# Puppet Tutorial

*By Alessandro Franceschi*
*Example42*
*11-07-2015*

# Puppet Essentials -

# Overview

## Introduction to Puppet

## Configuration management tools

## Puppet Ecosystem and related software

# What is Puppet

A **Configuration Management** Tool

A framework for **Systems Automation**

A Declarative Domain Specific Language (**DSL**)

An **OpenSource software** in written Ruby

Works on Linux, Unix (Solaris, AIX, *BSD), MacOS, Windows (Supported Platforms)

Developed by **Puppet Labs**

Used by http://puppetlabs.com/customers/companies/ ...

... and many others

... and **many** others

# Configuration Management advantages

**Infrastructure as Code**: Track, Test, Deploy, Reproduce, Scale

Code commits log shows the **history of change** on the infrastructure

**Reproducible setups**: Do once, repeat forever

**Scale** quickly: Done for one, use on many

**Coherent** and consistent server setups

**Aligned Environments** for devel, test, qa, prod nodes

Alternatives to Puppet: Chef, CFEngine, Salt, Ansible

# References and

# Ecosystem

Puppet Labs - The Company behind Puppet

Puppet - The OpenSource version

Puppet Enterprise - The commercial version

The Community - Active and vibrant

Puppet Documentation - Main and Official reference

Puppet Modules on: Module Forge and GitHub

# Software related to Puppet:

Facter - Complementary tool to retrieve system's data

MCollective - Infrastructure Orchestration framework

Hiera - Key-value lookup tool where Puppet data can be placed

PuppetDB - Stores all the data generated by Puppet

Puppet DashBoard - A Puppet *Web frontend* and

Puppet DashBoard - A Puppet *Web frontend* and External Node Classifier (ENC)

The Foreman - A well-known third party provisioning tool and Puppet ENC

Geppetto - A Puppet IDE based on Eclipse

# Installation

Debian, Ubuntu
Available by default

```
apt-
get install puppet       # On clients (n
odes)
apt-
get install puppetmaster # On server (ma
ster)
```

RedHat, Centos, Fedora
Add EPEL repository or RHN Extra channel

```
yum install puppet        # On clients (
nodes)
yum install puppet-
server # On server (master)
```

Use PuppetLabs repositories for latest updates

Installation Instructions for different OS

# Puppet Versions

From version 2 Puppet follows Semantic Versioning standards to manage versioning, with a pattern like: MAJOR.MINOR.PATCH

This implies that MAYOR versions might not be backwards compatible, MINOR versions may introduce new features keeping compatibility and PATCHes are for backwards compatible bug fixes.

This is the history of the main Puppet versions and some relevant changes
Check also Puppet Language History for a more complete review:

0.9.x - First public beta

0.10.x - 0.21.x - Various "early" versions

0.22.x - 0.25.x - Improvements and OS wider support

2.6.x - Many changes. Parametrized classes

2.7.x - Windows support

3.0.x - Many changes. Disabled dynamic variables scope. Data bindings

3.2.x - Future parser (experimental, will be default in Puppet 4)

4.x.x - Release in early 2015. New parser, new type system and lot of changes, some backwards incompatibilities

# Language Basics - Overview

## Introduction to Puppet language and terms

## Resource types

## Classes and defines

## Variables and parameters

# Puppet Language

A **Declarative** Domain Specific Language (DSL)

It defines **STATES** (Not procedures)

Puppet code is written in **manifests** (files with **.pp** extension)

In the code we declare **resources** that affect elements of the system (files, packages, services ...)

Resources are grouped in **classes** which may expose parameters that affect their behavior.

Classes and configuration files are organized in **modules**.

Consult the **official glossary** to give the correct meaning to Puppet terms

# Nodes classification

When clients connect, the Puppet Master generates a

**catalog** with the list of of the resources that clients have to apply locally.

The Puppet Master has to *classify* nodes and define for each of them:

The **classes** to include
The **parameters** to pass
The Puppet **environment** to use

The **catalog** is generated by the Master according to the logic of our Puppet code and data.

In our code we can define our **variables** and use other ones that may come from different sources:

**facts** generated directly by the client
**parameters** obtained from node's classification
Puppet **internal** variables

# Resource Types (Types)

Resource Types are single **units of configuration** composed by:

A **type** (package, service, file, user, mount, exec ...)

A **title** (how is called and referred)

Zero or more **arguments**

```
type { 'title':
  argument  => value,
  other_arg => value,
}
```

Example for a **file** resource type:

```
file { 'motd':
  path    => '/etc/motd',
  content => 'Tomorrow is another day',
}
```

# Resource Types reference

Find online the complete Type Reference for the latest or earlier versions.

From the shell the command line interface:

```
puppet describe file
```

For the full list of available descriptions try:

```
puppet describe --list
```

Give a glance to Puppet code for the list of **native** resource types:

```
ls $(facter rubysitedir)/puppet/type
```

# Simple samples of resources

Installation of OpenSSH package

```
package { 'openssh':
  ensure => present,
}
```

Creation of /etc/motd file

```
file { 'motd':
  path => '/etc/motd',
}
```

Start of httpd service

```
service { 'httpd':
  ensure => running,
  enable => true,
}
```

# More complex samples of resources

Management of nginx service with parameters defined in module's variables

```
service { 'nginx':
  ensure     => $::nginx::manage_service
_ensure,
  name       => $::nginx::service_name,
  enable     => $::nginx::manage_service
_enable,
}
```

Creation of nginx.conf with content retrieved from different sources (first found is served)

```
file { 'nginx.conf':
  ensure  => present,
  path    => '/etc/nginx/nginx.conf',
  source  => [
      "puppet:///modules/site/nginx.conf
--${::fqdn}",
      "puppet:///modules/site/nginx.conf
" ],
}
```

# Resource Abstraction Layer

Resources are abstracted from the underlying OS

Resource **types** have different providers for different OS

The **package** type is known for the great number of **providers**

```
ls $(facter rubysitedir)/puppet/provider
/package
```

Use **puppet resource** to interrogate the RAL:

```
puppet resource user

puppet resource user root

puppet resource package

puppet resource service
```

Or to directly modify resources:

```
puppet resource service httpd ensure=run
ning enable=true
```

# Classes - Definition

Classes are **containers** of different resources. Since Puppet 2.6 they can have parameters

Example of a class **definition**:

```
class mysql (
  root_password => 'default_value',
  port          => '3306',
) {
  package { 'mysql-server':
    ensure => present,
  }
  service { 'mysql':
    ensure    => running,
  }
  [...]
}
```

Note that when we define a class we just describe what it does and what parameters it has, we don't actually add it and its resources to the catalog.

# Classes - Declaration

When we have to use a class previously defined, we **declare** it.

This can be done in 2 different ways:

"Old style" class declaration, without parameters:

```
include mysql
```

(Inside a catalog we can have multiple includes of the same class but that class it's applied only once.)

"New style" (from Puppet 2.6) class declaration with explicit parameters:

```
class { 'mysql':
  root_password => 'my_value',
  port          => '3307',
}
```

(Syntax is the same of normal resources and the same class can be declared, in this way, only once inside the same catalog)

# Defines

Also called: **Defined resource types** or **defined types**

Similar to parametrized classes but can be used multiple times (with different titles).

**Definition** of a define:

```
define apache::virtualhost (
  $ensure   = present,
  $template = 'apache/virtualhost.conf.e
rb' ,
  [...] ) {

  file { "ApacheVirtualHost_${name}":
    ensure  => $ensure,
    content => template("${template}"),
  }
}
```

**Declaration** of a define:

```
apache::virtualhost { 'www.example42.com
':
  template => 'site/apache/www.example42
.com-erb'
}
```

# Variables

We need them to provide different configurations for different kind of servers.

They can be defined in different places and by different actors:

Can be provided by client nodes as **facts**

Can be **defined by users** in Puppet code, on Hiera on in the ENC

Can be **built-in** and be provided directly by Puppet

## Facts

**Facter** runs on clients and collects **facts** that the server can use as variables

```
al$ facter

architecture => x86_64
fqdn => Macante.example42.com
hostname => Macante
interfaces => lo0,eth0
ipaddress => 10.42.42.98
```

```
ipaddress_eth0 => 10.42.42.98
kernel => Linux
macaddress => 20:c9:d0:44:61:57
macaddress_eth0 => 20:c9:d0:44:61:57
memorytotal => 16.00 GB
netmask => 255.255.255.0
operatingsystem => Centos
operatingsystemrelease => 6.3
osfamily => RedHat
virtual => physical
```

# User Variables

We can define custom variables in different ways:

## In Puppet manifests:

```
$role = 'mail'

    $package = $::operatingsystem ? {
      /(?
i:Ubuntu|Debian|Mint)/ => 'apache2',
      default                   => 'http
d',
    }
```

### In an External Node Classifier (ENC)

Commonly used ENC are Puppet DashBoard, the Foreman, Puppet Enterprise.

