Puppet Training

A Puppet course covering:

- Introduction to DevOps and Configuration management tools
- Puppet and its ecosystem
- Core concepts and terminology
- Puppet Language basics
- Learning References
- Installation and basic usage
- How to choose the components of a Puppet architecture
- Understanding the Puppet DSL
- How to choose, integrate, use and write modules
- How to perform common tasks with Puppet
- How to classify nodes and organize code and data
- How to manage interdependencies across nodes

The [DevOps] Guide to Puppet, Universe, and Everything

This slides deck is part of The [DevOps] Guide to Puppet, Universe, and Everything, an Open Source holistic documentation project composed of:

- A sample Puppet architectures evolved into the Psick project
- A book [WIP]
- A set of cheatsheets [WIP]
- A slides deck [This project]

License

Developed and sponsored by

example42

Puppet Training - Contents

Introduction to Puppet - Takeaways

- **Know what is Puppet**
- **Understand the principles behind configuration management tools**
- **Know where to find information about Puppet**
- **Have a general view of Puppet software ecosystem**
- **Understand the logic and core principles of Puppet**

What is Puppet?

Puppet is an Open Source Configuration Management software developed by the Puppet company.

More widely Puppet is a framework for Systems Automation since it automates the configuration and ongoing management of our servers, in a centrally managed way.

It allows to define, with a declarative Domain Specific Language (DSL), what we want to configure and manage on an IT infrastructure: from components of an Operating System (OS) like packages, services to and files to network, cloud, storage resources.

Puppet agent can be installed on Linux, Unix (Solaris, AIX, *BSD), MacOS, Windows (List of Supported Platforms) and can be used also to remotely configure network and storage devices or manage cloud resources.

It is used by many companies with a number of managed systems than ranges from few dozens to several thousands.

Puppet releases also Puppet Enterprise (PE) based on the Open Source code base and oriented to enterprises that need official support and want Puppet infrastructure easier to setup and with better reporting and management features.

DevOps and Configuration management

DevOps is a term that involves a remarkable number of concepts, nuances and definitions.

We won't try to give another one, but we can safely say that DevOps is (also) aboutculture, processes, people and tools.

A complete DevOps tools chain contains software of these categories:

- Source Code Management (We use them when writing Puppet code)
- Repository and software Management (Puppet can configure them)
- Software build (Puppet can configure them)
- Configuration Management (Puppet is here)
- Testing (We can test our Puppet code)
- Monitoring and data analysis (Puppet can configure them)
- Systems and Applications Deployment (Puppet can be also here)
- Continuous integration (We can manage Puppet code deployments in a CI pipeline)
- Cloud (Puppet code can manage cloud resources)
- Project management and Issue tracking (We can use them to manage our Puppet projects)
- Messaging and Collaboration (We can use them to collaborate on Puppet works)
- Containerization and Virtualization (Puppet can configure them)
- Databases (Puppet can configure them)
- Application servers (Puppet can configure them)

Configuration Management principles

Configuration management tools allow to programmatically define how servers have to be configured. Their main benefits are:

- Automation: No need to manually configure systems
- Reproducibility: Setup once, repeat forever
- Recoverability: Setup again in case of outage
- Scale: Done for one, use on many
- Coherent and consistent server setups
- Aligned Environments for development, test, qa, production nodes

Configuration management tools typically describe the systems setups via code or data (Infrastructure as Code), this involves a paradigm change in system administration:

- Systems are managed centrally in an automated way: no more manually
- Code is versioned with a Source Control Management tool (git is the most used in Puppet world)
- Commits history shows the history of change on the infrastructure (who, what, when and why)
- Code can be tested before deployment in production
- Code has to be deployed after a CI/CD pipeline

Configuration Management tools

Common alternatives to Puppet:

Chef

- **Has Chef clients that connect to Chef server**
- **Has characteristic similar to Puppet**
- **Community code is shared on the Chef Supermarket**
- **Chef code is Ruby with dedicated extensions**
- **Software developed in Ruby.**

CFEngine

- **The first and oldest of the bunch**
- **CFEngine3 is a complete and modern rework**
- **Different daemons for different functions in a distributed environment**
- **Based on the Promise theory**
- **Software developed in C by Prof. Mark Burgess.**

Salt

- **Manages deployments on multiple clouds in a fast way**
- **Salt code is composed of states (Puppet resources) written in YAML files**
- **Formulas are equivalent to modules**
- **Sotfware developed in Python**

Ansible

- **Quick setup, no agents, communications over SSH**
- **Ansible code is YAML based and written on playbooks**
- **Roles are equivalent to modules, they are shared on the Ansible Galaxy**
- **Can centralise multi node task executions, software deployments and configuration management.**
- **Software developed in Python. Bought by RedHad in October 2015**

Learning resources

Useful resources to start learning Puppet:

- The website of the Puppet company
- The official Puppet Documentation site -
- The Learning VM, based on Puppet Enterprise, for a guided tour in Puppet world
- A list of the available Documentation resources

