puppet labs

Learn Puppet:

Quest Guide for the Learning VM

Table of Contents:

Learning VM Setup

About the Learning Virtual Machine

The Learning Virtual Machine (VM) is a sandbox environment equipped with everything you'll need to get started learning Puppet and Puppet Enterprise (PE). Because we believe exploration and playfulness are key to successful learning, we've done our best to make getting started with Puppet a fun and frictionless process. The VM is powered by CentOS Linux and for your convenience, we've pre-installed Puppet Enterprise (PE) along with everything you'll need to put it into action.

Before you get started, however, we'll walk you through a few steps to get the VM configured and running.

The Learning VM comes in two flavors. You downloaded this guide with either a VMware (.vmx) file or an Open Virtualization Format (.ovf) file. The .vmx version works with VMware Player or VMware Workstation on Linux and Windows based machines, and VMware Fusion for Mac. The .ovf file is suitable for Oracle's Virtualbox as well as several other virtualization players that support this format.

We've included instructions below for VMware Fusion, VMware Player, and Virtualbox.

If you run into issues getting the Learning VM set up, feel free to contact us at learningvm@puppetlabs.com, and we'll do our best to help out.

Getting started with the Learning VM

If you haven't already downloaded VMware Player, VMware Fusion, or Oracle Virtualbox, please see the links below:

- VMWare Player
- VMWare Fusion
- VirtualBox

You'll also need an SSH client to interact with the Learning VM over a Secure Shell (SSH) connection. This will be more comfortable than interacting with the virtualization software directly. If you're using Mac OS, you will be able to run SSH by way of the default Terminal application or a third party application like iTerm. If you are on a Windows OS, we recommend PuTTY, a free SSH client for Windows.

Once you have an up-to-date virtualization application and the means to SSH to the Learning VM you're ready to configure the Learning VM itself.

If you're reading this guide, you've probably already extracted the .zip file that contains the Learning VM. Keep that .zip file around in case you want to create a fresh instance of the Learning VM without having the re-do the download.

VM Setup

Start by launching your virtualization software. (Don't be tempted by any dialogues or wizards that pop up the first time you open the software. These will walk you through creating a *new* virtual machine, and will mislead you if you're trying to open the *existing* Learning VM file.)

Depending on what virtualization software you're using, there are some slight variations in how you'll open Learning VM file.

- In **VMware Player** there will be an *Open a Virtual Machine* option on the
- Welcome screen. You can also select *File* > *Open*... from the *Player* menu
- in the top left.
- For **VMWare Fusion**, select *File* > *Open*... from the menu bar.
- For **VirtualBox**, select *File* > *Import Appliance*... from the menu bar.
- If you're using different virtualization software, just be sure to *open* or
- *import*, rather than *create new*.

Don't launch the VM just yet. There are a few configuration steps that you should complete before launching the Learning VM for the first time. (If you skipped ahead and already launched the VM, shut it down by logging in with the credentials `root` and `puppet` and entering the command `shutdown -P now`. And if you run into errors, remember that you can simply delete the VM and create another by unpacking the .zip archive and following the instructions above.)

With the Learning VM selected in the library or manager window, open the **Settings** panel. There are a few things to adjust here.

First, under **Network** or **Network Adapter**, confirm that the **Network Adapter** is enabled, and configure it to use **Bridged** networking.

Next, you'll need to increase the memory allocation and processors to the VM to ensure that it has the resources neccessary to run smoothly. These options are under **System** in VirtualBox and **Processors & Memory** in VMware Fusion. Allocate 4 GB of memory (4096 MB) and two processor cores. You can run the Learning VM with less memory and fewer processor cores, but you may encounter performance issues.

Now that your settings are configured, select **Start** or **Power On** to boot up the VM.

> ### Input Capture
>
> Virtualization software uses mouse and keyboard capture to 'own' these devices and communicate input to the guest operating system. The keystroke to release the mouse and keyboard will be displayed at the top right of the VM window.

Next Steps

Once the VM is booted, you may have to hit `enter` to see to the login prompt. Log in using the following credentials:

- username: **root**
- password: **puppet**

All you'll want to do for now is get the Learning VM's IP address. Use the Facter tool bundled with Puppet Enterprise tool to find it.

```
facter ipaddress
```

Make a note of the IP address displayed. You'll need it to open an SSH connection to the Learning VM and in order to access to the PE Console later.

For the Learning VM's quest tool to work smoothly, you'll need to log out before starting your SSH session. The file that tracks your command line history and helps us test completion of some tasks will only be created after you log out for the first time. Enter the command:

```
exit
```

Now that you have the IP address, open an SSH connection to the Learning VM.

On a Linux system or a Mac, you can open a Terminal application and run the following command, replacing `<ip-address>` with the IP address of your Learning VM:

```
ssh root@<ip-address>
```

If you are on a Windows system, use an SSH client. We recommend Putty. Enter the IP address into the *Hostname* textbox and click *Open* to start your session.

Use the same credentials:

- username: **root**

- password: **puppet**

Now that the Learning VM is configured and you're connected, you're all set to take on your first quest! We hope you have fun learning Puppet!

In addition to the VM, the following resources may be handy in your journey to learn Puppet:

- Puppet users group
- Puppet Ask - Q&A site
- #puppet IRC channel on irc.freenode.net
- Learning VM Issue Tracker
- You can also email us at learningvm@puppetlabs.com

Welcome

Quest Objectives

- Learn about the value of Puppet and Puppet Enterprise
- Familiarize yourself with the Quest structure and tool

The Learning VM

> *Any sufficiently advanced technology is indistinguishable from magic.*
> *-Arthur C. Clarke*

Welcome to the Quest Guide for the Learning Virtual Machine. This guide will be your companion as you make your way through a series of interactive quests on the accompanying VM. This first quest serves as an introduction to Puppet and gives you an overview of the quest structure and the integrated quest tool. We've done our best to keep it short so you can get on to the meatier stuff in the quests that follow.

You should have started up the VM by now, and have an open SSH session from your terminal or SSH client.

If you need to, return to the Setup section and review the instructions to get caught up. Remember, the credentials to log in to the Learning VM via SSH are:

- username: **root**
- password: **puppet**

If you're comfortable in a Unix command-line environment, feel free to take a look around and get a feel for what you're working with.

Getting Started

The Learning VM includes a quest tool that will provide structure and feedback as you progress. You'll learn more about this tool below, but for now, type the following command to start your first quest: the "Welcome" quest.

```
quest --start welcome
```

What is Puppet?

Puppet is an open-source IT automation tool. The Puppet Domain Specific Language (DSL) is a Ruby-based coding language that provides a precise and adaptable way to describe a desired state for each machine in your infrastructure. Once you've described a desired state, Puppet does the work to bring your systems in line and keep them there.

The easy-to-read syntax of Puppet's DSL gives you an operating-system-independent language to specify which packages should be installed, what services you want running, which users accounts you need, how permissions are set, and just about any other detail of a system you might want to manage. If you're the DIY type or have unique needs, you can write the Puppet code to do all these things from scratch. But if you'd rather not re-invent the wheel, a wide variety of pre-made Puppet modules can help you get the setup you're looking for without pounding out the code yourself.

Why not just run a few shell commands or write a script? If you're comfortable with shell scripting and concerned with a few changes on a few machines, this may indeed be simpler. The appeal of Puppet is that allows you to describe all the details of a configuration in a way that abstracts away from operating system specifics, then manage those configurations on as many machines as you like. It lets you control your whole infrastructure (think hundreds or thousands of nodes) in a way that is simpler to maintain, understand, and audit than a collection of complicated scripts.

Puppet Enterprise (PE) is a complete configuration management platform, with an optimized set of components proven to work well together. It combines a version of open source Puppet (including a preconfigured production-grade Puppet master stack), with MCollective, PuppetDB, Hiera, and more than 40 other open source projects that Puppet Labs has integrated, certified, performance-tuned, and security-hardened to make a complete solution for automating mission-critical enterprise infrastructure.

In addition to these integrated open source projects, PE has many of its own features, including a graphical web interface for analyzing reports and controlling your infrastructure, orchestration features to keep your applications running smoothly as you coordinate updates and maintenance, event inspection, role-based access control, certification management, and cloud provisioning tools.

Task 1:

Now that you know what Puppet and Puppet Enterprise are, check and see what versions of are running on this Learning VM. Type the following command:

```
puppet -V   # That's a capital 'V'
```

You will see something like the following:

3.7.3 (Puppet Enterprise 3.7.1)

This indicates that Puppet Version 3.7.3 Puppet Enterprise 3.7.1 are installed.

What is a Quest?

At this point we've introduced you to the Learning VM and Puppet. You'll get your hands on Puppet soon enough. But first, what's a quest? This guide contains collection structured tutorials that we call *quests*. Each *quest* includes interactive *tasks* that give you a chance to try things out for yourself as you learn them.

If you executed the `puppet -V` command earlier, you've already completed your first task. (If not, go ahead and do so now.)

The Quest Tool

The Learning VM includes a quest tool that will help you keep track of which quests and tasks you've completed successfully and which are still pending. We've written a couple of tasks in this quest to demonstrate the features of the quest tool itself.

Task 2 :

To explore the command options for the quest tool, type the following command:

```
quest --help
```

The `quest --help` command provides you with a list of all the options for the `quest` command. You can invoke the quest command with each of those options, such as:

```
quest --progress     # Displays details of tasks completed
quest --completed    # Displays completed quests
quest --list         # Shows all available quests
quest --start <name> # Provide the name of a quest to start tracking
progress
```

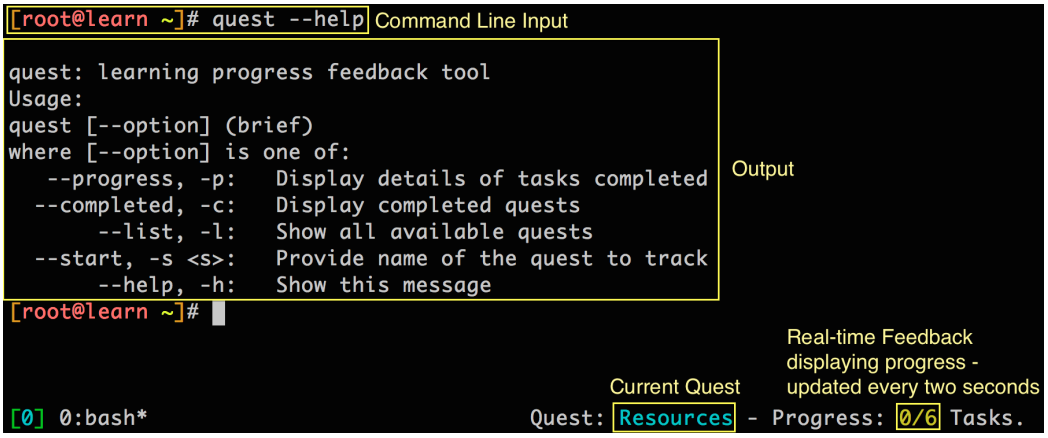
The VM comes with several adjustments to enable the use of the quest tool and progress tracking, including changes to how bash is configured. Please don't replace the .bashrc file. If you would like to make changes, append them to the existing file.

Task 3 :

Find out how much progress you have made so far:

```
quest --progress
```

While you can use the quest commands to find more detailed information about your progress through the quests, you can check the quest status display at the bottom right of your terminal window to keep up with your progress in real time.



Typing `clear` into your terminal will remove everything on your terminal screen.

Structure of this Quest Guide

We've organized the quests for the Learning VM around the belief that the more quickly you're exposed to real Puppet code in realistic conditions, the more quickly you'll learn. As far as is reasonable, we've tried to construct quests around plausible use cases, and to this end, most quests will introduce several related concepts at once.

The first several quests, up to and including the Modules quest, are your foundations. Learning about these things is like tying your shoe laces: no matter where you're trying to get to, you're going to get tripped up if you don't have a solid understanding of things like *resources*, *classes*, *manifests*, and *modules*.

We want to show that once you've taken care of these basics, though, there's quite a lot you can do with Puppet using modules from the Puppet Forge. After the foundations section, we've included some quests that will walk you through downloading, configuring, and deploying existing Puppet modules.

Finally, we introduce you to the Puppet language constructs you'll need to get started writing and deploying your own modules: things like *variables*, *conditionals*, *class parameters*, and *resource ordering*. With these concepts under your belt, you'll be in a much better position not just to create your own Puppet code, but to understand what's going on under the hood of modules you want to deploy.

Review

In this introductory quest we gave a brief overview of what Puppet is and the advantages of using Puppet to define and maintain the state of your infrastructure.

We also introduced the concept of the quest and interactive task. You tried out the quest tool and reviewed the mechanics completing quests and tasks.

Now that you know what Puppet and Puppet Enterprise are, and how to use the quest tool, you're ready to move on to the next quest: The Power of Puppet.

The Power of Puppet

Prerequisites

- Welcome Quest

Quest objectives

- Using existing Puppet modules, configure the Learning VM to serve a web version of the Quest Guide.
- Learn how the Puppet Enterprise (PE) console's node classifier can manage the Learning VM's configuration.

Getting started

In this quest you will use the Puppet Enterprise (PE) console in conjunction with existing modules to cut away much of the complexity of a common configuration task. You'll configure the Learning VM to serve the content of this Quest Guide as a locally accessible static HTML website. We'll show you how you can use Puppet and freely available Puppet modules to fully automate the process instead of writing code or using standard terminal commands.

As you go through this quest, remember that while Puppet can simplify many tasks, it's a powerful and complex tool. We will explain concepts as needed to complete and understand each task in this quest, but sometimes we'll hold off on a fuller explanation of some detail until a later quest. Don't worry if you don't feel like you're getting the whole story right away; keep at it and we'll get there when the time is right!