### In an Hiera backend

```
$syslog_server = hiera(syslog_server)
```

# Built-in variables

Puppet provides some useful built in variables, they can be:

## Set by the client (agent)

`$clientcert` - the name of the node's certificate. By default its `$::fqdn`
`$clientversion` - the Puppet version on the client

## Set by the server (master)

`$environment` (default: production) - the Puppet environment where are placed modules and manifests.
`$servername`, `$serverip` - the Puppet Master FQDN and IP

`$serverversion` - the Puppet version on the server
`$settings::<name>` - any configuration setting on the Master's `puppet.conf`

### Set by the server during catalog compilation

`$module_name` - the name of the module that contains the current resource's definition
`$caller_module_name` - the name of the module that contains the current resource's declaration

# Environments

Puppet environments allow isolation of Puppet code and data: for each environment we can have different paths manifest files, Hiera data and modules.

Puppet's environments DO NOT necessarily have to match the operational environments of our servers.

Environments mamanagement has changed from Puppet 3.6 onwards:

Earlier versions were based on so-called **Config file Environments** where each environment had to be

**Environments** where each environment had to be defined in **puppet.conf**.

From 3.6 **directory environments** have been introduced and the older approach has been deprecated.

On Puppet 4.x only directory environments are supported.

# Config file Environments

The "old" config file environments are defined inside `puppet.conf` with a syntax like:

```
[test]
   modulepath = $confdir/environments/test/modules:$condfir/modules
   manifest = $confdir/environments/test/manifests
```

For the default **production** environment there were the config parameters, now deprecated, `manifest` and `modulepath` in the `[main]` section.

modulepath in the [main] section.

# Directory Environments

Directory Environments are configured in `puppet.conf` as follows:

```
[main]
environmentpath = $configdir/
```

Then inside the `/etc/puppet/environments/$environment/` directory we have:

```
modules/   # Directory containing module
s
manifests/ # Directory containing site.p
p
environment.conf # Conf file for the env
ironment
```

# Nodes - Default classification

A node is identified by the PuppetMaster by its **certname**, which defaults to the node's **fqdn**

In the first manifest file parsed by the Master, `site.pp`, we can define nodes with a syntax like:

```
node 'web01' {
  include apache
}
```

We can also define a list of matching names:

```
node 'web01' , 'web02' , 'web03' {
  include apache
}
```

or use a regular expression:

```
node /^www\d+$/ {
  include apache
}
```

A node can inherit another node and include all the classes and variables defined for it, this feature is now deprecated and is not supported anymore on Puppet 4.

# Nodes classification via an ENC

Puppet can query an external source to retrieve the classes and the parameters to assign to a node. This source is called External Node Classifier (ENC) and can be anything that, when interrogated via a script with the clients' certname as first parameter, returns a yaml file with the list of classes and parameters.

Common ENC are **Puppet DashBoard**, **The Foreman** and **Puppet Enterprise** (where the functionality of ENC is enabled by default).

To enable the usage of an ENC set these parameters in `puppet.conf`

```
# Enable the usage of a script to classi
fy nodes
node_terminus = exec

# Path of the script to execute to class
ify nodes
external_nodes = /etc/puppet/node.rb
```

# Nodes classification with Hiera

Hiera provides a **hiera_include** function that allows the inclusion of classes as defined on Hiera. This is an approach that can be useful when there's massive usage of Hiera as backend for Puppet data.

In `/etc/puppet/manifests/site.pp` just place:

```
hiera_include('classes')
```

and place, as an array, the classes to include in our Hiera data source under the key `classes`.

## The Catalog

## The Catalog

The **catalog** is the complete list of resources, and their relationships, that the Puppet Master generates for the client.

It's the result of all the puppet code and logic that we define for a given node in our manifests and is applied on the client after it has been received from the master.

The client uses the RAL (Resource Abstraction Layer) to execute the actual system's commands that convert abstract resources like

```
package { 'openssh': }
```

to their actual fullfillment on the system (`apt-get install openssh` , `yum install openssh` ...).

The catalog is saved by the client in `/var/lib/puppet/client_data/catalog/$certname.json`

# Language Basics - Practice

On a test machine **install Puppet** if not already installed.

Practice with the **commands**:

puppet describe
puppet resource
facter

Create a **manifest** file and in that file manage the installation of the package and the management of the service of *nginx*.
Use `puppet apply` to apply the resources declared in our manifest.

# Puppet Configuration and basic usage - Overview

---

## Operational modes: apply / agent

## Configuration files and options

## Useful Paths

# Anatomy of a Puppet run

## Operational modes

## Masterless - puppet apply

Our Puppet code (written in manifests) is applied directly on the target system.

No need of a complete client-server infrastructure.

Have to distribute manifests and modules to the managed nodes.

Command used: `puppet apply` (generally as root)

## Master / Client - puppet agent

We have clients, our managed nodes, where Puppet client is installed.

And we have one or more Masters where Puppet server runs as a service

Client/Server communication is via https (**port 8140**)

Clients certificates have to be accepted (**signed**) on the Master

Command used on the client: `puppet agent` (generally as **root**)

Command used on the server: `puppet master` (generally as **puppet**)

# Masterless setup

Puppet manifests are deployed directly on nodes and applied locally:

```
puppet apply --
modulepath /modules/ /manifests/file.pp
```

More fine grained control on what goes in production for what nodes

Ability to trigger multiple truly parallel Puppet runs

No single point of failure, no Master performance issues

Need to define a fitting deployment workflow. Hints: Rump - supply_drop

With Puppet > 2.6 we can use file sources urls like:

```
puppet:///modules/example42/apache/vhost
.conf

# they point to
$modulepath/example42/apache/vhost.conf
```

StoreConfigs usage require access to Puppet DB from every node

# Master / Client Setup

A Puppet server (running as 'puppet') listening on 8140 on the Puppet Master (the server )

A Puppet client (running as 'root') on each managed node

node

Client can be run as a service (default), via cron (with random delays), manually or via MCollective

Client and Server have to share SSL certificates. New client certificates must be signed by the Master CA

It's possible to enable automatic clients certificates signing on the Master (be aware of security concerns)

## Certificates management

On the Master we can use `puppet cert` to manage certificates

List the (client) certificates to sign:

```
puppet cert list
```

List all certificates: signed (+), revoked (-), to sign ( ):

```
puppet cert list --all
```

Sign a client certificate:

```
puppet cert sign <certname>
```

Remove a client certificate:

```
puppet cert clean <certname>
```

Client stores its certificates and the server's public one in `$vardir/ssl**` (`/var/lib/puppet/ssl` on Puppet OpenSource)

Server stores clients public certificates and in `$vardir/ssl/ca` (`/var/lib/puppet/ssl/ca`). DO NOT remove this directory.

# Certificates management - First run

By default the first Puppet run on a client fails:

```
client # puppet agent -t
       # "Exiting; no certificate found
and waitforcert is disabled"
```

An optional `--waitforcert 60` parameter makes client wait 60 seconds before giving up.

The server has received the client's CSR which has to

The server has received the client's CSR which has to be manually signed:

```
server # puppet cert sign <certname>
```

Once signed on the Master, the client can connect and receive its catalog:

```
client # puppet agent -t
```

If we have issues with certificates (reinstalled client or other certs related problemes):

Be sure client and server times are synced

Clean up the client certificate. On the client remove it:

```
client # mv /var/lib/puppet/ssl /var/lib
/puppet/ssl.old
```

On the Master clean the old client certificate:

```
server # puppet cert clean <certname>
```

# Puppet configuration: puppet.conf

It's Puppet main configuration file.
On opensource Puppet is generally in:

```
/etc/puppet/puppet.conf
```

On Puppet Enterprise:

```
/etc/puppetlabs/puppet/puppet.conf
```

When running as a normal user can be placed in the home directory:

```
/home/user/.puppet/puppet.conf
```

Configurations are divided in [stanzas] for different Puppet sub commands

Common for all commands: **[main]**

For puppet agent (client): **[agent]** (Was [puppetd] in Puppet pre 2.6)

For puppet apply (client): **[user]** (Was [puppet])

For puppet master (server): **[master]** (Was [puppetmasterd] and [puppetca])

Hash sign (#) can be used for comments.

