In Puppet, the combined configuration to be applied to a host is called a ***catalog***, and the process of applying it is called a ***run***.

The Puppet **client** software is called the ***agent***. Puppet calls the definition of the host itself a***node***. The Puppet **server** is called the ***master***. 

Packages, Files, and Services

The most common types of resources we'll manage with Puppet are packages, files, and services.

Packages

With Puppet, to install nginx, we give a resource declaration like this:

**# /etc/puppet/manifests/site.pp**

**node 'puppet-agent' {**

**package { 'nginx':**

**ensure => installed,**

**}**

**}**

Then, Puppet will take the necessary actions by running **apt-get** behind the scenes.

Let's look at the code in detail:

**node 'puppet-agent' {**

**...**

**}**

Note that the **node** keyword introduces a node declaration, a list of resources that are to be applied only to node 'puppet-agent'.

**package { 'nginx':**

**ensure => installed,**

**}**

In our case, there is one type of resource, package. As with the file resource, the resource declaration consists of the following:

1. The type of resource: package
2. The name of the instance: nginx
3. A list of attributes  
   Each resource type has a different list of attributes that we can control. A useful attribute for package resources is ensure. We use this attribute to install (or sometimes remove) packages.
4. **ensure => installed,**

Run Puppet:

**root@puppet-agent:~# puppet agent --test**

**Info: Retrieving plugin**

**Info: Caching catalog for puppet-agent.ec2.internal**

**Info: Applying configuration version '1419668689'**

**Notice: /Stage[main]/Main/Node[puppet-agent]/Package[nginx]/ensure: ensure changed 'purged' to 'present'**

**Notice: Finished catalog run in 4.08 seconds**

When we apply the manifest, Puppet checks whether the nginx package is installed. If this is the first time we've applied the manifest, the package probably won't be present, so Puppet prints a message telling us that the package is being installed as shown above

Once the resource has been created the first time, subsequent Puppet runs will do nothing because the state of the system already matches the manifest:

**root@puppet-agent:~# puppet agent --test**

**Info: Retrieving plugin**

**Info: Caching catalog for puppet-agent.ec2.internal**

**Info: Applying configuration version '1419668689'**

**Notice: Finished catalog run in 0.03 seconds**

Removing Package

Sometimes we need to make sure a package is removed entirely from a machine:

**package { 'nginx':**

**ensure => absent,**

**}**

Using **ensure => absent** will remove the package if it's installed.

Updating Package

Another value that ensure can take on a package resource is latest. This will cause Puppet to check which version of the package is available in the repository (if we're using Ubuntu, this includes any additional APT sources that we may have configured, such as the Puppet Labs repo). If it is newer than the installed version, Puppet will upgrade the package to the latest version:

**package { 'nginx':**

**ensure => latest,**

**}**

Modules

To make our Puppet manifests more readable and maintainable, it's a good idea to arrange them into modules. A Puppet module is a way of grouping related resources. In our example, we're going to make an nginx module that will contain all Puppet code relating to nginx.

Create the file **modules/nginx/manifests/init.pp** with the following contents:

**# /etc/puppet/modules/nginx/manifests/init.pp**

**class nginx {**

**package { 'nginx':**

**ensure => installed,**

**}**

**}**

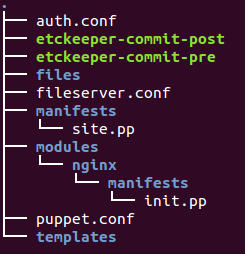
Edit the **manifests/sites.pp** file as follows:

**# /etc/puppet/manifests/site.pp**

**node 'puppet-agent' {**

**include nginx**

**}**



Run puppet:

**ubuntu@puppet-agent:~$ sudo puppet agent --test**

**Info: Retrieving plugin**

**Info: Caching catalog for puppet-agent.ec2.internal**

**Info: Applying configuration version '1419714937'**

**Notice: Finished catalog run in 0.04 seconds**

Note that we've modified our puppet code for node from:

**# /etc/puppet/manifests/site.pp**

**node 'puppet-agent' {**

**package { 'nginx':**

**ensure => installed,**

**}**

**}**

to:

**# /etc/puppet/manifests/site.pp**

**node 'puppet-agent' {**

**include nginx**

**}**

As we can see the nginx resource has been replaced by the line **include nginx**. We're telling Puppet, "Look for a class called nginx and include all the resources on this node."

