Lab 6 - le code de Puppet

Le catalogue

Nous allons commencer à utiliser le terme **catalogue** dans les sections suivantes, mais qu'est-ce que le catalogue?

En termes simples, il s'agit simplement de la version **compilée** de votre code Puppet (manifest).

Quand l'agent Puppet s'exécute, il renvoie tous les facts sur le système sur lequel il s'exécute au Puppet Master. Tout le code Puppet se situe sur le **Puppet Master**, et master prend tout les variables Puppet, les facts et le code (logique conditionnelle, fonctions, etc.) et les compile en un ensemble statique d'instructions à exécuter côté agent.

Toutes ces instructions sont exécuté côté agent dans un certain ordre tel que déterminé par les dépendances définies dans votre code Puppet.

Un autre concept important à comprendre est que lorsque le catalogue est compilé, toutes les fonctions et la logique conditionnelle que vous avez dans vos manifestes **est évaluée du côté** master, pas du côté agent. Les variables Puppet telles que les facts système sont également interpolées à ce moment. La totalité de la prise de décision dans un manifest a déjà eu lieu lorsque l'agent reçoit son catalogue, de sorte que l'agent exécute simplement les instructions qu'il reçoit dans le bon ordre.

Définition de classes et déclaration de classes

Tout d'abord, nous devons comprendre la différence entre **définir** une classe et **déclarer** une classe.

Définition des classes

- blocs nommés de code Puppet qui sont généralement stockés dans des modules pour une utilisation ultérieure
- ne sont PAS appliqués tant qu'ils ne sont pas appelés par leur nom (déclarés)
- ils peuvent être ajoutés au catalogue d'un nœud en les déclarant dans vos manifests

Déclaration d'une classe dans un manifest Puppet

Vous pouvez déclarer des classes

- dans les définitions de nœud au niveau supérieur dans le manifeste du site (site.pp)
- dans d'autres classes (par exemple, classes de profil lors de l'utilisation du paradigme Rôles et profils)

Exemple de déclaration d'une classe au top-scope (dans le site.pp) à l'aide de l'instruction **include**:

```
puppet
    include common_hosts
```

La classe **common_hosts** devrait être définie dans un fichier nommé `common_hosts.pp` et se trouver dans la base de code Puppet. **include** lit simplement le fichier manifeste et déclare (ajoute au catalogue) les ressources de cette classe.

La bonne chose à propos de **include** est que si la classe contenue a déjà été déclarée ailleurs, elle ne sera plus ajoutée au catalogue (ce qui serait une erreur de Puppet.) Les ressources ne peuvent être déclarées **qu'une seule** fois. Si vous essayez de déclarer la même ressource (identifiée par son nom) deux ou plusieurs fois, puppet lancera une erreur.

Si vous mettez cette instruction include dans votre `site.pp`, en dehors de toute définition de nœud, elle s'appliquerait à chaque nœud. (Remarque: ce n'est pas le même que l'inclure dans la section `node default {}`, qui ne s'applique que si aucune autre définition de nœud ne correspond.)

En savoir plus sur la classification des nœuds

Exemple de déclaration d'une classe pour un nœud spécifique:

```
node foo_server {
   include foo_server_code
}
```

Cela rechercherait une classe appelée **foo_server_code** et la déclarerait UNIQUEMENT pour le nœud nommé **foo_server**.

Remarque: n'utilisez pas le caractère tiret dans les noms de classe. Le tiret-bas est autorisé.

Vous pouvez également spécifier plusieurs nœuds dans une seule définition de nœud, ou même utilisez une expression régulière pour faire correspondre plusieurs nœuds. Vous pouvez en savoir plus sur Node Definitions dans la documentation officielle de PuppetLabs.

Écrivons du code

Faisons un bref aperçu de ce que Puppet peut faire.