If you have questions to ask about Puppet usage use these:

- All the Puppet Community references
- The Slack channel
- Ask Puppet, the official Q&A site
- The discussion groups on Google Groups: puppet-users, puppet-dev, puppet-security-announce

To explore and use existing Puppet code:

- Puppet modules on Module Forge
- Puppet modules on GitHub

To inform yourself about what happens in Puppet World

- Planet Puppet - Puppet blogosphere
- The PuppetConf website
- The ongoing PuppetCamps all over the world

To find more and deeper information:

- Puppet Labs tickets - The official ticketing system
- Developer reference - The commented Puppet code

Essential Puppet concepts

When approaching Puppet it's important to understand its basic concepts and terminology.

A complete official glossary is online, use it.

Here we outline a few essential Puppet concepts, be sure to understand them all:

- **A typical Puppet setup is based on a client-server architecture: the client is also calledagent (or agent node or simply node), the server is called master**
- **When the puppet client connects to the server, it sends it's own sslcertname and a list of facts (units of information about the client's system generated by the command facter)**
- **Based on the client certname and its facts, the master replies with a catalog that describes what has to be configured on the client**
- **The catalog is based on Puppet language (a Domain Specific Language, DSL), which is written inmanifests (files with .pp extension).**
- **In the Puppet manifests, using Puppet DSL, we declare resources that affect elements of the system to manage (files, packages, services ...)**
- **Resources are grouped in classes which may expose parameters that affect their behavior.**
- **The values of the class parameters can be defined in different ways, one of them isHiera**
- **Hiera is a very common and versatile key-pair lookup tool used to store the values of parameters**
- **Classes and the configuration files that are shipped to nodes are organized inmodules**
- **Modules to manage virtually any IT resource are shared on thePuppet Forge**
- **The control-repo is the [git] repository which typically contains all our Puppet code and data and the references to the used external modules**

Puppet ecosystem - Puppet Inc

Puppet Inc software products

Puppet ecosystem is composed of many software projects. The ones developed by PuppetLabs are:

- Puppet - Official Puppet Open Source documentation
- Puppet Enterprise - The commercial enterprise version of Puppet
- Facter - Complementary tool to retrieve system's data. It's installed with Puppet agent
- Puppet Server - The next generation server service
- Hiera - Key-value lookup tool where Puppet data can be placed
- PuppetDB - Stores all the data generated by Puppet
- MCollective - Infrastructure Orchestration framework (superseded by Bolt)
- Bolt - Remote execution and orchstration tool Orchestration framework
- Razor - A provisioning system for bare metal servers
- PDK - The Puppet Development Kit
- r10k - A tool to manage deployments of Puppet code

Puppet ecosystem - Community

Community resources

Some of the most known Puppet related community projects are:

- Puppet Community - Modules and tooling for and by the Puppet Community
- Puppet CookBook - A collection of task oriented solutions in Puppet
- Puppet DashBoard - A Puppet *Web frontend* and External Node Classifier (ENC)
- The Foreman - A well-known third party provisioning tool and Puppet ENC
- PuppetBoard - A web frontend for PuppetDB
- Puppet Lint - A tool to check Puppet code style
- Rpec Puppet - A tool to make unit tests of Puppet code
- Puppet Module Info - Automatically generated documentation for Puppet modules

History of Puppet Versions

During its history Puppet has constantly improved and added features, still most of the core principles are the same and have't changed.

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.x.x` - **Many changes. Disabled dynamic variables scope. Data bindings**
- `3.2.x` - **Future parser (experimental, will be default in Puppet 4)**
- `4.x.x` - **Released in early 2015. New parser, new type system and lot of changes, various language cleanups and some backwards incompatibilities**
- `4.0.9` - **Released in February 2017. Hiera 5 is with data in modules is introduced.**
- `5.x.x` - **Released in June 2017. No major changes. Ruby 2.4 used.**
- `6.x.x` - **Released in September 2018. Some core types moved to separated modules. Resource API. CA moved to puppetserver. Deferred data type.**

In terms of code base migrations the biggest "jumps", which involve more or less large refactoring, have been:

- **from** `0.x` **to** `2.6` **with the introduction of parametrized classes**
- **from** `2.x` **to** `3.x` **with the removal of variables dynamic scoping**
- **from** `3.x` **to** `4.x` **with the introduction of the Future parser**

Language Basics - Takeaways

- Nodes classification: How to assign classes to nodes
- Know what is the Catalog
- Understand usage of variables, parameters and facts
- Know how are declared Puppet resources
- Understand the RAL (Resource Abstraction Layer) principles
- Know how are Puppet classes and defines
- Understand modules structure, usage and paths conventions
- Know how to write erb and epp templates

Classification

When clients connect, the Puppet Master generates a catalog with the list of the resources that clients have to apply locally.

The Puppet Master has to classify nodes setting for each of them what are the classes to include.