The **class** keyword declares a group of resources (here, the package resource for nginx) identified by the name nginx. We can then use the **include** keyword elsewhere to include all the resources in the class at once.

But why we do this?   
That's because this enables us to include the nginx class on many nodes without repeating the same resource declarations over and over as shown below:

**node 'node1' {**

**include nginx**

**}**

**node 'node2' {**

**include nginx**

**}**

**node 'node3' {**

**include nginx**

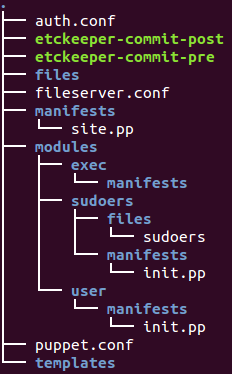
**}**

Also, by grouping resources into classes and modules helps us organize our code, and it's easy to read and maintain.

exec resources

We want Puppet to run a certain command directly using an **exec resource**. This is a very flexible and powerful resource, and we can use it to implement almost anything in Puppet.

**ubuntu@ip-172-31-45-62:/etc/puppet$ sudo mkdir -p modules/exec/manifests**



**# modules/exec/manifests/init.pp**

**class exec {**

**exec { 'Run a command':**

**command => '/bin/echo `/bin/date` >/tmp/output.txt',**

**}**

**}**

The line

**exec { 'Run a command':**

declares an **exec** resource with the name 'Run a command'. The name can be anything; it's not otherwise used by Puppet, except that like all resource names it can't be the same as another instance of the same resource type.

**echo** and **date**, are specified with their full path. This is because Puppet wants to be sure exactly which command we mean. When Puppet runs, it applies the exec resource by running the command:

**command => '/bin/echo `/bin/date` >/tmp/output.txt',**

This command will write the following text to **/tmp/output.txt**:

With the node definition:

**# manifests/site.pp**

**node 'puppet-agent' {**

**include user**

**include sudoers**

**include exec**

**}**

Run Puppet:

**ubuntu@puppet-agent:/etc/puppet$ sudo puppet agent --test**

**Info: Retrieving plugin**

**Info: Caching catalog for puppet-agent.ec2.internal**

**Info: Applying configuration version '1419811309'**

**Notice: /Stage[main]/Exec/Exec[Run a command]/returns: executed successfully**

**Notice: Finished catalog run in 0.11 seconds**

Check the output produced:

**ubuntu@puppet-agent:/etc/puppet$ cat /tmp/output.txt**

**Mon Dec 29 00:01:59 UTC 2014**

Bottom of Form

Command search paths - I

As in the previous section, Puppet requires us to specify the full path to any command referenced in an exec resource. However, we can provide a list of paths for Puppet to search for commands, using the **path** attribute. For example:

**class exec {**

**exec { 'Run a command':**

**#command => '/bin/echo `/bin/date` >/tmp/output.txt',**

**command => 'echo `date` >/tmp/output.txt',**

**path => ['/bin', '/usr/bin'],**

**}**

**}**

Now when Puppet sees a command name, it will search the directories you specify looking for the matching commands.

Command search paths - II

If we want to specify a set of default search paths for all exec resources, we can put this in our**manifests/site.pp** file:

**Exec {**

**path => ['/bin', '/usr/bin'],**

**}**

Note the capital 'E' for **Exec**. This means "make this the default for all exec resources." Then we can use unqualified commands without an explicit path attribute. So, the**modules/exec/manifests/init.pp** now looks like this:

**class exec {**

**exec { 'Run a command':**

**command => 'echo `date` >/tmp/output.txt',**

**}**

**}**

Check a request on master

**$ sudo puppet cert list -all**

**"ip-172-31-53-211.ec2.internal" (SHA256) 13:2E:06:C1:D8:51:E6:34:61:85:3C:B5:82:99:F4:FB:96:04:B3:CB:6F:05:67:AD:D6:93:E7:C7:E4:2E:BB:A0**

**+ "ip-172-31-53-212.ec2.internal" (SHA256) 07:C0:0E:C0:B0:30:51:89:02:AE:1B:AF:3C:C5:40:1D:5A:E5:DF:BE:00:18:23:22:2E:FA:8B:6A:7B:E3:89:60 (alt names: "DNS:ip-172-31-53-212.ec2.internal", "DNS:puppet", "DNS:puppet.ec2.internal")**

Sign a request

As mentioned in previous sections, in an agent/master deployment, an admin must approve a certificate request for each agent node before that node can fetch configurations. Agent nodes will request certificates the first time they attempt to run.