# Main configuration options

To view all or a specific configuration setting:

```
puppet config print all
puppet config print modulepath
```

## Important options under [main] section:

**vardir**: Path where Puppet stores dynamic data.

**ssldir**: Path where SSL certifications are stored.

## Under [agent] section:

**server**: Host name of the PuppetMaster. (Default:

puppet)

**certname**: Certificate name used by the client. (Default is its fqdn)

**runinterval**: Number of minutes between Puppet runs, when running as service. (Default: 30)

**report**: If to send Puppet runs' reports to the **report_server. (Default: true)

## Under [master] section:

**autosign**: If new clients certificates are automatically signed. (Default: false)

**reports**: How to manage clients' reports (Default: store)

**storeconfigs**: If to enable store configs to support exported resources. (Default: false)

Full configuration reference on the official site.

# Common command-line parameters

All configuration options can be overriden by command-line options.

A very common option used when we want to see immediately the effect of a Puppet run (it's actually the combination of: -onetime, -verbose, -ignorecache, -no-daemonize, -no-usecacheonfailure, -detailed-exit-codes, -no-splay, and -show_diff):

```
puppet agent --
test # Can be abbreviate to -t
```

Run puppet agent in foreground and debug mode:

```
puppet agent --test --debug
```

Run a dry-run puppet without making any change to the system:

```
puppet agent --test --noop
```

Run puppet using an environment different from the default one:

```
puppet agent --test --
environment testing
```

Wait for certificate approval (by default 120 seconds) in the first Puppet run (useful during automated first fime installation if PuppetMaster's autosign is false):

```
puppet agent --test --waitforcert 120
```

# Useful paths

**/var/log/puppet** contains logs (but also on normal syslog files, with facility daemon), both for agents and master

**/var/lib/puppet** contains Puppet operational data (catalog, certs, backup of files...)

**/var/lib/puppet/ssl** contains SSL certificate

**/var/lib/puppet/clientbucket** contains backup copies of the files changed by Puppet

**/etc/puppet/manifests/site.pp** (On Master) The first manifest that the master parses when a client connects in order to produce the configuration to apply to it (Default on Puppet < 3.6 where are used **config-file environments**)

**/etc/puppet/environments/production/manifests/site.pp** (On Master) The first manifest that the master parses when using **directory environments** (recommended from Puppet 3.6 and default on Puppt >= 4)

**/etc/puppet/modules** and **/usr/share/puppet/modules** (On Master) The default directories where modules are searched

**/etc/puppet/environments/production/modules** (On Master) An extra place where modules are looked for when using **directory environments**

# Other configuration

# files:

## auth.conf

Defines ACLs to access Puppet's REST interface. Details

## fileserver.conf

Used to manage ACL on files served from sources different than modules Details

## puppetdb.conf

Settings for connection to PuppetDB, if used. Details

## tagmail.conf , autosign.conf , device.conf , routes.yaml

These are other configuration files for specific functions. Details

## /etc/puppet/environments/production/environment

Contains environment specific settings

## Anatomy of a Puppet Run - Part 1: Catalog

# Run - Part II catalog compilation

Execute Puppet on the client

Client shell # puppet agent -t

If pluginsync = true (default from Puppet 3.0) the client retrieves all extra plugins (facts, types and providers) present in modules on the server's $modulepath

Client output # Info: Retrieving plugin

The client runs facter and send its facts to the server

Client output # Info: Loading facts in /var/lib/puppet/lib/facter/... [...]

The server looks for the client's hostname (or certname, if different from the hostname) and looks into its nodes list

The server compiles the catalog for the client using also client's facts

Server's logs # Compiled catalog for in environment production in 8.22 seconds

If there are not syntax errors in the processed Puppet code, the server sends the catalog to the client, in PSON format.

# Anatomy of a Puppet Run - Part 2: Catalog application

Client output # Info: Caching catalog for

The client receives the catalog and starts to apply it locally
If there are dependency loops the catalog can't be applied and the whole tun fails.

Client output # Info: Applying configuration version '1355353107'

All changes to the system are shown here. If there are errors (in red or pink, according to Puppet versions) they are relevant to specific resources but do not block the application of the other resources (unless they depend on the failed ones).

At the end ot the Puppet run the client sends to the server a report of what has been changed

Client output # Finished catalog run in 13.78 seconds

The server eventually sends the report to a Report Collector

# Puppet Configuration and basic usage - Practice

Use the command `puppet config print` to explore Puppet's configuration options.

Give a look to the various Puppet related directories and their contents:
`/etc/puppet`, `/var/lib/puppet`, `/var/log/puppet`

If the lab setup allows it, run Puppet on a new client and sign its certificate on the master.

Explore and describe the output of a Puppet run in apply and agent mode.

# Resources reference - Overview

## Where to find docs about Puppet resources

## Main resources: package, service, file, user, exec

## Language Style

# Online documentation on types

### Complete Type Reference

This is the complete reference for Puppet types based on the latest version.

Check here for the reference on all the older versions.

Check the Puppet Core Types Cheatsheet for an handy PDF with the essential reference.

# Inline documentation on types

### Types reference documentation

We can check all the types documentation via the command line:

```
puppet describe <type>
```

For a more compact output:

```
puppet describe -s <type>
```

To show all the available resource types:

```
puppet describe -l
```

### Inspect existing resource types

To interactively inspect and modify our system's resources

```
puppet resource <type> [name]
```

Remember we can use the same command to CHANGE our resources attibutes:

```
puppet resource <type> <name> [attribute
=value] [attribute2=value2]
```

# Managing packages

Installation of packages is managed by the **package** type.

The main arguments:

```
package { 'apache':
  name       => 'httpd',  # (namevar)
  ensure     => 'present' # Values: 'abse
nt', 'latest', '2.2.1'
  provider   => undef,    # Force an expl
icit provider
}
```

# Managing services

Management of services is via the **service** type.

The main arguments:

```
service { 'apache':
  name       => 'httpd',  # (namevar)
  ensure     => 'running' # Values: 'stop
ped', 'running'
  enable     => true,     # Define if to
enable service at boot (true|false)
  hasstatus => true,      # Whether to us
e the init script' status to check
                          # if the servic
e is running.
  pattern    => 'httpd',  # Name of the p
rocess to look for when hasstatus=false
}
```

## Managing files

Files are the most configured resources on a system, we manage them with the **file** type:

```
file { 'httpd.conf':
    # (namevar) The file path
    path      => '/etc/httpd/conf/httpd.
```

```
conf',
    # Define the file type and if it sho
uld exist:
    # 'present','absent','directory','li
nk'
    ensure    => 'present',
    # Url from where to retrieve the fil
e content
    source    => 'puppet://[puppetfilese
rver]/<share>/path',
    # Actual content of the file, altern
ative to source
    # Typically it contains a reference
to the template function
    content   => 'My content',
    # Typical file's attributes
    owner     => 'root',
    group     => 'root',
    mode      => '0644',
    # The sylink target, when ensure =>
link
    target    => '/etc/httpd/httpd.conf'
,
    # Whether to recursively manage a di
rectory (when ensure => directory)
    recurse   => true,
}
```

# Executing commands

We can run plain commands using Puppet's **exec** type. Since Puppet applies it at every run, either the command can be safely run multiple times or we have to use one of the **creates**, **unless**, **onlyif**, **refreshonly** arguments to manage when to execute it.

```
exec { 'get_my_file':
    # (namevar) The command to execute
    command   => "wget http://mysite/myf
ile.tar.gz -O /tmp/myfile.tar.gz',
    # The search path for the command. M
ust exist when command is not absolute
    # Often set in Exec resource default
s
    path      => '/sbin:/bin:/usr/sbin:/
usr/bin',
    # A file created by the command. It
if exists, the command is not executed
    creates   => '/tmp/myfile.tar.gz',
    # A command or an array of commands,
 if any of them returns an error
    # the command is not executed
    onlyif    => 'ls /tmp/myfile.tar.gz
&& false',
    # A command or an array of commands,
 if any of them returns an error
    # the command IS executed
    unless    => 'ls /tmp/myfile.tar.gz'
,
}
```

# Managing users

Puppet has native types to manage users and groups, allowing easy addition, modification and removal. Here are the main arguments of the **user** type:

```
user { 'joe':
    # (namevar) The user name
    name      => 'joe',
    # The user's status: 'present','absent','role'
    ensure    => 'present',
    # The user's  id
    uid       => '1001',
    # The user's primary group id
    gid       => '1001',
    # Eventual user's secondary groups (use array for many)
    groups    => [ 'admins' , 'developers' ],
    # The user's password. As it appears in /etc/shadow
    # Use single quotes to avoid unanted evaluation of $* as variables
    password  => '$6$ZFS5JFFRZc$FFDSvPZSSFGVdXDlHe',
    # Typical users' attributes
    shell     => '/bin/bash',
    home      => '/home/joe',
    mode      => '0644',
}
```

# Puppet language style

Puppet language as an official Style Guide which defines recommended rules on how to write the code.

