



# SALTSTACK

## La gestion de configuration et de cloud

*Les Jeudis du Libre, Mons*

*20/04/2017*

**Sébastien Wains**

 [sebastien.wains@gmail.com](mailto:sebastien.wains@gmail.com)

 @sebw

 @SebastienWains

# Qui suis-je ?

Administrateur systèmes Linux chez ETNIC

Gradué de l'Institut Supérieur Economique (ISE) de Mons en... comptabilité

Blogger (<https://blog.wains.be>)

Red Hat Certified Engineer

Contribution à des projets Open Source



## Que permet Salt ?

- exécution à distance (sa fonction première au début du projet)
- gestion de configuration
- récupération d'informations
- orchestration
- monitoring
- provisioning
- auto-scaling
- compliance



## Les avantages de la gestion de configuration

- efficacité
- stabilité
- contrôle
- suivi
- documentation
- visibilité
- compliance
- ...












**Combien de serveurs pour commencer ?**



## Les précurseurs

Projet	Année
CFEngine	1993
Puppet	2005
Chef	2009

## Les petits "nouveaux"

Projet	Année	GitHub
Rexify	2010	 Watch ▾ 65  Star 527  Fork 187 ,
Salt	2011	 Watch ▾ 555  Unstar 7,187  Fork 3,296
Ansible	2012	 Watch ▾ 1,542  Star 20,352  Fork 6,369



## Un peu d'histoire

**En 2010, l'ETNIC avait environ 150 serveurs Linux (70% virtuels)**

- Des ressources techniques : DNS, relais SMTP, webmail, forward et reverse proxy, serveurs web, database, etc.
- Des projets et applications métiers : ESB, data warehouse, gestion électronique documentaire, SAP, formulaires intelligents, CMS, etc.

Utilisateurs :

- 160 employés ETNIC
- 5000 utilisateurs au Ministère de la Communauté Française
- 1100 utilisateurs à l'ONE
- 130000 enseignants

# Un peu d'histoire

## Les problèmes constatés dès les premières semaines

- Authentification `root` par mot de passe...
- ...connu de presque tout le monde (devs, ops, consultants :-())
- Des distributions différentes (SuSE Enterprise, OpenSuse, Debian, RedHat, RedHat Enterprise)
- Compilation au lieu d'utilisation de packages
- Aucune gestion des mises à jour

# Un peu d'histoire

## Les problèmes constatés dès les premières semaines

- Des services SSH, NTP, SMTP, DNS mal ou pas configurés
- Des inconsistences entre environnements d'un même projet ou nodes d'un même cluster
- Pas d'authentification centralisée
- Services sécurisés par SSL self-signed ou pas du tout
- Installation OS entièrement à la main depuis un ISO
- Monitoring quasi absent
- Un équivalent temps plein pour remettre de l'ordre dans tout ça...

# Un peu d'histoire

## Par où commencer ?

### Les challenges techniques

- La situation est problématique partout
- Il faut définir :
  - des standards
  - des politiques de sécurité, de mises à jours
  - des nouvelles méthodes de travail
- Il faut :
  - améliorer le processus de création de nouveaux serveurs
  - et de leur configurations
- Et migrer, migrer, migrer le legacy !
- En plus de travailler sur les nouveaux projets