To sign a certificate request, use the puppet cert sign command, with the hostname of the certificate we want to sign. For example, to sign 'puppet\_agent.example.com', we would use the following command:

**$ sudo puppet cert sign ip-172-31-53-211.ec2.internal**

**Notice: Signed certificate request for ip-172-31-53-211.ec2.internal**

**Notice: Removing file Puppet::SSL::CertificateRequest ip-172-31-53-211.ec2.internal at '/var/lib/puppet/ssl/ca/requests/ip-172-31-53-211.ec2.internal.pem'**

Once Puppet agent has been signed, the Puppet master can now communicate and control the node that the signed certificate belongs to.

Puppet Forge modules

The modules in the Puppet Forge repository is publically-available modules, and they can be useful when we develop our own infrastructure. The Puppet Forge modules can be quickly installed with built-in **puppet module** command.

Since Apache and MySQL are available via Puppet Forge, we will demonstrate how they can be used to help us set up our lamp stack.

Install Apache and MySQL modules

On our Puppet master, install the **puppetlabs-apache** module:

**$ sudo puppet module install puppetlabs-apache**

We will see the following output, which indicates the modules installed correctly:

**Notice: Preparing to install into /etc/puppet/modules ...**

**Notice: Downloading from https://forgeapi.puppetlabs.com ...**

**Notice: Installing -- do not interrupt ...**

**/etc/puppet/modules**

**--- puppetlabs-apache (v1.1.1)**

**--- puppetlabs-concat (v1.1.2)**

**--- puppetlabs-stdlib (v4.3.2)**

Also, we need to install the **puppetlabs-mysql** module:

**$ sudo puppet module install puppetlabs-mysql**

**Notice: Preparing to install into /etc/puppet/modules ...**

**Notice: Downloading from https://forgeapi.puppetlabs.com ...**

**Notice: Installing -- do not interrupt ...**

**/etc/puppet/modules**

**--- puppetlabs-mysql (v2.3.1)**

**--- puppetlabs-stdlib (v4.3.2)**

Now, the apache and mysql modules are available.

Main manifest

Now let's edit our main manifest so it uses the new modules to install our lamp stack. On the Puppet master, edit the main manifest (**/etc/puppet/manifests/site.pp**):

**node default { }**

**node 'ip-172-31-53-211' {**

**class { 'apache': # use the "apache" module**

**default\_vhost => false, # don't use the default vhost**

**default\_mods => false, # don't load default mods**

**mpm\_module => 'prefork', # use the "prefork" mpm\_module**

**}**

**include apache::mod::php # include mod php**

**apache::vhost { 'example.com': # create a vhost called "example.com"**

**port => '80', # use port 80**

**docroot => '/var/www/html', # set the docroot to the /var/www/html**

**}**

**class { 'mysql::server':**

**root\_password => 'password',**

**}**

**file { 'info.php': # file resource name**

**path => '/var/www/html/info.php', # destination path**

**ensure => file,**

**require => Class['apache'], # require apache class be used**

**source => 'puppet:///modules/apache/info.php', # specify location of file to be copied**

**}**

**}**

The apache module can be passed parameters that override the default behavior of the module. We are passing in some basic settings that disable the default virtual host that the module creates, and make sure we create a virtual host that can use php.

Using the MySQL module is similar to using the Apache module. We will keep it simple since we are not actually using the database this time.

The file resource ensures **info.php** gets copied to the proper location. This time, we will use the source parameter to specify a file to copy. This file resource declaration is slightly different from the one we used in our previous example. The main difference is that we are specifying the source parameter instead of the content parameter. Source tells puppet to copy a file over, instead of simply specifying the file's contents. The specified source,**puppet:///modules/apache/info.php** gets interpreted by Puppet into**/etc/puppet/modules/apache/files/info.php**, so we must create the source file in order for this resource declaration to work properly.

info.php

Let's create the **info.php** file with the following command:

**sudo sh -c 'echo "<?php phpinfo(); ?>" > /etc/puppet/modules/apache/files/info.php'**

Pulling the configuration

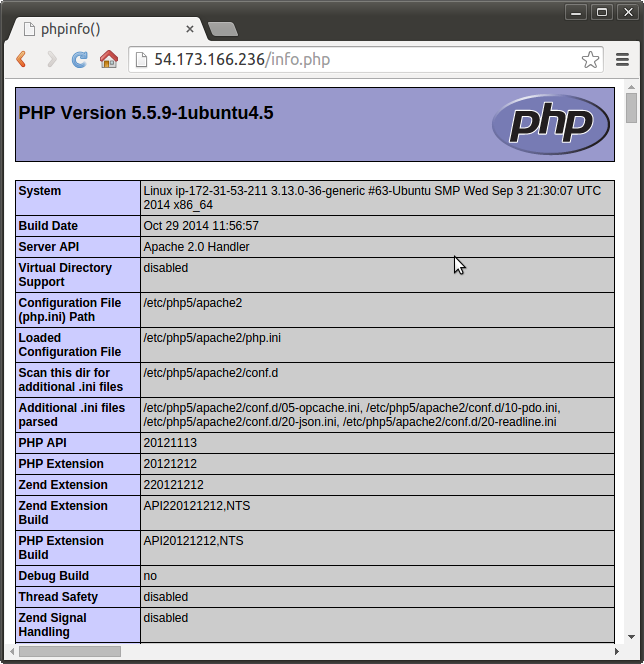
The next time our 'ip-172-31-53-211' Puppet agent node pulls its configuration from the master ('ip-172-31-53-212'), it will evaluate the main manifest and apply the module that specifies a lamp stack setup.

But if we want to try it out immediately, run the following command on the 'ip-172-31-53-211' Puppet agent node:

**$ sudo puppet agent --test**

Once it completes, we can see that a basic lamp stack is set up:

Master node:



Why do we need classes?

We don't want a huge **/etc/puppet/manifests/site.pp** file, and the files should be splitted into chunks of logically related code out into their own files, and then refer to those chunks by name when we need them.

"Classes are Puppet's way of separating out chunks of code, and modules are Puppet's way of organizing classes so that you can refer to them by name." - [Learning Puppet - Modules and Classes](https://docs.puppetlabs.com/learning/modules1.html)

classes

1. Defining a class makes it available by name, but doesn't automatically evaluate the code inside it.  
   Before we can use a class, we must define it, which is done with the class keyword, a name, curly braces, and a block of code:
2. **class my\_class {**
3. **... puppet code ...**
4. **}**

This manifest does nothing.

1. Declaring a class evaluates the code in the class, and applies all of its resources.  
   This one actually does something.
2. **class my\_class {**
3. **... puppet code ...**
4. **}**
5. **include my\_class**

class names

Class names must start with a lowercase letter, and can contain lowercase letters, numbers, and underscores. Class names can also use a double colon (::) as a namespace separator.

Class & Variable Scope

Each class definition introduces a new variable scope. Any variables we assign inside the class won't be accessible by their short names outside the class; to get at them from elsewhere, we have to use the fully-qualified name, e.g. $ntp::service\_name, as shown in the example below (**modules/ntp/manifests/init.pp**):

**class ntp {**

**case $operatingsystem {**

**centos, redhat: {**

**$service\_name = 'ntpd'**

**$conf\_file = 'ntp.conf.el'**

**}**

**debian, ubuntu: {**

**$service\_name = 'ntp'**

**$conf\_file = 'ntp.conf.debian'**

**}**

**}**

**package { 'ntp':**

**ensure => installed,**

**}**

**file { 'ntp.conf':**

**path => '/etc/ntp.conf',**

**ensure => file,**

**require => Package['ntp'],**

**source => "/root/examples/answers/${conf\_file}"**

**}**

**service { 'ntp':**

**name => $service\_name,**

**ensure => running,**

**enable => true,**

**subscribe => File['ntp.conf'],**

**}**

**}**

We can assign new, local values to variable names that were already used at top scope. For example, we could specify a new local value for **$var**.

define vs declaration

**Defining** makes a class available, and **declaring** evaluates it. We can see that in action by trying to apply our manifest in the previous section:

**ubuntu@ip-172-31-45-62:/etc/puppet$ sudo puppet apply modules/ntp/manifests/init.pp**

**Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds**

**Notice: Finished catalog run in 0.02 seconds**

We can see it does nothing, because we only defined the class.