When you're ready to get started, type the following command:

```
quest --start power
```

Forging ahead

A **module** is a bundle of Puppet code packaged along with the other files and data you need manage some aspect of a system. Need to set up NTP? There's a module for that. Manage system users? That too. But likely you'll want to do both of these things and more. Modules let you mix and match reusable bits of Puppet code to make achieving

your desired configuration as painless as possible. Modules are designed to be, well, *modular*.

But where do these modules come from? The Puppet Forge is a public repository of modules contributed by members of the Puppet community, including many written and maintained by Puppet Labs employees and partners. The Forge also includes a list of PE Supported Modules, which Puppet Labs has rigorously tested and is committed to supporting and maintaining through their lifecycle.

Task 1:

To get started setting up the Quest Guide website, you'll need to download and install Puppet Labs' `apache` module from the Forge. (If you're offline or behind a firewall, check the aside below for instructions on using the cached version of the module.)

The `apache` module gives you everything you need to automate installing, configuring, and starting an Apache webserver. In your terminal, enter the following command to install the module:

```
puppet module install puppetlabs-apache
```

Offline?

If you don't have internet access, run the following terminal commands to used a cached version of the module:

```
puppet module install /usr/src/forge/puppetlabs-apache-*.tar.gz
--ignore-dependencies
```

This command tells Puppet to download the Puppet Labs `apache` module from the Forge and place it in the directory specified as Puppet's *modulepath*. The modulepath defines the directory on your puppet master where Puppet saves modules you install and accesses modules you already have installed. For Puppet Enterprise, this defaults to `/etc/puppetlabs/puppet/environments/production/modules`.

To help set up the Quest Guide website, we've also prepared an `lvmguide` module. It's already in the VM's module path, so there's no need to fetch it from the Forge. This small `lvmguide` module draws on some resources from the `apache` module and uses some code and content of its own to finish the configuration of the Quest Guide website.

The lvmguide and apache modules

Before using these modules, you should know a little more about how they work.

The `lvmguide` *module* includes Puppet code that defines an `lvmguide` *class*. In Puppet, a *class* is simply a named block of Puppet code organized in a way that defines a set of associated system resources. A class might install a package, customize an associated configuration file for that package, and start a service provided by that package. These are related and interdependent processes, so it makes sense to organize them into a single configurable unit: a class.

While a module can include many classes, it will often have a main class that shares the name of the module. This class serves as the access point for the module's functionality and calls on other classes within the module or from pre-requisite modules as needed.

Put your modules to use

In order to configure the Learning VM to serve you the Quest Guide website, you'll need to *classify* it with the `lvmguide` class. Classification tells Puppet which classes to apply to which machines in your infrastructure. Though there are a few different ways to classify nodes, we'll be using the PE console's node classifier for this quest.

Task 2 :

To access the PE console you'll need the Learning VM's IP address. Remember, you can use the `facter` tool packaged with PE.

```
facter ipaddress
```

Open a web browser on your host machine and go to `https://<IPADDRESS>`, where `<IPADDRESS>` is the Learning VM's IP address. (Be sure to include the `s` in `https`)

Your browser may give you a security notice because the PE console certificate is self-signed. Go ahead and click through this notice to continue to the console.

When prompted, use the following credentials to log in:

- username: **admin**
- password: **learningpuppet**

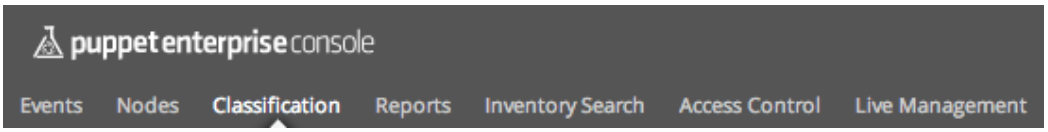
You can see a list of all the system facts accessible through facter by running the `facter` command.

Create a node group

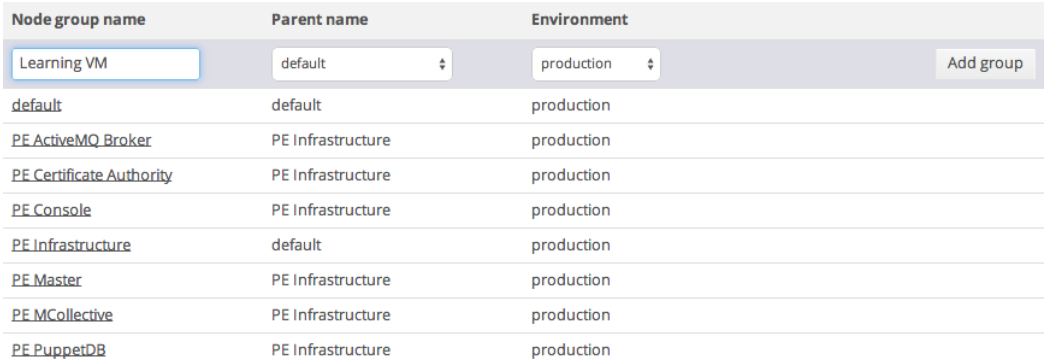
Now that you have access to the PE console, we'll walk you through the steps needed to classify the "learning.puppetlabs.vm" node (i.e. the Learning VM) with the `lvmguide` class.

First, you'll create a **Learning VM** node group. Node groups allow you to segment all the nodes in your infrastructure into separately configurable groups based on information collected by the `facter` tool.

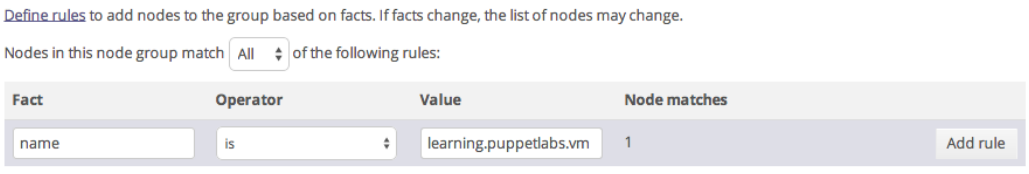
Click on *Classification* in the console navigation bar. It may take a moment to load.



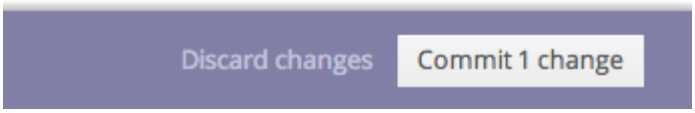
From here, enter "Learning VM" as a new node group name, and click *Add group* to create your new node group.



Click on the new group to set the rules for this group. You only want the Learning VM in this group, so create a rule that will match on the name `learning.puppetlabs.vm`.



Click *Add rule*, then click the *Commit 1 change* button in the bottom right of the console interface to commit your change.



Add a class

Now that the `lvmguide` class is available, you can use it to classify the node `learning.puppetlabs.vm`. Under the *Classes* tab, enter `lvmguide` in the text box, then click the *Add class* and *Commit 1 change* buttons to confirm your changes.

Run puppet

Now that you have classified the `learning.puppetlabs.vm` node with the `lvmguide` class, Puppet knows how the system should be configured, but it won't make any changes until a Puppet run occurs.

The puppet agent daemon runs in the background on any nodes you manage with Puppet. Every 30 minutes, the puppet agent daemon requests a *catalog* from the puppet master. The puppet master parses all the classes applied to that node and builds the catalog to describes how the node is supposed to be configured. It returns this catalog to the node's puppet agent, which then applies any changes necessary to bring the node into the line with the state described by the catalog.

Task 3 :

Instead of waiting for the puppet agent to make its scheduled run, use the `puppet agent` tool to trigger one yourself. In the terminal, type the following command:

puppet agent --test

This may take a minute to run. This is about the time it takes for the software packages to be downloaded and installed as needed. After a brief delay, you will see text scroll by in your terminal indicating that Puppet has made all the specified changes to the Learning VM.

Check out the Quest Guide website! In your browser's address bar, type the following URL: `http://<IPADDRESS>`. (Though the IP address is the same, using `https` will load the PE console, while `http` will load the Quest Guide as a website.)

From this point on you can either follow along with the website or with the PDF, whichever works best for you.

IP troubleshooting

The website for the quest guide will remain accessible for as long as the VM's IP address remains the same. If you move your computer or laptop to a different network, or if you suspend your laptop and resumed work on the Learning VM later, the website may not be accessible.

In case any of the above issues happen, and you end up with a stale IP address, run the following commands on the Learning VM to get a new IP address. (Remember, if you're ever unable to establish an SSH session, you can log in directly through the interface of your virtualization software.)

Refresh your DHCP lease:

```
service network restart
```

Find your IP address:

```
facter ipaddress
```

Explore the lvmguide class

To understand how the `lvmguide` class works, you can take a look under the hood. In your terminal, use the `cd` command to navigate to the module directory. (Remember, `cd` for 'change directory.')

```
cd /etc/puppetlabs/puppet/environments/production/modules
```

Next, open the `init.pp` manifest.

```
vim lvmguide/manifests/init.pp
```

```
class lvmguide (
  $document_root = '/var/www/html/lvmguide',
  $port          = '80',
) {

  # Manage apache, the files for the website will be
  # managed by the quest tool
  class { 'apache':
    default_vhost => false,
  }
  apache::vhost { 'learning.puppetlabs.vm':
    port    => $port,
    docroot => $document_root,
  }
}
```

(To exit out of the file without saving any changes, make sure you're in `command` mode in vim by hitting the `esc` key, and enter the command `:q!`.)

Don't worry about understanding each detail of the syntax just yet. For now, we'll just give you a quick overview so the concepts won't be totally new when you encounter them again later on.

Class title and parameters:

```
class lvmguide (
  $document_root = '/var/www/html/lvmguide',
  $port = '80',
) {
```

The class `lvmguide` takes two parameters, as defined in the parentheses following the class name. Parameters allow variables within a class to be set as the class is declared. Because you didn't specify parameter values, the two variables `$document_root` and `$port` were set to their defaults, `/var/www/html/lvmguide` and `80`.

Include the apache module's apache class:

```
class { 'apache':
    default_vhost => false,
  }
```

The `lvmguide` class declares another class: `apache`. Puppet knows about the `apache` class because it is defined by the `apache` module you installed earlier. The `default_vhost` parameter for the `apache` class is set to `false`. This is all the equivalent of saying "Set up Apache, and don't use the default VirtualHost because I want to specify my own."

Include the apache module's vhost class:

```
apache::vhost { 'learning.puppetlabs.vm':
    port    => $port,
    docroot => $document_root,
  }
```

This block of code declares the `apache::vhost` class for the Quest Guide with the title `learning.puppetlabs.vm`, and with `$port` and `$docroot` set to those class parameters we saw earlier. This is the same as saying "Please set up a VirtualHost website serving the 'learning.puppetlabs.vm' website, and set the port and document root based on the parameters from above."

The files for the website

The files for the quest guide are put in place by the `quest` command line tool, and thus we don't specify anything about the files in the class. Puppet is flexible enough to help you manage just what you want to, leaving you free to use other tools where more appropriate. Thus we put together a solution using Puppet to manage a portion of it, and our `quest` tool to manage the rest.

It may seem like there's a lot going on here, but by the time you get through this quest guide, a quick read-through will be enough to get the gist of well-written Puppet code. One advantage of a declarative language like Puppet is that the code tends to be much more self-documenting than code written in an imperative language.

Repeatable, portable, testable

It's cool to install and configure an Apache httpd web server with a few lines of code and some clicks in the console, but keep in mind that the best part can't be shown with the Learning VM: once the `lvmguide` module is installed, you can apply the `lvmguide` class to as many nodes as you like, even if they have different specifications or run different operating systems.

And once a class is deployed to your infrastructure, Puppet gives you the ability to manage the configuration from a single central point. You can implement your updates and changes in a test environment, then easily move them into production.

Updated content

Before continuing on to the remaining quests, let's ensure that you have the most up to date version of the quest-related content. Now that we have the website configured, please run the following command:

```
quest update
```

This will download an updated PDF, files for the quest guide website, as well as the tests for the quests.

You can find a copy of the update Quest Guide PDF at: `http://<IPADDRESS>/Quest_Guide.pdf`, or in the `/var/www/html/lvmguide/` directory on the VM.

Review

Great job on completing the quest! You should now have a good idea of how to download existing modules from the Forge and use the PE console node classifier to apply them to a node. You also learned how to use the `puppet agent --test` command to manually trigger a puppet run.

Though we'll go over many of the details of the Puppet DSL in later quests, you had your first look at a Puppet class, and some of the elements that make it up.

Resources

Prerequisites

- Welcome Quest
- Power of Puppet Quest

Quest Objectives

- Understand how resources on the system are modeled in Puppet's Domain Specific Language (DSL).
- Use Puppet to inspect resources on your system.
- Use the Puppet Apply tool to make changes to resources on your system.
- Learn about the Resource Abstraction Layer (RAL).

Getting Started

In this quest, you will be introduced to **resources**, the fundamental building blocks of Puppet's declarative modeling syntax. You will learn how to inspect and modify resources on the Learning VM using Puppet command-line tools. A thorough understanding of how the Puppet resource syntax reflects the state of a system will be an important foundation as you continue to learn the more complex aspects of Puppet and its DSL.

When you're ready to get started, type the following command:

```
quest --start resources
```

Resources

> *For me, abstraction is real, probably more real than nature. I'll go further and say that abstraction is nearer my heart. I prefer to see with closed eyes.*
> *-Joseph Albers*

Resources are the fundamental units for modeling system configurations. Each resource describes some aspect of a system and its state, like a service that should be running or a package you want installed. The block of code that describes a resource is called a

resource declaration. These resource declarations are written in Puppet code, a Domain Specific Language (DSL) built on Ruby.