Classes contain the resources to manage on the node.

Nodes classification can be done in different ways (more details will follow):

- Using the `node` statement in Puppet code
- Using an External Node Classifier (ENC): a separated tool that provides classification info
- Using Hiera, Puppet's companion tool to manage configuration data
- Using a nodeless setup: the classes to include are defined according to variables and facts.

A Catalog of abstracted resources

The catalog is the complete list of resources, and their relationships, that the Puppet Master generates for the client node and sends it in Json format.

It's the result of all the puppet code and Hiera data that we define for a given node 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 fulfillment on the system, such as

```
apt-get install openssh # On Debian derivatives
yum install openssh     # On RedHad derivatives
```

The catalog is saved by the client in:

```
$libdir/client_data/catalog/$certname.json
```

The `$libdir` depends on the Puppet version and OS used, find it with the command `puppet config print libdir`.

Resource Types (Types)

Resources (also called Resource Types and Types) are singleunits of configuration that potentially can manage any "object" in an Operating System or more widely in an IT infrastructure.

A resources is composed by:

- **A type (package, service, file, user, mount, exec, interface, ec2_instance ...)**
- **A title (how the resource is called and referred in Puppet code)**
- **Zero or more arguments**

The syntax is always as follows:

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

Example for a file resource type:

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

Resource Types documentation

Information about resource types and their documentation can be found in different places:

- **Online is published the completeType Reference for the latest or earlier versions of Puppet.**
- **The `puppet describe` command:**

```
puppet describe <resource_type>
```

for example, to find from the command line information about the file resource:

```
puppet describe file
```

To list the description of ll the available types, run:

```
puppet describe --list
```

- **Give a glance to Puppet code for the list ofnative resource types (the ones written in Ruby language, based on the types and providers pattern):**

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

Simple samples of resources

The most common native resources, shipped with Puppet by default are: package, service, file, user, group, cron, exec, mount. Here are some simple examples.

Installation of OpenSSH package:

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

Creation of /etc/motd file:

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

Start of httpd service:

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

Creation of oscar user:

```
user { 'oscar':
  ensure => present,
  uid    => 1024,
}
```

Each of these resources have several other arguments that allows to define every characteristic of the resource.

More complex examples

Here are some more complex examples with usage of variables, resource references, arrays and relationships.

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

Installation of the Apache package triggering a restart of the relevant service:

```
package { 'httpd':
  ensure => $ensure,
  name   => $apache_package,
  notify => Class['Apache::Service'],
}
```

Resource Abstraction Layer (RAL)

Resources are abstracted from the underlying OS, this is achieved via theResource Abstraction Layer (RAL) composed of resource types that can have different providers.

A type specifies the attributes that a given resource may have, a provider implements the relevant type on the underlying Operating System.

For example the `package` type is known for the great number of providers (yum, apt, msi, gem ...).

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

With the command `puppet resource` we can represent the current status of a system's resources in Puppet language (note this can be done for any resource, even the ones not managed by Puppet):

To show all the existing users on a system (or only the root user):

```
puppet resource user
puppet resource user root
```

To show all the installed packages:

```
puppet resource package
```

To show all the system's services:

```
puppet resource service
```

It's also possible to directly modify them with `puppet resource` (note that this is not generally the way Puppet is used to manage the system's resources):

```
puppet resource service httpd ensure=running enable=true
```

Managing packages

Installation of packages is managed by the package type.

The main arguments:

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

Managing services

Management of services is via the service type.

The main arguments:

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

Executing commands

We can run plain commands using Puppet'sexec 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':
    command => 'wget http://mysite/myfile.tar.gz -O /tmp/myfile.tar.gz',
    path    => '/sbin:/bin:/usr/sbin:/usr/bin',
    creates => '/tmp/myfile.tar.gz',
    onlyif  => 'ls /tmp/myfile.tar.gz && false',
    unless  => 'ls /tmp/myfile.tar.gz',
    user    => 'my_user',
}
```

The parameters are as follows:

- **command, (namevar, if not specified, title is used) The actual command to execute**
- **path, the search paths where to look for the command, if not set, command must have an absolute path**
- **creates, a method to test if to run command, it refers to a file created by the command. It if exists, the command is not executed**
- **onlyif, alternative test method, if command here returns an error the main command IS NOT executed**
- **unless, alternative test method, if command here returns an error the main command IS executed**
- **user, which user the command should be run as (default root)**

Resource 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]

The metaparameters that manage dependencies are widely used to apply the resources in a catalog following the expected order.

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']
```

Variables

Variables is Puppet codes are basically constants: once defined in a class we can't change them.

