Puppet for configuration management

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

- 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

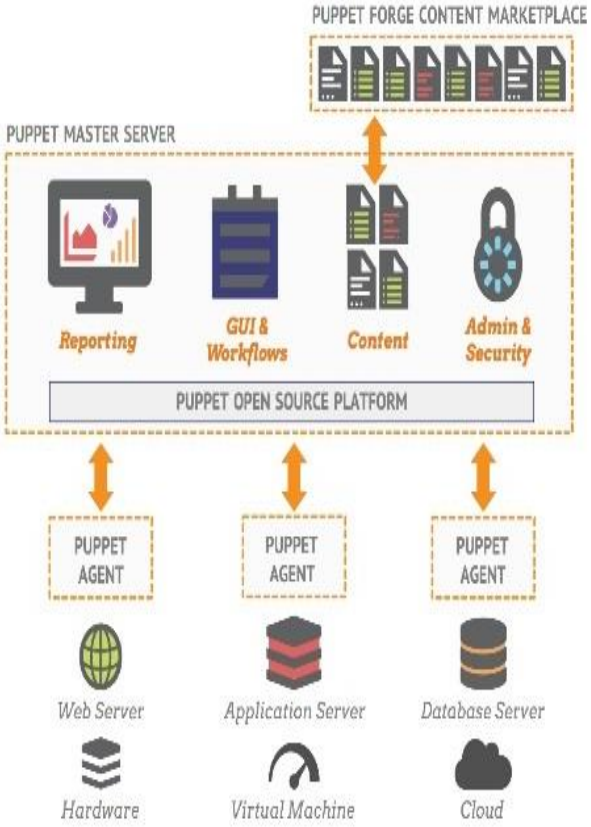
And if its wrong, SCM can determine errors

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

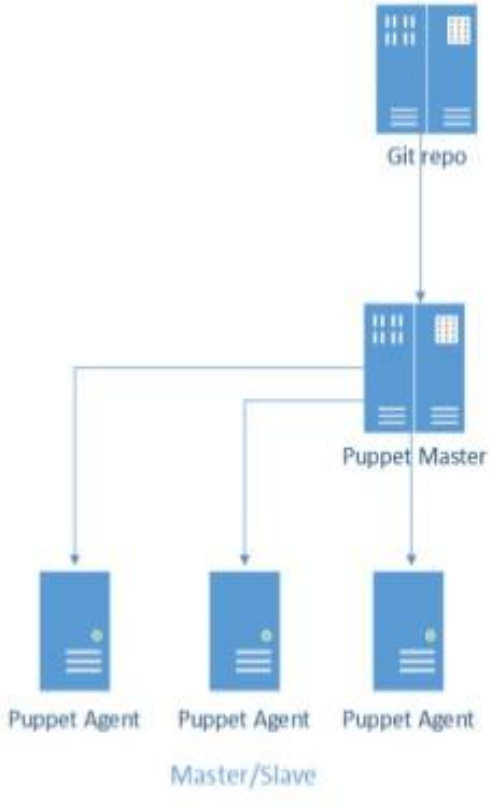
Puppet Setup

- 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

- 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 Open Source Or Enterprise

- Open source Puppet is great for individuals managing a small set of servers
- But when you have complex or large infrastructure that's used and managed by different teams, Puppet Enterprise is the way to go

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 behaviour.
- Classes and configuration files are organized in modules.