Puppet's Domain Specific Language

Puppet's DSL is a *declarative* language rather than an *imperative* one. This means that instead of defining a process or set of commands, Puppet code describes (or declares) only the desired end state, and relies of built-in *providers* to deal with implementation.

When Luke Kanies was initially designing Puppet, he experimented with several languages before settling on Ruby as the best match for his vision of a transparent and readable way to model system states. While the Puppet DSL has inherited many of these appealing aspects of Ruby, you're better off thinking of it as a distinct language. While a bit of Ruby knowledge certainly won't hurt in your quest to master Puppet, you don't need to know any Ruby to use Puppet, and you may even end up in trouble if you blindly assume that things will carry over.

One of the points where there is a nice carry over from Ruby is the *hash* syntax. It provides a clean way to format this kind of declarative model, and is the basis for the *resource declarations* you'll be learning about in this quest.

A nice feature of Puppet's declarative model is that it goes both ways; that is, you can inspect the current state of any existing resource in the same syntax you would use to declare a desired state.

Task 1:

Using the *puppet resource* tool, take a look at your root user account. Note the pattern of the command will be: *puppet resource <type> <name>*.

```
puppet resource user root
```

You'll see something like the following.

```
user { 'root':
  ensure           => 'present',
  comment          => 'root',
  gid              => '0',
  home             => '/root',
  password         => '$1$jrm5tnjw$h8JJ9mCZLmJvIxvDLjw1M/',
  password_max_age => '99999',
  password_min_age => '0',
  shell            => '/bin/bash',
  uid              => '0',
}
```

It's a little abstract, but a nice portrait, don't you think?

Resource Type

To be sure that you have a solid understanding of how resources are represented, we'll go through this example point by point.

Take a look at your first line in the above resource declaration.

```
user { 'root':
```

The word `user`, right *before* the curly brace, is the **resource type**.

Puppet includes a variety of built-in resource types, which allow you to manage various aspects of a system. Below are some of the core resource types you'll likely encounter most often:

- `user` A user
- `group` A user group
- `file` A specific file
- `package` A software package
- `service` A running service
- `cron` A scheduled cron job
- `exec` An external command
- `host` A host entry

If you are curious to learn about all of the different built-in resources types available for you to manage, see the Type Reference Document

Resource Title

Take another look at the first line of the resource declaration.

```
user { 'root':
```

The single quoted word 'root' just before the colon is the resource **title**. Puppet uses a resource's title as a unique identifer for that resource. This means that no two resources of the same type can share a title. In the case of the user resource, the title is also the name of the user account being managed.

Attribute Value Pairs

Now that we've covered the *type* and *title*, take a look at the body of the resource declaration.

```
user { 'root':
  ensure           => 'present',
  comment          => 'root',
  gid              => '0',
  home             => '/root',
  password         => '$1$jrm5tnjw$h8JJ9mCZLmJvIxvDLjw1M/',
  password_max_age => '99999',
  password_min_age => '0',
  shell            => '/bin/bash',
  uid              => '0',
}
```

After the colon in that first line comes a hash of **attributes** and their corresponding **values**. Each line consists of an attribute name, a `=>` (pronounced 'hash rocket'), a value, and a final comma. For instance, the attribute value pair `home => '/root',` indicates that your home is set to the directory `/root`.

So to bring this all together, a resource declaration will match the following pattern:

```
type {'title':
    attribute => 'value',
}
```

> ### The Trailing Comma
>
> Though the comma at the end of the final attribute value pair isn't strictly necessary, it is best practice to include it for the sake of consistency. Leave it out, and you'll inevitably forget to insert it when you add another attribute value pair on the following line!

So in the world of Puppet, you and everything around you can be respresented as a resource, and resources follow this tidy declarative syntax. As pretty as they are, presumably you don't want to just look at resources all day, you want to change them!

You can, and easily. But before making any changes, take a moment to learn a bit more about the user type. You'll want a way of knowing *what* you're changing before you start changing attributes.

Task 2 :

Use the *puppet describe* tool to get a description of the *user* type, including a list of its parameters.

```
puppet describe user | less
```

(You can use the `jk` key mapping or the arrow keys to scroll, and `q` to exit less.)

No need to read all the way through, but take a minute to skim the *describe* page for the *user* type. Notice the documentation for some of the attributes you saw for the *root* user.

Puppet Apply

You can use the Puppet resource declaration syntax with the *puppet apply* tool to make quick changes to resources on the system. (Note, though, that while *puppet apply* is great for tests and exploration, it's limited to this kind of one-off change. We'll get to the more robust ways to manage resources in later quests.)

Task 3 :

You can use the *puppet apply* tool with the *-e* (*--execute*) flag to execute a bit of Puppet code. In this example, you'll create a new user called *galatea*. Puppet uses some defaults for unspecified user attributes, so all you'll need to do to create a new user is set the 'ensure' attribute to 'present'. This 'present' value tells Puppet to check if the resource exists on the system, and to create the specified resource if it does not.

```
puppet apply -e "user { 'galatea': ensure => 'present', }"
```

Use the `puppet resource` tool to take a look at user *galatea*. Type the following command:

```
puppet resource user galatea
```

Notice that while the *root* user had a *comment* attribute, Puppet hasn't created one for your new user. As you may have noticed looking over the *puppet describe* entry for the user type, this *comment* is generally the full name of the account's owner.

Task 4 :

While puppet apply with the `-e` flag can be handy for quick one-liners, you can pass an `--execute` (incidentally, also shortened to `-e`) flag to the `puppet resource` tool to edit and apply changes to a resource.

```
puppet resource -e user galatea
```

You'll see the same output for this resource as before, but this time it will be opened in a text editor (vim, by default). To add a *comment* attribute, insert a new line to the resource's list of attribute value pairs. (If you're not used to Vim, note that you must use the `i` command to enter insert mode before you can insert text.)

```
comment => 'Galatea of Cyprus',
```

Save and exit (`esc` to return to command mode, and `:wq` in vim), and the resource declaration will be applied with the added comment. If you like, use the `puppet resource` tool again to inspect the result.

Quest Progress

Have you noticed that when you successfully finish a task, the 'completed tasks' in the lower right corner of your terminal increases? Remember, you can also check your progress by entering the following command:

```
quest --progress
```

The Resource Abstraction Layer

As we mentioned at the beginning of this quest, Puppet takes the descriptions expressed by resource declarations and uses *providers* specific to the operating system to realize them. These providers abstract away the complexity of managing diverse implementations of resource types on different systems. As a whole, we call this system of resource types and providers the **Resource Abstraction Layer** or **RAL**.

In the case of users, Puppet can use providers to manage users with LDAP, Windows ADSI, AIX, and several other providers depending on a node's system. Similarly, when you wish to install a package, you can stand back and watch Puppet figure out whether to use 'yum', 'apt', 'rpm', or one of several other providers for package management. This lets you set aside the implementation-related details of managing the resources, such as the names of commands (is it `adduser` or `useradd`?), arguments for the commands, and file formats, and lets you focus on the end result.

Review

So let's rehash what you learned in this quest. First, we covered two very important Puppet topics: the Resource Abstraction Layer and the anatomy of a resource. To dive deeper into these topics, we showed you how to use the `puppet describe` and `puppet resource` tools, which also leads to a better understanding of Puppet's Language. We also showed you how you can actually change the state of the system by declaring resources with the `puppet apply` and `puppet resource` tools. These tools will be useful as you progress through the following quests.

Manifests and Classes

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest

Quest Objectives

- Understand the concept of a Puppet manifest
- Construct and apply manifests to manage resources
- Understand what a *class* means in Puppet's Language
- Learn how to use a class definition
- Understand the difference between defining and declaring a class

Getting Started

In the Resources quest you learned about resources and the syntax used to declare them in the Puppet DSL. You used the `puppet resource`, `puppet describe`, and `puppet apply` tools to inspect, learn about, and change resources on the system. In this quest, we're going to cover two key Puppet concepts that will help you organize and implement your resource declarations: *classes* and *manifests*. Proper use of classes and manifests is the first step towards writing testable and reusable Puppet code.

When you're ready to get started, enter the following command to begin:

```
quest --start manifests_classes
```

Manifests

> *Imagination is a force that can actually manifest a reality.*
> *--James Cameron*

A manifest is a text file that contains Puppet code and is appended by the `.pp` extension. It's the same stuff you saw using the `puppet resource` tool and applied with the `puppet apply` tool, just saved as a file.

While it's nice to be able to edit and save your Puppet code as a file, manifests also give you a way to keep your code organized in a way that Puppet itself can understand. In theory, you could put whatever bits and pieces of syntactically valid Puppet code you like into a manifest. However, for the broader Puppet architecture to work effectively, you'll need to follow some patterns in how you write your manifests and where you put them. A key aspect of proper manifest management is related to Puppet *classes*.

Classes

In Puppet's DSL a **class** is a named block of Puppet code. A class will generally manage a set of resources related to a single function or system component. Classes often contain other classes; this nesting provides a structured way to bring together functions of different classes as components of a larger solution.

Using a Puppet class requires two steps. First, you'll need to *define* it by writing a class definition and saving it in a manifest file. When Puppet runs, it will parse this manifest and store your class definition. The class can then be *declared* to apply it to nodes in your infrastructure.

There are a few different ways to tell Puppet where and how to apply classes to nodes. You already saw the PE Console's node classifier in the Power of Puppet quest, and we'll discuss other methods of node classification in a later quest. For now, though, we'll show you how to write class definitions and use *test* manifests to declare these classes locally.

One more note on the topic of classes. In Puppet, classes are *singleton*, which means that a class can only be declared *once* on a given node. In this sense, Puppet's classes are different than the kind of classes you may have encountered in Object Oriented programming, which are often instantiated multiple times.

Cowsayings

You had a taste of how Puppet can manage users in the Resources quest. In this quest we'll use the *package* resource as our example.

First, you'll use Puppet to manage the *cowsay* package. Cowsay lets you print a message in the speech bubble of an ascii art cow. It may not be a mission critical software (unless your mission involves lots of ascii cows!), but it works well as a simple example. You'll also install the *fortune* package, which will give you and your cow access to a database of sayings and quotations.

We've already created a `cowsayings` module directory in Puppet's *modulepath*, and included two subdirectories: `manifests` and `tests`. Before getting started writing manifests, change directories to save yourself some typing:

```
cd /etc/puppetlabs/puppet/environments/production/modules
```

Cowsay

Task 1:

You'll want to put the manifest with your cowsay class definition in the manifests directory. Use vim to create a `cowsay.pp` manifest:

```
vim cowsayings/manifests/cowsay.pp
```

Enter the following class definition, then save and exit (`:wq`):

```
class cowsayings::cowsay {
  package { 'cowsay':
    ensure => 'present',
  }
}
```

Now that you're working with manifests, you can use some validation tools to check your code before you apply it. Use the `puppet parser` tool to check the syntax of your new manifest:

```
puppet parser validate cowsayings/manifests/cowsay.pp
```

The parser will return nothing if there are no errors. If it does detect a syntax error, open the file again and fix the problem before continuing.

If you try to apply this manifest, nothing on the system will change. (Give it a shot if you like.) This is because you have *defined* a cowsay class, but haven't *declared* it anywhere. Whenever Puppet runs, it parses everything in the *modulepath*, including your cowsay class definition. So Puppet knows that the cowsay class contains a resource declaration for the cowsay package, but hasn't yet been told to do anything with it.

Task 2:

To actually declare the class, create a `cowsay.pp` test in the tests directory.

```
vim cowsayings/tests/cowsay.pp
```

In this manifest, *declare* the cowsay class with the `include` keyword.

```
include cowsayings::cowsay
```

Save and exit.

Before applying any changes to your system, it's always a good idea to use the `--noop` flag to do a 'dry run' of the Puppet agent. This will compile the catalog and notify you of the changes that Puppet would have made without actually applying any of those changes to your system.

```
puppet apply --noop cowsayings/tests/cowsay.pp
```

You should see an output like the following:

```
Notice: Compiled catalog for learn.localdomain in environment production in
0.62 seconds
Notice: /Stage[main]/Cowsayings::Cowsay/Package[cowsay]/ensure: current_value
absent, should be present (noop)
Notice: Class[Cowsayings::Cowsay]: Would have triggered 'refresh' from 1
events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 1.08 seconds
```

Task 3 :

If your dry run looks good, go ahead and run `puppet apply` again without the `--noop` flag. If everything went according to plan, the cowsay package is now installed on the Learning VM. Give it a try!

```
cowsay Puppet is awesome!
```

Your bovine friend clearly knows what's up.

```
 ____________________
< Puppet is awesome! >
 --------------------
       \   ^__^
        \  (oo)\_______
           (__)\       )\/\
               ||----w |
               ||     ||
```

Fortune

But this module isn't just about cowsay; it's about cow *sayings*. With the fortune package, you can provide your cow with a whole database of wisdom.

Task 4 :

Create a new manifest for your fortune class definition:

```
vim cowsayings/manifests/fortune.pp
```

Write your class definition here:

```
class cowsayings::fortune {
  package { 'fortune-mod':
    ensure => 'present',
  }
}
```

Task 5 :

Again, you'll want to validate your new manifests syntax with the `puppet parser validate` command. When everything checks out, you're ready to make your test manifest:

```
vim cowsayings/tests/fortune.pp
```

As before, use `include` to declare your `cowsayings::fortune` class.

Task 6 :

Apply the `cowsayings/tests/fortune.pp` manifest with the `--noop` flag. If everything looks good, apply again without the flag.

Now that you have both packages installed, you can use them together. Try piping the output of the `fortune` command to `cowsay`:

```
fortune | cowsay
```

So you've installed two packages that can work together to do something more interesting than either would do on its own. This is a bit of a silly example, of course, but it's not so different than, say, installing packages for both Apache and PHP on a webserver.

Main Class: init.pp

Often a module will gather several classes that work together into a single class to let you declare everything at once.

