**Getting Started With Puppet Code: Manifests and Modules**

PostedAugust 21, 2014 308.5k views [Configuration Management](https://www.digitalocean.com/community/tags/configuration-management?type=tutorials) [Conceptual](https://www.digitalocean.com/community/tags/conceptual?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

**Introduction**

After setting up Puppet in an agent/master configuration, you may need some help writing Puppet manifests and modules. In order to use Puppet effectively, you must understand how manifests and modules are constructed. This tutorial covers Puppet code basics, and will show you how to construct manifests and modules that will help you get started with using Puppet to manage your server environment. We will show three different ways to use Puppet to configure a LAMP stack on an Ubuntu 14.04 VPS.

**Prerequisites**

Before starting this tutorial, you must have a working agent/master Puppet setup. If you do not already have this, follow this tutorial: [How To Install Puppet To Manage Your Server Infrastructure](https://www.digitalocean.com/community/tutorials/how-to-install-puppet-to-manage-your-server-infrastructure).

You also need to be able to create at least one new VPS to serve as the Puppet agent node that the Puppet master will manage.

**Create a New Agent Node**

Create a new Ubuntu 14.04 VPS called "lamp-1", [add it as a Puppet agent node](https://www.digitalocean.com/community/tutorials/how-to-install-puppet-to-manage-your-server-infrastructure#InstallPuppetAgent), and sign its certificate request on the Puppet master.

**Puppet Code Basics**

Before getting into writing Puppet code that will configure our systems, let's step back and review some of the relevant Puppet terminology and concepts.

**Resources**

Puppet code is composed primarily of *resource declarations*. A resource describes something about the state of the system, such as a certain user or file should exist, or a package should be installed. Here is an example of a user resource declaration:

user { 'mitchell':

ensure => present,

uid => '1000',

gid => '1000',

shell => '/bin/bash',

home => '/home/mitchell'

}

Resource declarations are formatted as follows:

resource\_type { 'resource\_name'

attribute => value

...

}

Therefore, the previous resource declaration describes a user resource named 'mitchell', with the specified attributes.

To list all of the default resource types that are available to Puppet, enter the following command:

puppet resource --types

We will cover a few more resource types throughout this tutorial.

**Manifests**

Puppet programs are called manifests. Manifests are composed of puppet code and their filenames use the .pp extension. The default main manifest in Puppet installed via apt is /etc/puppet/manifests/site.pp.

If you have followed the prerequisite Puppet tutorial, you have already written a manifest that creates a file and installs Apache. We will also write a few more in this tutorial.

**Classes**

In Puppet, classes are code blocks that can be called in a code elsewhere. Using classes allows you reuse Puppet code, and can make reading manifests easier.

**Class Definition**

A class definition is where the code that composes a class lives. Defining a class makes the class available to be used in manifests, but does not actually evaluate anything.

Here is how a class **definition** is formatted:

class example\_class {

...

code

...

}

The above defines a class named "example\_class", and the Puppet code would go between the curly braces.

**Class Declaration**

A class declaration occurs when a class is called in a manifest. A class declaration tells Puppet to evaluate the code within the class. Class declarations come in two different flavors: normal and resource-like.

A **normal class declaration** occurs when the include keyword is used in Puppet code, like so:

include example\_class

This will cause Puppet to evaluate the code in *example\_class*.

A **resource-like class declaration** occurs when a class is declared like a resource, like so:

class { 'example\_class': }

Using resource-like class declarations allows you to specify *class parameters*, which override the default values of class attributes. If you followed the prerequisite tutorial, you have already used a resource-like class declaration ("apache" class) when you used the PuppetLabs Apache module to install Apache on *host2*:

node 'host2' {

class { 'apache': } # use apache module

apache::vhost { 'example.com': # define vhost resource

port => '80',

docroot => '/var/www/html'

}

}

Now that you know about resources, manifests, and classes, you will want to learn about modules.

**Modules**

A module is a collection of manifests and data (such as facts, files, and templates), and they have a specific directory structure. Modules are useful for organizing your Puppet code, because they allow you to split your code into multiple manifests. It is considered best practice to use modules to organize almost all of your Puppet manifests.

To add a module to Puppet, place it in the /etc/puppet/modules directory.

We will cover the details necessary to write your own basic module. If you want to learn more details, check out the [PuppetLabs Module Fundamentals](https://docs.puppetlabs.com/puppet/latest/reference/modules_fundamentals.html) reference guide.