The standard de facto tool to check the code style is puppet-lint.

Badly formatted example:

```
file {
 "/etc/issue":
     content => "Welcome to $fqdn",
     ensure => present,
      mode => 644,
     group => "root",
     path => "${issue_file_path}",
     }
```

Corrected style:

```
file { '/etc/issue':
  ensure  => present,
  content => "Welcome to ${fqdn}",
  mode    => 0644,
```

```
  group   => 'root',
  path    => $issue_file_path,
}
```

# Resources reference - Practice

Create a manifest that contains all the explored resources: package, service, file, exec, user.

Use Pupppet Lint to verify its style.

# Language reference - Overview

## Referencing a resource

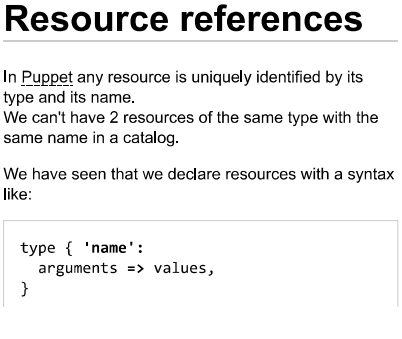
## Managing resources ordering

## Metaparameters

## Conditionals and

## Comparison operators

## Nodes and classes inheritance



### Resource references

In Puppet any resource is uniquely identified by its type and its name.
We can't have 2 resources of the same type with the same name in a catalog.

We have seen that we declare resources with a syntax like:

```
type { 'name':
  arguments => values,
}
```

When we need to reference to them in our code the syntax is like:

```
Type['name']
```

Some examples:

```
file { 'motd': ... }
apache::virtualhost { 'example42.com': .
... }
exec { 'download_myapp': .... }
```

are referenced, respectively, with

```
File['motd']
Apache::Virtualhost['example42.com']
Exec['download_myapp']
```

# Resource defaults

It's possible to set default argument values for a resource in order to reduce code duplication. The syntax is:

```
Type {
  argument => value,
}
```

Common examples:

```
Exec {
  path => '/sbin:/bin:/usr/sbin:/usr/bin
',
}

File {
  mode  => 0644,
  owner => 'root',
  group => 'root',
}
```

Resource defaults can be overriden when declaring a specific resource of the same type.

Note that the "Area of Effect" of resource defaults might bring unexpected results. The general suggestion is:

Place **global** resource defaults in /etc/pupept/manifests/site.pp outside any node definition.

Place **local** resource defaults at the beginning of a class that uses them (mostly for clarity sake, as they are parse-order independent).

# Nodes inheritance

On the PuppetMaster we can define with the **node** definition the resources to apply to any node.

It is possible to have an inheritance structure for nodes, so that resources defined for a node are automatically included in an inheriting node.

An example:

```
node 'general' { ... }

node 'www01' inherits general { ... }
```

In Puppet versions prior to 3, it was possible to use nodes inheritance also to set variables and override them at different inheritance levels, and refer to these variables with their "short" name (not fully qualified). When using this approach it was important to avoid the

When using this approach it was important to avoid the inclusion on classes in the inheritance tree, since some variables could be evaluated in an unexpected way.

This is no more possible because variables are not more dynamically scoped, and generally speaking nodes inheritance has been deprecated.

# Class inheritance

In Puppet classes are just containers of resources and have nothing to do with OOP classes. Therefore the meaning of class inheritance is somehow limited to few specific cases.

When using class inheritance, the main class ('puppet' in the sample below) is always evaluated first and all the variables and resource defaults it sets are available in the scope of the child class ('puppet::server').

Moreover the child class can override the arguments of a resource defined in the main class. Note the syntax used when referring to the existing resource File['/etc/puppet/puppet.conf']:

```
class puppet {
```

```
  file { '/etc/puppet/puppet.conf':
    content => template('puppet/client/p
uppet.conf'),
  }
}

class puppet::server inherits puppet {
  File['/etc/puppet/puppet.conf'] {
    content => template('puppet/server/p
uppet.conf'),
  }
}
```

# Run Stages

In Puppet 2.6 it has been introduced the concept of Run Stages to help users in managing the order of dependencies when applying resources.

Puppet (> 2.6) provides a default **main** stage, we can

add any number or further stages with the stage resource type:

```
stage { 'pre':
  before => Stage['main'],
}
```

Which is equivalent to:

```
stage { 'pre': }
Stage['pre'] -> Stage['main']
```

We can assign any class to a defined stage with the stage metaparameter:

```
class { 'yum':
  stage => 'pre',
}
```

Do not abuse of run stages (be vary of dependency cycles)!

Official documentation on Run Stages

# Metaparameters

Metaparameters are parameters available to any resource type, they can be used for different purposes:

Manage dependencies (**before**, **require**, **subscribe**, **notify**, **stage**)

Manage resources' application policies (**audit**, **noop**, **schedule**, **loglevel**)

Add information to a resource (**alias**, **tag**)

Official documentation on Metaparameters

# Managing dependencies

Puppet language is declarative and not procedural: it defines states, the order in which resources are written in manifests does not affect the order in which they are applied to the desired state.

To manage resources ordering, there are 3 different

methods, which can cohexist:

1 - Use the metaparameters **before**, **require**, **notify**, **subscribe**

2 - Use the **Chaining arrows** (compared to the above metaparamers: **->** , **<-** , **<~** , **~>**)

3 - Use run stages

## Managing dependencies - before | notify

In a typical Package/Service/Configuration file example we want the package to be installed first, configure it and then start the service, eventually managing its restart if the config file changes.

This can be expressed with metaparameters:

```
package { 'exim':
  before => File['exim.conf'],
}

file { 'exim.conf':
  notify => Service['exim'],
}

service { 'exim':
}
```

which is equivalent to

```
Package['exim'] -
> File['exim.conf'] ~> Service['exim']
```

# Managing dependencies - require | subscribe

The previous example can be expressed using the alternative reverse metaparameters:

```
package { 'exim':
}

file { 'exim.conf':
  require => Package['exim'],
}

service { 'exim':
  subscribe => File['exim.conf'],
}
```

which can also be expressed like:

```
Service['exim'] <~ File['exim.conf'] <-
Package['exim']
```

# Conditionals

Puppet provides different constructs to manage conditionals inside manifests:

**Selectors** allows to set the value of a variable or an argument inside a resource declaration according to the value of another variable.
Selectors therefore just returns values and are not used to manage conditionally entire blocks of code.

**case statements** are used to execute different blocks of code according to the values of a variable. It's possible to have a default block for unmatched entries.

Case statements are NOT used inside resource declarations.

**if elsif else** conditionals, like case, are used to execute different blocks of code and can't be used inside resources declarations.
We can can use any of Puppet's comparison expressions and we can combine more than one for complex patterns matching.

**unless** is somehow the opposite of **if**. It evaluates a boolean condition and if it's *false* it executes a block of code. It doesn't have elsif / else clauses.