Before creating the main class for cowsayings, however, a note on **scope**. You may have noticed that the classes you wrote for cowsay and fortune were both prepended by `cowsayings::`. When you declare a class, this scope syntax tells Puppet where to find that class; in this case, it's in the cowsayings module.

For the main class of a module, however, things are a little different. The main class shares the name of the module itself. Instead of following the pattern of the manifest for the class it contains, however, Puppet recognizes the special file name `init.pp` as designating the manifest that will contain a module's main class.

Task 7 :

So to contain your main `cowsayings` class, create an `init.pp` manifest in the `cowsayings/manifests` directory:

```
vim cowsayings/manifests/init.pp
```

Here, you'll define the `cowsayings` class. Within it, use the same `include` syntax you used in your tests to declare the `cowsayings::cowsay` and `cowsayings::fortune` classes.

```
class cowsayings {
  include cowsayings::cowsay
  include cowsayings::fortune
}
```

Save the manifest, and check your syntax with the `puppet parser` tool.

Task 8 :

Next, create a test for the `init.pp` manifest in the tests directory.

```
vim cowsayings/tests/init.pp
```

Here, just declare the `cowsayings` class:

```
include cowsayings
```

At this point, you've already got both packages you want installed on the Learning VM. Applying the changes again wouldn't actually do anything. For the sake of demonstration, go ahead and use a `puppet apply -e` to delete them so you can test the functionality of your new `cowsayings` class:

```
puppet apply -e "package { 'fortune-mod': ensure => 'absent', } \
 package {'cowsay': ensure => 'absent', }"
```

Task 9 :

Good. Now that the packages are gone, do a `--noop` first, then apply your `cowsayings/tests/init.pp` test.

Review

We covered a lot in this quest. We promised manifests and classes, but you got a little taste of how Puppet modules work as well.

A *class* is a collection of related resources and other classes which, once defined, can be declared as a single unit. Puppet classes are also singleton, which means that unlike classes in object oriented programming, a Puppet class can only be declared a single time on a given node.

A *manifest* is a file containing Puppet code, and appended with the `.pp` extension. In this quest, we used manifests in the `./manifests` directory each to define a single class, and used a corresponding test manifest in the `./tests` directory to declare each of those classes.

There are also a few details about classes and manifests we haven't gotten to just yet. As we mentioned in the Power of Puppet quest, for example, classes can also be declared with *parameters* to customize their functionality. Don't worry, we'll get there soon enough!

Modules

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes

Quest Objectives

- Understand the purpose of Puppet modules
- Learn the module directory structure
- Write and test a simple module

Getting Started

If you want to get things done effeciently in Puppet, the **module** will be your best friend. You got a little taste of module structure in the Manifests and Classes quest. In this quest, we'll take you deeper into the details.

In short, a Puppet module is a self-contained bundle of all the Puppet code and other data needed to manage some aspect of your configuration. In this quest, we'll go over the purpose and structure of Puppet modules, before showing you how to create your own.

When you're ready, type the following command:

```
quest --start modules
```

Why Meddle with Modules?

> *Creation always builds upon something else. There is no art that doesn't reuse.*
> *-Lawrence Lessig*

There's no hard-and-fast technical reason why you can't toss all the resource declarations for a node into one massive class. But this wouldn't be very Puppetish.

Puppet's not just about bringing your nodes in line with a desired configuration state; it's about doing this in a way that's transparent, repeatable, and as painless as possible.

Modules allow you to organize your Puppet code into units that are testable, reusable, and portable, in short, *modular*. This means that instead of writing Puppet code from scratch for every configuration you need, you can mix and match solutions from a few well-written modules. And because these modules are separate and self-contained, they're much easier to test, maintain, and share than a collection of one-off solutions.

Though there are some technical aspects to how Puppet treats modules, at their root they're little more than a conventional directory structure and some naming standards. The module file structure gives Puppet a consistent way to locate whatever classes, files, templates, plugins, and binaries are required to fulfill the function of the module.

Modules and the module directory structure also provide an important way to manage scope within Puppet. Keeping everything nicely tucked away in its own module means you have to worry much less about name collisions and confusion.

Finally, because modules are standardized and self-contained, they're easy to share. Puppet Labs hosts a free service called the Forge where you can find a wide array of modules developed and maintained by others.

The modulepath

All modules accessible by your Puppet Master are located in the directories specified by the *modulepath* variable in Puppet's configuration file. On the Learning VM, this configuration file is `/etc/puppetlabs/puppet/puppet.conf`.

Task 1:

You can find the modulepath on any system with Puppet installed by running the `puppet agent` command with the `--configprint` flag and the `modulepath` argument:

```
puppet agent --configprint modulepath
```

This will tell you that Puppet looks in the directories `/etc/puppetlabs/puppet/environments/production/modules`, `/etc/puppetlabs/puppet/modules`, and then in `/opt/puppet/share/puppet/modules` to find available modules.

Throughout the quests in the Learning VM, you will work in the `/etc/puppetlabs/puppet/environments/production/modules` directory. This is where you keep modules for your production environment. (Site specific modules you need to be available for all environments are kept in `/etc/puppetlabs/puppet/modules`, and

modules required by Puppet Enterprise itself are kept in the `/opt/puppet/share/puppet/modules` directory.)

Module Structure

Now that you have an idea of why modules are useful and where they're kept, it's time to delve a little deeper into the anatomy of a module.

A module consists of a pre-defined structure of directories that help Puppet reliably locate the module's contents.

Use the `ls` command to see what modules are already installed:

```
ls /etc/puppetlabs/puppet/environments/production/modules
```

You'll probably recognize some familiar names from previous quests.

To get clear picture of the directory structure of the modules here, you can use a couple flags with the `tree` command to limit the output to directories, and limit the depth to two directories.

```
tree -L 2 -d /etc/puppetlabs/puppet/environments/production/modules/
```

You'll see a list of directories, like so:

```
/etc/puppetlabs/puppet/environments/production/modules/
└── apache
    ├── files
    ├── lib
    ├── manifests
    ├── spec
    ├── templates
    └── tests
    ...
```

Each of the standardized subdirectory names you see tells Puppet users and Puppet itself where to find each of the various components that come together to make a complete module.

Now that you have an idea of what a module is and what it looks like, you're ready to make your own.

You've already had a chance to play with the *user* and *package* resources in previous quests, so this time we'll focus on the *file* resource type. The *file* resource type is also a nice example for this quest because Puppet uses some URI abstraction based on the module structure to locate the sources for files.

The module you'll make in this quest will manage some settings for Vim, the text editor you've been using to write your Puppet code. Because the settings for services and applications are often set in a configuration file, the *file* resource type can be very handy for managing these settings.

Change your working directory to the modulepath if you're not already there.

```
cd /etc/puppetlabs/puppet/environments/production/modules
```

Task 2 :

The top directory will be the name you want for the module. In this case, let's call it "vimrc." Use the `mkdir` command to create your module directory:

```
mkdir vimrc
```

Task 3 :

Now you need three more directories, one for manifests, one for tests, and one for files.

```
mkdir vimrc/{manifests,tests,files}
```

If you use the `tree vimrc` command to take a look at your new module, you should now see a structure like this:

```
vimrc
├── files
├── manifests
└── tests

3 directories, 0 files
```

Task 4 :

We've already set up the Learning VM with some custom settings for Vim. Instead of starting from scratch, copy the existing `.vimrc` file into the `files` directory of your new module. Any file in the `files` directory of a module in the Puppet master's modulepath will be available to client nodes through Puppet's built-in fileserver.

```
cp ~/.vimrc vimrc/files/vimrc
```

Task 5 :

Once you've copied the file, open so you can make an addition.

```
vim vimrc/files/vimrc
```

We'll keep things simple. By default, line numbering is disabled. Add the following line to the end of the file to tell Vim to turn on line numbering.

```
set number
```

Save and exit.

Task 6 :

Now that your source file is ready, you need to write a manifest to tell puppet what to do with it.

Remember, the manifest that includes the main class for a module is always called `init.pp`. Create the `init.pp` manifest in your module's manifests directory.

```
vim vimrc/manifests/init.pp
```

The Puppet code you put in here will be pretty simple. You need to define a class `vimrc`, and within it, make a *file* resource declaration to tell Puppet to take the `vimrc/files/vimrc` file from your module and use Puppet's file server to push it out to the specified location.

In this case, the `.vimrc` file that defines your Vim settings lives in the `/root` directory. This is the file you want Puppet to manage, so its full path (i.e. `/root/.vimrc`) will be the *title* of the file resource you're declaring.

This resource declaration will then need two attribute value pairs.

First, as with the other resource types you've encountered, `ensure => 'present',` tells Puppet to ensure that the entity described by the resource exists on the system.

Second, the `source` attribute tells Puppet what the managed file should actually contain. The value for the source attribute should be the URI of the source file.

All Puppet file server URIs are structured as follows:

```
puppet://{server hostname (optional)}/{mount point}/{remainder of path}
```

However, there's some URI abstraction magic built in to Puppet that makes these URIs more concise.

First, the optional server hostname is nearly always omitted, as it defaults to the hostname of the Puppet master. Unless you need to specify a file server other than

the Puppet master, your file URIs should begin with a triple forward slash, like so: `puppet:///`.

Second, nearly all file serving in Puppet is done through modules. Puppet provides a couple of shortcuts to make accessing files in modules simpler. First, Puppet treats `modules` as a special mount point that will point to the Puppet master's modulepath. So the first part of the URI will generally look like `puppet:///modules/`

Finally, because all files to be served from a module must be kept in the module's `files` directory, this directory is implicit and is left out of the URI.

So while the full path to the vimrc source file is `/etc/puppetlabs/puppet/environments/production/modules/vimrc/files/vimrc`, Puppet's URI abstraction shortens it to `/modules/vimrc/vimrc`. Combined with the implicit hostname, then, the attribute value pair for the source URI is:

```
source => 'puppet:///modules/vimrc/vimrc',
```

Putting this all together, your init.pp manifest should contain the following:

```
class vimrc {
  file { '/root/.vimrc':
    ensure => 'present',
    source => 'puppet:///modules/vimrc/vimrc',
  }
}
```

Save the manifest, and use the `puppet parser` tool to validate your syntax:

```
puppet parser validate vimrc/manifests/init.pp
```

Remember, this manifest *defines* the `vimrc` class, but you'll need to *declare* it for it to have an effect. That is, we've described what the `vimrc` class is, but you haven't told Puppet to actually do anything with it.

Task 7 :

To test the `vimrc` class, create a manifest called `init.pp` in the `vimrc/tests` directory.

```
vim vimrc/tests/init.pp
```

All you'll do here is *declare* the `vimrc` class with the `include` directive.

```
include vimrc
```

Task 8 :

Apply the new manifest with the `--noop` flag. If everything looks good, drop the `--noop` and apply it for real.

You'll see something like the following:

```
Notice: /Stage[main]/Vimrc/File[/root/.vimrc]/content: content changed
'{md5}99430edcb284f9e83f4de1faa7ab85c8' to
'{md5}f685bf9bc0c197f148f06704373dfbe5'
```

When you tell Puppet to manage a file, it compares the md5 hash of the target file against that of the specified source file to check if any changes need to be made. Because the hashes did not match, Puppet knew that the target file did not match the desired state, and changed it to match the source file you had specified.

To see that your line numbering settings have been applied, open a file with Vim. You should see the number of each line listed to the left.

Review

In this quest, you learned about the structure and purpose of Puppet modules. You created a module directory structure, and wrote the class you need to manage a configuration file for Vim. You also saw how Puppet uses md5 hashes to determine whether a target file matches the specified source file.

In the quests that follow, you'll learn more about installing and deploying pre-made modules from the Puppet Forge.

NTP

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes
- Modules

Quest Objectives

- Use the `puppet module` tool to find and install modules on the Puppet Forge
- Learn how you can use the `site.pp` manifest to classify nodes.
- Use class parameters to adjust variables in a class as you declare it.

Getting Started

In the previous Modules Quest we primarily learned about the structure of a module and how to create a module. In this quest, you'll learn how you can use an existing module from the Puppet Forge to manage an important service on your machine: NTP.

```
quest --start ntp
```

What's NTP

> *Time is the substance from which I am made. Time is a river which carries me along, but I am the river; it is a tiger that devours me, but I am the tiger; it is a fire that consumes me, but I am the fire.*
> *-Jorge Luis Borges*

Security services, shared filesystems, certificate signing, logging systems, and many other fundamental services and applications (including Puppet itself!) need accurate and coordinated time to function reliably. Given variable network latency, it takes some clever algorithms and protocols to get this coordination right.

The Network Time Protocol (NTP) lets you keep time millisecond-accurate within your network while staying synchronized to Coordinated Universal Time (UTC) by way of

publicly accessible timeservers. (If you're interested in the subtleties of how NTP works, you can read all about it here)

NTP is one of the most fundamental services you will want to include in your infrastructure. Puppet Labs maintains a supported module that makes the configuration and management of NTP simple.

Package/File/Service

We'll show you how to install and deploy the NTP module in a moment, but first, take a look at the current state of your system. This way, you'll be able to keep track of what Puppet changes and understand why the NTP module does what it does.

To get the NTP service running, there are three key resources that Puppet will manage. The puppet resource tool can show you the current state of each of these resources.

First, check the state of the NTP *package*:

```
puppet resource package ntp
```

check the NTP configuration *file*:

```
puppet resource file /etc/ntp.conf
```

finally, see if the Network Time Protocol Daemon (NTPD) *service* is running:

```
puppet resource service ntpd
```

You'll see that the NTP package is installed on the Learning VM, that the configuration file exists, but that the ntpd service is 'stopped'.

