement basé sur la documentation de SaltStack pour faire ce tutoriel. J'ai, par contre, fait mon tutoriel en suivant les différents et nombreux liens qui peuvent vite perdre une personne qui ne connaît rien à SaltStack. Et créer une arborescence plus simpliste. Je me suis permit également de traduire cette doc.

Merci à SaltStack pour leurs efforts et ce fabuleux projet !

CHAPITRE 2

Installation

Ce tuto va montrer comment installer Saltstack sur un serveur et un client Debian.

Pour wheezy, les lignes suivantes doivent être présente dans `/etc/apt/sources.list` ou dans un fichier dans `/etc/apt/sources.list.d` :

```
deb http://debian.saltstack.com/debian wheezy-saltstack main
```

Ensuite on importe le clé pour le dépot. Nous aurons besoin de celle utilisé lors de la signature du paquet.

```
wget -q -O- "http://debian.saltstack.com/debian-salt-team-joehealy.gpg.key" | apt-key add -
```

On fait un `Update` sur notre BDD de notre dépot :

```
apt-get update
```

2.1 Installation des paquets

On instal le `Salt master`, `minion` ou `syndic` depuis le dépot avec la commande `apt-get`. Chacun de ces exemples montrent l'installation d'un seul `daemon`, mais nous pouvons également le faire en une seul fois avec la même commande :

Sur le serveur :

```
apt-get install salt-master
```

Sur le client :

```
apt-get install salt-minion
```

Note : Un autre paquet est installé, mais je ne connait pas encore l'utilité, `salt-syndic`

2.2 Configuration de Salt

La configuration de Salt est très simple. La configuration par défaut du `master` marchera pour la plupart des installations et la seul exigence pour la mise en place d'un `minion` est de lui fixer l'adresse IP du `master` dans son fichier de configuration.

Le fichier de configuration sera dans le dossier `/etc/salt` et sera nommé `/etc/salt/master`, et `/etc/salt/minion`.

2.2.1 Master Configuration

Par défaut le `Salt master` écoute sur les ports 4505 et 4506 sur toutes les interfaces (0.0.0.0). Pour lier Salt à une IP spécifique, il faut redefinir l'instruction `interface` dans le fichier de configuration du `master`, typiquement `/etc/salt/master`, comme suit :

```
- #interface: 0.0.0.0
+ interface: 192.168.0.25
```

Après avoir réactualiser le fichier de configuration, il faut redémarrer le `Salt master`

2.2.2 Configuration du minion

Bien qu'il y ai beaucoup d'options de configuration pour les `Salt minion`, les configurer est très simple. Par défaut, un `Salt minion` va essayé de se connecter au nom DNS `salt` ; si le `minion` est capable de résoudre le nom correctement, auncune configuration n'est requise.

Si le nom DNS `salt` ne peut pas être résolu pour pointer sur le `master`, il faudra redefinir l'instruction `master` dans le fichier de configuration du `minion`, typiquement `/etc/salt/minion`, comme suit :

```
- #master: salt
+ master: 192.168.0.25
```

Après avoir réactualiser le fichier de configuration, il faut redémarrer le `Salt master`

2.3 Démarrer Salt

Démarrer le `master` en arrière plan (pour *daemonizer* le processus, ajouter l'option `-d`) :

```
salt-master
```

Démarrer le `minion` en arrière plan (pour *daemonizer* le processus, ajouter l'option `-d`) :

```
salt-minion
```

2.4 Quelques soucis ?

La manière la plus simple pour dépanner **Salt** est des lancer le `master` et le `minion` en arrière plan avec un niveau de log fixé à `debug` :

```
salt-master --log-level=debug
```

Au premier lancement du minion, un message d'erreur apparait :

```
[ERROR ] The Salt Master has cached the public key for this node, this salt minion will wait for 10 s
```

Il faut donc lancer la commande suivante sur le serveur :

```
salt-key -a <nom_DNS_minion>
```

`nom_DNS_minion` est le nom du minion

2.4.1 Gestion des clés

Salt utilise un cryptage AES pour toutes les communications entre le `Master` et le `Minion`. Ce qui permet de s'assurer que les commandes envoyées aux `Minion` ne peuvent être falsifiées, et la communication entre le `Master` et le `Minion` est authentifiée avec une clé de confiance et accéptée.

