SALTSTACK

La gestion de configuration et de cloud

Les Jeudis du Libre, Mons
20/04/2017

Sébastien Wains

sebastien.wains@gmail.com
@sebw
@SebastienWains

Qui suis-je ?

Administrateur systèmes Linux chez ETNIC

Gradué en comptabilité et gestion (Institut Supérieur Economique de Mons)

Blogger https://blog.wains.be

Red Hat Certified Engineer

Contributeur Open Source

Que permet Salt ?

- gestion de configuration (serveurs et devices divers)
- provisioning "cloud"
- exécution à distance asynchrone (sa fonction première au début du projet)
- récupération d'informations
- orchestration
- monitoring
- auto-scalability
- compliance
- extensible
- ...

Ce qui m'a séduit par rapport à la concurrence

- Orchestration "event-driven" via un bus d'événement `bus event`
- Haute disponibilité du serveur maître `salt-master`
- Ecrit en Python et développement de "plugins" facile
- Configuration YAML et templating Jinja (**attention à la syntaxe !**)
- Mode master/slave ou masterless (code sur le serveur géré "minion")
- Mode push ET pull
- Salt Cloud pour instancier le serveur avant de gérer sa configuration
- Salt API pour intégrer avec d'autres outils
- Gestion de configuration, exécution à distance, récupération d'informations dans un seul package
- Simple !
- Communauté enthousiaste et dynamique

Ce qui m'a séduit par rapport à la concurrence

- SaltStack fourni des dépôts pour toutes les plateformes habituelles [0]
- Support de Windows et MacOS
- Langage impératif ET déclaratif [1]
- Salt SSH pour gérer les "dumb" devices qui embarquent Python 2.6+
- Salt Proxy pour gérer certains "super dumb" devices sans stack Python

[0] https://repo.saltstack.com
[1] https://docs.saltstack.com/en/getstarted/flexibility.html

Salt à l'ETNIC aujourd'hui

- Dernière version Salt Community stable
- 5 salt-master (1x lab par sysadmin, 1x non prod, 1x prod)
- 260 serveurs gérés
- 12 minutes pour déployer et intégrer une VM RHEL7 sur VMware avec Salt Cloud et Rundeck
- Intégrations avec RedHat Satellite, Rundeck [0], iTop [1], Jenkins, Mattermost
- Code dans un dépôt Git interne (Gitlab CE)
- Développement sur base d'un workflow collaboratif par fork [2]

[0] http://www.rundeck.org
[1] https://www.combodo.com/itop
[2] https://www.atlassian.com/git/tutorials/comparing-workflows/

Fonctionnement de base

Glossaire

`master` : serveur de gestion

`minion` : serveur géré

`event bus` : bus de communication pour les échanges entre master et minions

`modules` : module d'exécution distante ayant différentes fonctions (ex : pkg.install)

`states` : états de configuration (package installé, fichier configuré, service démarré, etc.)

`grains` : informations "statiques" des minions à disposition du master

`pillar` : informations dynamiques stockées sur le master à disposition des minions

`top.sls` : les fichiers d'assignation de `states` et `pillars` aux minions

`init.sls` : manifest d'un `state`, `pillar`

Fonctionnement de base

Glossaire

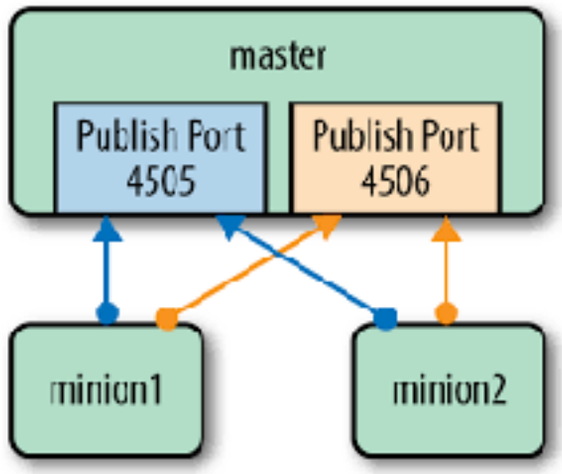
`beacons` : fonctionnalité permettant de monitorer des processus hors Salt (charge système, RAM, fichier, nombre de sessions HAproxy, etc.) et envoyer des messages sur l'`event bus`

`reactors` : actions déclenchées en réaction à un évenement sur l'`event bus`

`mine` : endroit sur le master où les minions poussent des informations à disposition des autres minions