As you continue to work with Puppet, you'll find that this *package/file/service* pattern is very common. These three resource types correspond to the common sequence of installing a package, customizing that package's functionality with configuration file, and starting the service provided by that package.

The *package/file/service* pattern also describes the typical relationships of dependency among these three resources: a well-written class will define these relationships, telling Puppet to restart the service if the configuration file has been modified, and re-create the configuration file when the package is installed or updated. You'll be working with an existing module in this quest, so these dependencies are already taken care of; we'll get into the specifics of how they can be managed in a later quest.

But now, on to the installation!

Installation

Before you classify the Learning VM with the NTP class, you'll need to install the NTP module from the forge. While the module itself is called `ntp`, modules in the forge are prefixed by the account name of the uploader. So to get the Puppet Labs NTP module, you'll specify `puppetlabs-ntp`. When you look at the module saved to the modulepath on your Puppet master, however, it will be named `ntp`. Keep this in mind, as trying to install multiple modules of the same name can lead to conflicts!

Task 1:

Use the puppet module tool to install the Puppet Labs `ntp` module.

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

This command tells the puppet module tool to fetch the module from the Puppet Forge and place it in Puppet's modulepath: `/etc/puppetlabs/puppet/environments/production/modules`.

Classification with the site.pp Manifest

Now that the NTP module is installed, all the included classes are available to use in node classification.

In the Power of Puppet quest, you learned how to classify a node with the PE Console. In this quest, we introduce another method of node classification: the `site.pp` manifest.

`site.pp` is the first manifest the Puppet agent checks when it connects to the master. It defines global settings and resource defaults that will apply to all nodes in your infrastructure. It is also where you will put your *node definitions* (sometimes called `node statements`). A node definition is a block of Puppet code that specifies a set one or more nodes and declares the classes that Puppet will enforce on that set.

In a sense, this node definition is a bit like the test manifests you've been using so far. While classes are generally defined in separate manifests, the node definition, like a test manifest, is a place where you actually declare them. Of course tests are just that, tests, while the node definitions in your `site.pp` manifest describe what you actually want your infrastructure to look like.

Because it's more ammenable to monitoring with the Learning VM quest tool, we'll be primarily using this `site.pp` method of classification in this Quest Guide. Once you've learned the basic mechanics of node definitions and class declarations, however, much of this knowledge will be portable to whatever methods of classification you decide to use later, including the PE Console node classifier you saw in the Power of Puppet quest.

Task 2 :

Open the site.pp manifest in your text editor.

```
vim /etc/puppetlabs/puppet/environments/production/manifests/site.pp
```

Skip to the bottom of the file. (You can use the vim shortcut `G`)

You'll see a `default` node definition. This is a special node definition that Puppet will apply to any node that's not specifically included in any other node definition.

Use the `include` syntax to add the `ntp` class to your default node definition:

```
node default {
  include ntp
}
```

Task 3 :

Note that triggering a puppet run with the `puppet agent` tool is useful for learning and testing, but that in a production environment you would want to let the puppet agent run as scheduled, every 30 minutes, by default. Because you'll be running puppet right after making changes to the `site.pp` manifest puppet may not have a chance to refresh its cache. If your changes to the `site.pp` manifest aren't reflected in a puppet run triggered by the `puppet agent -t` command, try running the command again.

Test the `site.pp` manifest with the `puppet parser validate` command, and run `puppet agent -t` to trigger a puppet run.

Once the puppet run is complete, use the puppet resource tool to inspect the `ntpd` service again. If the class has been successfully applied, you will see that the service is running.

Synching up

To avoid disrupting processes that rely on consistent timing, the ntpd service works gradually. It adds or removes a few microseconds to each tick of the system clock so as to slowly bring it into syncronization with the NTP server.

If you like, run the `ntpstat` command to check on the synchronization status. Don't worry about waiting to get synchronized. Because the Learning VM is virtual, its clock will probably be set based on the time it was created or last suspended. It's likely to be massively out of date with the time server, and it may take half an hour or more to get synchronized!

Class Defaults and Class Parameters

The `ntp` class includes default settings for most of its parameters. The `include` syntax you used let you concisely declare the class without modifying these defaults.

One of these defaults, for instance, tells Puppet which time servers to include in the NTP configuration file. To see what servers were specified by default, you can check the configuration file directly. Enter the command:

```
cat /etc/ntp.conf | grep server
```

You'll see a list of the default servers:

```
server 0.centos.pool.ntp.org
server 1.centos.pool.ntp.org
server 2.centos.pool.ntp.org
```

These ntp.org servers aren't actually time servers themselves; rather, they're access points that will pass you on to one of a pool of public timeservers. Most servers assigned through the ntp.org pool are provided by volunteers running NTP as an extra service on a mail or web server.

While these work well enough, you'll get more accurate time and use less network resources if you pick public timeservers in your area.

To manually specify which timeservers your NTPD service will poll, you'll need to override the default ntp.org pool servers set by the NTP module.

This is where Puppet's *class parameters* come in. *Class parameters* provide a method to set variables in a class as it's declared. The syntax for parameterized classes looks similar to the syntax for resource declarations. Have a look at the following example:

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

The `servers` parameter in our class declaration takes a list of servers as a value, not just one. This list of values, separated by commas (`,`) and wrapped in brackets (`[]`), is called an *array*. Arrays allow you assign a list of values to a single variable or attribute.

Task 4 :

In your `site.pp`, replace the `include ntp` line with a parameterized class declaration based on the example above. Use the servers from the example, or, if you know of a nearer timeserver, include that. You should always specify at least *three* timeservers for

NTP to function reliably. You might, for instance, include two from the ntp.org pool and one known nearby timeserver.

Task 5 :

Once you've made your changes to the `site.pp` manifest and used the puppet parser tool to validate your syntax, use the puppet agent tool to trigger a puppet run.

You will see in the output that Puppet has changed the `/etc/ntp.conf` file and triggered a refresh of the `ntpd` service.

Review

We covered some details of finding and downloading modules from the Puppet Forge with the `puppet module` tool. We also covered the common Package/File/Service pattern, and how it's used by the NTP module to install, configure, and run the ntpd service.

Rather than just running tests, you learned how to use the `site.pp` manifest to include classes within a node declaration.

After getting the ntpd service running, we went over class parameters, and showed how they can be used to set class parameters as a class is declared.

MySQL

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes
- Modules
- NTP

Quest Objectives

- Install and configure a MySQL server.
- Add a MySQL user, add a database, and grant permissions.

Getting Started

In this quest, we'll continue to explore how existing modules from the Puppet forge can simplify otherwise complex configuration tasks. You will use Puppet Labs' MySQL module to install and configure a server, then explore the custom resource types included with the module. To get started, enter the following command.

```
quest --start mysql
```

WhySQL?

The Puppet Labs MySQL module is a great example of how a well-written module can build on Puppet's foundation to simplify a complex configuration task without sacrificing robustness and control.

The module lets you install and configure both server and client MySQL instances, and extends Puppet's starndard resource types to let you manage MySQL *users*, *grants*, and *databases* with Puppet's standard resource syntax.

Server Install

Task 1:

Before getting started configuring your MySQL server installation, fetch the `puppetlabs-mysql` module from the Puppet Forge with the `puppet module` tool.

```
puppet module install puppetlabs-mysql
```

With this module installed in the Puppet master's module path, all the included classes are available to classify nodes.

Task 2 :

Edit `/etc/puppetlabs/puppet/environments/production/manifests/site.pp` to classify the LVM with the MySQL server class. Using class parameters, specify a root password and set the server's max connections to '1024.'

```
class { '::mysql::server':
    root_password    => 'strongpassword',
    override_options => { 'mysqld' => { 'max_connections' => '1024' } },
  }
```

In addition to some standard parameters like the `root_password`, the class takes a hash of `override_options`, which you can use to address any configuration options you would normally set in the `/etc/my.cnf` file. Using a hash lets you set any options you like in the MySQL configuration file without requiring each to be written into the class as a separate parameter. The structure of the `override_options` hash is analogous to the `[section]`, `var_name = value` syntax of a `my.cnf` file.

Task 3 :

Use the `puppet parser validate` tool to check your syntax, then trigger a puppet run:

```
puppet agent -t
```

If you want to check out your new database, you can connect to the MySQL monitor with the `mysql` command, and exit with the `\q` command.

To see the result of the 'max_connections' override option you set, grep the `/etc/my.cnf` file:

```
cat /etc/my.cnf | grep -B 9 max_connections
```

And you'll see that Puppet translated the hash into appropriate syntax for the MySQL configuration file:

```
[mysqld]
...
max_connections = 1024
```

Scope

It was easy enough to use Puppet to install and manage a MySQL server. The `puppetlabs-mysql` module also includes a bunch of classes that help you manage other aspects of your MySQL deployment.

These classes are organized within the module directory structure in a way that matches Puppet's scope syntax. Scope helps to organize classes, telling Puppet where to look within the module directory structure to find each class. It also separates namespaces within the module and your Puppet manifests, preventing conflicts between variables or classes with the same name.

Take a look at the directories and manifests in the MySQL module. Use the `tree` command with a filter to include only `.pp` manifest files:

```
tree -P *.pp /etc/puppetlabs/puppet/environments/production/modules/mysql/
manifests/
```

You'll see something like the following:

```
/etc/puppetlabs/puppet/environments/production/modules/mysql/manifests/
├── backup.pp
├── bindings
|   ├── java.pp
|   ├── perl.pp
|   ├── php.pp
|   ├── python.pp
|   └── ruby.pp
├── bindings.pp
├── client
|   └── install.pp
├── client.pp
├── db.pp
├── init.pp
├── params.pp
├── server
|   ├── account_security.pp
|   ├── backup.pp
|   ├── config.pp
|   ├── install.pp
|   ├── monitor.pp
|   ├── mysqltuner.pp
|   ├── providers.pp
|   ├── root_password.pp
|   └── service.pp
└── server.pp
```

Notice the `server.pp` manifest in the top level of the `mysql/manifests` directory.

You were able to declare this class as `mysql::server`. Based on this scoped class name, Puppet knows to find the class definition in a manifest called `server.pp` in the manifest directory of the MySQL module.

So `mysql::server` corresponds to:

```
/etc/puppetlabs/puppet/environments/production/modules/mysql/manifests/server.pp
```

To take an example one level deeper, the `mysql::server::account_security` class corresponds to:

```
/etc/puppetlabs/modules/mysql/manifests/server/account_security.pp
```

We won't be calling the `mysql` class directly in this quest, but it's worth reiterating the special case of a module's self-named class. This will always be found in a manifest called `init.pp` in the top level of the module's manifests directory.

So the `mysql` class is found here:

```
/etc/puppetlabs/modules/mysql/manifests/init.pp
```

Account Security

For security reasons, you will generally want to remove the default users and the 'test' database from a new MySQL installation. The `account_security` class mentioned above does just this.

Task 4 :

Go back to your `site.pp` manifest and include the `mysql::server::account_security` class. Remember, you don't need to pass any parameters to this class, so a simple `include` statement will work in place of a parameterized class declaration.

Trigger a Puppet run, and you will see notices indicating that the test database and two users have been removed:

```
Notice:
/Stage[main]/Mysql::Server::Account_security/Mysql_database[test]/ensure:
removed
Notice:
/Stage[main]/Mysql::Server::Account_security/Mysql_user[@localhost]/ensure:
removed
Notice:
/Stage[main]/Mysql::Server::Account_security/Mysql_user[root@127.0.0.1]/ensure:
removed
```

Types and Providers

The MySQL module includes some custom *types and providers* that let you manage some critical bits of MySQL as resources with the Puppet DSL just like you would with a system user or service.

A **type** defines the interface for a resource: the set of *properties* you can use to define a desired state for the resource, and the *parameters* that don't directly map to things on the system, but tell Puppet how to manage the resource. Both properties and parameters appear in the resource declaration syntax as attribute value pairs.

A **provider** is what does the heavy lifting to bring the system into line with the state defined by a resource declaration. Providers are implemented for a wide variety of supported operating systems. They are a key component of the Resource Abstraction Layer (RAL), translating the universal interface defined by the **type** into system-specific implementations.

The MySQL module includes custom types and providers that make `mysql_user`, `mysql_database`, and `mysql_grant` available as resources.

Database, User, Grant:

Task 5 :

These custom resource types make creating a new database with Puppet pretty simple.

Just add the following resource declaration to your node definition in the `site.pp` manifest.

```
mysql_database { 'lvm':
      ensure  => 'present',
      charset => 'utf8',
  }
```

Similarly, with a user, all you have to do is specify the name and host as the resource title, and set the ensure attribute to 'present'. Enter the following in your node definition as well.

```
mysql_user { 'lvm_user@localhost':
    ensure => 'present',
  }
```

Now that you have a user and database, you can use a grant to define the privileges for that user. Note that the `*` character will match any table, meaning that the `lvm_user` has access to all tables in the `lvm` database.

```
mysql_grant { 'lvm_user@localhost/lvm.*':
    ensure      => 'present',
    options     => ['GRANT'],
    privileges  => ['ALL'],
    table       => 'lvm.*',
    user        => 'lvm_user@localhost',
  }
```

Once you've added declarations for these three custom resources, use the `puppet parser validate` command on the `site.pp` manifest to check your syntax, and trigger a puppet run with

```
puppet agent -t
```

Review

In this quest, you learned how to install and make configuration changes to a MySQL server. You also got an overview of how classes are organized withing the module structure and how their names within your Puppet manifests reflect this organization.

The MySQL module we used for this quest provides a nice example of how custom types and providers can extend Puppet's available resources to make service or application specific elements easily configurable through Puppet's resource declaration syntax.

Variables and Class Parameters

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes
- Modules

Quest Objectives

- Learn how variables and parameters can make your modules adaptable.

Getting Started

In this quest you'll get a taste of how variables fit into good module design, and you'll learn how to integrate them into your own Puppet classes and resource declarations.