## Sample: Assign a variable value

**Selector for variable's assignement**

```
$package_name = $osfamily ? {
   'RedHat' => 'httpd',
   'Debian' => 'apache2',
   default  => undef,
 }
```

**case**

```
case $::osfamily {
   'Debian': { $package_name = 'apache2'
 }
   'RedHat': { $package_name = 'httpd' }
   default: { notify { "Operating system
 $::operatingsystem not supported" } }
 }
```

**if elsif else**

```
if $::osfamily == 'Debian' {
   $package_name = 'apache2'
 } elsif $::osfamily == 'RedHat' {
   $package_name = 'httpd'
 } else {
   notify { "Operating system $::operati
ngsystem not supported" }
 }
```

# Comparing strings:

# operators

Puppet supports some common comparison operators: `== != < > <= >= =~ !~` and the somehow less common `in`

```
if $::osfamily == 'Debian' { [ ... ] }

 if $::kernel != 'Linux' { [ ... ] }

 if $::uptime_days > 365 { [ ... ] }

 if $::operatingsystemrelease <= 6 { [ .
.. ] }
```

### Comparing strings: in operator and combinations

### in operator

The in operator checks if a string present in another string, an array or in the keys of an hash

```
if '64' in $::architecture

 if $monitor_tool in [ 'nagios' , 'icing
a' , 'sensu' ]
```

### Expressions

# Expressions combinations

It's possible to combine multiple comparisons with **and** and **or**

```
if ($::osfamily == 'RedHat')
and ($::operatingsystemrelease == '5') {
   [ ... ]
}

if ($::osfamily == 'Debian') or ($::osfa
mily == 'RedHat') {
   [ ...]
}
```

# Exported resources

When we need to provide to an host informations about resources present in another host, we need **exported resources**: resources declared in the catalog of a node (based on its facts and variables) but applied (collected) on another node.

Resources are declared with the special @@ notation which marks them as exported so that they are not applied to the node where they are declared:

```
@@host { $::fqdn:
  ip  => $::ipaddress,
}

@@concat::fragment { "balance-fe-
${::hostname}",
  target  => '/etc/haproxy/haproxy.cfg',
  content => "server ${::hostname} ${::i
paddress} maxconn 5000"
  tag     => "balance-fe",
}
```

Once a catalog containing exported resources has been applied on a node and stored by the PuppetMaster (typically on PuppetDB), the exported resources can be collected with the spaceshift syntax (where is possible to specify search queries):

```
Host << || >>
Concat::Fragment <<| tag == "balance-
fe" |>>
Sshkey <<| |>>
Nagios_service <<||>>
```

# Exported resources - Configuration

In order to use exported resources we need to enable on the Puppet Master the **storeconfigs** option and specify the backend to use.
We can do this configuring a PuppetMaster to use PuppetDB:

```
storeconfigs = true
storeconfigs_backend = puppetdb
```

In earlier Puppet versions storeconfigs is based on ActiveRecord, which is considerable slower.

# Modules - Overview

## Modules structure and conventions

## Erb templates

## Patterns for modules reusability

## Principles of modules testing

## Modules documentation

### Modules

Self Contained and Distributable *recipes* contained in a directory with a predefined structure

Used to manage an application, system's resources, a local site or more complex structures

Modules must be placed in the Puppet Master's modulepath

```
puppet config print modulepath
/etc/puppet/modules:/usr/share/puppet/mo
dules
```

Puppet module tool to interface with Puppet Modules Forge

```
puppet help module
[...]
ACTIONS:
  build        Build a module release package.
  changes      Show modified files of an
 installed module.
  generate     Generate boilerplate for
a new module.
  install      Install a module from the
 Puppet Forge or an archive.
  list         List installed modules
  search       Search the Puppet Forge for a module.
  uninstall    Uninstall a puppet module
.
  upgrade      Upgrade a puppet module.
```

GitHub, also, is full of Puppet modules

# Paths of a module

Modules have a standard structure:

```
mysql/            # Main module director
y

mysql/manifests/  # Manifests directory.
 Puppet code here. Required.
mysql/lib/        # Plugins directory. R
uby code here
mysql/templates/  # ERB Templates direct
ory
mysql/files/      # Static files directo
ry
mysql/spec/       # Puppet-
rspec test directory
mysql/tests/      # Tests / Usage exampl
es directory

mysql/Modulefile  # Module's metadata de
scriptor
```

This layout enables useful conventions

# Modules paths conventions

Classes and defines autoloading:

```
include mysql
# Main mysql class is placed in: $module
path/mysql/manifests/init.pp

include mysql::server
# This class is defined in: $modulepath/
mysql/manifests/server.pp

mysql::conf { ...}
# This define is defined in: $modulepath
/mysql/manifests/conf.pp

include mysql::server::ha
# This class is defined in: $modulepath/
mysql/manifests/server/ha.pp
```

Provide files based on Erb Templates (Dynamic content)

```
content => template('mysql/my.cnf.erb'),
# Template is in: $modulepath/mysql/temp
lates/my.cnf.erb
```

Provide static files (Static content). Note we can't use content AND source for the same file.

```
source => 'puppet:///modules/mysql/my.cn
f'
# File is in: $modulepath/mysql/files/my
.cnf
```

# Erb templates

Files provisioned by Puppet can be Ruby ERB templates

In a template all the Puppet variables (facts or user assigned) can be used :

```
# File managed by Puppet on <%= @fqdn %>
search <%= @domain %>
```

But also more elaborated Ruby code

```
<% @dns_servers.each do |ns| %>
nameserver <%= ns %>
<% end %>
```

The computed template content is placed directly inside the catalog
(Sourced files, instead, are retrieved from the puppetmaster during catalog application)

## Principes behind a Reusable Module

Data Separation

```
- Configuration data is defined outside the module
- Module behavior can be managed via parameters
- Allow module's extension and override via external data
```

Reusability

```
- Support different OS. Easily allow new additions.
- Customize behavior without changing module code
- Do not force author idea on how configurations should be provided
```

Standardization

```
- Follow PuppetLabs style guidelines (puppet-lint)
- Have coherent, predictable and intuitive interfaces
- Provide contextual documentation (pupp
```

```
  Provide contextual documentation (pupp
et-doc)
```

Interoperability

```
- Limit cross-module dependencies
- Allow easy modules cherry picking
- Be self contained, do not interfere wi
th other modules' resources
```

# Modules reusability patterns: Template + Options hash

Check this blog post for details.

Classes and defines should expose parameters that allow to override the used templates and set a custom hash of configurations.

```
class redis (
  $config_file_template = 'redis/redis.c
onf.erb',
  $options_hash         = {},
) {
  file { '/etc/redis/redis.conf':
    content => template($config_file_tem
plate),
  }
}
```

# Template + Options hash example:

Given the previous class definition, we can configure it with this sample Hiera data, in YAML format:

```
---
redis::config_file_template: 'site/redis
/redis.conf.erb'
redis::options_hash:
  port: '12312'
  bind: '0.0.0.0'
  masterip: '10.0.42.50'
  masterport: '12350'
  slave: true
```

The referenced template stays in our site module, in `$modulepath/site/templates/redis/redis.conf.erb` and may look like:

```
port <%= @options_hash['port'] %>
bind <%= @options_hash['bind'] %>
<% if @options_hash['slave'] == true -%>
slaveof <%= @options_hash['masterip'] %>
 <%= @options_hash['masterport'] %>
<% end -%>
```

# Modules reusability patterns: Users override on included classes

Sometimes modules need some prerequisites or have to manage resources related to other applications or may provide the same functionality in diffret ways. Whenever this is needed, let users provide custom versions for the relevant classes.

This module manages the Puppet Master in a dedicated class an exposes a parameter that allows users to provide a custom class (or an alternative class eventually provided by the same module) instead of the default one.

```
class puppet (
  $server_class = '::puppet::server',
) {

  if $server_class {
    include $server_class
  }
}
```

Using Hiera we ca override with:

```
---
puppet::server_class: '::site::puppet::haserver'
```

This implies that we need to create in `$modulepath/site/manifests/puppet/haserver.pp` a class like:

```
class ::site::puppet::haserver {
  # The resources for our HA Puppet Master setup
}
```

# Testing Modules

Puppet code testing can be done at different levels

Puppet code testing can be done at different levels with different tools

**puppet parser validate <manifest.pp>** - Checks the syntax of a manifest

**puppet-lint <manifest.pp>** - A gem that checks the style of a manifest

**puppet-rspec** - A gem that runs rspec unit tests on a module (Based on compiled catalog)

**cucumber-puppet** - A gem that runs cucumber tests a module (Based on compiled catalog) OBSOLETE

**puppet-rspec-system** - A gem that creates Vagrant VM and check for the expected results of real Puppet runs

**Beaker** - A gem that runs acceptance tests on multiple Vagrant VM

## Modules documentation with Puppet Doc

Puppetdoc generates documentation from manifests comments:

```
$ puppet doc [--all] --mode rdoc [--
outputdir ] [--debug|--verbose] [--
trace]
  [--modulepath ] [--manifestdir ] [--
config ]
```

Comment classes as below:

```
# Class: apache
#
# This module manages Apache
#
# Parameters:
#
# Actions:
#
# Requires:
#
# [Remember: No empty lines between comm
ents and class definition]
class apache {
  ...
}
```

# Hiera - Overview

## Principles behind Hiera

## Configuring hierarchies

## Hiera plugins

## Using Hiera from Puppet and the CLI

# Introduction to Hiera

Hiera is the **key/value lookup tool** of reference where to store Puppet user data.