Avant que les commandes puissent être envoyées au `Minion`, sa clé doit être accépté par le `Master`. Lancer la commande `salt-key` pour lister les clés connus sur le `Master` :

```
[root@master ~]# salt-key -L
Unaccepted Keys:alpha
        bravo
        charlie
        delta
Accepted Keys:
```

Cet exemple montre que le `Master` est conscient qu'il y a 4 `Minion`, mais aucunes des clé n'a été accépté. Pour accépter les clés et permettre aux `Minion` d'être controllé par le `Master`, on va utiliser encore une fois la commande `salt-key` :

```
[root@master ~]# salt-key -A
[root@master ~]# salt-key -L
Unaccepted Keys:
Accepted Keys:alpha
        bravo
        charlie
        delta
```

La commande `salt-key` permet de signer les clés individuellement ou en lot. L'exemple ci-dessus, qui utilise l'option `-A` permet d'accepter toutes les clés en attente. Pour accepter chaque clé individuellement, il faudra utiliser la même option mais en minuscule, `-a`.

CHAPITRE 3

Mes premières commandes

La communication entre le `Master` et le `Minion` peut être vérifié en lançant la commande `test.ping` :

```
[root@master ~]# salt alpha test.ping
alpha: True
```

La communications entre le `Master` et tous les `Minions` peut etre testé d'une façon similaire :

```
[root@master ~]# salt '*' test.ping
alpha: True
bravo: True
charlie: True
delta: True
```

Chacun des `Minion` devrait envoyer une réponse `True` comme le montre l'exemple précédent.

Par exemple, la commande suivante doit retourner l'usage du disque de tous nos `Minions` :

```
[root@master ~]# salt '*' disk.usage
```

CHAPITRE 4

Apprendre à connaitre les fonctions

Salt est installé avec une grande librairie de fonctions à éxécuter, et les fonctions **Salt** sont auto-documentées. Pour voir quelles fonctions sont disponibles sur les `Minions` il suffit d'éxécuter la fonction `sys.doc` :

```
[root@master ~] salt '*' sys.doc
```

Cela va afficher une très grande liste des fonctions disponibles et leur documentation.

4.1 Les fonctions utiles à connaitre

Le module `cmd` contient des fonctions qui vont être lancées sur les `Minions`, tel que `cmd.run` et `cmd.run_all` :

```
[root@master ~] salt '*' cmd.run 'ls -l /etc'
```

La fonction `pkg` récupère automatiquement le système de paquet local sur les `Minions`. Cela veut dire que `pkg.install` va installer un paquet depuis `yum` sur les système Red Hat, `apt` sur Debian, etc, ... :

```
[root@master ~] salt '*' pkg.install vim
```

Nous allons voir comment installer un serveur Web, puis lancer le service :

```
[root@master ~] salt '*' pkg.install nginx
[root@master ~] salt '*' service.start nginx
```

La fonction `network.interfaces` va lister toutes les interfaces sur un `Minion`, avec leur adresse IP, le netmask, la MAC adresse, etc ... :

```
[root@master ~] salt '*' network.interfaces
```

4.2 Changement du format de sortie

Le format de sortie par defaut utilisé par les commandes **Salt** est appelé *nestedoutputter*, mais il existe plusieurs autres *outputter* qui peuvent être utilisés pour changer les messages de sortie. Par exemple, l'*outputter* `pprint` peut être utilisé pour afficher les données retournées en utilisant le module Python `pprint` :

```
[root@master ~] salt <myminion> grains.item pythonpath --out=pprint
```

CHAPITRE 5

Mon premier fichier SLS

Le système **state** est construit sur les formules *SLS*. Ces formules sont construites dans un fichier sur le serveur de fichier **Salt**. Pour faire une formule très basique *SLS*, il suffit d'ouvrir un fichier sous `/srv/salt` nommé `vim.sls`. Les **state** suivant assure que *vim* est installé sur un système sur lequel un **state** a été appliqué.