If you completed the NTP and MySQL quests, you've already seen how parameterized classes can be used to adapt a module to your specific needs. In this quest, you'll see how to include parameters in your own classes.

To explore these two concepts, you'll be writing a module to manage a user account. First, you'll write a simple class using a few variables, then you'll add parameters to your class so that those variables can be set when the class is declared.

When you're ready to get started, type the following command to begin:

```
quest --start variables
```

Variables

> *Beauty is variable, ugliness is constant.*
> *-Douglas Horton*

Puppet's variable syntax lets you assign a name to a bit of data, so you can use that variable name later in your manifest to refer to the value assigned to it. In Puppet's syntax, variable names are prefixed with a `$` (dollar sign), and a value is assigned with the `=` operator.

Assigning a short string to a variable, for example, would look like this:

```
$myvariable = 'look, a string!'
```

Once you have defined a variable you can use it anywhere in your manifest you would have used the assigned value.

The basics of variables will seem familiar if you know another scripting or programming language. However, there are a few caveats you should be aware of when using variables in Puppet:

1. Unlike resource declarations, variable assignments are parse-order dependent. This means that you must assign a variable in your manifest *before* you can use it.
2. If you try to use a variable that has not been defined, the Puppet parser won't complain. Instead, Puppet will treat the variable as having the special `undef` value.
3. You can only assign a variable once within a single scope. Once it's assigned, the value cannot be changed. (If this makes you wonder how accurate the term "variable" is, you're not alone!)

Variable Interpolation

Variable interpolation gives you a way to insert a string stored as a variable into another string. For instance, if you want Puppet to manage the files in the `/var/www/html/lvmguide` directory you set up in the Power of Puppet quest, you can assign this directory path to a variable:

```
$doc_root = '/var/root/www/html/lvmguide'
```

Once the variable is set, you can use the variable interpolation syntax to insert it into a string. The variable name is preceded by a `$` and wrapped in curly braces (`${var_name}`). For example, you might use it in the title of a few *file* resource declarations:

```
file { "${doc_root}index.html":
  ...
}
file { "${doc_root}about.html":
  ...
}
```

Not only is this more concise, but using variables allows you to set the directory once, depending, for instance, on the kind of server you're running, and let that specified directory be applied throughout your class.

Note that a string that includes an interpolated variable must be wrapped in double quotation marks (`"..."`), rather than the single quotation marks that surround an ordinary string. These double quotation marks tell Puppet to find and parse special syntax within the string, rather than interpreting it literally.

Manage a Web Content with Variables

To better understand how variables work in context, we'll walk you through creating a simple `web` module to drop some new files into the directory served by the Apache service you set up in the Power of Puppet quest.

Task 1 :

First, you'll need to create the directory structure for your module.

Make sure you're in the `modules` directory for Puppet's modulepath.

```
cd /etc/puppetlabs/puppet/environments/production/modules/
```

Now create an `web` directory:

```
mkdir web
```

...and your `manifests` and `tests` directories:

```
mkdir web/{manifests,tests}
```

Task 2 :

Now you're ready to create your main manifest, where you'll define the `web` class.

```
vim web/manifests/init.pp
```

```
class web {

  $doc_root = '/var/www/html/lvmguide'

  $english = 'Hello world!'
  $french = 'Bonjour le monde!'

  file { "${doc_root}/hello.html":
    ensure => 'present',
    content => "<em>${english}</em>",
  }

  file { "${doc_root}/bonjour.html"}
    ensure => 'present',
    content => "<em>${french}</em>",
  }

}
```

Note that if you wanted to make a change to the `$doc_root` directory, you'd only have to do this in one place. While there are more advanced forms of data separation in Puppet, the basic principle is the same: The more distinct your code is from the underlying data, the more resuable it is, and the less difficult it will be to refactor when you have to make changes later.

Task 3 :

Once you've validated your manifest with the `puppet parser` tool, create a test for your manifest with an `include` statement for the web class you created.

Task 4 :

Run the test, using the `--noop` flag for a dry run before triggering your real `puppet apply`.

From your web browser on your host machine, connect to `http://<LVM's IP>/hello.html` and `http://<LVM's IP>/bonjour.html`, and you'll see pages you've set up.

Class Parameters

> *Freedom is not the absence of obligation or restraint, but the freedom of movement within healthy, chosen parameters.*
> *-Kristin Armstrong*

Now that you've created your basic `web` class and replaced some of the values in your resource declarations with variables, we'll move on to **class parameters**. Class parameters give you a way to set the variables within a class as it's **declared** rather than when the class is **defined**.

When defining a class, include a list of parameters and optional default values between the class name and the opening curly brace. So a parameterized class is defined as below:

```
class classname ( $parameter = 'default' ) {
  ...
}
```

Once defined, a parameterized class can be **declared** with a syntax similar to that of resource declarations, including key value pairs for each parameter you want to set.

```
class {'classname':
  parameter => 'value',
}
```

Say you want to make these pages available not just on the Learning VM, but on each node in your infrastructure, but that you want a few changes on each one. Instead of rewriting the whole class or module with these minor changes, you can use class parameters to customize these values as the class is declared.

Task 5 :

To get started re-writing your `web` class with parameters, reopen the `web/manifests/init.pp` manifest. You've already written variables into the resource declarations, so turning it into a parameterized class will be quick. Just add your parameters in a pair of parenthesis following the name of the class:

```
class web ( $page_name, $message ) {
```

Now create a third file resource declaration to use the variables set by your parameters:

```
file { "${doc_root}/${page_name}.html":
  ensure => 'present',
  content => "<em>${message}</em>",
}
```

Task 6 :

As before, use the test manifest to declare the class. You'll open `web/tests/init.pp` and replace the simple `include` statement with the parameterized class declaration syntax to set each of the class parameters:

```
class {'web':
  page_name => 'hola',
  message   => 'Hola mundo!',
}
```

Task 7 :

Now give it a try. Go ahead and do a `--noop` run, then apply the test.

Your new page should now be available as `http://<LVM's IP>/hola.html`!

Review

In this quest you've learned how to take your Puppet manifests to the next level by using variables. You learned how to assign a value to a variable and then reference the variable by name whenever you need its content. You also learned how to interpolate variables.

In addition to learning about variables, interpolating variables, and facts, you also gained more hands-on learning with constructing Puppet manifests using Puppet's DSL. We hope you are becoming more familar and confident with using and writing Puppet code as you are progressing.

Conditional Statements

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes
- Modules
- Variables and Class Parameters

Quest Objectives

- Learn how to use conditional logic to make your manifests adaptable.
- Understand the syntax and function of the `if`, `unless`, `case`, and `selector` statements.

Getting Started

Conditional statements allow you to write Puppet code that will return different values or execute different blocks of code depending on conditions you specify. In conjunction with Facter, which makes details of a machine available as variables, this lets you write Puppet code that flexibly accomodates different platforms, operating systems, and functional requirements.

To start this quest enter the following command:

```
quest --start conditionals
```

Writing for Flexibility

> *The green reed which bends in the wind is stronger than the mighty oak which breaks in a storm.*
> *-Confucius*

Because Puppet manages configurations on a variety of systems fulfilling a variety of roles, great Puppet code means flexible and portable Puppet code. While the *types* and *providers* that form the core of Puppet's *resource abstraction layer* do a lot of heavy

lifting around this kind of adaptation, there are some things better left in the hands of competent practitioners rather than hard-coded in Puppet itself.

As you move from general platform-related implementation details to specific application-related implementation details, it starts making less sense to rely on Puppet to make decisions automatically, and much more sense for a module developer or user to make his or her own choices.

It's sensible, for example, for Puppet's `package` providers to take care of installing and maintaining packages. The inputs and outputs are standardized and stable enough that what happens in between, as long as it happens reliably, can be safely hidden by abstraction; once it's done, the details are no longer important.

What package is installed, on the other hand, isn't something you can safely forget. In this case, the inputs and outputs are not so neatly delimited. Though there are often broadly equivalent packages for different platforms, the equivalence isn't always complete; configuration details will often vary, and these details will likely have to be accounted for elsewhere in your Puppet module.

While Puppet's built-in providers can't themselves guarantee the portability of your Puppet code at this higher level of implementation, Puppet's DSL gives you the tools to build adaptability into your modules. **facts** and **conditional statements** are the bread and butter of this functionality.

Facts

> *Get your facts first, then distort them as you please.*
> *-Mark Twain*

You already encountered the *facter* tool when we asked you to run `facter ipaddress` in the setup section of this quest guide. While it's nice the be able to run facter from the command line, its real utility is to make information about a system available to use as variables in your manifests.

Combined with conditionals, which we'll get to in a moment, **facts** give you a huge amount of power to write portability into your modules.

To get a full list of facts available to facter, enter the command:

```
facter -p | less
```

While facter is an important component of Puppet and is bundled with Puppet Enterprise, it's actually one of the many separate open-source projects integrated into the Puppet ecosystem.

Any of the facts you see listed here can be used within your Puppet code with the syntax `$::factname`. The double colons `::` indicate that the fact is defined in what's called *top scope*, that is, before any variables in your node definitions or classes are assigned.

While you could technically use a fact without the `::`, you would risk having it mistakenly overridden by a locally defined variable with the same name. The `::` tells Puppet to look directly in the top scope instead of using the first variable it finds with a matching name. The Puppet style guide suggests using this syntax consistently to avoid naming collisions.

Conditions

> *Just dropped in (to see what condition my condition was in)*
> *-Mickey Newbury*

Conditional statements return different values or execute different blocks of code depending on the value of a specified variable. This is key to getting your Puppet modules to perform as desired on machines running different operating systems and fulfilling different roles in your infrastructure.

Puppet supports a few different ways of implementing conditional logic:

- `if` statements,
- `unless` statements,
- case statements, and
- selectors.

The 'if' Statement

Puppet's `if` statements behave much like those in many other programming and scripting languages.

An `if` statement includes a condition followed by a block of Puppet code that will only be executed **if** that condition evaluates as **true**. Optionally, an `if` statement can also include any number of `elsif` clauses and an `else` clause. Here are some rules:

- If the `if` condition fails, Puppet moves on to the `elsif` condition (if one exists).
- If both the `if` and `elsif` conditions fail, Puppet will execute the code in the `else` clause (if one exists).
- If all the conditions fail, and there is no `else` block, Puppet will do nothing and move on.

Lets say you want to give the user you're creating with your accounts module administrative priveleges. You have a mix of CentOS and Debian systems in your infrastructure. On your CentOS machines, you use the `wheel` group to manage superuser privileges, while you use an `admin` group on the Debian machines. With the `if` statement and the `operatingsystem` fact from facter, this kind of adjustment is easy to automate with Puppet.

Before getting started, make sure you're working in the `modules` directory:

```
cd /etc/puppetlabs/puppet/environments/production/modules
```

Task 1:

Create an `accounts` directory:

```
mkdir accounts
```

And your `tests` and `manifests` directories:

```
mkdir accounts/{manifests,tests}
```

Task 2:

Open the `accounts/manifests/init.pp` manifest in Vim.

At the beginning of the `accounts` class definition, you'll include some conditional logic to set the `$groups` variable based on the value of the `$::operatingsystem` fact. In both cases, you'll add the user the new group made solely for that user, defined by the `$name` parameter. If the operating system is CentOS, you'll also add the user to the `wheel` group, and if the operating system is Debian you'll add the user to the `admin` group.

So the beginning of your class definition should looks something like this:

```
class accounts ($name) {

  if $::operatingsystem == 'centos' {
    $groups = 'wheel'
  }
  elsif $::operatingsystem == 'debian' {
    $groups = 'admin'
  }
  else {
    fail( "This module doesn't support ${::operatingsystem}." )
  }

  notice ( "Groups for user ${name} set to ${groups}" )

  ...

}
```

Note that the string matches are *not* case sensitive, so 'CENTOS' would work just as well as 'centos'. Finally, in the `else` block, you'll raise a warning that the module doesn't support the current OS.

Once you've written the conditional logic to set the `$groups` variable, edit the `user` resource declaration to assign the `$groups` variable to the `groups` attribute.

```
class accounts ($name) {

  ...

  user { $name:
    ensure => 'present',
    home   => "/home/${name}",
    groups => $groups,
  }

  ...

}
```

Make sure that your manifest can pass a `puppet parser validate` check before continuing on.

Task 3 :

Create a test manifest (`accounts/tests/init.pp`) and declare the accounts manifest with the name parameter set to `dana`.

```
class {'accounts':
  name => 'dana',
}
```

Task 4 :

The Learning VM is running CentOS, so to test what would happen on a Debian OS you'll have to override the `operatingsystem` fact with a little environment variable magic. To provide a custom value for any facter fact as you run a `puppet apply`, you can include `FACTER_factname=new_value` before your new terminal command.

Combining this with the `--noop` flag, you can do a quick test of how your manifest would run on a different system before setting up a full testing environment.

```
FACTER_operatingsystem=Debian puppet apply --noop accounts/tests/init.pp
```

Look in the list of notices, and you'll see the changes that would have been applied.

Task 5 :

Try one more time with an unsupported operating system to check the fail condition:

```
FACTER_operatingsystem=Darwin puppet apply --noop accounts/tests/init.pp
```

Task 6 :

Now go ahead and run a `puppet apply --noop` on your test manifest without setting the environment variable. If this looks good, drop the `--noop` flag to apply the catalog generated from your manifest.

You can use the `puppet resource` tool to verify the results.

The 'unless' Statement

The `unless` statement works like a reversed `if` statement. An `unless` statements takes a condition and a block of Puppet code. It will only execute the block **if** the condition is **false**. If the condition is true, Puppet will do nothing and move on. Note that there is no equivalent of `elsif` or `else` clauses for `unless` statements.