To declare a class, we should use the **include** function with the class's name:

**node 'puppet-agent' {**

**include ntp**

**...**

**}**

Run puppet:

**ubuntu@puppet-agent:~$ sudo puppet agent --test**

**Info: Retrieving plugin**

**Info: Caching catalog for puppet-agent.ec2.internal**

**Info: Applying configuration version '1419831895'**

**Notice: /Stage[main]/Exec/Exec[Run a command]/returns: executed successfully**

**Notice: Finished catalog run in 0.18 seconds**

Modules

Now we know how to define and declare classes. We could have done everything in a single manifest (**site.pp**), but we actually splitted up our manifests into an easier to understand structure, and used **manifests/site.pp**.

**node 'puppet-agent' {**

**include ntp**

**include nginx**

**include user**

**include sudoers**

**include exec**

**Exec {**

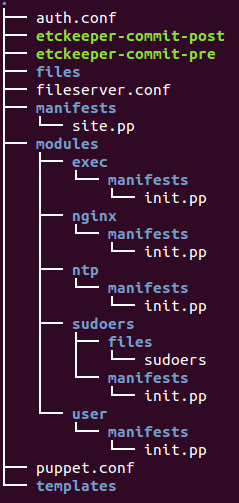
**path => ['/bin', '/usr/bin'],**

**}**

**}**

By stowing the implementation of a feature in a module, our main manifest can become much smaller, more readable, and policy-focused - we can tell at a glance what will be configured on our nodes, and if we need implementation details on something, we can delve into the module.

Module Structure



1. A module is a directory.
2. The module's name must be the name of the directory.
3. It contains a manifests directory, which can contain any number of **.pp** files.
4. The manifests directory should always contain an **init.pp** file.
   1. This file must contain a single class definition. The class's name must be the same as the module's name.

###################################################################

Services

To manage services, use the service resource type. The ensure attribute controls whether or not the service should be running. To specify that the service should be running, use

**ensure => running**

To specify that it should be stopped, use

**ensure => stopped**

The enable attribute controls whether or not a service is started at boot time. To start the service at boot time, use

**enable => true**

If we don't want it to start on boot, use

**enable => false**

Resource dependencies

We can specify a dependency between resources using the require attribute:

**require => Package['nginx']**

If resource 'resB' requires resource 'resA', then Puppet will make sure the resources are applied in the right order.

Files

We can have Puppet deploy a copy of a file using the source attribute:

**file { '/etc/nginx/sites-enabled/default':**

**source => 'puppet:///modules/nginx/myfile',**

**}**

File resources can trigger a service to be restarted using the notify attribute. This is useful for configuration files, for which changes often don't take effect until the relevant service is restarted:

**notify => Service['nginx'],**

Services

The following code declares a resource of type service:

**class nginx {**

**package { 'apache2.2-common':**

**ensure => absent,**

**}**

**package { 'nginx':**

**ensure => installed,**

**require => Package['apache2.2-common'],**

**}**

**service { 'nginx':**

**ensure => running,**

**require => Package['nginx'],**

**}**

**}**

The first part of the code:

**package { 'apache2.2-common':**

**ensure => absent,**

**}**

On Ubuntu, the default setup includes the Apache web server, which would conflict with nginx if we tried to run it at the same time. So by specifying **ensure => absent**, we remove the Apache package.

The middle section declares the nginx package:

**package { 'nginx':**

**ensure => installed,**

**require => Package['apache2.2-common'],**

**}**

The require attribute tells Puppet that this resource depends on another resource, which must be applied first. In this case, we want the removal of **Apache** to be applied before the installation of **nginx**.

In the last part, we declare the nginx service:

**service { 'nginx':**

**ensure => running,**

**require => Package['nginx'],**

**}**

Service resources manage daemons, or background processes, on the server. The **ensure**attribute tells Puppet what state the service should be in:

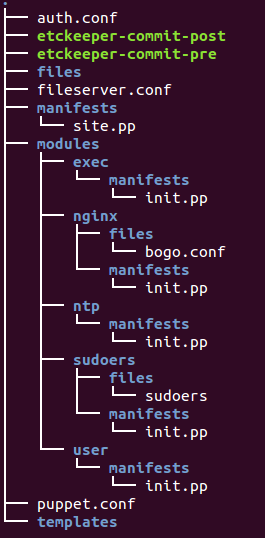
**ensure => running,**

Files

Though nginx is installed and running, but it's not serving a website yet. To do that, we have to have Puppet install a config file on the server to define an nginx **virtual host**. This will tell nginx how to respond to requests for the 'bogo' website.