Puppet 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=running enable=true
```

- Require types that define a resource
- Require providers that define interaction

```
file{‘/data’ :
   ensure => directory,
}
package{‘mypackage’ :
   provider => ‘gem’,
}
```

https://docs.puppetlabs.com/references/latest/type.html
https://docs.puppetlabs.com/references/latest/type.html#package

Exercise 1: Install Puppet on Ubuntu 14.04 LTS

- Install Puppet Master and 2 Agents on Amazon AWS
- Run some basic Puppet commands
- Applying a simple manifest

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
```

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

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

Exercise 2: Write Puppet Manifest

- Create some Puppet Manifests and apply them

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

Resource Type Reference

- 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 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" ],
}
```

Exercise 3: Resources & Resource Types

- Run some Puppet resource commands
- Apply resources to node

Puppet Nodes

- A node definition or node statement is a block of Puppet code that will only be included in matching nodes' catalogs. This feature allows you to assign specific configurations to specific nodes.
- Node statements are an optional feature of Puppet. They can be replaced by or combined with an external node classifier, or you can eschew both and use conditional statements with facts to classify nodes.
- Unlike more general conditional structures, node statements only match nodes by name. By default, the name of a node is its certname (which defaults to the node's fully qualified domain name).

Roles Define Policy

- 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 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

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.

Exercise 4: Puppet Nodes

- Create some Puppet Manifests for Nodes and apply them

Exercise 5: Combining Resource Types in a Single Manifest

- Create some Puppet Manifests for Nodes and apply them

Puppet Resource Collectors

- Resource collectors select a group of resources by searching the attributes of every resource in the catalog
- This search is independent of evaluation-order (that is, it even includes resources which haven’t yet been declared at the time the collector is written)
- Collectors realize virtual resources, can be used in chaining statements, and can override resource attributes

```
User <| title == 'luke' |> # Will collect a single user resource whose title is 'luke’
User <| groups == 'admin' |> # Will collect any user resource whose list of supplemental groups includes 'admin’
Yumrepo['custom_packages'] -> Package <| tag == 'custom' |> # Will create an order relationship with several package resources
```

Puppet Virtual Resources

- A virtual resource declaration specifies a desired state for a resource without enforcing that state
- Puppet manages the resource by realizing it elsewhere in your manifests
- This divides the work done by a normal resource declaration into two steps
- Although virtual resources are declared once, they can be realized any number of times, similar to a class.

Uses of Virtual Resources

- Virtual resources are useful for:
  - Resources whose management depends on at least one of multiple conditions being met
  - Overlapping sets of resources required by any number of classes
  - Resources which should only be managed if multiple cross-class conditions are met
- Because they both offer a safe way to add a resource to the catalog in multiple locations, virtual resources can be used in some of the same situations as classes
- The features that distinguish virtual resources are:
  - Searchability via resource collectors, which helps to realize overlapping clumps of virtual resources.
  - Flatness, such that you can declare a virtual resource and realize it a few lines later without having to clutter your modules with many single-resource classes.

Example Virtual Resources

- Virtual resources are used in two steps: declaring and realizing
- Declare: modules/apache/manifests/init.pp

```
@a2mod { 'rewrite':
        ensure => present,
}
```

note: The a2mod resource type is from the puppetlabs-apache module

- Realize: modules/wordpress/manifests/init.pp

```
realize A2mod['rewrite']
```

- Realize again: modules/freight/manifests/init.pp

```
realize A2mod['rewrite']
```

Example Virtual Resources

- To declare a virtual resource, prepend @ (the “at” sign) to the resource type of a normal resource declaration:

```
@user {'deploy':
  uid     => 2004,
  comment => 'Deployment User',
  group   => 'www-data',
  groups  => ["enterprise"],
  tag     => [deploy, web],
}
```

- To realize one or more virtual resources by title, use the realize function, which accepts one or more resource references:

```
realize(User['deploy'], User['zleslie'])
```

Puppet Exported Resources

- An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes
- It does not manage the resource on the target system
- Any node, including the node that exports it, can collect the exported resource and manage its own copy of it

Uses of Puppet Exported Resources

- Exported resources enable the Puppet compiler to share information among nodes by combining information from multiple nodes' catalogs
- This helps manage things that rely on nodes knowing the states or activity of other nodes
- The common use cases are monitoring and backups
- A class that manages a service like PostgreSQL, exports a nagios_service resource which describes how to monitor the service, including information such as its hostname and port
- The Nagios server collects every nagios_service resource, and automatically starts monitoring the Postgres server

Exported Resource Example