It provides an highly customizable way to lookup for parameters values based on a custom hierarchy using many different backends for data storage.

It provides a command line tool `hiera` that we can use to interrogate direclty the Hiera data and functions to be used inside Puppet manifests: `hiera()` , `hiera_array()` , `hiera_hash()` , `hiera_include()`

Hiera is installed by default with Puppet version 3 and

is available as separated download on earlier version (Installation instructions).

We need Hiera only on the PuppetMaster (or on any node, if we have a masterless setup).

Currently version 1 is available, here is its official documentation.

## Hiera configuration: hiera.yaml

Hiera's configuration file is in yaml format and is called **hiera.yaml** here we define the hierarchy we want to use and the backends where data is placed, with backend specific settings.

Hiera's configuration file path is different according to how it's invoked:

### From the Command Line and Ruby code

Default path: `/etc/hiera.yaml`

### From Puppet

Default path for Puppet OpenSource: `/etc/puppet/hiera.yaml`

Default path for Puppet Enterprise: `/etc/puppetlabs/puppet/hiera.yaml`

It's good practice the symlink these alternative configuration files in order to avoid inconsistencies when using Hiera from the shell line or within Puppet manifests:

```
ln -
s /etc/hiera.yaml /etc/puppet/hiera.yaml
```

## Default configuration

By default Hiera does not provide a configuration file. The default settings are equivalent to this:

```
---
:backends: yaml
:yaml:
  :datadir: /var/lib/hiera
:hierarchy: common
:logger: console
```

# Hiera Backends

One powerful feature of Hiera is that the actual key-value data can be retrieved from different backends.

With the `:backends` global configuration we define which backends to use, then, for each used backend we can specify backend specific settings.

## Build it backends:

**yaml -** Data is stored in yaml files (in the `:datadir` directory)

**json -** Data is stored in json files (in the `:datadir` directory)

**puppet -** Data is defined in Puppet (in the `:datasouce` class)

## Extra backends:

Many additional backends are available, the most interesting ones are:

**gpg -** Data is stored in GPG encripted yaml files

**http -** Data is retrieved from a REST service

**mysql -** Data is retrieved from a Mysql database

**redis -** Data is retrieved from a Redis database

## Custom backends

It's relatively easy to write custom backends for Hiera. Here are some development instructions

# Hierarchies

With the `:hierarchy` global setting we can define a string or an array of data sources which are checked in order, from top to bottom.

When the same key is present on different data sources by default is chosen the top one. We can override this setting with the `:merge_behavior` global configuration. Check this page for details.

In hierarchies we can interpolate variables with the %{} notation (variables interpolation is possible also in other parts of hiera.yaml and in the same data sources).

This is an example Hierarchy:

```
---
    :hierarchy:
      - "nodes/%{::clientcert}"
      - "roles/%{::role}"
      - "%{::osfamily}"
      - "%{::environment}"
      - common
```

Note that the referenced variables should be expressed with their fully qualified name. They are generally facts or Puppet's top scope variables (in the above example, **$::role** is not a standard fact, but is generally useful to have it (or a similar variable that identifies the kind of server) in our hierarchy).

If we have more backends, for each backend is evaluated the full hierarchy.

We can find some real world hierarchies samples in this Puppet Users Group post

More information on hierarchies here.

# Using Hiera in Puppet

The data stored in Hiera can be retrieved by the PuppetMaster while compiling the catalog using the hiera() function.

In our manifests we can have something like this:

```
$my_dns_servers = hiera("dns_servers")
```

Which assigns to the variable ***$my_dns_servers*** (can have any name) the top value retrieved by Hiera for the key ***dns_servers***

We may prefer, in some cases, to retrieve all the values retrieved in the hierarchy's data sources of a given key and not the first use, use hiera_array() for that.

```
$my_dns_servers = hiera_array("dns_serve
rs")
```

If we expect an hash as value for a given key we can use the hiera() function to retrieve the top value found or hiera_hash to merge all the found values in a single hash:

```
$openssh_settings = hiera_hash("openssh_
settings")
```

## Extra parameters

All these hiera functions may receive additional parameters:

Second argument: **default** value if no one is found

Third argument: **override** with a custom data source added at the top of the configured hierarchy

```
$my_dns_servers = hiera("dns_servers","8
.8.8.8","$country")
```

# Puppet 3 data bindings

With Puppet 3 Hiera is shipped directly with Puppet and an automatic hiera lookup is done for each class' parameter using the key **$class::$argument**: this functionality is called data bindings or automatic parameter lookup.

For example in a class definition like:

```
class openssh (
    template = undef,
  ) { . . . }
```

Puppet 3 automatically looks for the Hiera key openssh::template if no value is explitely set when declaring the class.

To emulate a similar behaviour on pre Puppet 3 we should write something like:

```
class openssh (
    template = hiera("openssh::template"
),
  ) { . . . }
```

If a default value is set for an argument that value is used only when user has not explicitly declared value for that argument and Hiera automatic lookup for that argument doesn't return any value.

# Using hiera from the command line

Hiera can be invoken via the command line to interrogate the given key's value:

```
hiera dns_servers
```

This will return the default value as no node's specific information is provided. More useful is to provide the whole facts' yaml of a node, so that the returned value can be based on the dynamic values of the hierarchy. On the Pupppet Masters the facts of all the managed clients are collected in $vardir/yaml/facts so this is the best place to see how Hiera evaluates keys for different clients:

```
hiera dns_servers --
yaml /var/lib/puppet/yaml/facts/<node>.y
aml
```

We can also pass variables useful to test the hierarchy, directly from the command line:

```
hiera ntp_servers operatingsystem=Centos
    hiera ntp_servers operatingsystem=Ce
ntos hostname=jenkins
```

To have a deeper insight of Hiera operations use the debug (-d) option:

```
hiera dns_servers -d
```

To make an hiera array lookup (equivalent to hiera_array()):

```
hiera dns_servers -a
```

To make an hiera hash lookup (equivalent to hiera_hash()):

```
hiera openssh::settings -h
```

# Puppet architectures - Overview

## The Components of a Puppet

# architecture

# Where to define classes

# Where to define parameters

# Where to place files

# Puppet security

## Components of a Puppet architecture

### Tasks we deal with

Definition of the **classes** to be included in each node

Definition of the **parameters** to use for each node

Definition of the configuration **files** provided to the nodes

### Components

**/etc/puppet/manifests/site.pp** - The default manifests loaded by the Master

**ENC** - The (optional) Enternal Node Classifier

**ldap** - (Optional) LDAP backend

**Hiera** - Data key-value backend

**Public modules** - Public shared modules

**Site modules** - Local custom modules

# Where to define classes

The classes to include in each node can be defined on:

**/etc/puppet/manifests/site.pp** - Top or Node scope variables

**ENC** - Under the classes key in the provided YAML

**ldap** - puppetClass attribute

**Hiera** - Via the `hiera_include()` function

**Site modules** - In roles and profiles or other grouping classes

# Where to define parameters

The classes to include in each node can be defined on:

**/etc/puppet/manifests/site.pp** - Under the node statement

**ENC** - Following the ENC logic

**ldap** - puppetVar attribute

**Hiera** - Via the `hiera()`, `hiera_hash()`, `hiera_array()` functions of Puppet 3 Data Bindings

**Shared modules** - OS related settings

**Site modules** - Custom and logical settings

**Facts** - Facts calculated on the client

# Where to define files

**Shared modules** - Default templates populated via

module's params

**Site modules** - All custom static files and templates

**Hiera** - Via the **hiera-file** plugin

**Fileserver** custom mount points

# Code workflow management

Puppet code should stay under a SCM (Git, Subversion, Mercurial... whatever )

We must be able to test the code before committing to production

Testing Puppet code syntax is easy, testing its effects is not

Different testing methods for different purposes

Different code workflow options

Recommended: Couple Puppet environments with code branches More info

# Puppet Security considerations

Client / Server communications are encrypted with SSL x509 certificates
By default the CA on the PuppetMaster requires manual signing of certificate requests

```
# In puppet.conf under [master] section:
autosign = false
```

On the Puppet Master (as unprivileged user) runs a service that must be reached by every node (port 8140 TCP)

On the Client Puppet runs a root but does not expose a public service

Sensitive data might be present in the catalog and in reports

Sensitive data might be present in the puppet code (Propagated under a SCM)

Security advisories

# PuppetDB - Overview

## What PuppetDB does

## Installation and configuration

## API and query language

## The puppetdbquery module

# Introduction to PuppetDB

PuppetDB is a "fast, scalable and reliable" data warehouse for Puppet.