Et si vous souhaitez installer des packages?

```
package { 'bind-utils':
  ensure => 'installed'
}

package { 'dstat':
  ensure => 'installed'
}

package { 'git':
  ensure => 'installed'
}n

package { 'tcpdump':
  ensure => 'installed'
}

package { 'net-tools':
  ensure => 'installed'
}
```

Ce code installera la dernière version de ces 5 packages, et puppet ne suivra **PAS** les versions installées. Si une version plus récente devient disponible dans les dépôts logiciels de l'hôte (par exemple, les dépôts yum configurés), Puppet ne le remarquera pas et ne fera rien. Si vous voulez que Puppet installe le dernier package, et suivre la version, et mettre à jour quand une nouvelle version est libéré, alors vous devez utiliser:

```
ensure => 'latest'
```

... dans votre ressource `package`.

Vous pouvez également spécifier une version du package si vous souhaitez épingler la version à une version très spécifique, mais cela ne fonctionne que pour les systèmes qui ont un fournisseur «versionable». Yum le permet, mais certains systèmes de paquets plus anciens tels que «up2date» ne le font pas.

Ajoutons ce code à un fichier manifeste et enveloppons-le dans une définition de classe. Une classe nous permet de faire référence à un code par son nom, et éventuellement de passer les paramètres auxquels le code de la classe pourrait accéder pour contrôler plus ce que fait le code.

Installer certains packages

Configurons puppet pour nous assurer que certains packages sont installés sur notre nœud d'agent.

Connectez-vous à votre nœud Puppet Master et devenez root

```
[root@puppet ~]# cd /etc/puppetlabs/code/environments/production/manifests
[root@puppet manifests]# vi common_packages.pp
```

Nous ajouterons ce nouveau manifeste dans l'environnement **production** dans le répertoire **manifests**.

Notez que Puppet recherche du code dans le répertoire des manifestes en fonction de l'environnement dont il fait partie, et la valeur 'manifest' de la configuration .

```
[root@puppet]# puppet config print environment
production
```

```
[root@puppet]# puppet config print manifest
/etc/puppetlabs/puppet/environments/production/manifests
```

Dans votre fichier common_packages.pp, ajoutez votre code enveloppé dans une classe avec le même nom du fichier, moins l'extension .pp, comme ceci:

```
class common_packages {

  package { 'bind-utils':
    ensure => 'installed'
  }

  package { 'dstat':
    ensure => 'installed'
  }

  package { 'git':
    ensure => 'installed'
  }

  package { 'tcpdump':
    ensure => 'installed'
  }

  package { 'net-tools':
    ensure => 'installed'
  }
}
```

Nous avons maintenant **défini** une classe, mais ce code ne fera rien tant que nous ne l'aurons pas épinglé à un nœud. Dès que vous enregistrez le manifeste, puppet le *"verra"* lors de sa prochaine exécution. Il saura que la classe a été **définie**, mais n'appliquera le code à aucun nœud jusqu'à ce

que nous **déclarions** la classe pour un nœud. Rappelez-vous cette chose appelée «classification des nœuds»?

Classifions le nœud d'agent avec la classe common_packages.

Nous éditerons notre `site.pp`

```
[root@puppet]# vi site.pp
```

À la fin de «site.pp», ajoutez ce qui suit:

```
node 'agent.example.com' {
   include common_packages
}
```

Enregistrez-le, puis exécutez 'puppet agent -t' sur votre machine virtuelle d'agent, et vous devriez voir les packages s'installer ...

```
[root@agent ~]# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for agent.example.com
Info: Applying configuration version '1479239887'
Notice: /Stage[main]/Common_packages/Package[bind-utils]/ensure: created
Notice: /Stage[main]/Common_packages/Package[dstat]/ensure: created
Notice: /Stage[main]/Common_packages/Package[git]/ensure: created
Notice: /Stage[main]/Common_packages/Package[tcpdump]/ensure: created
Notice: Applied catalog in 19.05 seconds
```

Avez-vous remarqué que seuls certains packages ont été installés? Si vous faites un `yum info net-tools` vous remarquerez qu'il était déjà installé, donc Puppet n'a rien fait pour ce package. Remarque: net-tools est requis pour Puppet, donc quand nous avons installé puppet, ce package a été automatiquement installé.