Fonctionnement de base

- Les minions sont connectés constamment au master
- Sur le master, deux ports écoutent et doivent être accessibles pour les minions



- A la première connexion d'un minion, le master doit accepter sa clé
- Le salt-master peut tourner en utilisateur non privilégié
- Le salt-minion doit tourner en root
- Fonctionne avec SELinux en mode `Enforcing` actif sur le master et ses minions

Recommandations avant d'écrire nos premiers states

Infrastructure as Code [0] : un bug dans le code = un downtime éventuel

Nous sommes à présent des sysadmins développeurs :

- Définir des guidelines de développement (syntaxe, structure, etc.)
- Définir un workflow de développement (centralisé, par branche, par fork, etc.)
- Stocker le code dans un outil de gestion de versions (Git, etc.)

Ne rien pousser en production qui n'a pas été testé et validé (principe des 4 yeux)

Principe KISS : Keep It Simple, Stupid

> La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer. ~ Antoine de Saint-Exupéry

[0] https://en.wikipedia.org/Infrastructure_as_Code

Installation (RHEL/CentOS)

Sur chaque serveur :

```
yum install https://repo.saltstack.com/yum/redhat/salt-repo-latest-1.el7.noarch.rpm
```

Sur le master :

```
yum install salt-master --enablerepo=salt-latest
```

Sur les minions :

```
yum install salt-minion --enablerepo=salt-latest
```

Configuration du master

/etc/salt/master :

```
file_roots:
  base:
    - /srv/salt/states

pillar_roots:
  base:
    - /srv/salt/pillars
```

Démarrage du service : `systemctl start salt-master`

Démarrage en mode debug : `salt-master -l debug`

Configuration des minions

/etc/salt/minion :

```
master: salt-master
```

Démarrage du service : `systemctl start salt-minion`

Démarrage en mode debug : `salt-minion -l debug`

Accepter un minion

Depuis le master : `salt-key -a salt-minion`

Vérifier le statut de nos minions

```
[root@salt-master ~]# salt-key

Accepted Keys:
salt-minion

Denied Keys:

Unaccepted Keys:

Rejected Keys:
```

On vérifie s'il répond bien

Via la fonction d'**exécution distante**.

Ici avec le module `test` et la fonction `ping` :

```
[root@salt-master ~]# salt 'salt-minion' test.ping
salt-minion:
    True
```

Attention, ceci n'est pas un ping ICMP ou TCP !

Voir `/usr/lib/python2.7/site-packages/salt/modules/test.py` sur le minion.

Un autre exemple d'exécution distante

```
salt 'cible' module.fonction [arguments] [options salt]
```

Activation de SELinux :

```
# salt 'salt-minion' selinux.setenforce Enforcing --output=json
{
    "salt-minion": "Enforcing"
}
```

top.sls pour les states

```
[root@salt-master /]# cat /srv/salt/states/top.sls
base:
  '*':
    - motd

  'G@os:RedHat':
    - selinux

  'G@ETNIC_ROLE:frontend':
    - elastic
```

Appliquer le state `motd` sur tous les `minions`.

Pour dont le grain "OS" est `RedHat`, appliquer le state `selinux`.

Pour les machines avec le grain ETNIC_ROLE `frontend`, appliquer le state `elastic`.

Conseil : le top.sls doit être le plus **générique** possible. Ne pas cibler sur base du nom mais préférer le rôle.

Un premier état de configuration "state" : `motd`

```
[root@salt-master ~]# cat /srv/salt/states/motd/init.sls
ma_conf_motd:                                <-- ID unique
  file.managed:                              <-- module.fonction
    - name: /etc/motd                        <-- le fichier géré
    - source: salt://motd/motd.jinja         <-- template à utiliser
    - template: jinja                        <-- type de template
```

Syntaxe YAML, respecter les espaces !