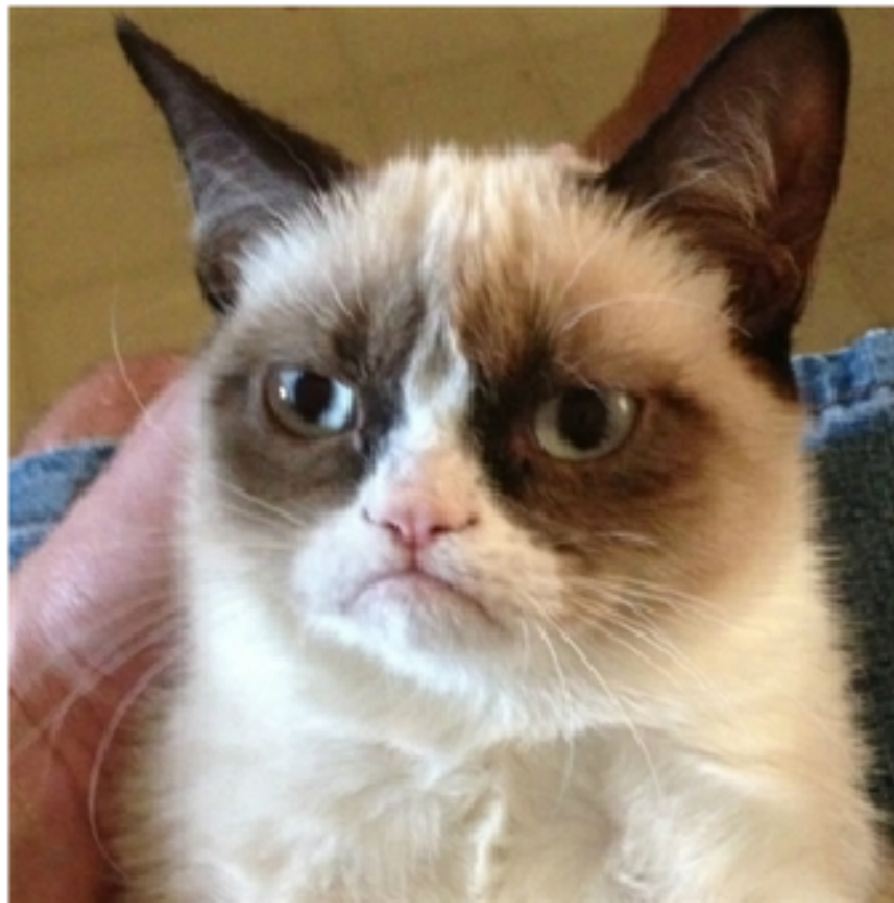
### Les challenges ne sont pas que techniques

# **Il faut faire face à la résistance au changement**

**Des accès root en production ?**



**L'accueil a parfois été mitigé**





## Un peu d'histoire

**Pour faire pour un mieux et dans l'immédiat je commence sur ces standards**

- Installation d'un serveur : template VMware RHEL5 installation minimale
- Installation des services de base et leurs configurations : script bash

## Un peu d'histoire

Mais après à peine quelques semaines...

```
$ wc -l install_rhel.sh  
664 install_rhel.sh
```

Et je ne gère alors que le strict minimum !



# Pourquoi pas... Puppet ?

## Fin 2011 je participe à une formation Puppet

- à l'époque Puppet est toujours en mode "pull"
- la recommandation du formateur pour notre infrastructure était deux masters à raison d'un pull toutes les 30 minutes ==> risque de delta
- la gestion de configuration, l'exécution à distance et la récupération d'informations depuis les nodes se font via trois composants installés séparément (Puppet, MCollective, Facter)
- la syntaxe n'est pas très claire (Ruby DSL)

**Conclusion : pas vraiment convaincu**

# Pourquoi Salt ? La genèse...

## Premiers tests en mai 2013

- deux principaux types de noeuds :
  - un serveur `salt-master`
  - des clients `salt-minion`
- gestion de configuration de services de base pour commencer, afin de remplacer le script (SSH, SMTP, NTP)
- remote execution ( `yum upgrade x` , `uptime` )
- récupération d'informations sur le parc (CPU, mémoire, version OS)
- code dans SVN
- test du code via `test=True` avant de pousser en production

## Les avantages (en 2013)

- Orchestration "event-driven" via un `event bus` sur le master
- Haute disponibilité du rôle `salt-master` possible
- Ecrit en Python avec des possibilités d'extensions intéressantes
- YAML et Jinja (mais attention à la syntaxe !)
- Modèle client/serveur ( `salt-minion` / `salt-master` )
- `salt-syndic` pour les grosses infrastructures ("proxy")
- Mode masterless possible (code sur le minion)