We can set variables in our Puppet code with this syntax:

```
# Normal variable assignment
$role = 'mail'

# The value of a variable is based on another variable (here used the **selector** costruct)
$package = $::operatingsystem ? {
  /(?i:Ubuntu|Debian|Mint)/ => 'apache2',
  default                   => 'httpd',
}
```

Puppet automatically provides also some internal variables, the most common are:

- **The name of the node (the certname setting in its puppet.conf)**

```
$clientcert # Default is the client's Fully Qualified Domain Name)
```

- **The Puppet's environment where the Master looks for the code to compile**

```
$environment # Default is "production"
```

- **The Master's FQDN and IP address**

```
$servername $serverip
```

- **Any configuration setting of the Puppet Master's puppet.conf**

```
$settings::<setting_name>:
```

- **The name of the module that contains the current resource's definition**

```
$module_name
```

Facter and facts

Facter is a tools shipped with Puppet. It runs on clients and collectsfacts which are sent to the server. In our code we can use facts to manage resources in different ways or with different arguments.

Here follows a list of the most common and useful facts:

```
al$ facter

architecture => x86_64
fqdn => Macante.example42.com
hostname => Macante
ipaddress_eth0 => 10.42.42.98
macaddress_eth0 => 20:c9:d0:44:61:57
operatingsystem => Centos
operatingsystemrelease => 6.3
osfamily => RedHat
virtual => physical
```

It's easy to create custom facts. They can be of 2 types:

- **Native facts written in ruby and shipped with modules (in the `lib/facter` directory)**
- **External facts can be simple ini-file like texts (with `.txt` extension), Yaml files or even commands in any language, which returns a fact name and its value. External faccts are located in the nodes' `/etc/facter/facts.d` directory and can be shipped also from modules (in the `facts.d` directory)**

Classes

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

Example of a class definition (here we describe what the class does and what parameters it has, we don't actually add it and its resources to the catalog):

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

When we have to use a class previously defined, we declare it. Class declaration can be done in 2 different ways:

Class declaration without parameters (inside a catalog we can have multiple `include` of the same class but that class it's applied only once):

```
include mysql
```

This is similar to:

```
require mysql # Include and require the class
contain mysql # Include the class and be sure that all classes contained there are handled together
```

Class declaration with explicit parameters, available from Puppet 2.6. Syntax is the same of normal resources and the same class can be declared, in this way, only once inside the same catalog:

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

Defines

Also called: Defined resource types or defined types

Defined resource types are written in Puppet DSL and can make use of existing resource types or of other defined resource types.

They are similar to parametrized classes but can be used multiple times (with different titles), they are used as normal native types like package, service etc.

Definition of a define:

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

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

Declaration of a define:

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

Classes and defines parameters

Classes and defines can have parameters, according to the values given to these parameters user can customized their behaviour and generated results. Parameters are variables usable inside the same class or define (or in templates used there).

Starting from Puppet 4 parameters can be validated using the (very flexible and powerful) Puppet Data type system.

Class definition in older Puppet versions:

```
class motd (
  $content => 'This is the motd',
) {
  file { '/etc/motd':
    content => $content,
  }
}
```

Same class with parameters validation from Puppet 4:

```
class motd (
  String $content => 'This is the motd',
) {
  file { '/etc/motd':
    content => $content,
  }
}
```

A variable defined in a class can be used another class by using its fully qualified name ($:). For example we can refer in a class different than `motd` its `$content` variable with: `$::motd::content`. The only condition is that the class where the variable is set must be parsed (classified, in this case) before the class that uses it.

Puppet Modules

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

They are used to manage an application, system's resources, a local site or more complex structures.

Common places where to find public modules are the [Puppet Forge]() and [GitHub]().

Modules must be placed in the Puppet Master's modulepath, which is a double colon separated list of directories where Puppet looks for modules.

To see what's the local modulepath:

```
puppet config print modulepath
```

Default module path, for root user, has changed over time. On Puppet pre 4 it is:

```
/etc/puppet/modules:/usr/share/puppet/modules
```

Starting fromm Puppet 4, modulepath changes (as may other Puppet related directories):

```
/etc/puppetlabs/code/environments/production/modules:/etc/puppetlabs/code/modules:/opt/puppetlabs/puppet/modules
```

The puppet module subcommand

Puppet has a module 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.
```

To search a module on the Forge:

```
puppet module search elasticsearch
```

To install / uninstall / upgrade a module the module's author must be specified:

```
puppet module install elastic/elasticsearch
puppet module uninstall elastic/elasticsearch
puppet module upgrade elastic/elasticsearch
```

Paths of a module

Modules have a standard structure:

```
mysql/              # Main module directory

mysql/manifests/    # Manifests directory. Puppet code here in .pp files
mysql/lib/          # Plugins directory. Ruby code that extends Puppet here.
mysql/templates/    # ERB and EPP Templates directory
mysql/files/        # Static files directory
mysql/spec/         # Puppet tests directory
mysql/examples/     # Tests / Usage examples directory
mysql/tasks         # Tasks (in any language) directory (from Puppet )
mysql/plans         # Plans directory
mysql/type          # Data types directory

mysql/Modulefile    # Module's metadata descriptor (old, deprecated)
mysql/metadata.json # Module's metadata descriptor
```

This layout enables useful conventions over configurations.

Modules paths conventions

Classes and defines can be autoloaded, that is found automatically by Puppet as long as the relevant module is in the $modulepath and defined in the correct manifests, as follows:

