Puppet Configuration Management

What is Puppet

- A Configuration Management Tool
- A framework for Systems Automation
- A Declarative Domain Specific Language (DSL)
- An Open Source software written in Ruby
- Works on Linux, Unix (Solaris, AIX, *BSD), MacOS, Windows (Supported Platforms)
- Developed by Puppet Labs

Configuration Management

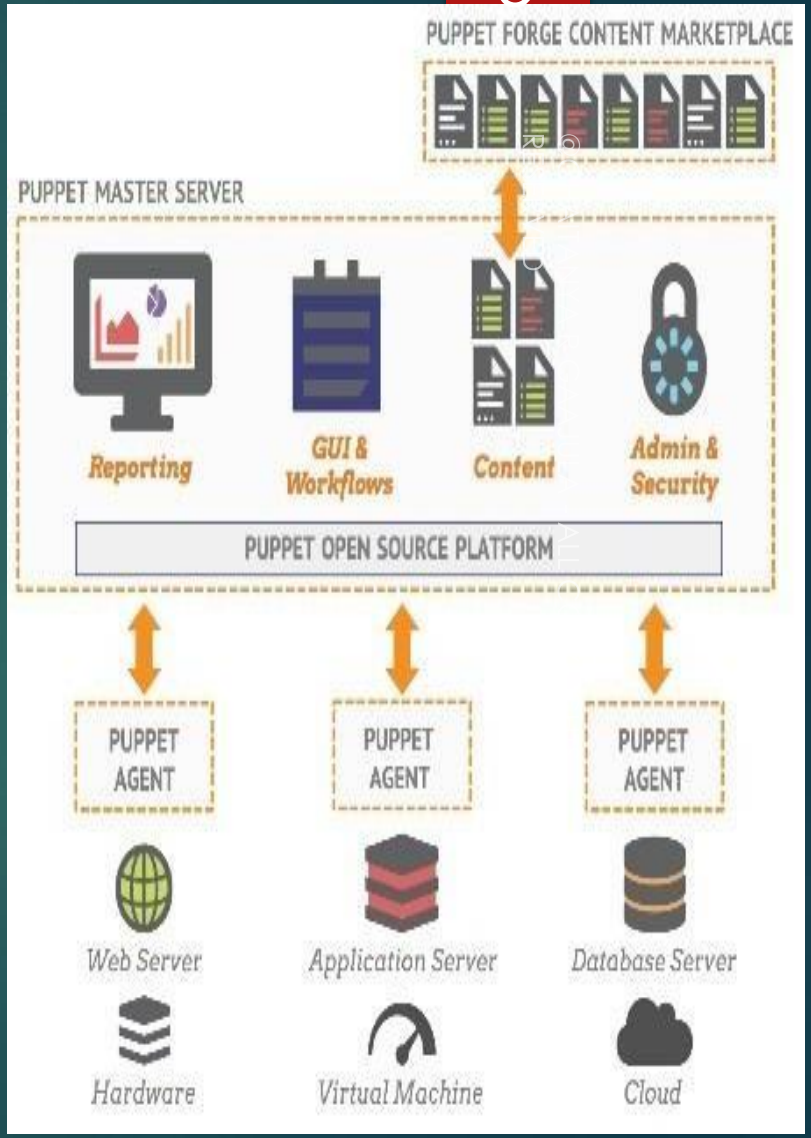
- Infrastructure as Code: Track, Test, Deploy, Reproduce, Scale
- Code commits log shows the history of change on the infrastructure
- Reproducible setups: Do once, repeat forever
- Scale quickly: Done for one, use on many
- Coherent and consistent server setups
- Aligned Environments for development, test, QA, prod nodes
- Alternatives to Puppet: Chef, CFEngine, Salt, Ansible

Configuration Management Tools

- Software configuration management includes tracking and controlling changes in the software
- SCM includes revision control and establishing baselines based on that
- Once a correct configuration is determined, SCM tools can replicate it across many hosts And if its wrong, SCM can determine errors
- With cloud computing the purposes of SCM tools have become merged
  - SCM tools are virtual appliances to be instantiated as VMs that can be saved with state and version Tools model and manage cloud-based virtual resources like VMs, storage space, software bundles