`/srv/salt/vim.sls` :

```
vim:
   pkg.installed
```

A partir de là, on installe `vim` sur le `Minion` en appelant le *SLS* directement :

```
salt '*' state.sls vim
```

Cette commande va invoquer le système **state** et lancer le *SLS* `vim`.

Maintenant, pour renforcer la formule *SLS* `vim`, un `vimrc` peut-être ajouté :

`/srv/salt/vim.sls` :

```
vim:
  pkg.installed: []

/etc/vimrc:
  file.managed:
    - source: salt://vimrc
        - mode: 644
        - user: root
        - user: root
```

Maintenant le fichier `vimrc` désiré doit être copié dans le serveur de fichier **Salt** dans `/srv/salt/vimrc`. Dans **Salt**, tout est fichier, donc aucune redirection de chemin n'a besoin d'être justifié. Le fichier `vimrc` est placé juste à coté du fichier `vim.sls`. La même commande que précédemment peut être éxécuté pour toutes les formules `vim` *SLS* et vont inclure le fichier de configuration.

5.1 Ajout de profondeur

Evidemment, maintenir les formules *SLS* dans un simple repertoire à la racine du serveur de fichier ne va pas favoriser pour un gros déploiement. C'est pourquoi plus de profondeur. Commençons par faire une formule `nginx` d'une meilleur façon, faisons un sous-fichier `nginx` et ajoutons un fichier `init.sls` :

`/srv/salt/nginx/init.sls` :

```
nginx:
  pkg.installed: []
        service.running:
          -require:
                - pkg: nginx
```

Quelques concepts sont introduits dans cette formule *SLS*.

D'abord, nous avons la déclaration du service qui va s'assurer que le service `nginx` est démarré.

Bien sûr, le service `nginx` ne peut pas être démarré à moins que le paquet soit installé d'où la déclaration `require` qui implante une dependance entre les 2.

La déclaration `require` s'assure que le composant requit est exécuté avant et que le résultat soit un succés.

Cette nouvelle formule *SLS* a un nom spécial `init.sls`. Quand une formule *SLS* est nommé `init.sls` elle hérite du nom du dossier qui la contient. Cette formule peut être référencé via la commande suivante :

```
salt '*' state.sls nginx
```

Maintenant que les sous-repertoires peuvent être utilisés, la formule `vim.sls` peut être effacé. Pour faire les chose d'une manière plus flexible, bougez le `vim.sls` et `vimrc` dans un sous-repertoire appelé `edit` et changez le fichier `vim.sls` pour refleter le changement :

`/srv/salt/edit/vim.sls` :

```
vim:
  pkg.installed

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
        - mode: 644
        - user: root
        - group: root
```

Seulement le chemin source du fichier `vimrc` a changé. Maintenant la formule est référencé en tant que `edit.vim` parcqu'il réside dans le sous-dossier `edit`. Maintenant le sous-dossier peut contenir des formules pour `emacs`, `nano`, `joe` ou n'importe quel autre editeur de texte qui devra être déployé.

CHAPITRE 6

Salt State

Comme vu précédement, voici un fichier *SLS* simple, il est écrit en `YAML` :

```
apache:
  pkg.installed: []
  service.running
    - require:
          - pkg: apache
```

Ces données *SLS* vous s'assurer que le paquet nommé apache est installé, et que le service apache est démarré.Les composants peuvent être expliqué d'une façon simple.

La première ligne est l'`ID` pour un ensemble de données, et est appelé `ID Declaration`. Cet `ID` determine le nom de l'objet qui doit être manipulé.