```
include mysql
# Main mysql class is placed in: $modulepath/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
```

Templates (`.erb` **or** `.epp` **) are used in the relevant functions (** `template()` **or** `epp()` **) as follows**

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

content => epp('mysql/my.cnf.epp'),
# Epp template is in: $modulepath/mysql/templates/my.cnf.epp
```

Statically sourced files (files copied as is from master to clients) are referred as follows (note: `source` **and** `content` **parameters in the file type are mutually exclusive):**

```
source => 'puppet:///modules/mysql/my.cnf'
# Static file is in: $modulepath/mysql/files/my.cnf
```

ERB templates

They are files with `.erb` extension typically passed to the `template()` function and used as content for `file` resources.

Inside these files ruby code can be interpolated inside the `<%` and `%>` tags.

Example of an erb template:

```
# Show facts values
hostname = <%= @::fqdn %>

# Show values of variables outside local scope, two methods:
ntp_server = <%= scope.lookupvar('::ntp::server') %>
ntp_server = <%= scope['::ntp::server'] %>

# Iteration over an array
<% @::dns_servers.each do |ns| %>
nameserver <%= ns %>
<% end %>

# Conditional blocks of texts
<% if scope.lookupvar('puppet::db') == "puppetdb" -%>
storeconfigs_backend = puppetdb
<% end -%>
```

Note the ending `-%>` : when dash is present, no new line is introduced on the generated file.

EPP templates

They are files with `.epp` extension typically passed to the `epp()` function and used as content for `file` resources. As second argument the epp function expects an hash of variables usable in the template.

Inside these files Puppet code can be interpolated inside the `<%` and `%>` tags, the variables passed can be validated.

Example of an epp template, as used, for example, in `epp('template.epp',{dns_servers = ['1.1.1.1','8.8.8.8']})`:

```
# Show facts values and top scope variables using the fully qualified path ($:: ...)
hostname = <%= $::fqdn %>

# Show values of variables outside local scope using the fully qualified path:
ntp_server = <%= $::ntp::server') %>

# Iteration over an array (here the local scope dns_servers must be passed as argument of the epp function)
<% $dns_servers.each |$ns| { %>
nameserver <%= $ns %>
<% } %>

# Conditional blocks of texts
<% if $::puppet::db' == 'puppetdb' { -%>
storeconfigs_backend = puppetdb
<% } -%>
```

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

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 -%>
```

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 (puppet-doc)**

Interoperability

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

Puppet architectures - Takeaways

- **Understand Client/Master and Masterless setups**
- **Know the most important settings in puppet.conf**
- **Use Puppet Environments**
- **Know the Control repo parts**
- **Have an idea of the Roles and profiles pattern**
- **Overview of where things can be done in a Puppet architecture**

A typical Agent / Master Setup

A typical modern Puppet infrastructure has the following components:

- a Puppet server service (running as user `puppet`) listening on TCP port 8140 on the Puppet Master node (the server). This service can be supplied by different softwares:
    - The normal Puppet software in Ruby, running with themaster command (this is usually invoked viaPassenger, in not small setups)
    - The new puppetserver Clojure application, which runs inside a JVM
- a Puppet client (running as `root`) on each managed node. This is provided by thepuppet package (for Puppet 4, when provided via the Puppet upstream repositories, this is called puppet-agent).
- an optional PuppetDB service, which need to communicate with the Puppet server and usesPostgreSQL as data backend.
- an optional MCollective infrastructure based on agents, running on each node, a cli console, used from an administrative node for centralized control, and a Message Queue middleware service (typically ActiveMQ or RabbitMQ)
- more recently Bolt is used to replace Mcollective for remote execution and orchestration of commands, tasks and plans

A Puppet run on the client can be triggered in different ways:

- As a service (default configuration), which polls the server every 30 minutes (default)
- Via a cron job (typically with random delays to avoid requests congestion)
- Manually from the client node
- In a centralized way via MCollective or Bolt
- From the Puppet Enterprise (PE), via Orchestration features, based on MCollective, in older versions, or Bolt, starting from PE version 2016.x

Masterless setup

A masterless setup doesn't need a Puppet Master.

Puppet manifests are deployed directly on nodes and applied locally, we need to specify the manifest to apply, the path where to find modules and eventually the location of Hiera configuration file.