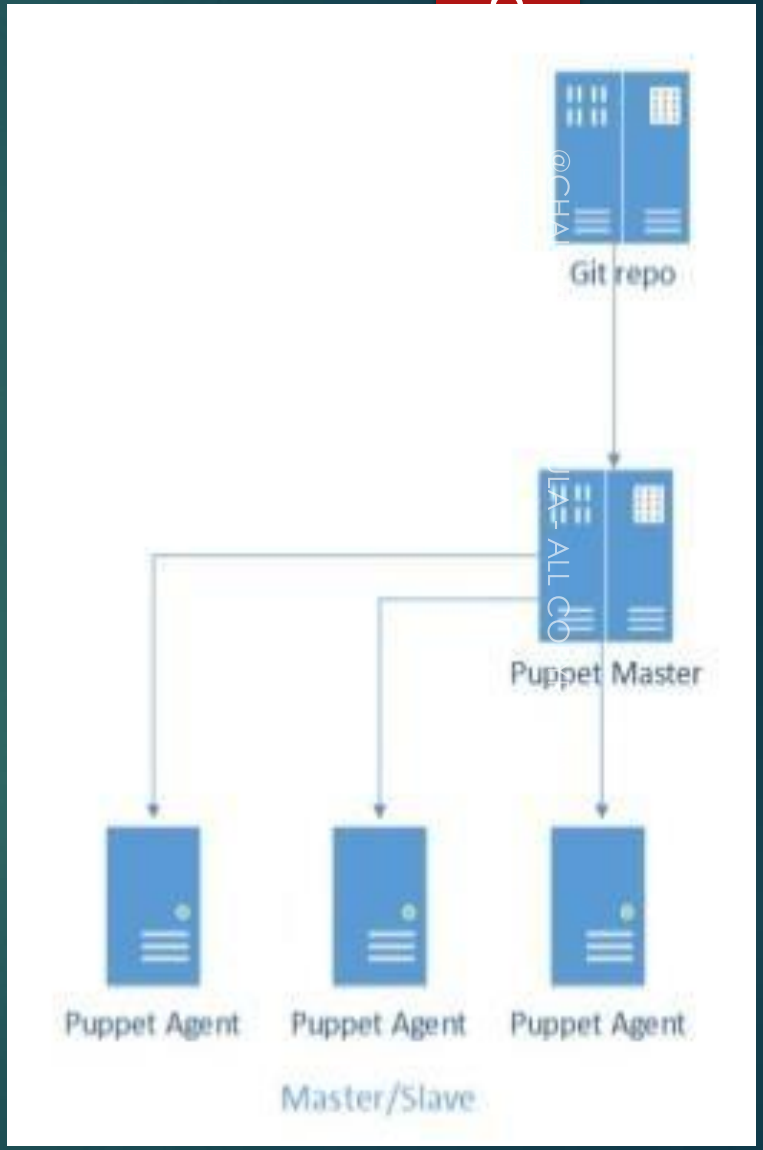
Puppet Setup

- System configuration in Puppet can be done in a client/server architecture,
  - using the Puppet agent and Puppet master applications
  - in a stand-alone architecture, using the Puppet apply application
- Starting with a catalog that describes the desired system state for a computer
- Catalog will list all resources as well as any dependencies between those resources, that need to be managed
- Puppet uses several sources of information to compile a catalog



Puppet Architecture Master Agent

- The Puppet master server manages the configuration information of all nodes
- Managed nodes run the Puppet agent application as a background service
- Puppet Server runs the Puppet master application
- Puppet Agent gathers facts about the node using Facter
- Puppet agent sends its configuration requirements to the Puppet master
- Puppet Master in turn compiles and returns that node's catalog, using Puppet Forge Content Marketplace
- Puppet agent upon receiving the catalog, applies it to the node
- If Agent finds any resources that are not in the state that they should be, it makes the necessary changes
- Agent sends a report to the Puppet master after applying changes
- Puppet agent nodes and Puppet masters communicate using HTTPS
- Each master and agent must have an identifying SSL certificate
- Upon examining other's certificate they decide whether to allow an exchange of information



