Brief introduction to Puppet

Scott Hazelhurst

Wits Bioinformatics

March 2013
Puppet

System/language for configuration of computer systems:
- Helps installation of new systems
- Simplifies rolling of new software
- Way of documenting system configuration

Basic structure
- *Master* has a description of each computer under control (e.g., installed software, configuration files, ...)
- *Clients* contact master to get their configuration – local puppet process implements
Aside – system installation

Ideally, couple use of Puppet with a system like Cobbler for automating system installation
Puppet is a declarative language:

- Describes the *state* the system should be in, rather than how to get there.
- Ruby-based, OO
- Extensible
- Under rapid development
  Version 2.6 discussed here – major changes in 2.7 and 3
Facter

*Facter* is a complementary package to Puppet

- Puppet uses Facter to determine facts about state of client machine
- Can be used from command-line: gives a uniform way of finding info about system

```
facter fqdn
facter operatingsystem
facter kernelrelease
```

These facts are available in your Puppet programs to guide configuration
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
- Update cycle
  - Client sends update request to master
  - Master sends information to client
  - Client updates itself
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
- Update cycle
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
- Update cycle
  - Client sends update request to master
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
- Update cycle
  - Client sends update request to master
  - Master sends information to client
Overview of puppet process

- Run puppet master daemon on server
- Client generates cert and identifies itself to master
- Master authorises and signs certificate
- Update cycle
  - Client sends update request to master
  - Master sends information to client
  - Client updates itself
Puppet setup on master

In /etc/puppet/manifests main file is site.pp

import "classes/*.pp"
import "groups/virt_groups.pp"
import "groups/global_special.pp"
import "users/virt_users.pp"
import "users_groups.pp"

import "nodes.pp"
In nodes.pp I have declarations for each node

$storagehome="homestore.core.wits.ac.za"

node 'cloud01' { include ubuntu }

node 'n02.core.wits.ac.za' { include worker }

node 'n01.core.wits.ac.za' {
    $bricka01_1 = "/dev/sdc1"

    include worker
    include atlassudoers
    include glustervolA01
}
And the *worker* class is defined as:

```plaintext
class worker {
    tag(gluster-mount)
    tag(mount-data)
    tag(want-sshconfig)

    include slnode
    include datasrc

    ... resource declarations ... 
}
```

*worker* is a subclass of *sl* is a subclass of *base*
Puppet resources

*Resources* are the building blocks of configuration
Puppet resources

*Resources* are the building blocks of configuration

- file
Puppet resources

*Resources* are the building blocks of configuration

- file
- package
Puppet resources

(Resources are the building blocks of configuration)

- file
- package
- cron
- host
- mount
- user / group
- ssh_authorised_key

Extensible in many ways
file {gluster-repo:
    name => "/etc/yum.repos.d/glusterfs-epel.repo",
    content => file("/etc/puppet/sources/repos/glusterfs-epel.repo")
}

file {check:
    name => "/root/os",
    content => $operatingsystem
}

file {sysconf:
    name => "/root/sysconfig",
    mode => 640,
    owner => root,
    group => admin,
    ensure => directory,
}

package {'gnuplot' : ensure => installed }

And

$plist = ["emi-wn","emi-torque-client","emi-mpi","texlive-utils"]

package {
    $plist : ensure => installed,
}


NB: execution of resources not determined by order in file. May need to specify explicitly

package{"sagrid-repo":
  ensure=>installed,
  require=>File["sagrid"]
}

package{"sagrid-gpg":
  ensure=>installed,
  require=>Package["sagrid-repo"]
}
if !tagged(emi-root) {

    mount { glite-wits-mount:
            name => "/opt/glite/yaim/etc/wits",
            atboot => true,
            fstype => nfs,
            ensure => present,
            options => 'rw,auto',
            device => "${storagehome:/opt/glite/yaim/etc/wits}",
            require => [Package[nfs-utils],File[glite-wits]]
        }

    }
}
host { "cream-ce.core.wits.ac.za":
    ensure => present,
    ip => "146.141.240.90"
}

cron { firewall:
    command => "/opt/sysconfig/filter/naughty",
    user => root,
    hour => 2,
    minute => 0
}
Exec

Can execute any process on the underlying machine

- Very powerful, flexible
- Can be overused — other resources are essentially *idempotent* in effect (benefit of a declarative model)
- With exec, must be very careful
- May also be unnecessary warning statements!

Try to use other puppet features if you can.
exec {emi-rpm-key:
    path => ["/usr/bin","/bin"],
    command => "rpm --import /root/sysconfig/RPM-GPG-KEY"}
exec {emi-rpm-key:
    path => ["/usr/bin","/bin"],
    command => "rpm --import /root/sysconfig/RPM-GPG-KEY",
    subscribe => File[emi-rpm-gpg-key],
    refreshonly => true,
    require => File[emi-rpm-gpg-key]
}
exec{perl-DBD-SQLite:
    command => "'/usr/bin/yum -y --enablerepo=epel install perl-DBD-SQLite'",
    path => ["/bin","/usr/bin"],
    onlyif  => "test -n 'rpm -aq perl-DBD-SQLite'" }
}
Virtual resources

Puppet has strict requirement that when client gets resources, resource declared in one place only

- Mostly OK – hierarchical class structure fits well
- But some resources are not so easy because they may be required in independent classes.

User/group management good example:

- Want to define users, groups in one or two places
- May have very varied needs of where users are defined
Solution: declare resource as virtual by using @prefix

@user { "dduck" :
  home => "/home/dduck",
  uid => "28021",
  gid => "28021",
  comment => "Donald Duck",
  require => Group["dduck"],
  ensure => present
}
Then, where you want the user, use realize

```plaintext
node 'testserver' {

    $users=["dduck","mmouse","spiderman"]

    realize [User[$users], Group[$users]]

}
```
Others

case $operatingsystem { 
  centos, redhat: { $service_name = 'ntpd' } 
  debian, ubuntu: { $service_name = 'ntp' } 
}

package { 'ntp':
  ensure => installed,
}

service { 'ntp':
  name => $service_name,
  ensure => running,
  enable => true,
  subscribe => File['ntp.conf'],
}