

Learn Puppet:

Quest Guide for the Learning VM

Table of Contents:

▪ Learning VM Setup	1
▪ Welcome	5
▪ The Power of Puppet	10
▪ Resources	21
▪ Manifests	28
▪ Variables	33
▪ Conditional Statements	40
▪ Resource Ordering	47
▪ Classes	53
▪ Modules	60
▪ The Forge and the Puppet Module Tool	67
▪ Afterword	71
▪ Troubleshooting	72
▪ Glossary of Puppet Vocabulary	73

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 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 to 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, in 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 necessary 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 `Factor` tool bundled with Puppet Enterprise tool to find it.

```
factor 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 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?

So what is Puppet, and why should you take the time to learn it?

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 keeping 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 let you get the setup you're looking for without churning out the code yourself.

And what's the 'Enterprise' part?

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:

3.4.3 (Puppet Enterprise 3.2.2)

This indicates that Puppet Version 3.4.3 Puppet Enterprise 3.2.2 are installed.

But why learn something new?

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.

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 yourself.

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, instead append your changes.

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.

```
[root@learn ~]# quest --help Command Line Input
quest: learning progress feedback tool
Usage:
quest [--option] (brief)
where [--option] is one of:
  --progress, -p:  Display details of tasks completed
  --completed, -c: Display completed quests
  --list, -l:      Show all available quests
  --start, -s <s>: Provide name of the quest to track
  --help, -h:      Show this message