```
class ssh {
  # Declare:
  @@sshkey { $::hostname:
      type => dsa,
      key => $::sshdsakey,
  }
  # Collect:
  Sshkey <<| |>>
}
```

Exported Resource Example

- To declare an exported resource, prepend @@ to the resource type of a standard resource declaration:

```
@@nagios_service { "check_zfs${::hostname}":
  use                 => 'generic-service',
  host_name           => $::fqdn,
  check_command       => 'check_nrpe_1arg!check_zfs',
  service_description => "check_zfs${::hostname}",
  target              => '/etc/nagios3/conf.d/nagios_service.cfg',
  notify              => Service[$nagios::params::nagios_service],
}
```

- To collect exported resources, use an exported resource collector. Collect all exported nagios_service resources:

```
Nagios_service <<| |>>
```

Puppet Relationships - Metaparameters

- Puppet uses four metaparameters to establish relationships, and you can set each of them as an attribute in any resource

  before — Applies a resource before the target resource

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
```

  require — Applies a resource after the target resource

```
file { '/etc/ssh/sshd_config':
  ensure  => file,
  mode    => '0600',
  source  => 'puppet:///modules/sshd/sshd_config',
  require => Package['openssh-server'],
}
```

Puppet Relationships - Metaparameters

- notify — Applies a resource before the target resource. The target resource refreshes if the notifying resource changes

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  notify => Service['sshd'],
}
```

- subscribe — Applies a resource after the target resource. The subscribing resource refreshes if the target resource changes

```
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

Puppet Relationships – Chaining Arrows

- You can create relationships between two resources or groups of resources using the -> and ~> operators.
  - -> (ordering arrow; a hyphen and a greater-than sign) — Applies the resource on the left before the resource on the right.
  - ~> (notifying arrow; a tilde and a greater-than sign) — Applies the resource on the left first. If the left-hand resource changes, the right-hand resource will refresh.
- The chaining arrows accept the following kinds of operands on either side of the arrow:
  - Resource references, including multi-resource references
  - Arrays of resource references
  - Resource declarations
  - Resource collectors

Puppet Relationships – Chaining Arrows

- An operand can be shared between two chaining statements, which allows you to link them together into a “timeline”:

```
Package['ntp'] -> File['/etc/ntp.conf'] ~> Service['ntpd’]
```

- Since resource declarations can be chained, you can use chaining arrows to make Puppet apply a section of code in the order that it is written:

```
# first:
package { 'openssh-server':
  ensure => present,
} ->

# and then:

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
} ~> # and then:
service { 'sshd':
  ensure => running,
  enable => true,
}
```

Puppet Relationships – require Function

- The require function declares a class and causes it to become a dependency of the surrounding container:

```
class wordpress {
  require apache
  require mysql
  ...
}
```

- The above example causes every resource in the apache and mysql classes to be applied before any of the resources in the wordpress class.

Puppet Relationships – Refresh Events

- **Receiving refresh events**
  - If a resource gets a refresh event during a run and its resource type has a refresh action, it will perform that action.
  - If a resource gets a refresh event but its resource type cannot refresh, nothing happens.
  - If a class or defined resource gets a refresh event, every resource it contains will also get a refresh event.
  - A resource can perform its refresh action up to once per run. If it receives multiple refresh events, they're combined and the resource only refreshes once.
- **Sending refresh events**
  - If a resource is not in its desired state and Puppet makes changes to it during a run, it will send a refresh event to any subscribed resources.
  - If a resource performs its refresh action during a run, it will send a refresh event to any subscribed resources.
  - If Puppet changes (or refreshes) any resource in a class or defined resource, that class or defined resource will send a refresh event to any subscribed resources.
- **No-op**
  - If a resource is in no-op mode (due to the global noop setting or the per-resource noop metaparameter), it will not refresh when it receives a refresh event. However, Puppet will log a message stating what would have happened.
  - If a resource is in no-op mode but Puppet would otherwise have changed or refreshed it, it will not send refresh events to subscribed resources. However, Puppet will log messages stating what would have happened to any resources further down the subscription chain.