Notre premier state : `motd`

Le template jinja :

```
[root@salt-master ~]# cat /srv/salt/states/motd/motd.jinja
Bonjour et bienvenue sur {{ grains['fqdn'] }}
Mon master est {{ grains['master'] }}
```

Grains : informations concernant nos minions

```
[root@salt-master ~]# salt 'salt-minion' grains.items
salt-minion:
    ----------
[...]
    fqdn:
        salt-minion
    master:
        10.211.55.26
    mem_total:
        988
    osfullname:
        Red Hat Enterprise Linux Server
    osmajorrelease:
        7
[...]
```

Notre premier state : `motd`

Récapitulons :

- assigner le rôle `motd` aux minions : `/srv/salt/states/top.sls`
- écrire notre state : `/srv/salt/states/motd/init.sls`
- créer un template : `/srv/salt/states/motd/motd.jinja`

Pour tester avant d'appliquer

```
salt '*' state.highstate -v test=True
```

Pour appliquer :

```
salt '*' state.highstate
```

Notre premier state : `motd`

```
[root@salt-master ~]# salt '*' state.highstate
salt-minion:
----------
          ID: ma_conf_motd
    Function: file.managed
        Name: /etc/motd
      Result: True
     Comment: File /etc/motd updated
     Started: 09:53:06.610581
    Duration: 78.358 ms
     Changes:
              ----------
              diff:
                  New file
              mode:
                  0644

Summary for salt-minion
------------
Succeeded: 2 (changed=1)
Failed:    0
------------
Total states run:     2
Total run time:  79.083 ms                                    79 ms !
```

Notre premier state : `motd`

Vérifions sur notre minion :

```
[root@salt-minion ~]# cat /etc/motd
Bonjour et bienvenue sur salt-minion
Mon master est 10.211.55.26
```

Un state plus avancé

```
{% if grains['os'] == 'RedHat' and grains['osmajorelease'] >= '6' %}

postfix-service:
  service.running:
    - name: postfix
    - enable: True
    - reload: False
    - require:
      - file: postfix-conf
    - watch:
      - file: postfix-conf

postfix-conf:
  file.managed:
    - name: /etc/postfix/master.cf
    - source: salt://postfix/master.cf.jinja
    - template: jinja

{% endif %}
```

La logique peut aussi être appliquée dans les templates !

Portabilité du code entre OS

Les modules se chargent de "deviner" les utilitaires à utiliser.

`pkg.installed` utilise le gestionnaire de package du système

`service.running` démarre le service via ce qu'il trouve comme système init

Oui mais...

Noms de packages différents entre distributions (`apache2` vs `httpd`) ?

Définir des "map files" !

Exemple : https://github.com/saltstack-formulas/vim-formula/blob/master/vim/map.jinja

Définir un nouveau grain manuellement

```
master# salt 'salt-minion' grains.setval ROLE ['frontend','elastic']
salt-minion:
    ----------
    ROLE:
        - frontend
        - elastic
```

On peut alors avoir un state tel que :

```
{% if grains['ROLE'] is defined %}
{% if 'frontend' in grains['ROLE'] %}
...
{% endif %}
{% endif %}
```

Ou :

```
{% for i in grains['ROLE'] %}
role-{{ i }}-conf:
  file.managed:
  ...
{% endfor %}
```

Définir un nouveau grain automatiquement

Alimenter des nouveaux grains avec des informations provenant de différentes sources externes : CMDB, LDAP, DB, API, etc.

Mise à jour au démarrage du minion ou via la commande master `saltutil.sync_grains`

Ce script Python sera placé sous `/srv/salt/states/_grains/satellite.py`

```python
#!/usr/bin/env python

import requests

def satellite_retrieve_info():
        node_id = __opts__['id']

        satellite_pillar = __pillar__.get('satellite', None)
        username = satellite_pillar['api_username']
        password = satellite_pillar['api_password']
        url = satellite_pillar['api_url']

        r = requests.get(url + '/hosts/' + node_id + '/',
        auth=(username, password))

        grains = {}
        grains['INFO_SATELLITE'] = r.json()

        return grains
```

Pillars : pour le stockage de données sensibles !

Imaginons un state `mysql` :

```
mysql-bob:
  mysql_user.present:
    - name: bob
    - host: localhost
    - password: hunter2
```

Ce fichier sera mis en cache sur les minions sous

`/var/cache/salt/minions/files/base/mysql/init.sls`

==> Problème potentiel de sécurité

Par contre, les pillars ne sont jamais conservés en cache !

Pillars : pour le stockage de données sensibles !

Alternative avec utilisation d'un pillar :

/srv/salt/pillars/top.sls

```
base:
  'salt-minion':
    - mysql
```

/srv/salt/pillars/mysql/init.sls

```
mysql:
  bob:
    password: hunter2
```

/srv/salt/states/mysql/init.sls

```
mysql-bob:
  mysql_user.present:
    - name: bob
    - host: localhost
    - password: {{ salt['pillar.get']('mysql:bob:password'), None }}
```

Pillars : pour le stockage de données sensibles !