It saves data generated by Puppet: nodes' **facts**, **catalogs** and **reports**.

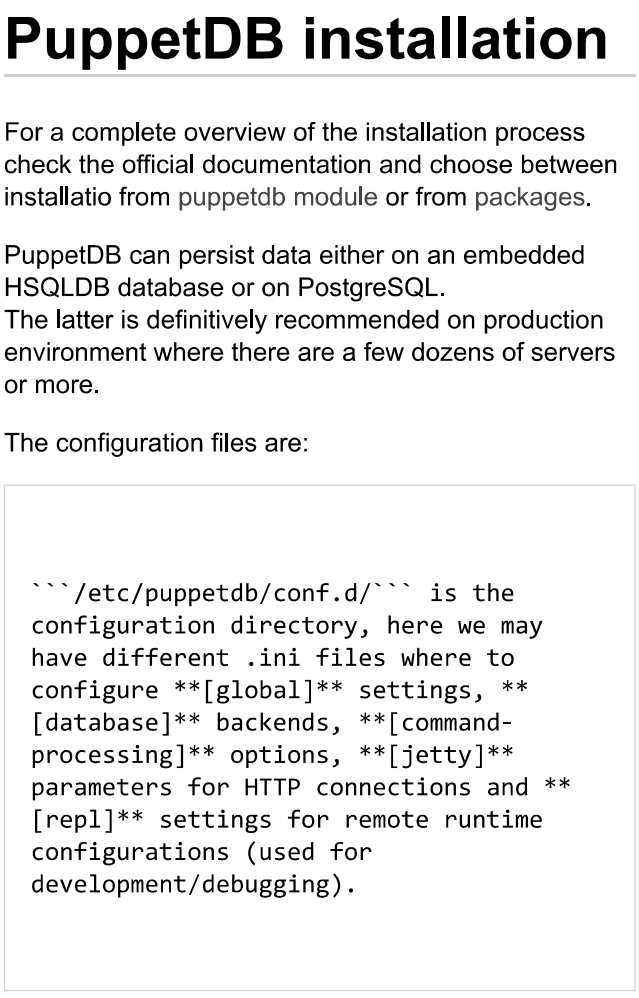
It can also work as a performant backend for **exported resources** being a recommended alternative to the older ActiveRecord storeconfigs interface.

It provides a faster backend also for Puppet's

**Inventory Service**.

PuppetDB is opensource, written in Clojure (it requires >= JDK 1.6 and PuppetMasters with Puppet >= 2.7.12), and is delivered as an autonomous software.

The best source from where to retrieve it are the official Puppetlabs repositories.



# PuppetDB installation

For a complete overview of the installation process check the official documentation and choose between installatio from puppetdb module or from packages.

PuppetDB can persist data either on an embedded HSQLDB database or on PostgreSQL.
The latter is definitively recommended on production environment where there are a few dozens of servers or more.

The configuration files are:

```
```/etc/puppetdb/conf.d/``` is the
configuration directory, here we may
have different .ini files where to
configure **[global]** settings, **
[database]** backends, **[command-
processing]** options, **[jetty]**
parameters for HTTP connections and **
[repl]** settings for remote runtime
configurations (used for
development/debugging).
```

`/etc/puppet/puppetdb.conf` is the configuration file for **Puppet** with the settings to be used by the PuppetDB terminus
```

# PuppetDB console and tools

## Integrated console: Anaylize PuppetDB performance

PuppetDB provides a performances dashboard out of the box, we can use it to check how the software is working: **http:// :8080/dashboard/** .
This is integral part of the PuppetDB software.

## Puppet Board: Query PuppetDB from the web

A nice frontend that allows interrogation of the PuppetDB from the web is PuppetBoard.

PuppetDB from the web is PuppetBoard.
This software is contributed from the community.

## Puppetdbquery module: Query PuppetDB from Puppet manifests

An incredibly useful module that provides faces, functions and hiera backends that work with PuppetDB is the puppetdbquery module. We can install it also from the Forge:

```
puppet module install dalen-
puppetdbquery
```

This software is contributed from the community (and is becoming a standard de facto for PuppetDB querying inside Puppet modules).

PuppetDB exposes an HTTP API that uses a Command/Query Responsibility Separation (CQRS) pattern:

### REST for reading

A standard REST API is used to query data.

The current (API v3) available endpoints are: `metrics`, `fact-names`, `facts`, `nodes`, `resources`, `reports`, `events`, `event-counts`, `aggregate-event-counts`, `server-time`

### COMMANDS for writing

Explicit **commands** are used (via HTTP using the /commands/ URL) used to populate and modify data.

The current **commands** are: `replace catalog`, `replace facts`, `deactivate node`, `store report`.

## API versions

There are different versions of the APIs as they evolve with PuppetDB versions.

As October 2013, Version 1 of the API is deprecated, Version 2 and 3 (the latter adds new endpoints and is recommended) are both supported.

We can access a specific version of the API using the relevant prefix:

```
http[s]://puppetdb.server/v#/<endpoint>/
[NAME]/[VALUE][?query=<QUERY STRING>]
```

## Query language

Queries to the REST endpoints can define search scope and limitations for the given endpoint. The query are sent as an "URL-encoded JSON array in prefix notation". Check the online tutorial for details.

# Using Puppetdbquery module

The PuppetDbQueryModule is developed by a community member, Erik Dalen, and is the most used and useful module available to work with PuppetDB: it provides command lines tools (as Puppet Faces), functions to query PuppetDB and a PuppetDB based Hiera backend.

# Query format

All the queries we can do with this module are in the format

```
Type[Name]
{attribute1=foo and attribute2=bar}
```

by default they are made on normal resources, use the

@@ prefix to query exported resources.

The comparison operators are: **=**, **!=**, **>**, **<** and **~**

The expressions can be combined with **and**, **not** and **or**

## Using Puppetdbquery module

## Command line

The module introduces, as a Puppet face, the **query** command:

```
puppet help query
puppet query facts '(osfamily=RedHat and
 operatingsystemversion=6)'
```

## Functions

The functions provided by the module can be used inside manifests to populate the catalog with data retrieved on PuppetDB.

**query_nodes** takes 2 arguments: the query to use and (optional) the fact to return (by default it provides the certname). It returns an array:

```
$webservers = query_nodes('osfamily=Debi
an and Class[Apache]')
$webserver_ip = query_nodes('osfamily=De
bian and Class[Apache]', ipaddress)
```

**query_facts** requires 2 arguments: the query to use to discover nodes and the list of facts to return for them. It returns a nested hash in JSON format.

```
query_facts('Class[Apache]
{port=443}', ['osfamily', 'ipaddress'])
```

# MCollective - Overview

## Essentials

## Installation and configuration

## The mco command

# MCollective essentials

An orchestration framework that allows massively parallel actions on the infrastructure's server

Based on 3 components:

One or more central consoles, from where the **mco** command can be issued

The infrastructure nodes, where an **agent** receives and executes the requested actions

A middleware **message broker**, that allows communication between master(s) and agents

Possible actions are based on plugins

Common plugins are: **service**, **package**, **puppetd**, **filemgr**...

Simple RCPs allow easy definitions of new plugins or actions.

Security is handled at different layers: transport, authentication and authorization via different plugins. Here is a Security Overview

# Installation

Some distros provide native mcollective packages but it's definitively recommended to use PuppetLabs repositories to have recent mcollective versions and packages for the needed dependencies.

On the nodes to be managed just install the **mcollective** package, it's configuration file is **/etc/mcollective/server.cfg**

On the administrative nodes, from where to control the infrastructure, install the **mcollective-client** package, it's configuration file is **/etc/mcollective/client.cfg**

On both clients and servers we have to install the agent plugins

On the middleware server(s) install **activemq** or **rabbitmq**

# Configuration

Configuration files are similar for server and client.