```
puppet apply --modulepath /puppetcode/modules/ --hiera_config /puppetcode/hiera.yaml /puppetcode/manifests/file.pp
```

To see the default values of these configuration options type:

```
puppet config print hiera_config modulepath
```

Such a setup involves pros, cons and new infrastructural challenges:

- **(+) No need of a Puppet infrastructure: just the puppet package on the node**
- **(+) There's not an external point of failure and a bottleneck on the Puppet Master (which still can scale)**
- **(+) We can trigger multiple parallel Puppet runs**
- **(+) Easier control of what version of our Puppet code is deployed and applied to each node**
- **(-) All the nodes have to store locally the Puppet code: we need a sane way to manage secrets**
- **(-) StoreConfigs usage require access to Puppet DB from every node**
- **(-) There aren't simple and ready solutions for centralized reporting and visualization of Puppet activity**
- **(=) We need to define a fitting deployment workflow on all the nodes. Hints:Rump - supply_drop**

Puppet configuration: puppet.conf

It's Puppet main configuration file. [Official reference]

On Open Source Puppet <= 3 is generally in:

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

On Puppet 4 and Puppet Enterprise is in:

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

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

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

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

Common stanza 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.

To view all or a specific configuration setting:

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

Options worth attention

Here are some options (and the relevant [stanza]) worth noting:

- **[main] vardir: Path where Puppet stores dynamic data.**
- **[main] ssldir: Path where SSL certifications are stored.**
- **[agent] server: Host name of the PuppetMaster. (Default: puppet)**
- **[agent] certname: Certificate name used by the client. (Default is its fqdn)**
- **[agent] runinterval: Number of minutes between Puppet runs, when running as service. (Default: 30)**
- **[agent] report: If to send Puppet runs' reports to thereport_server. (Default: true)**
- **[master] autosign: If new clients certificates are automatically signed. (Default: false)**

- **[master] reports: How to manage clients' reports (Default: store)**
- **[master] storeconfigs: If to enable store configs to support exported resources. (Default: false)**

For details check the full configuration reference on the official site.

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 management has changed from Puppet 3.6 onwards:

Earlier versions were based on so-called Config file 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 = $confdir/environments/test/modules:$confdir/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.

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 modules
manifests/ # Directory containing site.pp
environment.conf # Conf file for the environment
```

Certificates management

On the Puppet server is typically created a Certification Authority which manages all the nodes' certificates.

Starting from Puppet 6 the CA has been integrated in the puppetserver application and moved from the common puppet executable.

We are going to review here the most common commands to manage certificates in both the tools.

List the (client) certificates to sign:

```
puppet cert list                            # Puppet <  6
puppetserver ca list                        # Puppet >= 6
```

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

```
puppet cert list --all                      # Puppet <  6
puppetserver ca list --all                  # Puppet >= 6
```

Sign a node's certificate:

```
puppet cert sign <certname>                 # Puppet <  6
puppetserver ca sign --certname <certname>  # Puppet >= 6
```

Remove a client certificate:

```
puppet cert clean <certname>                # Puppet <  6
puppetserver ca clean --certname <certname> # Puppet >= 6
```

To see the path of the `ssldir` where all certs are written:

```
puppet config print ssldir
```

Client stores its certificates and the server's public one in `$ssldir`.

Server stores clients public certificates and in `$ssldir/ca`. DO NOT remove this directory.

The components of our Puppet architecture

What we have to do:

- Define and group the classes to include in each node
- Set the parameters both global to use for each node
- Manage the files used to configure things on our servers

Available components

- The default_manifest - The default manifests directory or single site.pp
- An ENC - (optional) An External Node Classifier. Can be Puppet Enterprise Console, The Foreman, Puppet Dashboard or any custom solution
- Hiera - A hierarchical and modular key-value backend
- Public modules - The available universe of public modules
- Site modules - Any custom module written for local needs

The components of our Puppet architecture

Where to define classes

- **default_manifest - 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

- **default_manifest - 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**

Nodes classification

We can classify nodes, that is define for each of them what classes and parameters to include, in different ways:

- Using the node statement
- Using an External Node Classifier
- Using Hiera

Classification via node statement

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 (nodes inheritance) 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 classify nodes
node_terminus = exec

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

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 the manifest file just write:

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

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

Nodeless setup

It's possible also to skip the nodes classification and define what to provide on each node just using (top scope or facts) variables.

For example we may define the `$::role` of our systems via a fact or a variable defined according to the hostname of our clients and just include, for each of them the relevant role class:

```
include "::roles::${::role}"
```

Roles and profiles

A common challenge we have when designing a Puppet architecture is how and where to define the logic that assigns to a group of nodes a specific set of classes (and the relevant resources).

There's not a single way or approach, but a consolidated best practices involves the usage of roles and profiles.

- A node has one and only one role (a variation involves more roles per node)
- Roles are classes that just include one or more profiles and describe the machines' function according to our business needs
- Profiles are classes that define a technological stack. They can include classes or declare them with parameters, they can have specific single resources. They either have parameters or hiera lookups inside the code.
- The modules that manage a single applications, typically the ones we find on the Forge for apache, nginx etc, are calledcomponent modules. Their classes and defines are supposed to be used in profile classes.