## Les avantages (en 2013)

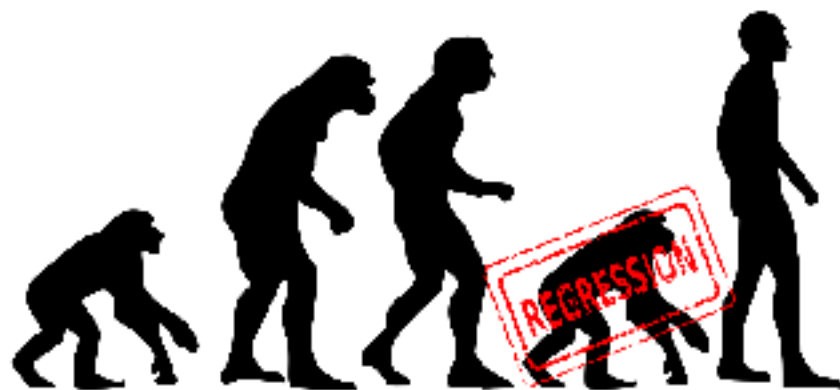
- Mode push ET pull
- Début d'un support de Windows
- Salt Cloud pour instancier le node avant de gérer sa configuration
- Gestion de configuration, exécution à distance, récupération d'informations dans un seul package
- Simple ! De l'installation à un premier fichier de configuration géré : 12 minutes
- Communauté enthousiaste et dynamique

## Les inconvénients (en 2013)

- Installation d'un agent (salt-minion) qui doit être dans la même version que le salt-master (pas d'alignement Debian/RedHat)
- Agent et ses dépendances (Python, ZeroMQ, msgpack) éparpillés dans les dépôts (Redhat, EPEL)
- Language déclaratif (ordre d'exécution aléatoire si pas de dépendances entre actions)
- Donne l'impression de partir un peu dans tous les sens :
  - tout est en chantier, rien n'est abouti
  - installation de Salt-API impossible
  - pas de support VMware dans Salt-Cloud
  - pull requests acceptés 5 minutes montre en main
  - failles de sécurité (dont une critique dans la PKI)

## Les inconvénients (en 2013)

- Release cycle trop rapide et sans test



- Quelques gros bugs :
  - `reload: True` qui fait un restart
  - ZeroMQ sous RHEL5 qui provoque la perte régulière des minions

## Salt aujourd'hui

- Pas de support Python 3
- SaltStack fourni des dépôts avec toutes les dépendances [0]
- Le support de Windows et MacOS a bien avancé
- Impératif ET déclaratif
- Ils ont engagé une équipe de testeurs : releases moins fréquentes, mieux testées, plus de régressions depuis longtemps
- Salt SSH pour gérer les "dumb" devices qui embarquent Python 2.6 ou plus
- Salt Proxy pour gérer les "super dumb" devices n'embarquant pas de stack Python
- Salt API fonctionne ! Intégrations possibles avec Jenkins, Rundeck, Satellite, etc.
- Un web GUI dans la version enterprise