We'll create a simple website for nginx to serve:

1. Create the directory **/var/www/bogo**:
2. **ubuntu@ip-172-31-45-62:~$ sudo mkdir -p /var/www/bogo**
3. Add an HTML file:
4. **ubuntu@ip-172-31-45-62:~$ sudo su -c 'echo "sample site" >/var/www/bogo/index.html'**
5. Now we want to create the virtual host file for Puppet to deploy.  
   Create the directory **modules/nginx/files**:
6. **ubuntu@ip-172-31-45-62:~$ sudo mkdir -p modules/nginx/files**
7. Create the file **modules/nginx/files/bogo.conf** with the following contents:
8. **server {**
9. **listen 80;**
10. **root /var/www/bogo;**
11. **server\_name bogo.com;**
12. **}**
13. Edit the file **modules/nginx/manifests/init.pp** so it looks like this:
14. **class nginx {**
15. **package { 'nginx':**
16. **ensure => installed,**
17. **}**
18. **service { 'nginx':**
19. **ensure => running,**
20. **require => Package['nginx'],**
21. **}**
22. **file { '/etc/nginx/sites-enabled/default':**
23. **source => 'puppet:///modules/nginx/bogo.conf',**
24. **notify => Service['nginx'],**
25. **}**
26. **}**



**file { '/etc/nginx/sites-enabled/default':**

This line declares a file resource with the path **/etc/nginx/sites-enabled/default**.

**source => 'puppet:///modules/nginx/bogo.conf',**

**source** is a file attribute that tells Puppet where to find a copy of the file:

**puppet:///modules/nginx/bogo.conf**

This looks like a URL, but it tells Puppet to look in the **modules/nginx/files** directory for a file named cat-pictures.conf.

**notify** is an attribute that tells Service['nginx'] to restart whenever there is change.

**notify => Service['nginx'],**

1. Run puppet and make sure everything worked properly, request the website:
2. **ubuntu@puppet-agent:~$ sudo puppet agent --test**
3. **Info: Retrieving plugin**
4. **Info: Caching catalog for puppet-agent.ec2.internal**
5. **Info: Applying configuration version '1419914163'**
6. **Notice: /Stage[main]/Exec/Exec[Run a command]/returns: executed successfully**
7. **Notice: Finished catalog run in 0.29 seconds**
8. **ubuntu@puppet-agent:~$ netstat -antlp**
9. **(No info could be read for "-p": geteuid()=1000 but you should be root.)**
10. **Active Internet connections (servers and established)**
11. **Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name**
12. **tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN -**
13. **tcp 0 0 0.0.0.0:22 0.0.0.0:\* LISTEN -**
14. **tcp 0 0 172.31.43.38:22 64.71.28.178:37040 ESTABLISHED -**
15. **tcp6 0 0 :::22 :::\* LISTEN -**
17. **ubuntu@puppet-agent:~$ curl localhost**
18. **<html>**
19. **<head><title>404 Not Found</title></head>**
20. **<body bgcolor="white">**
21. **<center><h1>404 Not Found</h1></center>**
22. **<hr><center>nginx/1.4.6 (Ubuntu)</center>**
23. **</body>**
24. **</html>**

package-file-service pattern

The following covers most services:

**class NAME {**

**package { NAME:**

**ensure => installed,**

**}**

**service { NAME:**

**ensure => running,**

**require => Package[NAME],**

**}**

**file { '/etc/NAME.conf':**

**source => 'puppet:///modules/NAME/NAME.conf',**

**notify => Service[NAME],**

**}**

**}**

1. The service NAME should be running
2. Before the service NAME is started, the package NAME should be installed
3. Before the service NAME is started, the file **/etc/NAME.conf** should be present (remember that "A notifies B" implies "B requires A")
4. If the file **/etc/NAME.conf** changes, restart the service NAME

templates

In the chapter, [Puppet packages, services, and files II with nginx](http://www.bogotobogo.com/DevOps/Puppet/puppet_packages_files_services2_nginx.php), we had Puppet deploy an nginx virtual host file for the 'SAMPLE' application where we simply used a file resource with the **SAMPLE.conf** file distributed from Puppet master to our EC2 agent node.

In this chapter, we'll learn how to template configuration files out with Puppet, filling in variables with the managed node's facts.