[0] 0:bash*
Current Quest: Resources - Progress: 0/6 Tasks.
```

Output

Real-time Feedback displaying progress - updated every two seconds

Figure 1

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.

Welcome

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 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. There's a lot to learn if you want to use it to its full potential. 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 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 use 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/modules/`.

Great job! You've just installed your first module from the Forge.

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 always 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.

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://<ip-address>`, where `<ip-address>` is the Learning VM's IP address. (Be sure to include the `s` in `https`)

If your browser gives you a security notice because the certificate is self-signed, go ahead and click accept to continue to the PE Console.

When prompted, use the following credentials to log in:



Despite some superficial similarities, Puppet's classes aren't like the classes in Object Oriented programming. You'll just get confused if you think of them this way!



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

Username: puppet@example.com

Password: learningpuppet

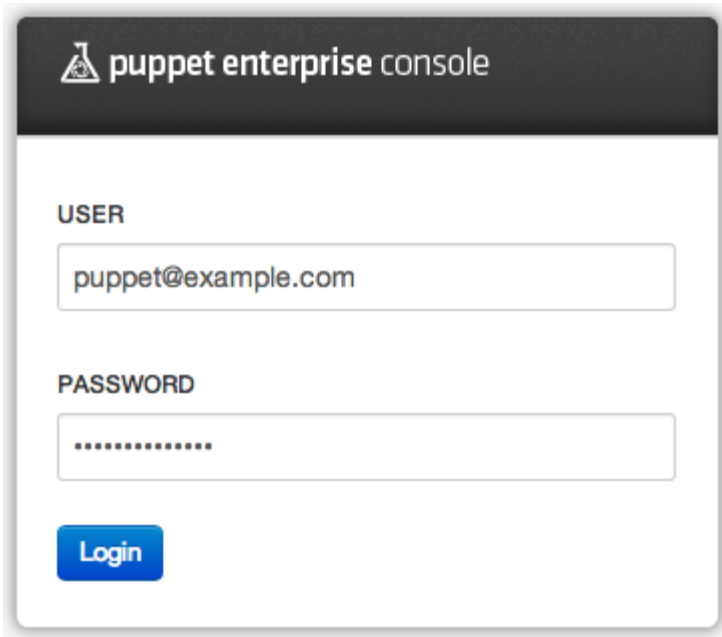
The image shows a login form for the Puppet Enterprise Console. At the top, there is a dark header bar with the Puppet logo (a triangle with a circle inside) and the text "puppet enterprise console". Below the header, the form has two sections: "USER" and "PASSWORD". The "USER" section contains a text input field with the value "puppet@example.com". The "PASSWORD" section contains a text input field with masked characters (dots). Below these fields is a blue "Login" button.

Figure 1

You're in! Now that you have access to the PE Console, we'll go over the steps you'll take to classify the "learning.puppetlabs.vm" node (i.e. the Learning VM) with the `lvmguide` class.