[0] <https://repo.saltstack.com>



# Salt à l'ETNIC aujourd'hui

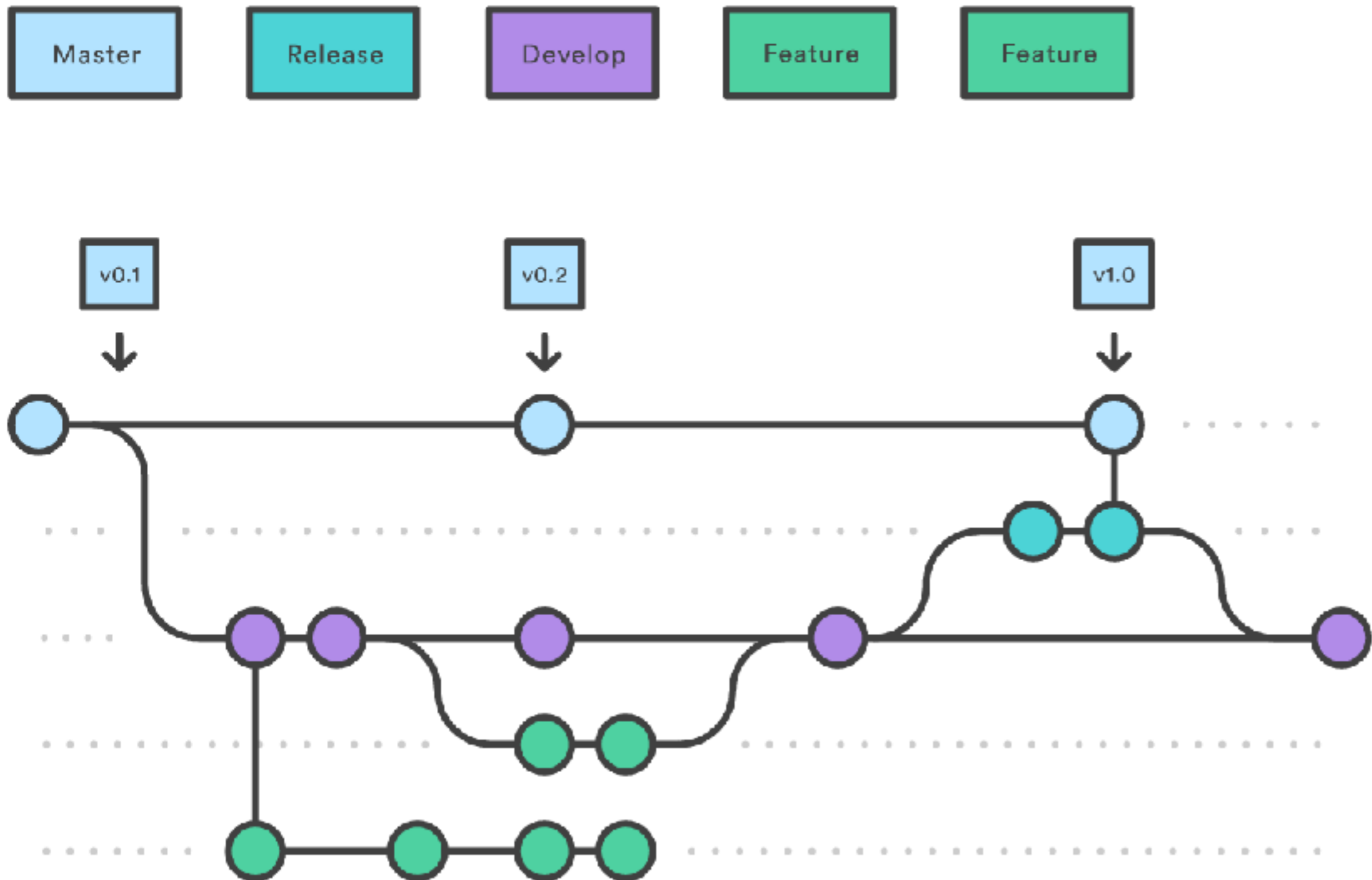
- L'équipe Linux a triplé il y a un an et Salt a été adopté immédiatement par les deux nouveaux membres
- Un consultant de SaltStack est venu auditer notre infrastructure
- Salt Community 2016.11 (dernière release stable)
- 260 serveurs Red Hat gérés (virtualisation 99%) 👍
- Cinq salt-master (3x lab, 1x non prod, 1x prod)
- Encore quelques serveurs legacy non gérables 💣
- Un nouveau serveur virtuel RHEL7 complètement provisionné et intégré en moins de 10 minutes grâce à Salt Cloud et Rundeck [0]
- Tout le code dans un dépôt Gitlab
- Workflow de développement [1]

[0] <http://www.rundeck.org>

[1] <https://www.atlassian.com/git/tutorials/comparing-workflows/>



# Workflow de développement





# Fonctionnement de base

## Glossaire

`master` : serveur de gestion

`minion` : serveur géré

`modules` : module d'exécution ayant différentes fonctions (ex : `file.managed`)