Rappels :

- Les pillars sont conçus pour stocker des informations sensibles
- Ils ne sont **jamais** stockés sur les minions
- A chaque appel d'un pillar, un canal de communication dédié et crypté est établi entre le master et le minion
- Problème de performance potentiel si beaucoup trop de pillars (pas dans la doc !)

Que se passe-t'il sur l'event bus ?

Sur le master `salt-run state.event pretty=True`


```
minion_start    {
    "_stamp": "2017-02-21T07:59:28.146292",
    "cmd": "_minion_event",
    "data": "Minion jdl-minion1 started at Tue Feb 21 09:21:32 2017",
    "id": "jdl-minion1",
    "pretag": null,
    "tag": "minion_start"
}
salt/minion/jdl-minion1/start   {
    "_stamp": "2017-02-21T07:59:28.153296",
    "cmd": "_minion_event",
    "data": "Minion jdl-minion1 started at Tue Feb 21 09:21:32 2017",
    "id": "jdl-minion1",
    "pretag": null,
    "tag": "salt/minion/jdl-minion1/start"
}
```

Les reactors

Réactions à des events sur le bus, configurés sur le master `/etc/salt/master` :

```
reactor:
  - 'salt/minion/*/start':
    - /srv/salt/reactors/start.sls
  - 'salt/cloud/*/destroyed':
    - /srv/salt/reactors/destroy/*.sls
```

/srv/salt/reactors/start.sls :

```
hipchat:
  local.hipchat.send_message:
    - tgt: jdl-master
    - kwarg:
        room_id: Notifications
        from_name: Master
        message: "Démarrage de {{ data['id'] }} à {{ data['_stamp'] }}"
        api_key: xxx
        api_version: v2
        notify: False
```

==> au démarrage d'un minion, envoyer une notification vers Hipchat.

Salt API

Installation : `yum install salt-api --enablerepo=salt-latest`

Configuration /etc/salt/master.d/api.conf

```
external_auth:
  pam:
    testapiaccount:
      - .*
      - '@wheel'
      - '@runner'
      - '@jobs'

rest_cherrypy:
  port: 8443
  host: 0.0.0.0
  disable_ssl: False
  ssl_crt: /etc/ssl/private/cert.pem
  ssl_key: /etc/ssl/private/key.pem
  webhook_url: /hook
  webhook_disable_auth: True
  debug: False
```

Redémarrer le master : `systemctl restart salt-master` (pour la partie auth)

Démarrer Salt API : `systemctl start salt-api` ou `salt-api -l debug`

Reactors sur webhooks API

Events sur le bus sur requêtes webhooks : `salt/netapi/*`

https://github.com/saltstack-formulas/salt-api-reactor-formula

Interfaçage REST

Salt fourni donc un service REST

Possibilités :

- développer une interface web de gestion pour Salt
- un job Jenkins déclenche une action Salt (ex : déploiement d'une application une fois compilée)

Salt peut consommer des services REST

```
salt-run http.query https://jenkins.example.org/ params='{"job": "true"}'
```

Possibilité : event sur le bus --> reactor --> requête vers une API

Exemple :

- ouverture automatique d'un ticket lors d'un événement sur le bus (beacons !)

Salt Cloud

Permet de créer des machines virtuelles à partir de profils, sur différentes plateformes "cloud" et de virtualisation telles que :

- VMware
- Proxmox
- Openshift
- Amazon
- Google
- Parallels
- etc.

Salt Cloud est installé avec le package salt-master.

Salt Cloud

Définir un "provider" sous `/etc/salt/cloud.providers.d/vmware.conf` :

```
vcenter01:
  driver: vmware
  user: 'DOMAIN\user'
  password: 'verybadpass'
  url: 'vcenter01.domain.com'
  protocol: 'https'
  port: 443
```

Salt Cloud

Définir un "profile" sous `/etc/salt/cloud.profiles.d/vmware.conf` :

```
vmware-centos7.3:
  provider: vcenter01
  clonefrom: template-centos73
  num_cpus: 4
  memory: 8GB
  devices:
    disk:
      Hard disk 2:
        size: 20
    network:
      Network adapter 1:
        name: VLAN30
        ip: 10.20.30.123
        gateway: [10.20.30.110]
        subnet_mask: 255.255.255.128
        domain: example.com
  domain: example.com
  dns_servers:
    - 10.20.30.21
  script: bootstrap-salt
  script_args: -H proxy.example.org:8080 stable 2016.11
```

Salt Cloud

Instancier une VM :

```
salt-cloud -p vmware-centos7.3 nom-vm.example.com
```

A la création de la VM, salt-minion est installé et attaché automatiquement au master grâce au script bootstrap fourni par Salt (options `script` et `script_args`)

Démonstration !