La seconde et troisième ligne contiennent le `state module function` à lancer, au format `<state_module>.<function>`. Le `state module function service.running` s'assure qu'un daemon est démarré.

Finallement, sur la ligne 5, nous avons le mot `require`. On l'appel un `Requisite Statement`, et il vérifie que le service `Apache` sera lancé seulement si l'installation du paquet c'est fait correctement.

6.1 Ajout de configurations et d'utilisateurs

Lorsque l'on installe un service comme *Apache*, beaucoup plus de composants doivent être ajoutés. La configuration d'*apache* doit être géré, et un utilisateur et un group doit être installé.

```
apache:
  pkg.installed: []
  service.running:
    - watch:
          - pkg: apache
          - file: /etc/httpd/conf/httpd.conf
          - user: apache
  user.present:
    - uid: 87
        - gid: 87
        - home: /var/www/html
        - shell: /bin/nologin
        - require:
          - group: apache
  group.present:
```

```
    - gid: 87
        - require:
          - pkg: apache

/etc/httpd/conf/httpd.conf:
  file.managed:
    - source: salt://apache/httpd.conf
        - user: root
        - group: root
        - mode: 644
```

Ces données *SLS* étendent considerablement le premier exemple, et inclus un fichier de configuration, un utilisateur, un groupe et un nouveau `requisite statement` : `watch`.

Ajouter plus de **state** est facile, dès lors que le nouvel **state** utilisateur et groupe sont en dessous de l'`ID apache`, l'utilisateur et le groupe seront l'utilisateur et le groupe d'`Apache`. Les `require` déclarations* vont s'assurer que l'utilisateur va être créé après le groupe, et que le groupe ne sera créé qu'après l'installation de paquet `Apache`.

Ensuite, la déclaration `require` sous *service* a été changé par `watch`, et il vérifie maintenant 3 **state** au lieu d'un seul. La déclaration* `watch` fait la même chose que `require`, il s'assure que les autres **state** soient démarrés avant le **state** `watch`, mais il ajoute un composant supplémentaire. La déclaration* `watch` va lancer le *state watcher function* (observateur d'état) pour tout changement effectué sur le *watched state*. Donc si le paquet a été updaté, le fichier de configuration changé, ou l'UID modifié, alors le *service state watcher* sera lancé. Le *service state watcher* va relancer le service, donc dans ce cas, un changement dans le fichier de configuration va également amorcer un redemarrage du service lié.

6.2 Aller au dela du fichier SLS unique

Lorsque l'on veut éditer Salt d'une manière évolutive, nous aurons besoin d'utiliser plus d'un fichier *SLS*. L'exemple précédent était dans un fichier *SLS* unique, mais 2 ou plusieurs fichiers peuvent être combinés pour construire notre "architecture".

L'exemple précédent nous présentait un fichier d'une manière bizarre - `salt://apache/httpd.conf`. Ce fichier a également besoin d'être présent.

Les fichiers *SLS* sont énoncés dans une structure de dossier sur le "*Salt Master*" ; un *SLS* est simplement un fichier et les fichiers à télécharger sont seulement des fichiers.

L'exemple pour apache serait énoncé à la racine du serveur de fichier de **Salt** de la manière suivante :

```
apache/init.sls
apache/httpd.conf
```

Donc le fichier `httpd.conf` est simplement un fichier dans le dossier apache, et est référencé directement.

Mais lorsque nous utilisons plus d'un fichier *SLS*, plusieurs composants peuvent être ajoutés à la "boite à outil". Considérons cet exemple pour SSH :