For further reference:

- The original blog post on Roles and Profiles by Craig Dunn
- The series of Gary Larizza's blog posts
- Some presentations on Roles and Profiles

The control repo

The current recommended approach to manage our Puppet code and data is to use a so calledcontrol-repo which is basically the content of a Puppet environment and contains typically:

- The environment.conf file to configure the environment itself
- A hieradata directory where to place Hiera data file (typically in yaml format)
- A manifests directory which contains manifests outside modules (maps to themanifests config setting). This has typically thesite.pp manifest with some node classification technique, top scope variables and resource defaults.
- An empty modules directory that contains Public modules to be installed via r10k
- A Puppetfile that defines the public modules to install via r10k
- A site directory which is added to the modulepath and contains local custom modules. In this directory you have, for example therole and the profile modules and any other custom module created for local needs
- An optional Vagrantfile and relevant files to have a local Vagrant environment where to test the Puppet

For a sample, check PuppetLabs' control repository template.

Introduction to Hiera

Hiera is the most used solution to manage the data we need to configure our systems according to the logic we need.

It allows to separate data from code, allowing us to avoid to place inside our Puppet manifests information which is strictly related to our infrastructure and its settings.

Hiera has been integrated in starting from Puppet 3: for each class parameter Puppet does an automatic lookup on a corresponding Hiera value.

Since Puppet 4.9 a totally new version of Hiera (5) has been introduced, which introduces the possibility to use Hiera data directly in modules.

Hiera can use different backends to store it's data. The default one is Yaml (all data is placed in simple Yaml files). Other backend are Json, Mysql, Redis and so on.

Data is looked on Hiera following a hierarchy based on Puppet variables, in this way we can attribute to an Hiera Key (for example `ntp::server`) different values according to different conditions (for example the role, the environment or the location of a server).

Inside Puppet code with can use the `hiera()` function (obsoleted by the `lookup()` function from Puppet 4.9) to look for a given Hiera key. For example the following code assign to the variable `$server` the value of the Hiera key called `ntp_server` . An optional parameter can be added to set the default value (here `ntp.pool.org`) to return if the key is not found on Hiera:

```
$server = hiera('ntp_server','ntp.pool.org')
```

On modern Puppet versions this is equivalent to:

```
$server = lookup('ntp_server',String,'first','ntp.pool.org')
```

Since Puppet 3 Puppet does an automatic lookup on Hiera for each parameter of a class. In the following example the parameter `server` (whose value can be referred, inside the same class, using the `$server` variable and, inside any other class with its *fully qualified name*: `$::ntp::server`) can be changed by setting, on Hiera a value for the key called `ntp::server` :

```
class ntp (
  server = 'ntp.pool.org',
) { }
```

Hiera configuration (old, pre Hiera 5)

Hiera 's configuration file for Puppet is `/etc/puppet/hiera.yaml` (On Puppet 4 `/etc/puppetlabs/puppet/hiera.yaml`). It's a Yaml file which looks like this:

```
---
  :backends:
    - yaml

  :hierarchy:
     - "%{::environment}/nodes/%{::fqdn}"
     - "%{::environment}/roles/%{::role}"
     - "%{::environment}/zones/%{::zone}"
     - "%{::environment}/common"
   :yaml:
     :datadir: /etc/puppet/hieradata
```

In the above sample it's configured thebackend(s) to use, the hierarchy based on Puppet variables (inside `%{}`) and the configuration specific to the used backend (here is set the datadir when Yaml files containing Hiera keys are placed, named accongding to the defined hierarchy).

Hiera configuration (new, starting from Hiera 5 - Puppet 4.9)

Starting from Hiera version 5 (available from Puppet version 4.9) hiera.yaml configuration file can be present in[3 different layers](#):

- **Global layer:** `/etc/puppetlabs/puppet/hiera.yaml`**. Equivalent to the single hiera.yaml in older versions.**
- **Environment layer:** `/etc/puppetlabs/code/environments/<environment>/hiera.yaml`**. Inside each Puppet environment directory**
- **Module layer** `hiera.yaml` **inside each module.**

The hierarchies defined in each of these files are traversed in the same order: first all the ones in the global layer, then the ones in the environment layer and then the ones in module layer.

This means that a common.yaml defined in `/etc/puppetlabs/puppet/hiera.yaml` **would override any more specific, variable dependent, hierarchy level in environment or module layers.**

For this reason when using Hiera 5 layes are geenrally used as follows:

- **Global layer is disabled (empty or missing hiera.yaml), used to set to global overrides, which "win" over everything, or used for special backends (like, as default on PE, the PE console)**
- **Environment layer is used for the most common data, where infrastructure related settings are placed**
- **Module layer is used generally by module authors, for OS related settings.**

The format of hiera.yaml is slightly different, a sample one might look as follows:

```
version: 5
hierarchy:
  - name: "Hierarchy"
    paths:
    - "nodes/%{::fqdn}.yaml"
    - "roles/%{::role}.yaml"
    - "zones/%{::zone}.yaml"
    - "common.yaml"
defaults:
  data_hash: yaml_data
  datadir: hieradata
```

If the above were an environment hiera.yaml, the data directory would be relative to the environment: `/etc/puppetlabs/code/environments/<environment>/hieradata/`**.**

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 legacy (now deprecated) `hiera()` function or the more modern `lookup()` one.