Installation et configuration master et minion

Exécution distante

Gestion de configuration

Grains

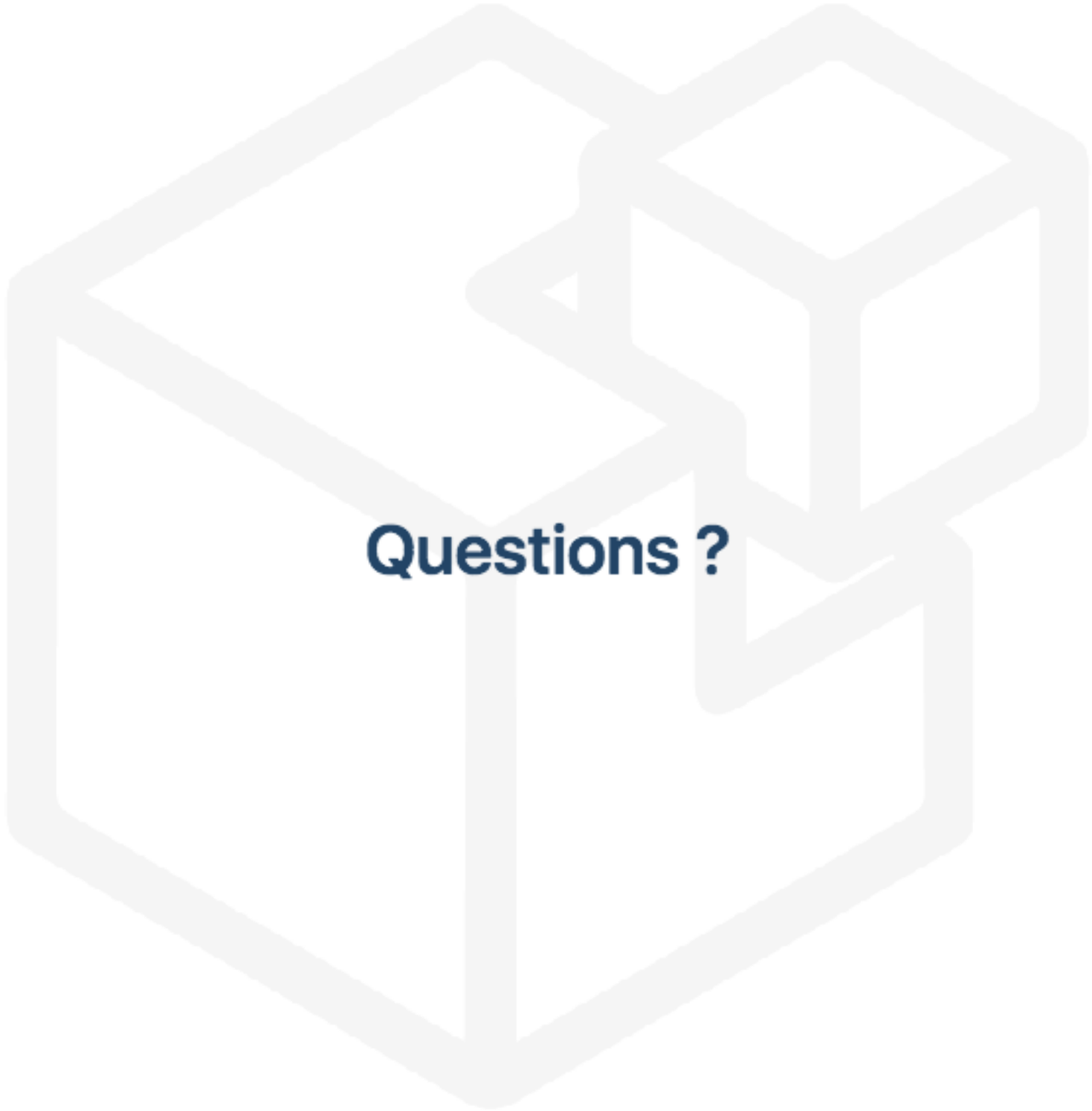
Pillars

Reactors

Beacons

Création de grains avec infos depuis une API

Salt API



Questions ?

Merci et à tout de suite !