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

`grains` : informations relativement statiques des minions

`pillar` : informations dynamiques stockées sur le master à disposition des minions pour lesquelles elles sont définies

`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** : action déclenchée en réaction à un événement sur l' **event bus**

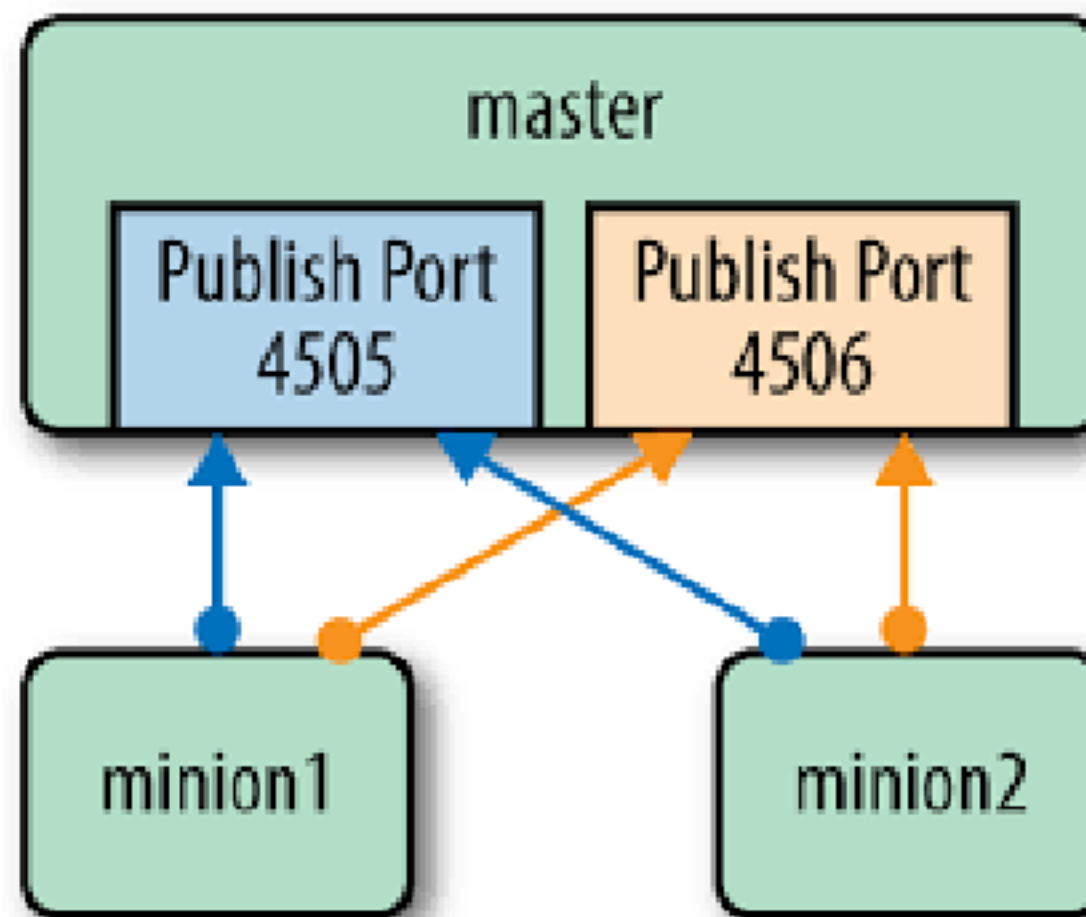
**mine** : fonction du master qui collecte des données générées par des minions pour les rendre disponibles auprès des autres minions. Ces données sont supposées très dynamiques, avec un rafraichissement configurable

## Remarque

Curieusement dans la documentation et la configuration, SaltStack parle de **states** et **grains** (pluriel) mais de **pillar** (singulier)