Often we may want to maintain configuration files for applications that are different between servers. If we have a couple of configurations, it's not touch to maintain multiple files, but what if we have a very large number of differing configurations? We can manage this situation by writing ERB templates and populating the templates with node-specific information. This can be done in Puppet with the **template()** function:

Puppet supports templates written in the **ERB** templating and it can be used to specify the contents of files.

Now we want to manage many different websites, it would quickly become tedious to supply an almost identical virtual host file for each site, altering only the name and domain of the site.

The best way of doing this is to give Puppet master a template file into which it could just insert these variables for each different site. The **template()** function serves just this purpose. Anywhere we have multiple files that differ only slightly, or files that need to contain dynamic information, we just use a template.

template evaluation

Templates are evaluated via a simple function as show in the following example:

**$value = template("my\_module/my\_template.erb")**

Template files should be stored in the **templates** directory of a Puppet module, which allows the template function to locate them with the simplified path format shown above. For example, the file referenced by **template("my\_module/my\_template.erb")** would be found on disk at **/etc/puppet/modules/my\_module/templates/my\_template.erb**.

Templates are always evaluated **by the parser**, **not by the client**. This means that if we're using a puppet master server, then the templates only need to be on the server, and we never need to download them to the client. The client sees no difference between using a template and specifying all of the text of the file as a string. - [Docs: Using Puppet Templates](https://docs.puppetlabs.com/guides/templating.html)

conf file - deploying a virtual host

Suppose new we want to build three new sites: SAMPLE1.com, SAMPLE2.com, and SAMPLE3.com. To prepare for this, we need to change the Puppet config for SAMPLE.com to use a template so that we can later use the same template for the new sites.

1. Here is our new **modules/nginx/manifests/init.pp** file:
2. **class nginx {**
3. **package { 'nginx':**
4. **ensure => installed,**
5. **}**
6. **service { 'nginx':**
7. **ensure => running,**
8. **enable => true,**
9. **require => Package['nginx'],**
10. **}**
11. **file { '/etc/nginx/sites-enabled/default':**
12. **ensure => 'absent',**
13. **}**
14. **}**

Since we previously used the file **/etc/nginx/sites-enabled/ default** as the virtual host for SAMPLE.com ([Puppet packages, services, and files II with nginx](http://www.bogotobogo.com/DevOps/Puppet/puppet_packages_files_services2_nginx.php)), we need to remove that now:

**file { '/etc/nginx/sites-enabled/default':**

**ensure => absent,**

**}**

1. We want to reate a new templates directory in the nginx module:
2. **ubuntu@ip-172-31-45-62:/etc/puppet$ sudo mkdir -p modules/nginx/templates**
3. Create the file **modules/nginx/templates/vhost.conf.erb** with the following contents. The template file is for the virtual host definition:
4. **server {**
5. **listen 80;**
6. **root /var/www/<%= @site\_name %>;**
7. **server\_name <%= @site\_domain %>;**
8. **}**

The <%= %> signs mark where parameters will go; we will supply **site\_name** and **site\_domain** later, when we use the template. Puppet will replace <%= @site\_name %> with the value of the site\_name variable.

1. Then in the **site.pp** file, we include the nginx module on the node:
2. **node 'puppet-agent' {**
3. **include nginx**
4. **$site\_name = 'SAMPLE'**
5. **$site\_domain = 'SAMPLE.com'**
6. **file { '/etc/nginx/sites-enabled/SAMPLE.conf':**
7. **content => template('nginx/vhost.conf.erb'),**
8. **notify => Service['nginx'],**
9. **}**
10. **...**
11. **}**

Note that before using the template, we need to set values for the variables **site\_name**and **site\_domain**:

**$site\_name = 'SAMPLE'**

**$site\_domain = 'SAMPLE.com'**

Also note that when we refer to these variables in Puppet code, we use a **$ prefix ($site\_name)**, but in the template it's an **@ prefix (@site\_name)**. This is because in templates we're actually writing Ruby, not Puppet!