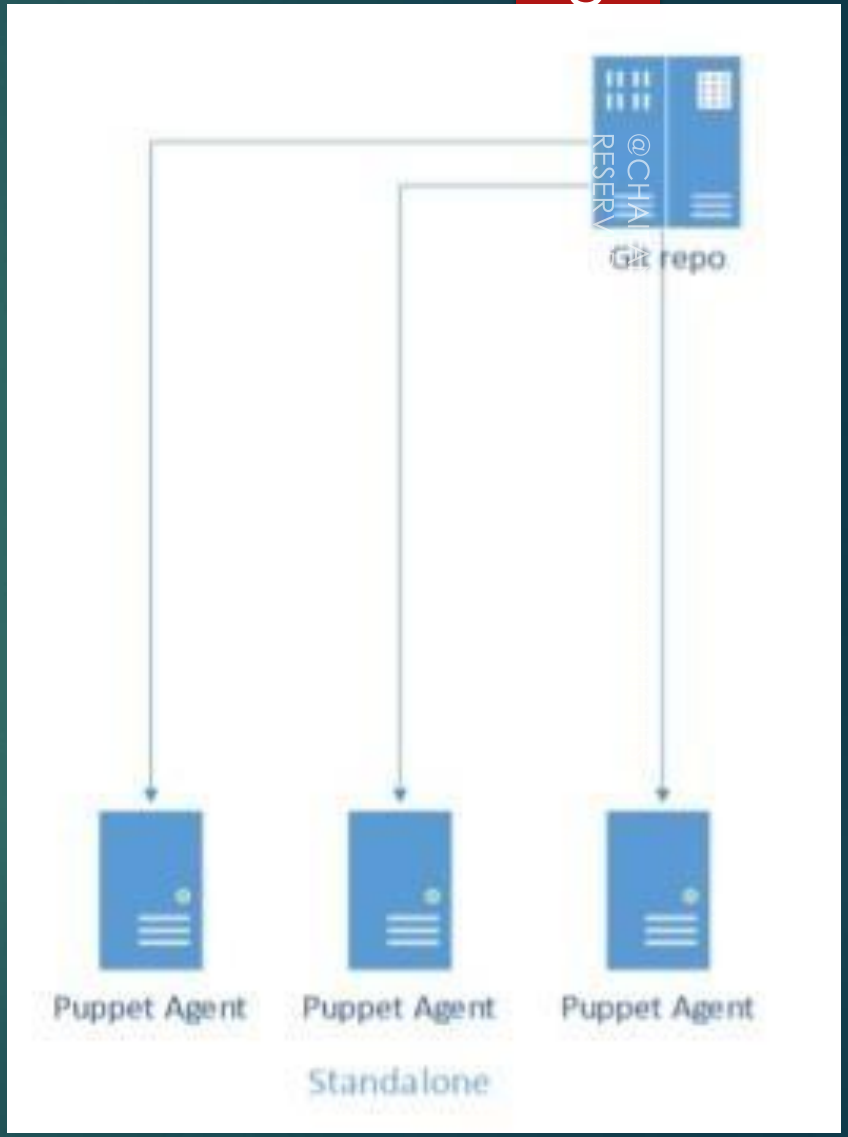
Puppet Master Agent Setup

- 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
- 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 careful of security concerns)

Puppet Architecture Standalone

- Each Puppet node has its complete configuration info using which it

  compiles catalog
- Nodes run Puppet apply application
- Puppet Apply uses several sources of configuration data which it uses to compile catalog for the node
- Puppet Apply applies catalog after compiling the catalog
- Puppet Apply makes the changes necessary nodes based on catalog
- Puppet Apply stores a report on disk for applied changes



Cross Platform Puppet

- Puppet Master and Agents can run on:
  - Linux
  - Windows
  - OS X
  - Other Unix (anything that can support Ruby/RubyGems)
- Puppet can determine individual run time environment information using Factor

Puppet Master

- When set up as an agent/master architecture, a Puppet master server controls the configuration information, and each managed agent node requests its own configuration catalog from the master
- In this architecture, managed nodes run the Puppet agent application, usually as a background service
- One or more servers run the Puppet master application, Puppet Server
- Periodically, each Puppet agent sends facts to the Puppet master, and requests a catalog
- The master compiles and returns that node's catalog, using several sources of information it has access to.
- Once it receives a catalog, Puppet agent applies it to the node by checking each resource the catalog describes.
- If it finds any resources that are not in their desired state, it makes the changes necessary to correct them.

Puppet Agent

- This is the main puppet client
- Its job is to retrieve the local machine's configuration from a remote server and apply it
- In order to successfully communicate with the remote server, the client must have a certificate signed by a certificate authority that the server trusts; the recommended method for this, at the moment, is to run a certificate authority as part of the puppet server (which is the default)
- The client will connect and request a signed certificate, and will continue connecting until it receives one
- Once the client has a signed certificate, it will retrieve its configuration and apply it

Puppet Agent

- Are software packages installed on each server
- “do the work” of installing and configuring software
- Can be configured to run regularly or on demand or not at all
- Can be executed in a “noop” mode for reporting purposes
- Can also execute the entire lifecycle with puppet apply locally (masterless puppet)

Puppet 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
```

Puppet Server Resources

- 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

Facter

- 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
```

Puppet Manifests

- Manifests are files that store specific configurations for resources
- Manifests for every resource are located on the Puppet master
- Each manifest ends with a .pp file extension
- Puppet site.pp manifest has global configurations that are applicable to all nodes
- Site manifests have node specific code as well
- Node definition is Puppet code included in catalog of nodes matching that node definition
- This allows you to assign specific configurations to specific nodes
- Manifests grouping several resources can also be created
- A class can be used to apply resources to specific nodes