`ssh/init.sls`

```
openssh-client:
  pkg.installed

/etc/ssh/ssh_config:
  file.managed:
    - user: root
        - group: root
        - mode: 644
```

```
        - source: salt://ssh/ssh_config
        - require:
          - pkg: openssh-client
```

ssh/server.sls

```
include:
  - ssh

openssh-server:
  pkg.installed

sshd:
  service.running:
    - require:
        - pkg: openssh-client
        - pkg: openssh-server
        - file: /etc/ssh/banner
        - file: /etc/ssh/sshd_config

/etc/ssh/sshd_config:
  file.managed:
    - user: root
        - group: root
        - mode: 644
        - source: salt://ssh/sshd_config
        - require:
          - pkg: openssh-server

/etc/ssh/banner:
  file:
    - managed
        - user: root
        - group: root
        - mode: 644
        - source: salt://ssh/banner
        - require:
          - pkg: openssh-server
```

Note : On peut noter les 2 façons différente pour gérer les fichiers dans Salt. Dans le section `/etc/ssh/sshd_config` ci-dessus, on utilise la déclaration `file.managed`, alors que dans la section `/etc/ssh/banner`, on utilise la déclaration `file`, et on ajoute l'attribut `managed` dans la déclaration d'état. Ces 2 manières différentes amènent la même chose ; la première solution - "file.managed" - sera simplement plus rapide.

Maintenant, notre arborescence sera :

```
apache/init.sls
apache/httpd.conf
ssh/init.sls
ssh/server.sls
ssh/banner
ssh/ssh_config
ssh/sshd_config
```

Cet exemple introduit la déclaration `include`. Cette déclaration `include` inclue un autre fichier *SLS* de sorte que les composants trouvés à l'intérieur peuvent être requis (`require`), surveillés (`watch`)ou comme nous le verrons

plus tard, enrichis (`extended`).

La déclaration `include` permet aux états d'être croisés. Lorsqu'un fichier *SLS* à une déclaration `include`, elle sera littéralement étendue pour inclure le contenu des fichiers *SLS* inclus

A noter que certains fichiers SLS sont nommés "init.sls", tandis que d'autres ne le sont pas. Plus d'info sur le pourquoi du comment peut être trouvé dans le paragraphe *Mon premier fichier SLS*.

6.3 Extension et inclusion des données SLS

Parfois les données *SLS* ont besoins d'être étendues. Peut-être que le service apache de vérifier des ressources additionnelles, ou sous certaines circonstances un fichier différent aurait besoin d'être inséré.

Dans ces exemples, le premier va ajouter une bannière customisé à SSH et le second va ajouter plusieurs "watcher" (observateurs) à apache pour inclure le `mod_python`.

`ssh/custom-server.sls`

```
include:
  - ssh.server

extend:
  /etc/ssh/banner:
    file:
      - source: salt://ssh/custom-banner
```

`python/mod_python.sls`

```
include:
  - apache

extend:
  apache:
    service:
          - watch:
          - pkg: mod_python

mod_python:
  pkg.installed
```

Le fichier `custom-server.sls` utilise la déclaration pour réécrire l'emplacement du fichier de la bannière à télécharger, et donc changer le fichier qui sera utiliser pour configurer la bannière.

Dans le nouveau `mod_python` *SLS* le paquet "mod_python" est ajouté, mais plus important, le service apache est étendu pour, aussi, vérifier le paquet "mod_python".

Note : La déclaration `extend` fonctionne différemment pour `require` ou `watch`. Il ajoute, à la place de remplacer le composant requis.

CHAPITRE 7

Comprendre le système de rendu

Section à faire, voir la documentation Salt

Comme les données *SLS* sont simplement des données, elles n'ont pas nécécessairement besoin d'être écrite en langage YAML. **Salt** écrit par défaut en YAML, parcque que ce langage est simple et facile à apprendre et à écrire. Mais les fichier *SLS* peuvent être rendu depuis n'importre quel moyen, du moment que le module est installé.

Le système de rendu par défaut est `yaml_jinja`. `yaml_jinja` va d'abord passer le modèle à travers le système de modèle `Jinja2`, puis à travers l'analyseur YAML. Les benefices sont que la condtruction du programme sont accessibles lorsque l'on crée les fichiers *SLS*.