Puppet 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

Puppet User Variables

- We can define custom variables in different ways:
- In Puppet manifests:

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

- 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)
```

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 Data Type - String

- Strings are unstructured text fragments of any length
- There are four ways to write literal strings in the Puppet language:
  - Bare words:
    - Bare word strings are most commonly used with resource attributes that accept a limited number of one-word values.
    - Begin with a lower case letter, and contain only letters, digits, hyphens (-), and underscores (_).
    - Not be a reserved word
  - Single-quoted strings:
    - Multi-word strings can be surrounded by single quotes
    - Single-quoted strings can't interpolate values
  - Double-quoted strings:
    - Strings can also be surrounded by double quotes
    - Double-quoted strings can interpolate values. Interpolation allows strings to contain expressions, which will be replaced with their values
  - Heredocs
    - Heredocs let you quote strings with more control over escaping, interpolation, and formatting.
    - They're especially good for long strings with complicated content.

```
service { "ntp":
  ensure => running,
  # bare word string }
```

```
"String content ${<EXPRESSION>} more content"
```

Puppet Data Type – Numbers, Booleans

- Numbers in the Puppet language are normal integers and floating point numbers.
- Booleans are one-bit values, representing true or false.
- The condition of an “if” statement expects an expression that resolves to a boolean value. All of Puppet’s comparison operators resolve to boolean values, as do many functions.

Puppet Data Type – Arrays, Hashes

- Arrays are ordered lists of values
- Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array
- Many functions also take arrays, including the iteration functions
- You can access items in an array by their numerical index (counting from zero)

```
[ 'one', 'two', 'three' ]
# Equivalent: [ 'one', 'two', 'three', ]
```

- Hashes map keys to values, maintaining the order of the entries according to insertion order
- Hashes are written as a pair of curly braces containing any number of key/value pairs
- A key is separated from its value by a => (arrow, fat comma, or hash rocket), and adjacent pairs are separated by commas

```
{ 'key1' => 'val1', key2 => 'val2' }
# Equivalent: { 'key1' => 'val1', key2 => 'val2', }
```

Puppet Data Type – Regular Expressions

- A regular expression is a pattern that can match some set of strings, and optionally capture parts of those strings for further use

```
if $host =~ /^www(\d+)\./
  { notify { "Welcome web server #$1": }
}
```

- Sensitive types in the Puppet language are strings marked as sensitive.
- The value is displayed in plain text in the catalog and manifest, but is redacted from logs and reports.
- Because the value is currently maintained as plain text, you should only use it as an aid to ensure that

```
$secret = Sensitive('myPassword’)
notice($secret)
```

Puppet Data Type – Undef Resource Reference

- Puppet's special undef value is roughly equivalent to nil in Ruby; it represents the absence of a value.
- Resource references identify a specific Puppet resource by its type and title.
- Several attributes, such as the relationship metaparameters, require resource references

```
subscribe => File['/etc/ntp.conf'],
before => Concat::Fragment['apache_port_header'],
```

- The general form of a resource reference is:
  - The resource type, capitalized (every segment must be capitalized if the resource type includes a namespace separator [::])
  - An opening square bracket
  - The title of the resource as a string, or a comma-separated list of titles
  - A closing square bracket

Puppet Data Type – Undef Resource Reference

- Puppet's special default value usually acts like a keyword in a few limited corners of the language

```
file { default:
  mode => '0600',
  owner => 'root',
  group => 'root',
  ensure => file,
}
```

- All of the resources in the block above will inherit attributes from default unless they specifically override them

Exercise 6: Applying Conditional Logic in Puppet

- Create a manifest with conditional logic code for Puppet
- Create manifest with conditional logic for facts received by facter

Exercise 7: Applying Conditional Logic using facts in Puppet

- Create manifest with conditional logic for facts received by facter

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.