Now we can use the template to generate the nginx virtual host file:

**file { '/etc/nginx/sites-enabled/SAMPLE.conf':**

**content => template('nginx/vhost.conf.erb'),**

**notify => Service['nginx'],**

**}**

This looks just like any other file resource, with a content attribute. Though we can give the contents of the file as a literal string:

**content => "Hello, SAMPLEtoSAMPLE\n",**

However, here we call the template function:

**content => template('nginx/vhost.conf.erb'),**

The argument to template tells Puppet where to find the template file. The path

**nginx/vhost.conf.erb**

Translates to

**modules/nginx/templates/vhost.conf.erb**

1. Run puppet:
2. **ubuntu@ip-172-31-45-62:/etc/puppet$ sudo puppet apply modules/nginx/manifests/init.pp**
3. **Notice: Compiled catalog for ip-172-31-45-62.ec2.internal in environment production in 0.02 seconds**
4. **Notice: Finished catalog run in 0.02 seconds**
5. Check the resulting virtual host file on agent node:
6. **ubuntu@puppet-agent:~$ cat /etc/nginx/sites-enabled/SAMPLE.conf**
7. **server {**
8. **listen 80;**
9. **root /var/www/SAMPLE;**
10. **server\_name SAMPLE.com;**
11. **}**

Puppet now evaluated the template, inserted the values of any variables referenced in**<%= %>** signs in **modules/nginx/templates/vhost.conf.erb**, and generated the final output as we can see in the **SAMPLE.conf**.

garethr/docker from Puppet forge

Let's install [garethr/docker](https://forge.puppetlabs.com/garethr/docker" \t "_blank) from Puppet forge. It's a module for installing and managing docker.

**$ sudo puppet module install garethr-docker**

**Notice: Preparing to install into /etc/puppet/modules ...**

**Notice: Downloading from https://forgeapi.puppetlabs.com ...**

**Notice: Installing -- do not interrupt ...**

**/etc/puppet/modules**

**|--- garethr-docker (v4.1.1)**

**|-- puppetlabs-apt (v2.1.1)**

**|-- puppetlabs-stdlib (v4.9.0)**

**|-- stahnma-epel (v1.1.1)**

docker\_example.pp

The **/etc/puppet/manifests/docker\_example.pp** should look like as shown below:

**include 'docker'**

**docker::image { 'ubuntu':**

**image\_tag => 'trusty',**

**}**

The module includes a single class:

**include 'docker'**

By default, this sets up the docker hosted repository if necessary for our OS, and installs the docker package and on Ubuntu, any required Kernel extensions.

**docker::image { 'ubuntu':**

**image\_tag => 'trusty'**

**}**

The image tags is equivalent to running **docker pull -t="trusty"** ubuntu. Note that the image will only install if an image of that name does not already exist.

manifest apply

Let's apply the puppet manifest (**/etc/puppet/manifests/docker\_example.pp**) in order to get docker installed on our puppet master:

**$ sudo puppet apply site.pp**

**...**

**Notice: Finished catalog run in 46.40 seconds**

Check if docker is installed

Let's check if docker has been installed:

**$ sudo docker version**

**Client version: 1.7.1**

**Client API version: 1.19**

**Go version (client): go1.4.2**

**Git commit (client): 786b29d**

**OS/Arch (client): linux/amd64**

**Server version: 1.7.1**

**Server API version: 1.19**

**Go version (server): go1.4.2**

**Git commit (server): 786b29d**

**OS/Arch (server): linux/amd64**

Also, we can check there is no running docker:

**$ sudo docker ps**

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

Check the installation of the ubuntu docker image:

**$ sudo docker images**

**REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE**

**ubuntu trusty 91e54dfb1179 3 weeks ago 188.4 MB**

It's looking good, and we have an image to work with.

manifest update

We can launch containers:

**docker::run { 'helloworld':**

**image => 'ubuntu',**

**command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',**

**}**

which is equivalent to running the following:

**docker run -d base /bin/sh -c "while true; do echo hello world; sleep 1; done"**

This will launch a Docker container managed by the local init system.

Launch docker container II

Let's run the updated puppet manifests:

**$ sudo puppet apply docker\_example.pp**

**Notice: Compiled catalog for puppet in environment production in 0.57 seconds**

**Notice: /Stage[main]/Main/Docker::Run[helloworld]/File[/etc/init.d/docker-helloworld]/ensure: created**

**Notice: /Stage[main]/Main/Docker::Run[helloworld]/Service[docker-helloworld]/ensure: ensure changed 'stopped' to 'running'**

**Notice: Finished catalog run in 2.47 seconds**

We can check if it ran successfully:

**$ sudo docker ps**

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

**f7d43960cb62 ubuntu "/bin/sh -c 'while t About a minute ago Up About a minute helloworld**

If we want, we can attach it:

**$ sudo docker attach f7d43960cb62**

**hello world**

**hello world**

**hello world**