Les autres moteur de rendu que l'on peut utiliser sont `yaml_mako` et `yaml_wempy` qui utilisent respectivement le `Mako` ou `Wempy` système de modèle à la place de `Jinja` et plus particulièrement, Python ou `py`, `pyds1` et `pyobjects`. Le moteur de rendu `py` accepte Python pour écrire les fichier *SLS* pour un maximum de flexibilité et puissance lorsque l'on écrit les données *SLS* ; quand le moteur de rendu `pydsl` amène la flexibilité, et un langage pour un domaine spécifique lors de la création des fichier *SLS* ; et le moteur de rendu `pyobjects` apporte une interface `Pythonic` pour construire des données **state**.

Note : Les moteurs de modèle décrient au-dessus ne sont pas uniqueent utilisable dans les fichier *SLS*. Ils peuvent être utilisé dans les **state** `file.manged`, ce qui fait la configuration des fichiers bien plus dynamiques et flexibles. Quelques exemples de l'utilisation de modèles dans les fichiers de configurationpeuvent être trouvés dans la documentation des *fichiers \state*.

7.1 Commençont à apprendre le moteur par défaut - `yaml_jinja`

Le moteur par défaut- `yaml_jinja`, permet d'utiliser le sytème de modélisation jinja. Un guide sur jinja est disponible ici.

Lorsque nous travaillons avec un moteur de rendu quelques bits de données utiles sont écrites. Dans le cas d'un moteur de modèlisation basé sur un modèle de rendu (?), trois composants critiques sont disponible, `salt`, `grains`, et `pillar`. L'objet `salt` permet à n'importe quel fonction **Salt** d'appeler depuis un modèle, et un `grains` permet aux `Grains` d'être accessible depuis un modèle. Quelques exemples :

`apache/init.sls` :

```
apache:
  pkg.installed:
    {% if grains['os'] == 'RedHat' %}
        - name: httpd
        {% endif %}
  service.running:
```

```
    {% if grains['os'] == 'RedHat' %}
        - name: httpd
    {% endif %}
        - watch:
          - pkg: apache
          - file: /etc/httpd/conf/httpd.conf
          - user: apache
  user.present:
    - uid: 87
        - gid: 87
        - home: /var/www/html
        - shell: /bin/nologin
        - require:
          - group: apache
  group.present:
    - gid: 87
        - require:
          - pkg: apache


/etc/httpd/conf/httpd.conf:
  file.managed:
    - source: salt://apache/httpd.conf
        - user: root
        - group: root
        - mode: 644
```

Cet exemple est simple. Si le grain `os` voit que le systèmed'exploitation est un RedHat, alors le nom du paquet Apache ainsi que son service sera `httpd`.

Voici une façon plus aggressive d'utiliser Jinja, en paramétrant un module "MooseFS distributed filesystem chunkserver" :

`mossefs/chunk.sls` :

```
include:
  - moosefs

{% for mnt in salt['cmd.run']('ls /dev/data/moose*').split() %}
/mnt/moose{{ mnt[-1] }}:
  mount.mounted:
    - device: {{ mnt }}
        - fstype: xfs
        - mkmnt: True
  file.directory:
    - user: mfs
        - group: mfs
        - require:
          - user: mfs
          - group: mfs
{% endfor %}


/etc/mfshdd.cfg:
  file.managed:
    - source: salt://mossefs/mfshdd.cfg
        - user: root
        - group: root
        - mode: 644
```

```
        - template: jinja
        - require:
          - pkg: mfs-chunkserver

/etc/mfschunkserver.cfg:
  file.managed:
    - source: salt://mossefs/mfschunkserver.cfg
        - user: root
        - group: root
        - mode: 644
        - template: jinja
        - require:
          - pkg: mfs-chunkserver

mfs-chunkserver:
  pkg.installed: []
mfschunkserver:
  service.running:
    - require:
{% for mnt in salt['cmd.run']('ls /dev/data/moose*') %}
      - mount: /mnt/moose{{ mnt[-1] }}
          - file: /mnt/moose{{ mnt[-1] }}
{% endfor %}
          - file: /etc/mfschunkserver.cfg
          - file: /etc/mfshdd.cfg
          - file: /var/lib/mfs
```

A CONTINUER

CHAPITRE 8

Pas à pas sur les Pillars

Les **Pillars** sont des arborescences de données définies sur le **Salt Master** et envoyées aux `Minions`. Ils permettent la confidentialité, l'envoie sécurisé des données ciblées seulement sur le `Minion` approprié.