Maintenant que ces 5 packages ont été installés, si vous exécutez à nouveau puppet, aucune modification supplémentaire ne sera apportée.

Le nœud par défaut

Maintenant, je veux revenir sur quelque chose que nous avons mentionné plus tôt. N'oubliez pas, le **nœud par défaut** s'applique uniquement à un noeud **si aucune autre** définition de nœud ne

lui correspond. Nous venons d'ajouter une définition de nœud pour 'agent.example.com', de sorte que la définition de nœud par défaut ne s'appliquera plus. L'ordre des définitions de nœuds n'a pas d'importance. Pour prouver ça, allez-y et éditez votre fichier `/etc/hosts` sur l'agent. SUPPRIMER la ligne avec gitlab dessus, puis réexécutez 'puppet agent -t'. Notez que Puppet n'a pas rajouté la ligne.

Profitons de cette occasion pour extraire le code de la définition de 'node default', et le placer dans un autre fichier manifest appelé `common_hosts.pp`

Enveloppez ce code dans une définition de classe comme celle-ci:

```
class common_hosts {

 # remove all unmanaged resources
 resources { 'host': purge => true }


 # add some host entries
 host { 'localhost':              ip => '127.0.0.1', }
 host { 'puppet.example.com':     ip => '192.168.198.10', host_aliases => [
'puppet' ] }
 host { 'agent.example.com':      ip => '192.168.198.11', host_aliases => [
'agent' ] }
 host { 'gitlab.example.com':     ip => '192.168.198.12', host_aliases => [
'gitlab' ] }


}
```

... enregistrez votre nouveau **common_hosts.pp**, revenez à notre site.pp et incluez-le pour le nœud d'agent à nouveau.

Supprimez les ressources host du site.pp et incluez à la place la nouvelle classe contenu dans votre nouveau **common_hosts.pp** comme suit:

```
node default {
   include common_hosts
}


node 'agent.example.com' {
  include common_hosts
  include common_packages
}
```

Maintenant, réexécutez `puppet agent -t` sur votre nœud agent, et notez que l'entrée gitlab a été rajoutée.

```
[root@agent ~]# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for agent.example.com
Info: Applying configuration version '1479240149'
Notice: /Stage[main]/Common_hosts/Host[gitlab.example.com]/ensure: created
Info: Computing checksum on file /etc/hosts
Notice: Applied catalog in 0.69 seconds
```

Le point important que nous illustrons ici est que la définition de nœud par défaut s'applique UNIQUEMENT si AUCUNE AUTRE définition de nœud ne lui correspond.

De plus, puisque nous incluons **common_hosts** à la fois dans les définitions du nœud par défaut et du nœud agent, nous pourrions choisir de supprimer cette inclusion des deux afin qu'elle s'applique globalement, même si nous ajoutons à l'avenir de nouveaux hôtes avec leurs propres définitions de nœuds.

Modifions notre site.pp pour qu'il ressemble à ceci:

```
node default {
}
node agent {
   include common_packages
}
include common_hosts
```

Notez maintenant que la définition de nœud par défaut est vide et que seul la définition de l'agent inclut la classe common_packages. chaque hôte obtiendra la classe common_hosts. Lançons à nouveau 'puppet agent -t' pour prouver que notre code fonctionne toujours.

Encore une fois, supprimez la ligne gitlab dans /etc/hosts sur l'agent et le master, et exécutez à nouveau Puppet, et vous devriez voir Puppet l'ajouter à nouveau sur les deux.

Nous voulons éventuellement avoir git sur le master, alors apportons une dernière modification à notre site.pp et déplaçons l'inclusion common_packages en dehors de la définition de nœud agent, en laissant vides les définitions de 'node default' et 'node agent'. Ajoutons également une définition de nœud vide pour le master au cas où nous voudrions le classer de manière unique plus tard. Enfin, réorganisons également le code dans un ordre plus intuitif. Cela ne fait aucune différence pour Puppet, mais je pense que c'est plus intuitif pour les humains de cette façon:

```
#
# Global - All code outside a node definition gets applied to all nodes
#

include common_packages
include common_hosts

#
# Node-specific
#

node 'puppet.example.com' {
}

node 'agent.example.com' {
}

#
# Default - if no other Node-specific definition matched
#

node default {
}
```

La prochaine fois que puppet s'exécutera sur le Master, vous remarquerez que ces packages sont installés.