In our manifests we can have something like this:

```
# Deprecated Hiera function for direct lookup
$my_dns_servers = hiera('dns_servers')

# Current lookup function alternative
$my_dns_servers = lookup('dns_servers')
```

Both of them assign to the variable *$my_dns_servers* (can have any name) the first 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. If the expected data is an array we can use hiera_array() for that (deprecated) or specify the unique merge method in the lookup() function.

```
# Deprecated Hiera function for merging arrays lookup
$my_dns_servers = hiera_array('dns_servers')

# Current lookup function alternative with unique merge method
$my_dns_servers =  lookup('ntp::servers', {merge => unique})
```

If we expect an hash as value for a given key we can use the hiera() legacy function or lookup() with the default "first" lookup method to retrieve the first value found in the hierarchies, but, if we want to merge the hash values found in all the hierarchy levels in a single hash we can use the legacy hiera_hash() function or the deep merge method in lookup():

```
# Deprecated Hiera function for merging hashes lookup
$openssh_settings = hiera_hash('openssh_settings')

# Current lookup function alternative with merge methods hash or deep
$openssh_settings = lookup('openssh_settings', {merge => hash}) # Normal hash merge
$openssh_settings = lookup('openssh_settings', {merge => deep}) # Deep hash merge
```

The difference between hash and deep merge methods is that, when exist the same subkeys in hash looked up via Hiera, with deep merge the subkeys values are merged recursively, with normal hash merge the first one is the returned value.

Finally it's possible to include classes via Hiera, this is the so called Hiera driven classification.

In these examples, the key looked for ('classes') contain an array of the names of the classes to include (merged across the hierarchies):

```
# Deprecated Hiera_ function for including classes
hiera_include('classes')

# Alternative in modern Puppet
```

```
lookup('classes', {merge => unique}).include
```

Puppet data bindings

Starting from Puppet version 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 explicitly 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 avalue for that argument and Hiera automatic lookup for that argument doesn't return any value.

In modern Puppet setups data bindings is widely used to configure the classes included (without parameters declaration) via any classification method.

Information available on the agent

To see the list of classes included in the local node ($classfile):

```
sudo cat /var/lib/puppet/state/classes.txt # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/state/classes.txt # Puppet 4
```

To see all the resources managed on the node ($resourcefile):

```
sudo cat /var/lib/puppet/state/resources.txt # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/state/resources.txt # Puppet 4
```

To check the summary of the last puppet run (with metrics on performances and resources changed$lastrunfile):

```
sudo cat /var/lib/puppet/state/last_run_summary.yaml # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/state/last_run_summary.yaml # Puppet 4
```

To check the full report of the last run (not easy to read output):

```
sudo cat /var/lib/puppet/state/last_run_report.yaml # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/state/last_run_report.yaml # Puppet 4
```

To see, in JSON pretty format, the whole content of the last saved catalog:

```
sudo cat /var/lib/puppet/client_data/catalog/<node_certname>.json | json_pp # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/client_data/catalog/<node_certname>.json | json_pp # Puppet 4
```

To see the Puppet agent logs:

```
sudo cat /var/log/puppet/agent.log # Puppet 3
sudo cat /var/log/puppetlabs/puppet/puppetd.log # Puppet 4
```

Information available on the Puppet Master

To see the facts of the Puppet Master's clients:

```
sudo cat /var/lib/puppet/yaml/facts/<node_certname>.yaml # Puppet 3
sudo cat /opt/puppetlabs/puppet/cache/yaml/facts.d/<node_certname>.yaml # Puppet 4
```

To check the value of an Hiera key for a node with given variables:

```
hiera <key_name> -c /etc/puppet/hiera.yaml <param=value> <param=value> # Puppet 3
hiera <key_name> -c /etc/puppetlabs/code/hiera.yaml <param=value> <param=value> # Puppet 4
```

Information available on the CA server

List of all the signed and requested certificates

```
sudo puppet cert -la
```

On the CA server, by default the same Puppet Master, you can manage all the certificates with the `puppet cert` command.

Information available on the PuppetDB server

Query facts about a node

```
sudo puppet node find <node_certname> | json_pp
```

To purge a node data from PuppetDB (useful when you have duplicated resources in a node that collects exported resources). Use if you know what you are doing:

```
sudo puppet node deactivate <node_certname>
```

You can use the `query` action provided by the puppetdbquery module. For example to list all the nodes, stored on PuppetDB than match a given fact:

```
sudo puppet query nodes osfamily=Debian
```

To list specific facts of nodes matching your query:

```
sudo puppet query facts osfamily=Debian --facts 'ipaddress,uptime'
```

Check the puppetdbquery [query format syntax](#) to learn more about the syntax of your queries.