```
# 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',
  }
}
```

Defining Classes

```
# A class with parameters
class apache (String $version = 'latest') {
  package {'httpd':
    ensure => $version, # Using the class parameter from above
    before => File['/etc/httpd.conf'],
  }
  file {'/etc/httpd.conf':
    ensure  => file,
    owner   => 'httpd',
    content => template('apache/httpd.conf.erb'), # Template from a module
  }
  service {'httpd':
    ensure    => running,
    enable    => true,
    subscribe => File['/etc/httpd.conf'],
  }
}
```

Class Parameters & Variables

- Parameters allow a class to request external data. If a class needs to configure itself with data other than facts, that data should usually enter the class via a parameter
- Each class parameter can be used as a normal variable inside the class definition
- The values of these variables are not set with normal assignment statements or looked up from top or node scope; instead, they are set based on user input when the class is declared
- If a class parameter lacks a default value, the module's user must set a value themselves
- You should supply defaults wherever possible
- Each parameter can be preceded by an optional data type
- If included, Puppet will check the parameter's value at runtime to make sure that it has the right data type, and raise an error if the value is illegal
- If no data type is provided, the parameter will accept values of any data type
- The special variables $title and $name are both set to the class name automatically, so they can't be used as parameters

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.

Classes – Using Include

- The include function is the standard way to declare classes.

```
include base::linux
include base::linux # no additional effect; the class is
only declared once
include Class['base::linux’]     # including a class
reference
include base::linux, apache      # including a list
$my_classes = ['base::linux', 'apache']
include $my_classes              # including an array
```

Classes – Using Require

- The require function (not to be confused with the require meta-parameter) declares one or more classes, then causes them to become a dependency of the surrounding container.

```
define apache::vhost (Integer $port, String $docroot,
String $servername, String $vhost_name) {
  require apache
  …
}
```

- In the above example, Puppet will ensure that every resource in the apache class gets applied before every resource in any apache::vhost instance

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

Exercise 8: Puppet Modules

- Create Puppet module and apply it to nodes
- Download a module from Puppet Forge and add it to a custom module

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

Templating Language

- 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
}
```

NTP Module

- To install the puppetlabs-ntp module:
- From the Puppet master, run

```
puppet module install puppetlabs-ntp
```

- The NTP module contains several classes. Classes are named chunks of Puppet code and are the primary means by which Puppet configures nodes. The NTP module contains the following classes:

```
ntp: the main class; this class includes all other NTP classes
ntp::install: this class handles the installation packages.
ntp::config: this class handles the configuration file.
ntp::service: this class handles the service.
```

- You’re going to add the ntp class to the default node in your main manifest

NTP Module

- Use your text editor to open site.pp.
- Add the following Puppet code to site.pp:

```
node default {
    class { 'ntp':
     servers => ['nist-time-server.eoni.com','nist1-
lv.ustiming.org','ntp-nist.ldsbc.edu']
     }
 }
```

- From the command line on your Puppet agent, trigger a Puppet run with

```
puppet agent –t
```

- To check if the NTP service is running, run puppet resource service ntpd on your Puppet agent. The output should be:

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

SSH Module

- Manage SSH client and server via Puppet
- To install the module from puppet forge

```
puppet module install saz-ssh --version 4.0.0
```

- Use your text editor to open site.pp.
- Add the following Puppet code to site.pp:

```
node default {
    include ssh
}

node default {
    include ssh::client
}
```

SUDO Module

- Manage SSH client and server via Puppet
- To install the module from puppet forge

```
puppet module install saz-sudo --version 5.0.0
```

- Use your text editor to open site.pp.
- Add the following Puppet code to site.pp:

```
node default {
    class { 'sudo': }
}

node default {
    include sudo
}
```


Cognixia
A Collabera LEARNING SOLUTIONS COMPANY
THANK YOU
45%