# Fonctionnement de base

Mode client/server construit autour d'un event bus



## Fonctionnement de base

- Les minions restent connectés constamment au master (message bus ZeroMQ)
- Le master doit accepter la clé d'un nouveau minion (PKI)
- Sur le master, deux ports écoutent :
  - TCP/4505 : le bus de communication avec les minions
  - TCP/4506 : pour les messages de retour des minions
- Le salt-master peut tourner en utilisateur non privilégié
- Le salt-minion doit tourner en root

# Installation

## 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=saltstack
```

## Sur les minions :

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

# Configuration du master

/etc/salt/master :

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

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

Démarrage du service : `service salt-master start`



# Configuration des minions

/etc/salt/minion :

```
master: salt-master
```

Démarrage du service : `service salt-minion start`

## Accepter un minion

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

## Vérifier le statut de nos minions

```
[root@salt-master ~]# salt-key
Accepted Keys:
salt-minion <===== notre minion est accepté :-)
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 !

Le minion exécute une fonction Python:

```
def ping():
    return True
```

Code simplifié issu de `/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]`

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

# Rappel avant d'écrire nos premières lignes de code

Infrastructure as code (un bug dans le code = un downtime éventuel)

Nous sommes à présent des développeurs ! Définir des guidelines de développement (syntaxe, structure des fichiers, etc.)

Système de contrôle de versions (SVN, Git, etc.) avec un workflow de développement

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

Et surtout :

## 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

## Arborescence sur le master

```
[root@salt-master /]# find /srv
/srv
/srv/salt
/srv/salt/pillars
/srv/salt/pillars/top.sls <----- top.sls pour les pillars
/srv/salt/pillars/passwords
/srv/salt/pillars/passwords/init.sls <-- un pillar
/srv/salt/states
/srv/salt/states/motd
/srv/salt/states/motd/init.sls <----- un state
/srv/salt/states/motd/motd.jinja
/srv/salt/states/selinux
/srv/salt/states/selinux/init.sls <----- un autre state
/srv/salt/states/top.sls <----- top.sls pour les states
```

## top.sls pour les states

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

  'G@os:RedHat':
    - selinux

  'G@ROLE:solr':
    - solr
```

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

Pour les OS de type `RedHat` , appliquer le state `selinux` .

Ne jamais cibler un serveur sur base de son nom ! Le `top.sls` doit être le plus **générique** possible.

## Notre premier state : **motd**

MOTD : "message of the day", message affiché à la connexion au serveur

Notre état de configuration (fichier .sls, SaLt State) est écrit en YAML :

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

`salt://` est un serveur HTTP embarqué dans Salt, on peut spécifier d'autres types de sources : `http://` , `https://` , `ftp://` , `file:///` , etc.

Par défaut jinja, plusieurs moteurs de templates supportés. But KISS !



## 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'] }}
```

## Notre premier state : motd

Pour lister les grains disponibles sur un minion :

```
[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**

Appliquons la configuration avec `state.highstate` :

```
[root@salt-master ~]# salt 'salt-minion' 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
```



**Les states se reposent sur l'exécution distante !**

# Allons plus loin...

Gestion d'un service et de sa configuration !

```
postfix-pkg:
  pkg.installed:
    - name: postfix

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

postfix-conf:
  file.managed:
    - name: /etc/postfix/master
    - source: salt://postfix/master
    - template: jinja
    - require:
      - pkg: postfix-pkg
```

# Impératif ET déclaratif

## Impératif :

- Salt exécute les actions dans l'ordre de définition
- plus simple à écrire mais moins flexible

## Déclaratif :

- on définit les dépendances entre les actions
- modèle plus puissant et flexible mais attention au spaghetti code

<https://docs.saltstack.com/en/getstarted/flexibility.html>

## Portabilité du code entre OS

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

`pkg.installed` utilisera le gestionnaire de package du système : `yum` , `apt` , `zypper` , etc.

`service.running` démarra le service via ce qu'il trouve parmi `sysVinit` , `systemd` , `upstart` .

## Oui mais...

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

Définir des "map files" ! (ressemble fortement à un template jinja)



## Un template plus avancé