Note : Grains et **Pillars** peuvent amener à confusion, il faut se souvenir que les **Grains** sont des données sur un `Minion` qui sont stockées ou générées depuis un `Minion`. C'est pouqruoi les informations tel que l'OS et la CPU se trouvent dans les **Grains**. **Pillar** est une information sur un `Minion` ou plusieurs `Minions` stockées ou générées sur le **Salt Master**.

Les Données **Pillar** sont utiles pour :

Des données hautement sensibles : L'information transférée depuis **Pillar** est garantie être seulement présenté aux `Minions` qui sont ciblés, ce qui fait de **Pillar** un objet adapté pour gérer des informations sécurisées, tel que des clés cryptographiques et de mots de passe.

La configuration des |minions| : Les modules `Minion` tel que les modules d'éxécution, d'état, et "*returners*" peuvent souvent être configurés depuis des données stockées dans le **Pillar**.

Les variables : Les variables qui ont besoins d'être assignées à des `Minions` spécifique ou des groupes de `Minions` peuvent être défini dans un **Pillar** pour ensuite être accesibles dans une formule **SLS** et des fichiers "*template*".

Des données arbitraires : Pillar peut contenir une structure de données basique, donc une liste de valeurs, ou un couple clé/valeur peut être défini ce qui le fait plus simple à le répéter dans un groupe de valeurs dans une formule **SLS**.

Pillar est donc un des plus important système lorsque nous utilisons `Salt`. Ce tutoriel est construit pour créer un **Pillar** simple et opérationnel en peu de temps puis se plonger dans les capacités de **Pillar** et trouver où sont les données.

8.1 Configurer Pillar

Pillar est démarré par défaut avec `Salt`. Pour voir les données **Pillar** des `Minions` :

```
salt '*' pillar.items
```

Note : Avant la version 0.16.2, cette fonction est nommé `pillar.data`. Ce nom de fonction est toujours supporté pour assurer la comptabilité.

Par défaut le contenu du fichier de configuration maître est chargé dans le **Pillar** de tous les `Minions`. Ce qui permet que le fichier de configuration maître soit utilisé pour une configuration globale des `Minion`.

Pillar comprend des fichiers **SLS** et un `top file`, comme l'arborescence des `states`. L'emplacement par défaut de **Pillar** est : `/srv/pillar`.

Note : L'emplacement de **Pillar** peut être configuré depuis l'option `pillar_roots` dans le fichier de configuration du maître. Il ne doit pas être dans un sous-répertoire de l'arborescence `state`.

Pour commencer à configurer le **Pillar**, le repertoire `srv/pillar` doit être présent :

```
mkdir /srv/pillar
```

Maintenant créons un "*top file*", en suivant le même format que notre "*top file*" utilisé pour les `states` :

`/srv/pillar/top.sls` :

```
base:
  '*':
                - data
```

Ce "*top file*" associe le fichier `data.sls` à tous les `Minions`. Maintenant, le fichier `/srv/pillar/data.sls` doit être rempli :

`/srv/pillar/data.sls` :

```
info: some data
```

Pour s'assurer que tous les minions ont les données du nouveau **Pillar**, il faut lancer une commande sur eux leur demandant de récupérer leurs **Pillars** depuis le *master* :

```
salt '*' saltutil.refresh_pillar
```

Maintenant que tous les `Minions` ont le nouveau **Pillar**, on peut le récupérer :

```
salt '*' pillar.items
```

La clé `info` devrait apparaitre dans le retour des données du **Pillar**.