Portée de premier niveau

Remarque: les définitions de nœuds ne peuvent être effectuées qu'au niveau supérieur dans site.pp

Top-Scope est une autre façon de dire **Globalement visible**.

Toute variable déclarée dans le `site.pp` est automatiquement une variable de **Top-Scope**. Les variables de **Top-Scope** sont accessibles dans tout votre code Puppet en les référençant avec `$::varname`

Affectant les valeurs par défaut des paramètres de ressource

Autre chose pour simplifier le code de notre classe common_packages: Notez que nous avons plusieurs packages, et que nous faisons le "ensure => 'installed'" pour tous. Pour réduire le code dupliqué, nous pourrions définir une valeur d'attribut par défaut pour le type de package comme ceci:

```
Package { ensure => 'installed' }
```

Cette valeur par défaut s'appliquerait à toute autre ressource de package que nous déclarons dans le manifest. Nous pouvons toujours remplacer la valeur par défaut par ressource si nous le souhaitons. Notre code pourrait donc ressembler à ceci:

```
class common_packages {

  Package { ensure => 'installed' }

  package { 'bind-utils': }
  package { 'dstat': }
  package { 'git': }
  package { 'tcpdump': }
  package { 'net-tools': }

}
```

Notez qu'il existe une nette différence lorsque vous utilisez la version minuscule du type de ressource, par rapport à la version majuscule. Les minuscules déclarent la ressource, tandis que les majuscules font référence au type de ressource, mais ne le déclarent PAS.

Quand on dit:

```
   Package { ensure => 'installed' }
```

... ce que nous disons vraiment, c'est: pour chaque ressource de package déclarée, prenez ces attributs par défaut à moins qu'ils ne soient remplacés par la déclaration de ressource individuelle.

Nous pourrions même mettre le code `Package ensure installed` dans le site.pp, et ce serait alors une valeur par défaut globale pour les ressources du package partout.

Nous pouvons remplacer la valeur par défaut que nous venons de définir en spécifiant la chaîne de version dans le paramètre ensure. Exemple d'épinglage à une version spécifique d'un package:

```
if ( $::operatingsystemmajrelease == '6' ) {
  package { 'nc': ensure => '1.84-24.el6' }
  package { 'nmap': ensure => '5.51-4.el6' }
}

if ( $::operatingsystemmajrelease == '7' ) {
  package { 'whois': ensure => '5.1.1-2.el7' }
}
```

Nous avons également ajouté une instruction conditionnelle qui vérifie le **fact** de la **version majeure du système d'exploitation**. Si nous exécutons un système EL6, les packages **nc** et **nmap** seront installés dans la version spécifique que nous avons spécifiée. Si nous exécutons un système EL7, le package **whois** sera installé dans la version spécifique que nous avons spécifiée. (Tout cela suppose que les versions que nous avons spécifiées sont disponibles dans notre référentiel de packages.)

La chose légèrement ennuyeuse ici est que si vous mainteniez ce package sur plusieurs plates-formes et versions, vous devrez gérer les chaînes de version pour chacune. Par exemple, pour le package **whois**, nous pourrions avoir plusieurs versions différentes pour chaque plateforme:

```
5.1.0-1.el5
5.1.1-3.el6
5.1.1-2.el7
```

Il est ennuyeux que la plate-forme (el5, el6 ou el7) soit incluse dans la release par exemple. Si vous ne l'incluez pas, Puppet ne reconnaîtra pas la version. Il serait préférable de dire simplement '5.1.1' et de demander au provider de déterminer si nous exécutons cette version, mais cela ne nous aiderait pas si la version '5.1.1' n'est pas disponible sur EL5 , et nous voulons y installer '5.1.0'.