Puppet Resources

- Resources are the fundamental unit for modelling system configurations
- Each resource describes some aspect of a system, like a specific service or package
- A resource declaration is an expression that describes the desired state for a resource and tells Puppet to add it to the catalog
- When Puppet applies that catalog to a target system, it manages every resource it contains, ensuring that the actual state matches the desired state

Simple Sample 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,
}
```

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" ],
}
```

Puppet 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

Puppet Values & Data Types

- Most of the things you can do with the Puppet language involve some form of data.
- An individual piece of data is called a value, and every value has a data type, which determines what kind of information that value can contain and how you can interact with it.
- Strings are the most common and useful data type, but you'll also have to work with others, including numbers, arrays, and some Puppet-specific data types like resource references

Puppet Data Types

- Strings
- Numbers
- Booleans
- Arrays
- Hashes
- Regular Expressions
- Sensitive
- Undef
- Resource References
- Default

Puppet Classes

- Classes are named blocks of Puppet code that are stored in modules for later use and are not applied until they are invoked by name
- They can be added to a node's catalog by either declaring them in your manifests or assigning them from an ENC
- Classes generally configure large or medium-sized chunks of functionality, such as all of the packages, config files, and services needed to run an application
- Classes are containers of different resources. Since Puppet 2.6 they can have parameters

Example of Class

```
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.

Defining Classes


```
# A class with no parameters
class base::linux {
  file { '/etc/passwd':
    owner => 'root',
    group => 'root',
    mode  => '0644',
  }
  file { '/etc/shadow':
    owner => 'root',
    group => 'root',
    mode  => '0440',
  }
}
```

Class Location

- Class definitions should be stored in modules. Puppet is automatically aware of classes in modules and can autoload them by name.
- Classes should be stored in their module's manifests/ directory as one class per file, and each filename should reflect the name of its class
- A class definition statement isn't an expression and can't be used where a value is expected.

Puppet Modules

- Modules are self-contained bundles of code and data with a specific directory structure
- These reusable, shareable units of Puppet code are a basic building block for Puppet
- Modules must have a valid name and be located in modulepath
- Puppet automatically loads all content from every module in modulepath making classes, defined types, and plug-ins (such as custom types or facts) available
- You can download and install modules written by Puppet or the Puppet community from the Puppet Forge

Module Structure

- Modules have a specific directory structure that allows Puppet to find and automatically load classes, defined types, facts, custom types and providers, functions, and tasks
- Module names should contain only lowercase letters, numbers, and underscores, and should begin with a lowercase letter
- Each manifest in a module's manifests folder should contain only one class or defined type
- You can serve files in a module's files directory to agent nodes
- Any ERB or EPP template can be rendered in a manifest

- `<MODULE NAME>`
  - `manifests`
  - `files`
  - `templates`
  - `lib`
    - `facter`
    - `puppet`
      - `functions`
      - `parser/functions`
      - `type`
      - `provider`
  - `facts.d`
  - `examples`
  - `spec`
  - `functions`
  - `types`
  - `tasks`

Puppet Forge

- https://forge.puppet.com/
- Go to Modules

Puppet Templates

- Templates are documents that combine code, data, and literal text to produce a final rendered output
- The goal of a template is to manage a complicated piece of text with simple inputs
- In Puppet, you'll usually use templates to manage the content of configuration files
- Templates are written in a templating language, which is specialized for generating text from data
- Puppet supports two templating languages:
  - Embedded Puppet (EPP) uses Puppet expressions in special tags
    - It's easy for any Puppet user to read, but only works with newer Puppet versions
  - Embedded Ruby (ERB) uses Ruby code in tags
    - You need to know a small bit of Ruby to read it, but it works with all Puppet versions

With Template

- You can put template files in the templates directory of a module
- EPP files should have the .epp extension, and ERB files should have the .erb extension

```
# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure  => file,
  content => epp('ntp/ntp.conf.epp', {'service_name' => 'xntpd', 'iburst_enable'
=> true}),
  # Loads
/etc/puppetlabs/code/environments/production/modules/ntp/templates/ntp.conf.epp
}

# template(<FILE REFERENCE>, [<ADDITIONAL FILES>, ...])
file { '/etc/ntp.conf':
  ensure  => file,
  content => template('ntp/ntp.conf.erb'),
  # Loads
/etc/puppetlabs/code/environments/production/modules/ntp/templates/ntp.conf.erb
}
```

THANK YOU