They are in a typical setting = value format and # are used for comments.

We have to define to connetion settings to the middleware, the settings for the chosen security plugins, the facts source and other general settings.

The setup of the whole infrastructure involves some parameters to be consistent on the mcollective config files and the message broker ones (credentials and eventually certificates).

There can be different setups, according to the security plugins used, the recommended one is well described in the Standard Mcollective Deployment

# Using the mco command

The **mco** command is installed on the Mcollective clients and can be used to perform actions on any node of the infrastructure.

General usage:

```
mco [subcommand] [options] [filters]
mco rpc [options] [filters] <agent> <act
ion> [<key=val> <key=val>]
```

Most used subcommands are **ping** , **rpc** , **inventory**, **find**

Most used options are: **-batch SIZE** (send requests in batches), **-q** (quiet), **-j** (produce Json output), **-v** (verbose)

Most used filters are: **-F <fact=val>** (Match hosts with the specified fact value), **-I** (Match host with the given name).

Sample commands:

```
mco help
mco help rpc
mco ping
mco rpc service status service=mysqld
mco ping -F datacenter=eu-1
```

# Puppet reporting - Overview

## Reporting overview

## Understanding puppet run output

# Reporting

In reports Puppet stores information of what has changed on the system after a Puppet run

Reports are sent from the client to the Master, if report is enabled

```
# On client's puppet.conf [agent]
report = true
```

# On the Master different report processors may be enabled

```
# On server's puppet.conf [master]
reports = log,tagmail,store,https
reporturl = http://localhost:3000/reports
```

## Understanding Puppet runs output

Puppet is very informative about what it does on the functions

```
/Stage[main]/Example42::CommonSetup/File
[/etc/motd]/content:
--- /etc/motd   2012-12-
13 10:37:29.000000000 +0100
+++ /tmp/puppet-file20121213-19295-
qe4wv2-0 2012-12-
13 10:38:19.000000000 +0100
@@ -4,4 +4,4 @@
-
Last Puppet Run: Thu Dec 13 10:36:58 CET
 2012
+Last Puppet Run: Thu Dec 13 10:38:00 CE
T 2012
Info: /Stage[main]/Example42::CommonSetu
p/File[/etc/motd]: Filebucketed /etc/mot
d to main with sum 623bcd5f8785251e2cd00
a88438d6d08
/Stage[main]/Example42::CommonSetup/File
[/etc/motd]/content: content changed '{m
d5}623bcd5f8785251e2cd00a88438d6d08' to
'{md5}91855575e2252c38ce01efedf1f48cd3'
```

The above lines refer to a change made by Puppet on /etc/motd
The File[/etc/motd] resource is defined in the class example42::commonsetup
A diff is shown of what has changed (the diff appears when running puppet agent -t in interactive mode)

The original file has been filebucketed (saved) with checksum 623bcd5f8785251e2cd00a88438d6d08
We can retrieve the original file in /var/lib/puppet/clientbucket/6/2/3/b/c/d/5/f/623bcd5f8785251e2cd00a88438d6d08/contents
We can search for all the old versions of /etc/motd with

```
grep -
R '/etc/motd' /var/lib/puppet/clientbuck
et/*
```

# Developing Puppet extensions - Overview

## Developing custom facts

## Developing custom types and provides

## Developing functions

# Developing Facts

Facts are generated on the client before the evaluation of the Puppet code on the server.

Facts provide variables that can be used in Puppet code and templates.

A simple custom type that just executes a shell command:

```
require 'facter'
```

```
require 'facter'
Facter.add("last_run") do
  setcode do
    Facter::Util::Resolution.exec('date'
)
  end
end
```

This file should be placed in /lib/facter/acpi_available.rb

## Developing Types

Resource types can be defined in Puppet language or in Ruby as plugins.

Ruby types require one or more **providers** which manage low-lovel interaction with the underlining OS to provide the more abstact resource defined in Types.

An example of a type with many providers is package, which has more than 20 providers that manage packages on different OS.

A sample type structure (A type called vcsrepo, must have a path like **/lib/puppet/type/vcsrepo.rb**

```
require 'pathname'

Puppet::Type.newtype(:vcsrepo) do
  desc "A local version control reposito
ry"

  feature :gzip_compression,
          "The provider supports explici
t GZip compression levels"

  [...] List of the features

  ensurable do

    newvalue :present do
      provider.create
    end

  [...] List of the accepted values

  newparam(:source) do
    desc "The source URI for the reposit
ory"
  end

  [...] List of the accepted parameters

end
```

# Developing Providers

The vcsrepo type seen before has 5 different providers for different source control tools.

Here is, as an example, the git provider for the vcsrepo type (Code (C) by PuppetLabs):
This file should be placed
**/lib/puppet/provider/vcsrepo/git.rb**

```
require File.join(File.dirname(__FILE__)
, '..', 'vcsrepo')

Puppet::Type.type(:vcsrepo).provide(:git
, :parent => Puppet::Provider::Vcsrepo)
do
  desc "Supports Git repositories"

  commands :git => 'git'
  defaultfor :git => :exists
  has_features :bare_repositories, :refe
rence_tracking

  def create
    if !@resource.value(:source)
      init_repository(@resource.value(:p
ath))
    else
      clone_repository(@resource.value(:
source), @resource.value(:path))
      if @resource.value(:revision)
        if @resource.value(:ensure) == :
bare
          notice "Ignoring revision for
bare repository"
        else
          checkout_branch_or_reset
        end
      end
      if @resource.value(:ensure) != :ba
re
        update_submodules
      end
    end
  end

  def destroy
    FileUtils.rm_rf(@resource.value(:pat
h))
  end

  [...]

end
```

# Developing Functions

Functions are Ruby code that is executed during compilation on the Puppet Master.

They are used to interface with external tools, provide debugging or interpolate strings.

Importants parts of the Puppet language like include and template are implemented as functions.

```
module Puppet::Parser::Functions
  newfunction(:get_magicvar, :type => :r
value, :doc => <<-EOS
This returns the value of the input vari
able. For example if you input role
it returns the value of $role'.
    EOS
  ) do |arguments|

    raise(Puppet::ParseError, "get_magic
var(): Wrong number of arguments " +
      "given (#
{arguments.size} for 1)") if arguments.s
ize < 1

    my_var = arguments[0]
    result = lookupvar("#{my_var}")
    result = 'all' if ( result == :undef
ined || result == '' )
    return result
  end
end
```

This file is placed in puppi/lib/puppet/parser/functions/get_magicvar.rb.

# Scaling Puppet

## Optimizing code for performance

### Optimize code for performance

Reduce the number of resources per node
For each resource Puppet has to serialize, deserialize, apply, report...
Avoid use of resources for too many entities (Do

manage hundreds of users with the User type)
Limit overhead of containing classes
(A module with the openssh::package, openssh::service, openssh::configuration subclasses pattern is uselessly filled with extra resources)

Do not use the file type to deliver too large files or binaries:
For each file Puppet has to make a checksum to verify if it has changed
With source => , the content of the file is retrieved with a new connection to the PuppetMaster
With content => template() , the content is placed inside the catalog.
Avoid too many elements in a source array for file retrieval:

```
source => [ "site/openssh/sshd.conf---
$::hostname ,
            "site/openssh/sshd.conf--
$environemnt-$role ,
            "site/openssh/sshd.conf-
$role ,
            "site/openssh/sshd.conf ],
```

This checks 3 files (and eventually gets 3 404 errors from server) before getting the default ones.

# Reduce PuppetMaster(s) load

Run Puppet less frequently (default 30 mins)

Evaluate centrally managed, on-demand, Puppet Runs (ie: via MCollective)

Evaluate Master-less setups

Use a setup with Passenger to have multiple PuppetMasters childs on different CPUs

Disable Store Configs if not used

```
storeconfigs = false
```

Alternatively enable Thin Store Configs and Mysql backend

```
storeconfigs = true
thin_storeconfigs = true
dbadapter = mysql
dbname = puppet
dbserver = mysql.example42.com
dbuser = puppet
dbpassword = <%= scope.lookupvar('secret
::puppet_db_password') %>
```

Evaluate PuppetDB as backend (faster)

```
storeconfigs = true
storeconfigs_backend = puppetdb
```