The 'case' Statement

Like `if` statements, case statements choose one of several blocks of Puppet code to execute. Case statements take a control expression, a list of cases, and a series of Puppet code blocks that correspond to those cases. Puppet will execute the first block of code whose case value matches the control expression.

A special `default` case matches anything. It should always be included at the end of a case statement to catch anything that did not match an explicit case.

For instance, if you were setting up an Apache webserver, you might use a case statement like the following:

```
case $::operatingsystem {
  'CentOS': { $apache_pkg = 'httpd' }
  'Redhat': { $apache_pkg = 'httpd' }
  'Debian': { $apache_pkg = 'apache2' }
  'Ubuntu': { $apache_pkg = 'apache2' }
  default: { fail("Unrecognized operating system for webserver.") }
}

package { $apache_pkg :
  ensure => present,
}
```

This would allow you to always install and manage the right Apache package for a machine's operating system. Accounting for the differences between various platforms

is an important part of writing flexible and re-usable Puppet code, and it's a paradigm you will encounter frequently in published Puppet modules.

The 'selector' Statement

Selector statements are similar to `case` statements, but instead of executing a block of code, a selector assigns a value directly. A selector might look something like this:

```
$rootgroup = $::osfamily ? {
  'Solaris'  => 'wheel',
  'Darwin'   => 'wheel',
  'FreeBSD'  => 'wheel',
  'default'  => 'root',
}
```

Here, the value of the `$rootgroup` is determined based on the control variable `$::osfamily`. Following the control variable is a `?` (question mark) symbol. In the block surrounded by curly braces are a series of possible values for the `$::osfamily` fact, followed by the value that the selector should return if the value matches the control variable.

Because a selector can only return a value and cannot execute a function like `fail()` or `warning()`, it is up to you to make sure your code handles unexpected conditions gracefully. You wouldn't want Puppet to forge ahead with an inappropriate default value and encounter errors down the line.

Review

In this quest, you saw how you can use facts from the `facter` tool along with conditional logic to write Puppet code that will adapt to the envoronment where you're applying it.

You used an `if` statement in conjunction with the `$::osfamily` variable from facter to determine how to set the group for an administrator user account.

We also covered a few other forms of conditional statement: 'unless', the case statement, and the selector. Though there aren't any hard-and-fast rules for which conditional statement is best in a given situation, there will generally be one that results in the most concise and readible code. It's up to you to decide what works best.

Resource Ordering

Prerequisites

- Welcome
- Power of Puppet
- Resources
- Manifests and Classes
- Modules
- Variables and Class Parameters
- Conditions

Quest Objectives

- Understand why some resources must be managed in a specific order.
- Use the `before`, `require`, `notify`, and `subscribe` metaparameters to effectively manage the order that Puppet applies resource declarations.

Getting Started

This quest will help you learn more about specifying the order in which Puppet should manage resources in a manifest. When you're ready to get started, type the following command:

```
quest --start ordering
```

Autorequires and Explicit Ordering

We are likely to read instructions from top to bottom and execute them in that order. When it comes to resource declarations in a Puppet manifest, Puppet does things a little differently. It works through the problem as though it were given a list of things to do, and it was left to decide the most efficient way to get them done.

The **catalog** is a compilation of all the resources that will be applied to a given system, and the relationships between those resources. For some resource types, Puppet is clever enough to figure out necessary relationships among resources itself. These implicit resource relationships are called **autorequires**.

You can see what a resource can autorequire with the `puppet describe` tool.

Take a look at the entry for the `group` resource:

```
puppet describe group
```

A few paragraphs down, you'll see the following section:

Autorequires: If Puppet is managing the user or group that owns a file, the file resource will autorequire them. If Puppet is managing any parent directories of a file, the file resource will autorequire them.

(The information you find with the `describe` tool can also be found in the type reference section of Puppets docs site.)

When you declared `user` and `file` resources in your `accounts` module, the autorequires of these resources ensured that everything went smoothly.

Sometimes, however, you will need to tell Puppet explicitly that a resource declaration is applied before another. For instance, if you wish to declare that a service should be running, you need to ensure that the package for that service is installed and configured before you can start the service. Just as Puppet's built-in providers don't (and shouldn't!) automatically decide what package you might want to use on a given operating system, Puppet doesn't try to guess what services are associated with a given package.

Often, more than one package provides the same service, and what if you are using a package you built yourself? Since Puppet cannot *always* conclusively determine the mapping between a package and a service (the names of the software package and the service or executable it provides are not always the same either), it is up to the user to specify the relationship between them.

To overcome this issue, Puppet's syntax includes a few different ways to explicitly manage resource ordering.

Relationship Metaparameters

One way of telling Puppet what order to use when managing resources is by including ordering **metaparameters** in your resource declarations.

Metaparameters are attributes that can be set in any resource to tell Puppet *how* to manage that resource. In addition to resource ordering, metaparameters can help with things like logging, auditing, and scheduling. For now, however, we'll be concentrating only on resource ordering metaparameters.

There are four metaparameter **attributes** that you can include in your resource declaration to order relationships among resources.

- `before` causes a resource to be applied **before** a specified resource.
- `require` causes a resource to be applied **after** a specified resource.

- `notify` causes a resource to be applied **before** the specified resource, just as with `before`. Additionally, notify will generate a refresh event for the specified resource when the notifying resource changes.
- `subscribe` causes a resource to be applied **after** the specified resource. The subscribing resource will be refreshed if the target resource changes.

The **value** of the relationship metaparameter is the title or titles (in an array) of one or more target resources.

Here's an example of how the `notify` metaparameter is used:

```
file {'/etc/ntp.conf':
  ensure => file,
  source => 'puppet:///modules/ntp/ntp.conf',
  notify => Service['ntpd'],
}

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

In the above, the file `/etc/ntp.conf` is managed. The contents of the file are sourced from the file `ntp.conf` in the ntp module's files directory. Whenever the file `/etc/ntp.conf` changes, a refresh event is triggered for the service with the title `ntpd`. By virtue of using the notify metaparameter, we ensure that Puppet manages the file first, before it manages the service, which is to say that `notify` implies `before`.

Refresh events, by default, restart a service (such as a server daemon), but you can specify what needs to be done when a refresh event is triggered, using the `refresh` attribute for the `service` resource type, which takes a command as the value.

In order to better understand how to explicitly specify relationships between resources, we're going to use SSH as our example. Setting the `GSSAPIAuthentication` setting for the SSH daemon to `no` will help speed up the login process when one tries to establish an SSH connection to the Learning VM.

Let's try to disable GSSAPIAuthentication, and in the process, learn about resource relationships.

Before getting started, ensure that you're in the `modules` directory:

```
cd /etc/puppetlabs/puppet/environments/production/modules
```

Task 1:

Create an `sshd` directory and create `tests`, `manifests`, and `files` subdirectories.

Task 2:

We've already prepared an `sshd_config` file to use as a base for your source file. Copy it into your module's `files` directory:

```
cp /root/examples/sshd_config sshd/files/sshd_config
```

Task 3 :

Create a `sshd/manifests/init.pp` manifest with the following class definition:

```
class sshd {

  file { '/etc/ssh/sshd_config':
    ensure => file,
    mode   => 600,
    source => 'puppet:///modules/sshd/sshd_config',
  }

}
```

This will tell Puppet to ensure that the file `/etc/ssh/sshd_config` exists, and that the contents of the file should be sourced from the file `sshd/files/sshd_config`. The `source` attribute also allows us to use a different URI to specify the file, something we discussed in the Modules quest. For now, we are using a file in `/root/examples` as the content source for the SSH daemon's configuration file.

Now let us disable GSSAPIAuthentication.

Task 4 :

Disable GSSAPIAuthentication for the SSH service

Edit the `sshd/files/sshd_config` file.
Find the line that reads:

```
GSSAPIAuthentication yes
```

and edit it to read:

```
GSSAPIAuthentication no
```

Save the file and exit the text editor.

Even though we have edited the source for the configuration file for the SSH daemon, simply changing the content of the configuration file will not disable the GSSAPIAuthentication option. For the option to be disabled, the service (the SSH server daemon) needs to be restarted. That's when the newly specified settings will take effect.

Let's now add a metaparameter that will tell Puppet to manage the `sshd` service and have it `subscribe` to the config file. Add the following Puppet code below your file resource in the `sshd` module's `init.pp` manifest:

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

Notice that in the above the `subscribe` metaparameter has the value `File['/etc/ssh/sshd_config']`. The value indicates that we are talking about a file resource (that Puppet knows about), with the *title* `/etc/ssh/sshd_config`. That is the file resource we have in the manifest. References to resources always take this form. Ensure that the first letter of the type ('File' in this case) is always capitalized when you refer to a resource in a manifest.

Task 5 :

Create a test manifest to include your `sshd` class.

Let's apply the change. Remember to check syntax and do a dry-run using the `--noop` flag before using `puppet apply` to run your test manifest.

You will see Puppet report that the content of the `/etc/ssh/sshd_config` file changed. You should also be able to see that the SSH service was restarted.

In the above example, the `service` resource will be applied **after** the `file` resource. Furthermore, if any other changes are made to the targeted file resource, the service will refresh.

Package/File/Service

Wait a minute! We are managing the service `sshd`, we are managing its configuration file, but all that would mean nothing if the SSH server package is not installed. So, to round it up, and make our manifest complete with regards to managing the SSH server on the VM, we have to ensure that the appropriate `package` resource is managed as well.

On CentOS machines, such as the VM we are using, the `openssh-server` package installs the SSH server.

- The package resource makes sure the software and its config file are installed.
- The file resource config file depends on the package resource.
- The service resources subscribes to changes in the config file.

The **package/file/service** pattern is one of the most useful idioms in Puppet. It's hard to overstate the importance of this pattern! If you only stopped here and learned this, you could still get a lot of work done using Puppet.

To stay consistent with the package/file/service idiom, let's dive back into the sshd init.pp file and add the `openssh-server` package to it.

Task 6 :

Manage the package for the SSH server

Add the following code above your file resource in your `sshd/manifests/init.pp` manifest

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

Make sure to check the syntax. Once everything looks good, go ahead and apply the manifest.

Notice that we use `before` to ensure that the package is managed before the configuration file is managed. This makes sense, since if the package weren't installed, the configuration file (and the `/etc/ssh/` directory that contains it would not exist. If you tried to manage the contents of a file in a directory that does not exist, you are destined to fail. By specifying the relationship between the package and the file, we ensure success.

Now we have a manifest that manages the package, configuration file and the service, and we have specified the order in which they should be managed.

Review

In this Quest, we learned how to specify relationships between resources, to provide for better control over the order in which the resources are managed by Puppet. We also learned of the Package-File-Service pattern, which emulates the natural sequence of managing a service on a system. If you were to manually install and configure a service, you would first install the package, then edit the configuration file to set things up appropriately, and finally start or restart the service.

Afterword

Thank you for embarking on the journey to learn Puppet. We hope that the Learning VM and the Quest Guide helped you get started on this journey.

We had a lot of fun writing the guide, and hope it was fun to read and use as well. This is just the beginning for us, too. We want to make Learning VM the best possible first step in a beginner's journey to learning Puppet. With time, we will add more quests covering more concepts.

If you are interested in learning more about Puppet, please visit the Puppet Labs Workshop.

To get started with Puppet Enterprise download it for free.

Please let us know about your experience with the Learning VM! Fill out our feedback survey or reach us at learningvm@puppetlabs.com. We look forward to hearing from you.

Troubleshooting

If you have feedback or run into any issues with the Learning VM, please visit our public issue tracker or email us at learningvm@puppetlabs.com.

Puppet Labs maintains a list of known Puppet Enterprise issues.

Common Issues:

I've completed a task, but it hasn't registered in the quest tool.

Some tasks are verified on the presence of a specific string in a file, so a typo can prevent the task from registering as complete even if your Puppet code validates and your puppet agent run is successful. You can check the actual spec tests we use with the quest tool in the `~/.testing/spec/localhost` directory.

While most tasks validate some change you have made to the system, there are a few that check for a specific line in your bash history. The file that tracks this history isn't created until you log out from the VM for the first time. In the setup instructions, we suggest that you log out from the VM before establishing an SSH connection. If you haven't done this, certain tasks won't register as complete. Also, because the quest tool looks for a specific command, the tasks may not register as complete if you've included a slight variation even if it has the same effect.

I can't access the PE Console with Safari.

This is a known issue with the way Safari handles certificates. You may encounter a dialog box prompting you to select a certificate. If this happens, you may have to click `Cancel` several times to access the console. This issue will be fixed in a future release of PE.

The Learning VM cannot complete a Puppet run, or does so very slowly.

It's likely that you haven't assigned enough memory or processor cores to the Learning VM, or that your host system doesn't have the available resources to allocate. Power down the VM and increase the processor cores to 2 and the memory to 4 GB. You may also need to close other processes running on the host machine to free up resources.

Glossary of Puppet Vocabulary

An accurate, shared vocabulary goes a long way to ensure the success of a project. To that end, this glossary defines the most common terms Puppet users rely on.

attribute

Attributes are used to specify the state desired for a given configuration resource. Each resource type has a slightly different set of possible attributes, and each attribute has its own set of possible values. For example, a package resource (like `vim`) would have an `ensure` attribute, whose value could be `present`, `latest`, `absent`, or a version number:

```
package {'vim':
      ensure   => present,
      provider => apt,
    }
```

The value of an attribute is specified with the `=>` operator; attribute/value pairs are separated by commas.

agent

(or **agent node**)

Puppet is usually deployed in a simple client-server arrangement, and the Puppet client daemon is known as the "agent." By association, a computer running puppet agent is usually referred to as an "agent node" (or simply "agent," or simply "node").

Puppet agent regularly pulls configuration catalogs from a puppet master server and applies them to the local system.

catalog

A catalog is a compilation of all the resources that will be applied to a given system and the relationships between those resources.

Catalogs are compiled from manifests by a puppet master server and served to agent nodes. Unlike the manifests they were compiled from, they don't contain any conditional logic or functions. They are unambiguous, are only relevant to one specific node, and are machine-generated rather than written by hand.

class

A collection of related resources, which, once defined, can be declared as a single unit. For example, a class could contain all of the elements (files, settings, modules, scripts, etc) needed to configure Apache on a host. Classes can also declare other classes.

Classes are singletons, and can only be applied once in a given configuration, although the `include` keyword allows you to declare a class multiple times while still only evaluating it once.

> Note:
>
> Being singletons, Puppet classes are not analogous to classes in object-oriented programming languages. OO classes are like templates that can be instantiated multiple times; Puppet's equivalent to this concept is defined types.

classify

(or **node classification**)

To assign classes to a node, as well as provide any data the classes require. Writing a class makes a set of configurations available; classifying a node determines what its actual configuration will be.

Nodes can be classified with node definitions in the site manifest, with an ENC, or with both.

declare

To direct Puppet to include a given class or resource in a given configuration. To declare resources, use the lowercase `file {'/tmp/bar':}` syntax. To declare classes, use the `include` keyword or the `class {'foo':}` syntax. (Note that Puppet will automatically declare any classes it receives from an external node classifier.)

You can configure a resource or class when you declare it by including attribute/value pairs.

Contrast with "define."

define

To specify the contents and behavior of a class or a defined resource type. Defining a class or type doesn't automatically include it in a configuration; it simply makes it available to be declared.

define (noun)

(or **definition**)

An older term for a defined resource type.

define (keyword)

The language keyword used to create a defined type.

defined resource type

(or **defined type**)

See "type (defined)."

ENC

See external node classifier.

environment

An arbitrary segment of your Puppet site, which can be served a different set of modules. For example, environments can be used to set up scratch nodes for testing before roll-out, or to divide a site by types of hardware.

expression

The Puppet language supports several types of expressions for comparison and evaluation purposes. Amongst others, Puppet supports boolean expressions, comparision expressions, and arithmetic expressions.

external node classifier

(or **ENC**)

An executable script, which, when called by the puppet master, returns information about which classes to apply to a node.

ENCs provide an alternate method to using the main site manifest (`site.pp`) to classify nodes. An ENC can be written in any language, and can use information from any pre-existing data source (such as an LDAP db) when classifying nodes.

An ENC is called with the name of the node to be classified as an argument, and should return a YAML document describing the node.

fact

A piece of information about a node, such as its operating system, hostname, or IP address.

Facts are read from the system by Facter, and are made available to Puppet as global variables.

Facter can also be extended with custom facts, which can expose site-specific details of your systems to your Puppet manifests.

Facter

Facter is Puppet's system inventory tool. Facter reads facts about a node (such as its hostname, IP address, operating system, etc.) and makes them available to Puppet.

Facter includes a large number of built-in facts; you can view their names and values for the local system by running `facter` at the command line.

In agent/master Puppet arrangements, agent nodes send their facts to the master.

filebucket

A repository in which Puppet stores file backups when it has to replace files. A filebucket can be either local (and owned by the node being managed) or site-global (and owned by the puppet master). Typically, a single filebucket is defined for a whole network and is used as the default backup location.

function

A statement in a manifest which returns a value or makes a change to the catalog.

Since they run during compilation, functions happen on the puppet master in an agent/master arrangement. The only agent-specific information they have access to are the facts the agent submitted.

Common functions include `template`, `notice`, and `include`.

global scope

See scope.

host

Any computer (physical or virtual) attached to a network.

In the Puppet docs, this usually means an instance of an operating system with the Puppet agent installed. See also "Agent Node".

host (resource type)

An entry in a system's `hosts` file, used for name resolution.

idempotent

Able to be applied multiple times with the same outcome. Puppet resources are idempotent, since they describe a desired final state rather than a series of steps to follow.

(The only major exception is the `exec` type; exec resources must still be idempotent, but it's up to the user to design each exec resource correctly.)

inheritance (class)

A Puppet class can be derived from one other class with the `inherits` keyword. The derived class will declare all of the same resources, but can override some of their attributes and add new resources.

> ***Note:*** *Most users should avoid inheritance most of the time. Unlike object-oriented programming languages, inheritance isn't terribly important in Puppet; it is only useful for overriding attributes, which can be done equally well by using a single class with a few parameters.*

inheritance (node)

Node statements can be derived from other node statements with the `inherits` keyword. This works identically to the way class inheritance works.

> Note:
>
> Node inheritance **should almost always be avoided.** Many new users attempt to use node inheritance to look up variables that have a common default value and a rare specific value on certain nodes; it is not suited to this task, and often yields the opposite of the expected result. If you have a lot of conditional per-node data, we recommend using the Hiera tool or assigning variables with an ENC instead.

master

In a standard Puppet client-server deployment, the server is known as the master. The puppet master serves configuration catalogs on demand to the puppet agent service that runs on the clients.

The puppet master uses an HTTP server to provide catalogs. It can run as a standalone daemon process with a built-in web server, or it can be managed by a production-grade web server that supports the rack API. The built-in web server is meant for testing, and is not suitable for use with more than ten nodes.

manifest

A file containing code written in the Puppet language, and named with the `.pp` file extension. The Puppet code in a manifest can:

- Declare resources and classes
- Set variables
- Evaluate functions
- Define classes, defined types, and
- nodes

Most manifests are contained in modules. Every manifest in a module should define a single class or defined type.

The puppet master service reads a single "site manifest," usually located at `/etc/puppet/manifests/site.pp`. This manifest usually defines nodes, so that each managed agent node will receive a unique catalog.

metaparameter

A resource attribute that can be specified for any type of resource. Metaparameters are part of Puppet's framework rather than part of a specific type, and usually affect the way resources relate to each other.

module

A collection of classes, resource types, files, and templates, organized around a particular purpose. For example, a module could be used to completely configure an Apache instance or to set-up a Rails application. There are many pre-built modules available for download in the Puppet Forge.

namevar

(or **name**)

The attribute that represents a resource's **unique identity** on the **target system.** For example: two different files cannot have the same `path`, and two different services cannot have the same `name`.

Every resource type has a designated namevar; usually it is simply `name`, but some types, like `file` or `exec`, have their own (e.g. `path` and `command`). If the namevar is something other than `name`, it will be called out in the type reference.

If you do not specify a value for a resource's namevar when you declare it, it will default to that resource's title.

node (definition)

(or **node statement**)

A collection of classes, resources, and variables in a manifest, which will only be applied to a certain agent node. Node definitions begin with the `node` keyword, and can match a node by full name or by regular expression.

When a managed node retrieves or compiles its catalog, it will receive the contents of a single matching node statement, as well as any classes or resources declared outside any node statement. The classes in every *other* node statement will be hidden from that node.

node scope

The local variable scope created by a node definition. Variables declared in this scope will override top-scope variables. (Note that ENCs assign variables at top scope, and do not introduce node scopes.)

noop

Noop mode (short for "No Operations" mode) lets you simulate your configuration without making any actual changes. Basically, noop allows you to do a dry run with all logging working normally, but with no effect on any hosts. To run in noop mode, execute `puppet agent` or `puppet apply` with the `--noop` option.

notify

A notification relationship, set with the `notify` metaparameter or the wavy chaining arrow. (`~>`)

notification

A type of relationship that both declares an order for resources and causes refresh events to be sent.

ordering

Which resources should be managed before which others.

By default, the order of a manifest is not the order in which resources are managed. You must declare a relationship if a resource depends on other resources.

parameter

Generally speaking, a parameter is a chunk of information that a class or resource can accept.

pattern

A colloquial term, describing a collection of related manifests meant to solve an issue or manage a particular configuration item. (For example, an Apache pattern.) See also module.

plusignment operator

The `+>` operator, which allows you to add values to resource attributes using the ('plusignment') syntax. Useful when you want to override resource attributes without having to respecify already declared values.

provider

Providers implement resource types on a specific type of system, using the system's own tools. The division between types and providers allows a single resource type `package` to manage packages on many different systems (using, for example, `yum` on Red Hat systems, `dpkg` and `apt` on Debian-based systems, and `ports` on BSD systems).

Typically, providers are simple Ruby wrappers around shell commands, so they are usually short and easy to create.

plugin

A custom type, function, or fact that extends Puppet's capabilities and is distributed via a module.

realize

To specify that a virtual resource should actually be applied to the current system. Once a virtual resource has been declared, there are two methods for realizing it:

1. Use the "spaceship" syntax `<|`|`>`
2. Use the `realize` function

A virtually declared resource will be present in the catalog, but will not be applied to a system until it is realized.

refresh

A resource gets **refreshed** when a resource it subscribes to (or which notifies it) is changed.

Different resource types do different things when they get refreshed. (Services restart; mount points unmount and remount; execs usually do nothing, but will fire if the `refreshonly` attribute is set.)

relationship

A rule stating that one resource should be managed before another.

resource

A unit of configuration, whose state can be managed by Puppet. Every resource has a type (such as `file`, `service`, or `user`), a title, and one or more attributes with specified values (for example, an `ensure` attribute with a value of `present`).

Resources can be large or small, simple or complex, and they do not always directly map to simple details on the client -- they might sometimes involve spreading information across multiple files, or even involve modifying devices. For example, a `service` resource only models a single service, but may involve executing an init script, running an external command to check its status, and modifying the system's run level configuration.

resource declaration

A fragment of Puppet code that details the desired state of a resource and instructs Puppet to manage it. This term helps to differentiate between the literal resource on disk and the specification for how to manage that resource. However, most often, these are just referred to as "resources."

scope

The area of code where a variable has a given value.

Class definitions and type definitions create local scopes. Variables declared in a local scope are available by their short name (e.g. `$my_variable`) inside the scope, but are hidden from other scopes unless you refer to them by their fully qualified name (e.g. `$my_class::my_variable`).

Variables outside any definition (or set by an ENC) exist at a special "top scope;" they are available everywhere by their short names (e.g. `$my_variable`), but can be overridden in a local scope if that scope has a variable of the same name.

Node definitions create a special "node scope." Variables in this scope are also available everywhere by their short names, and can override top-scope variables.

> Note:
>
> Previously, Puppet used dynamic scope, which would search for short-named variables through a long chain of parent scopes. This was deprecated in version 2.7 and will be removed in the next version.

site

An entire IT ecosystem being managed by Puppet. That is, a site includes all puppet master servers, all agent nodes, and all independent masterless Puppet nodes within an organization.

site manifest

The main "point of entry" manifest used by the puppet master when compiling a catalog. The location of this manifest is set with the `manifest` setting in puppet.conf. Its default value is usually `/etc/puppet/manifests/site.pp` or `/etc/puppetlabs/puppet/environments/production/manifests/site.pp`.

The site manifest usually contains node definitions. When an ENC is being used, the site manifest may be nearly empty, depending on whether the ENC was designed to have complete or partial node information.

site module

A common idiom in which one or more modules contain classes specific to a given Puppet site. These classes usually describe complete configurations for a specific system or a given group of systems. For example, the `site::db_slave` class might describe the entire configuration of a database server, and a new database server could be configured simply by applying that class to it.

subclass

A class that inherits from another class. See inheritance.

subscribe

A notification relationship, set with the `subscribe` metaparameter or the wavy chaining arrow. (`~>`)

template

A partial document which is filled in with data from variables. Puppet can use Ruby ERB templates to generate configuration files tailored to an individual system.

title

The unique identifier (in a given Puppet catalog) of a resource or class.

- In a class, the title is simply the name of the class.

- In a resource declaration, the title is the part after the first curly brace
- and before the colon; in the example below, the title is `/etc/passwd`:

```
file  { '/etc/passwd':
          owner => 'root',
          group => 'root',
        }
```

- In native resource types, the name or namevar will use the title
- as its default value if you don't explicitly specify a name.
- In a defined resource type or a class, the title is available for use
- throughout the definition as the `$title` variable.

Unlike the name or namevar, a resource's title need not map to any actual attribute of the target system; it is only a referent. This means you can give a resource a single title even if its name has to vary across different kinds of system, like a configuration file whose location differs on Solaris.

top scope

See scope.

type

A kind of resource that Puppet is able to manage; for example, `file`, `cron`, and `service` are all resource types. A type specifies the set of attributes that a resource of that type may use, and models the behavior of that kind of resource on the target system. You can declare many resources of a given type.

type (defined)

(or **defined resource type;** sometimes called a **define** or **definition**)

A resource type implemented as a group of other resources, written in the Puppet language and saved in a manifest. (For example, a defined type could use a combination of `file` and `exec` resources to set up and populate a Git repository.) Once a type is defined, new resources of that type can be declared just like any native or custom resource.

Since defined types are written in the Puppet language instead of as Ruby plugins, they are analogous to macros in other languages. Contrast with native types.

type (native)

A resource type written in Ruby. Puppet ships with a large set of built-in native types, and custom native types can be distributed as plugins in modules. See the type reference for a complete list of built-in types.

Native types have lower-level access to the target system than defined types, and can directly use the system's own tools to make changes. Most native types have one or more providers, so that they can implement the same resources on different kinds of systems.

variable

A named placeholder in a manifest that represents a value. Variables in Puppet are similar to variables in other programming languages, and are indicated with a dollar sign (e.g. `$operatingsystem`) and assigned with the equals sign (e.g. `$myvariable = "something"`). Once assigned, variables cannot be reassigned within the same scope; however, other local scopes can assign their own value to any variable name.

Facts from agent nodes are represented as variables within Puppet manifests, and are automatically pre-assigned before compilation begins.

variable scoping

See scope above.

virtual resource

A resource that is declared in the catalog but will not be applied to a system unless it is explicitly realized.