8.2 Des données un peu plus complexes

Contrairement aux **States**, **Pillar** n'a pas besoin de définir des formules. Cet exemple ajoute des utilisateurs avec leur UID respectif :

`/srv/pillar/users/init.sls` :

```
users:
  thatch: 1000
  shouse: 1001
  utahdave: 1002
  redbeard: 1003
```

Note : La même recherche sur les répertoire que pour les **States** existe pour **Pillar**, donc le fichier `users/init.sls` peut être référencé en tant que `users` dans le fichier "*top file*".

Nous allons devoir modifier le "*top file*" pour inclure notre nouveau fichier **SLS** :

`/srv/pillar/top.sls` :

```
base:
  '*':
    - data
        -users
```

Maintenant les données seront accessible aux `Minions`. Pour utiliser les données **Pillar** dans un **State**, nous pourrons utiliser *Jinja* :

`/srv/salt/users/init.sls` :

```
{% for user, uid in pillar.get('users', {}).items() %}
{{ user }}:
  user.present:
    - uid: {{ uid }}
{% endfor %}
```

Cette approche permet de définir d'une manière sécurisé les utilisateurs dans un **Pillar** pour ensuite appliquer les données de l'utilisateur dans un fichier **SLS**.

8.3 Paramétrer les States avec un Pillar

Les données d'un **Pillar** peuvent accéssibles depuis un fichier **State** pour customiser le comportement de chaque `Minion`. Tous les données d'un **Pillar** (et **Grains**) qui sont applicables pour chaque `Minion` sont remplacées dans le fichier **State** à travers des modèles avant d'être lancées. Typiquement, il va inclure les répertoires appropriés pour le `Minion` et passer sur les **States** qu'il ne doit pas appliquer.

Un exemple simple est de configurer un plan de noms de paquets dans **Pillar** pour chaque distribution Linux :

`/srv/pillar/pkg/init.sls` :

```
pkgs:
  {% if grains['os_family'] == 'RedHat' %}
  apache: httpd
  vim: vim-enhanced
  {% elif grains['os_family'] == 'Debian' %}
  apache: apache2
  vim: vim
  {% elif grains['os'] == 'Arch' %}
  apache: apache
  vim: vim
  {% endif %}
```

Le nouveau **SLS** `pkg` doit maintenant être ajouté à notre "*top file*" :

`/srv/pillar/top.sls` :

```
base:
  '*':
    - data
        - users
        - pkg
```

A présent les `Minions` vont se baser automatiquements d'après leur OS dans le **Pillar**, donc nous pouvont paramétrer sans risque le fichier **SLS** :

`/srv/salt/apache/init.sls` :

```
apache:
  pkg.installed:
    - name: {{ pillar['pkgs']['apache'] }}
```

Par contre, si aucun **Pillar** n'est créé(disponible, référencé), nous pouvons le créer dans le **State** directement :

Note : La fonction `pillar.get` utilisé dans cet exemple a été ajouté dans la version 0.14.0

`/srv/salt/apache/init.sls` :

```
apache:
  pkg.installed:
    - name {{ salt['pillar.get']('pkgs:apache', 'httpd') }}
```

Dans cet exemple précédent, si la valeur du **Pillar** `pillar['pkgs']['apache']` n'est pas configuré dans le **Pillar** du `Minion`, alors l'option par défaut `httpd` sera utilisé.

Note : Sous le capot, **Pillar** est simplement un dictionnaire Python, donc les méthodes du dictionnaire Python tel que `get` et `items` peuvent être utilisées.

8.4 PILLAR MAKES SIMPLE STATES GROW EASILY

http://docs.saltstack.com/en/latest/topics/tutorials/pillar.html#pillar-makes-simple-states-grow-easily

CHAPITRE 9

Le “Top File”

A ECRIRE VOIR : La documentation Salt

CHAPITRE 10

Les fichiers Salt State

Voir la doc

CHAPITRE 11

Indices and tables

— genindex
— modindex
— search