**Developing a Manifest**

To demonstrate how to write a Puppet manifests, classes, and modules, we will use Puppet to set up LAMP stack on Ubuntu (similar to the setup in [this tutorial](https://www.digitalocean.com/community/tutorials/how-to-install-linux-apache-mysql-php-lamp-stack-on-ubuntu-14-04)). If you have never set up a LAMP stack before, you will want to run through the linked tutorial to familiarize yourself with how to set it up manually.

From the LAMP stack tutorial, we know that we want an Ubuntu 14.04 server with the following *resources*:

* Apache package (apache2) installed
* Apache service (apache2) running
* MySQL Server package (mysql-server) installed
* MySQL Server service (mysql) running
* PHP5 package (php5) installed
* A test PHP script file (info.php)
* Update apt before installing packages

The following three sections will show different ways to use Puppet to achieve similar results, a working LAMP server. The first example will show how to write a basic manifest that is all in one file. The second example will show how to build and use a class and module, building upon the manifest developed in the first example. Finally, the third example will show how to use pre-existing, publicly available modules to quickly and easily set up a similar LAMP stack. If you want to try all three examples, for learning purposes, we recommend starting with a fresh VPS (as described in the [prerequisites](https://www.digitalocean.com/community/tutorials/getting-started-with-puppet-code-manifests-and-modules)) each time.

**Example 1: Install LAMP with a Single Manifest**

If you have not ever written a Puppet manifest before, this example is a good place to start. The manifest will be developed on a Puppet agent node, and executed via puppet apply, so an agent/master setup is not required.

You will learn how to write a manifest that will use following types of resource declarations:

* **exec**: To execute commands, such as apt-get
* **package**: To install packages via apt
* **service**: To ensure that a service is running
* **file**: To ensure that certain files exist

**Create Manifest**

On a fresh *lamp-1* VPS, create a new manifest:

sudo vi /etc/puppet/manifests/lamp.pp

Add the following lines to declare the resources that we just determined we wanted. The inline comments detail each resource declaration:

# execute 'apt-get update'

exec { 'apt-update': # exec resource named 'apt-update'

command => '/usr/bin/apt-get update' # command this resource will run

}

# install apache2 package

package { 'apache2':

require => Exec['apt-update'], # require 'apt-update' before installing

ensure => installed,

}

# ensure apache2 service is running

service { 'apache2':

ensure => running,

}

# install mysql-server package

package { 'mysql-server':

require => Exec['apt-update'], # require 'apt-update' before installing

ensure => installed,

}

# ensure mysql service is running

service { 'mysql':

ensure => running,

}

# install php5 package

package { 'php5':

require => Exec['apt-update'], # require 'apt-update' before installing

ensure => installed,

}

# ensure info.php file exists

file { '/var/www/html/info.php':

ensure => file,

content => '<?php phpinfo(); ?>', # phpinfo code

require => Package['apache2'], # require 'apache2' package before creating

}

Save and exit.

**Apply Manifest**

Now you will want to use the puppet apply command to execute the manifest. On *lamp-1*, run this:

sudo puppet apply --test

You will see many lines of output that show how the state of your server is changing, to match the resource declarations in your manifest. If there were no errors, you should be able to visit the public IP address (or domain name, if you set that up), and see the PHP info page that indicates that Apache and PHP are working. You can also verify that MySQL was installed on your server (it has not been secured, but we're not going to worry about that for now). Congrats! You set up a LAMP stack with Puppet.

This particular setup isn't too exciting, because we did not take advantage of our agent/master setup. The manifest is currently not available to other agent nodes, and Puppet is not continuously checking (every 30 minutes) that our server is in the state that the manifest described.

Now we want to convert the manifest that we just developed into a module, so it can be used by your other Puppet nodes.

**Example 2: Install LAMP by Creating a New Module**

Now let's create a basic module, based on the LAMP manifest that was developed in example 1. We will do this on the Puppet *master* node this time. To create a module, you must create a directory (whose name matches your module name) in Puppet's modules directory, and it must contain a directory called manifests, and that directory must contain an init.pp file. The init.pp file must only contain a Puppet class that matches the module name.

**Create Module**

On the Puppet *master*, create the directory structure for a module named lamp:

cd /etc/puppet/modules

sudo mkdir -p lamp/manifests

Now create and edit your module's init.pp file:

sudo vi lamp/manifests/init.pp

Within this file, add a block for a class called "lamp", by adding the following lines:

class lamp {

}

Copy the contents of LAMP manifest that you created earlier (or copy it from example 1 above) and paste it into the *lamp* class block. In this file, you created a class definition for a "lamp" class. The code within the class is will not be evaluated at this time, but it is available to be declared. Additionally, because it complies with the Puppet conventions for defining a module, this class can be accessed as a module by other manifests.