```
{% if grains['os'] == 'RedHat' and grains['osmajorrelease'] == '5' %}
...
{% elif grains['os'] == 'RedHat' and grains['osmajorrelease'] >= '6' %}
...
{% elif grains['os'] == 'Debian' %}
...
{% else %}
...
{% endif %}
```

Les grains à notre disposition sont nombreux, mais est-ce suffisant ?

# Définir un nouveau grain manuellement

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

On peut alors avoir un template tel que :

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

Ou :

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

# Définir un grain automatiquement

Il est possible de récupérer des informations provenant de différentes sources (CMDB, LDAP, DB, API) et de les stocker dans en grains sur nos minions.

Mise à jour au runtime ou `saltutil.sync_grains`

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

```
#!/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 : stockage de données sensibles !

Pour des raisons de performances, chaque `state` est mis en cache sur le minion à l'exécution de la commande `state.highstate`

Imaginons un state `mysql-users` :

```
bob:
  mysql_user.present:
    - host: localhost
    - password: eponge
```

Ce fichier sera mis en cache sur les minions sous

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

### **==> Problème de sécurité**

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

# Pillars : stockage de données sensibles !

Alternative avec utilisation d'un pillar :

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

```
mysql:
  bob:
    password: eponge
```

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

```
bob:
  mysql_user.present:
    - host: localhost
    - password: {{ salt['pillar.get']('mysql:bob:password'), 'default' }}
```

Rappel :

- Les pillars sont conçus pour stocker des informations sensibles
- Ils ne sont jamais stockés sur les minions
- Limiter leur utilisation aux données sensibles
- Problème de performances potentiel si beaucoup trop de pillars (pas dans la doc !)

# Différentes syntaxes pour faire la même chose ?

## Grains

```
{{ grains['ROLE'] }}
```

```
{{ salt['grains.get']('ROLE', None) }}
```

## Pillars

```
{{ pillar['PWD'] }}
```

```
{{ salt['pillar.get']('PWD', None) }}
```

Privilégier la méthode `salt['module.function']` plus avancée et permettant de définir des valeurs par défaut si la variable recherchée n'existe pas.

## Que se passe-t'il sur le bus ?

```
salt-run state.event pretty=True
```

```
salt/job/20161211013317058053/ret/salt-minion {
  "_stamp": "2016-12-11T00:33:17.060487",
  "arg": [
    "bonjour"
  ],
  "cmd": "_return",
  "fun": "event.send",
  "fun_args": [
    "bonjour"
  ],
  "id": "salt-minion",
  "jid": "20161211013317058053",
  "retcode": 0,
  "return": true,
  "tgt": "salt-minion",
  "tgt_type": "glob"
}
```

## Les runners

`state.highstate` s'exécute de manière concurrentielle sur les minions.

Le runner `state.orchestrate` s'exécute sur le master et permet d'appliquer les états de manière orchestrée dans un ordre défini et coordonné, par exemple :

- installer la base de données
- installer le serveur applicatif backend
- reconfigurer le HAproxy frontend



## Les reactors

```
reactor:  
  - 'salt/job/*/ret/salt-minion':  
    - /srv/salt/reactors/jenkins.sls
```

A chaque retour d'un minion, déclencher le reactor jenkins.

# Interfaçage REST

## Salt peut consommer des services REST

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

On peut donc imaginer une fonctionnalité webhook avec un reactor.

Par exemple :

- ouverture automatique d'un ticket lors d'un événement sur le bus
- déclencher un job Jenkins

## Salt fourni un service REST

Exemple :

- déclencher une action Salt après exécution d'un job Jenkins (déployer un nouvel artefact)

# Salt Cloud

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

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



## Démonstration !

**La production des Jeudis du Libre tient à rassurer le spectateur qu'aucun minion ne sera maltraité durant cette présentation.**



**L'ETNIC est régulièrement à la recherche de nouveaux talents !**

**<https://monjob.etnic.be>**



## Questions et (tentatives de) réponses



**Merci et à tout de suite ! 🍺🍺🍺**