Save and exit.

**Use Module in Main Manifest**

Now that we have a basic lamp module set up, let's configure our main manifest to use it to install a LAMP stack on *lamp-1*.

On the Puppet *master*, edit the main manifest:

sudo vi /etc/puppet/manifests/site.pp

Assuming the file is empty, add the following *node* blocks (replace "lamp-1" with the hostname of the Puppet agent that you want to install LAMP on):

node default { }

node 'lamp-1' {

}

A node block allows you to specify Puppet code that will only apply to certain agent nodes. The *default* node applies to every agent node that does not have a node block specified--we will leave it empty. The *lamp-1* node block will apply to your *lamp-1* Puppet agent node.

In the *lamp-1* node block, add the following code to use the "lamp" module that we just created:

include lamp

Now save and exit.

The next time your *lamp-1* Puppet agent node pulls its configuration from the master, it will evaluate the main manifest and apply the module that specifies a LAMP stack setup. If you want to try it out immediately, run the following command on the *lamp-1* agent node:

sudo puppet agent --test

Once it completes, you will see that a basic LAMP stack is set up, exactly like example 1. To verify that Apache and PHP are working, go to *lamp-1*'s public IP address in the a web browser:

http://lamp\_1\_public\_IP/info.php

You should see the information page for your PHP installation.

Note that you can reuse the "lamp" module that you created by declaring it in other node blocks. Using modules is the best way to promote Puppet code reuse, and it is useful for organizing your code in a logical manner.

Now we will show you how to use pre-existing modules to achieve a similar setup.

**Example 3: Install LAMP with Pre-existing Modules**

There is a repository of publically-available modules, at [the Puppet Forge](https://forge.puppetlabs.com/), that can be useful when trying to develop your own infrastructure. The Puppet Forge modules can be quickly installed with built-in puppet module command. It just so happens that modules for installing and maintaining Apache and MySQL are available here. We will demonstrate how they can be used to help us set up our LAMP stack.

**Install Apache and MySQL Modules**

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

sudo puppet module install puppetlabs-apache

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

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

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

Notice: Installing -- do not interrupt ...

/etc/puppet/modules

└─┬ puppetlabs-apache (v1.0.1)

├── puppetlabs-concat (v1.0.0) [/etc/puppet/modules]

└── puppetlabs-stdlib (v3.2.0) [/etc/puppet/modules]

Also, install the puppetlabs-mysql module:

sudo puppet module install puppetlabs-mysql

Now the *apache* and *mysql* modules are available for use!

**Edit the 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:

sudo vi /etc/puppet/manifests/site.pp

Assuming the file is empty, add the following node blocks (if you followed example 2, just delete the contents of the *lamp-1* node block):

node default { }

node 'lamp-1' {

}

Within the *lamp-1* node block, use a resource-like class declaration to use the *apache* module (the in-line comments explain each line):

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

}

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. For complete documentation of the PuppetLabs-Apache module, check out its [readme](https://forge.puppetlabs.com/puppetlabs/apache).

Using the MySQL module is similar to using the Apache module. We will keep it simple since we are not actually using the database at this point. Add the following lines within the node block:

class { 'mysql::server':

root\_password => 'password',

}

Like the Apache module, the MySQL module can be configured by passing parameters ([full documentation here](https://forge.puppetlabs.com/puppetlabs/mysql).

Now let's add the file resource that ensures info.php gets copied to the proper location. This time, we will use the *source* parameter to specify a file to copy. Add the following lines within the node block:

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

}

This file resource declaration is slightly different from before. 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.

Save and exit site.pp.

Create the info.php file with the following command:

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

The next time your *lamp-1* Puppet agent node pulls its configuration from the master, it will evaluate the main manifest and apply the module that specifies a LAMP stack setup. If you want to try it out immediately, run the following command on the *lamp-1* agent node:

sudo puppet agent --test

Once it completes, you will see that a basic LAMP stack is set up, exactly like example 1. To verify that Apache and PHP are working, go to *lamp-1*'s public IP address in the a web browser:

http://lamp\_1\_public\_IP/info.php

You should see the information page for your PHP installation.

**Conclusion**

Congratulations! You have used Puppet to set up an Ubuntu 14.04 LAMP stack.

Now that you are familiar with the basics of Puppet code, and are able to write basic manifests and modules, you should try to use Puppet to configure other aspects of your environment.

A good place to start is to use Puppet to manage your system users and your application configuration files. Remember that if you use Puppet to manage resources you must make changes to those particular resources on your Puppet master server, or they will be overwritten the next time your agent nodes do their periodic catalog pull request.

Good luck!