Add a Class

First, you need to add the `lvmguide` class to the list of classes available to the PE Console.

To do this, click the **Add classes** button on the **Classes** panel. (You may need to scroll to the bottom of the page to find the panel.)

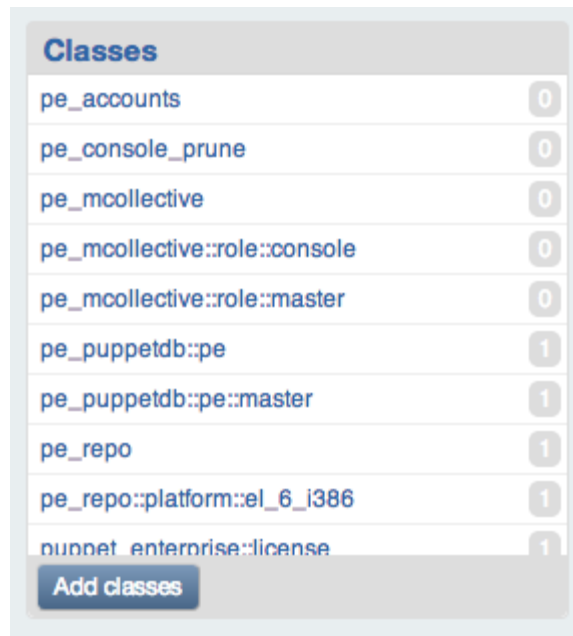


Figure 2

Type *lvmguid*e in the "Filter the list below" input box, and check the checkbox by the *lvmguid*e class that appears in the list.

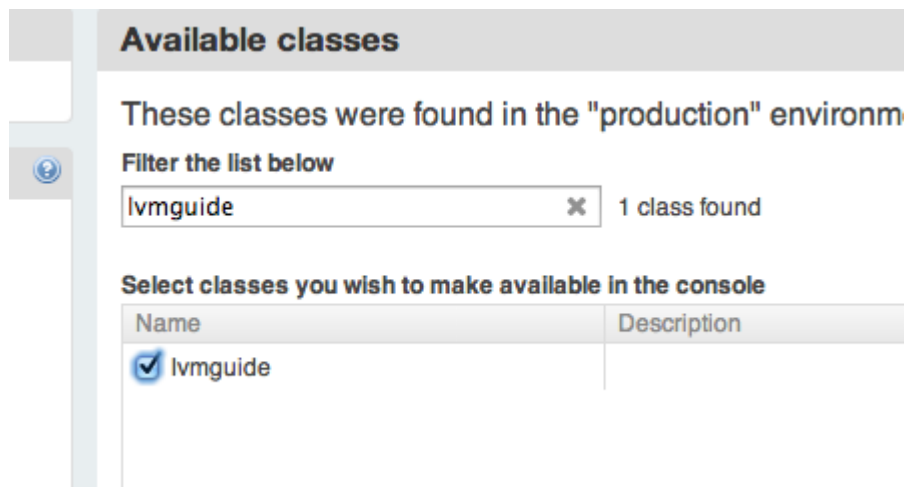


Figure 3

Now click the "Add selected classes" button.

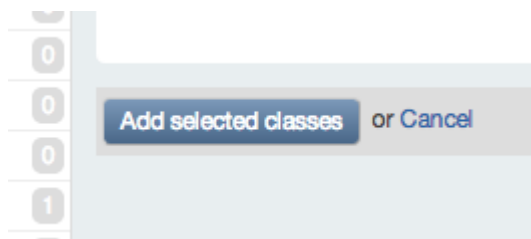


Figure 4

You should see a verification message at the top of the PE Console and the `lvmguide` class will appear in the list of available classes on the left side of the console interface.

Classify a Node

Now that the `lvmguide` class is available, you can use it to classify the node `learning.puppetlabs.vm`.

Click on the "Nodes" menu item in the navigation menu. (You may need to scroll to the top of the page to see the navigation menu.)

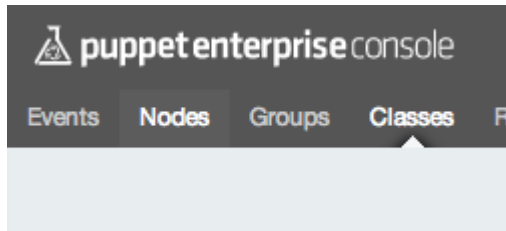


Figure 5

Click on the `learning.puppetlabs.vm` node hyperlink. (This should be the only one listed since the Learning VM is the only node you're managing with Puppet Enterprise.)

Nodes

Node			↓
Total			
✓	learning.puppetlabs.vm		20

Figure 6

Once you're on the node page for *learning.puppetlabs.vm*, click the "Edit" button located in the top-right corner of the screen.

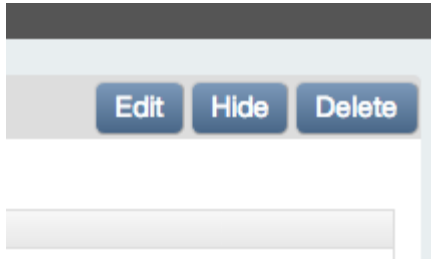


Figure 7

In the Classes section, type *lvmgulde* in the "add a class" input box

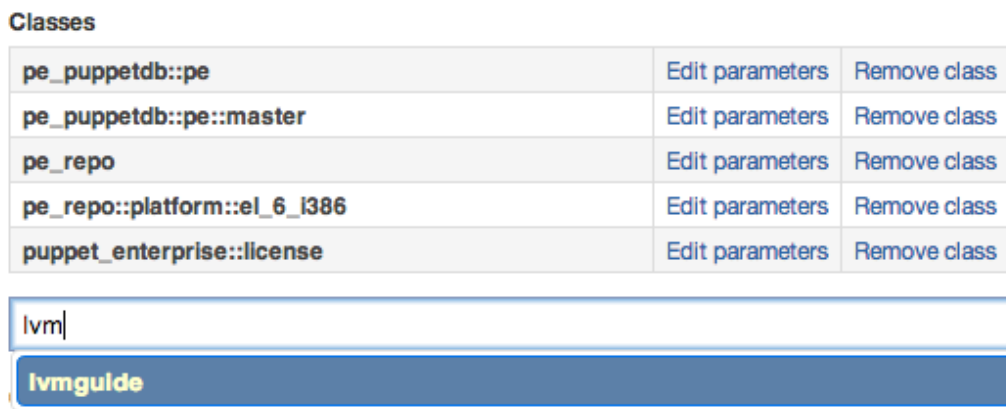


Figure 8

Click the "Update" button at the bottom.

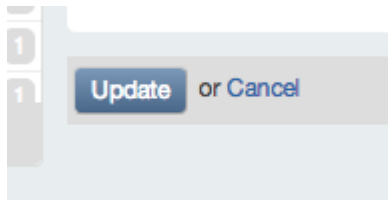


Figure 9

Excellent! If everything went according to plan, you've successfully classified the *learning.puppetlabs.vm* node with the *lvmgulde* class.

Run Puppet

Now that you have classified the *learning.puppetlabs.vm* node with the *lvmgulde* 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 your terminal, type the following command:

```
puppet agent --test
```

Please note this may take about 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! In your browsers address bar, type the following URL: `http://<ip-address>`. 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 after a while, 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/modules
```

Next, open the `init.pp` manifest.

```
nano 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, use the command `control-x` in nano, or `:q!` in vim.)

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 later on.

Class Title and Parameters:

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

The class `lvmguid` takes two parameters: `$document_root` and `$port`. The default values are set as `/var/www/html/lvmguide` and `80`.

Include the apache module's apache class:

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

The `lvmguid` 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 once you have a basic understanding of the syntax, a quick read-through will be enough to get the gist of well-written Puppet code. (We often talk about Puppet's DSL as self-documenting code.)

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://<your-vm's-ip-address>/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)
- Learn about the Resource Abstraction Layer (RAL)
- Use Puppet to inspect resources on your system

Getting Started

In this quest, you will be introduced to Resources and how system configurations are represented using Resource definitions. You will learn how to inspect resources on the Learning VM using Puppet. Please note though, that we are not going to use Puppet to manage any resources. Instead, we are going to use basic Unix commands in this quest, and then look at how the resultant resource changes are represented in Puppet's Domain Specific Language (DSL). As an aspiring practitioner of Puppet, it is important for you to have a thorough understanding of the Puppet syntax as well as the `puppet resource` and `puppet describe` tools. When you're ready to get started, type the following command:

```
quest --start resources
```

Resources

Resources are the fundamental units for modeling system configurations. Each resource describes some aspect of a system, like a service that must be running or a package that must be installed. The block of Puppet code that describes a resource is called a **resource declaration**. Resource declarations are written in Puppet's own Domain Specific Language.

Puppet's Domain Specific Language

Puppet uses its own configuration language, one that was designed to be accessible and does not require much formal programming experience. The code you see below is an example of what we're referring to. Since it is a **declarative** language, the definitions of resources can be considered as *models* of the state of resources.

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

The Trailing Comma

Though a comma isn't strictly necessary at the end of the final attribute value pair, it is best practice to include it for the sake of consistency.

You will not be using resource declarations to shape your environment just yet. Instead, you will exercise your power by hand and use Puppet only to inspect your actions using the `puppet resource` and `puppet describe` tools.

Anatomy of a Resource

Resources can be large or small, simple or complex. In the world of Puppet, you and everything around you (on the Learning VM) are resources. But let's say you wanted to learn more about a particular resource. How would one do that? Well, you have two options: `puppet describe` and `puppet resource`.

🔧 Task 1:

Let's say you want to learn more about the `user` resource type as it applies to all users in the Learning VM. Type the following command:

```
puppet describe user
```

The `puppet describe` command can list info about the currently installed resource types on a given machine.

🔧 Task 2:

Great! But how would one look at a specific resource? Well, to check and see how you look in the world of Puppet, type the following command :

```
puppet resource user root
```

The block of code below that describes you as the root user is called a **resource declaration**. It's a little abstract, but a nice portrait, don't you think?

```
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',  
}
```

The `puppet resource` can interactively inspect and modify resources on a single system and can also be useful for one-off jobs. However, Puppet was born for greater things which we'll discuss further in the Manifest Quest.

Resource Type

Let's take a look at your first line in the above resource declaration. Do you see the word `user`? It's right *before* the curly brace. This is called the **resource type**. Just as any individual cat or dog is a member of its species (*Felis catus* and *Canus lupis familiaris* to be precise) any instance of a resource must be a member of a resource type. Think of this type as a framework that defines the range of characteristics an individual resource can have.

Puppet allows you to describe and manipulate a variety of resource types. Below are some core resource types you will 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

...Wait, There's More!

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

Again, let's take a look at your first line in the above resource declaration. Do you see the single quoted word `'root'`? It's right *after* the curly brace. This is called the **title**. The title of a resource is used to identify it and **must** be unique. No two resources of the same type can share the same title. Also, don't forget to always add a colon (:) after the title. That's important to remember and often overlooked!

Attribute Value Pairs

One more time. Let's look at the resource declaration for user `root` listed above. After the colon (:) comes a list of **attributes** and their corresponding **values**. Each line consists of an attribute name, a `=>` (which we call a hash rocket), a value, and a final comma. For example, the attribute value pair `home => '/root'`, indicates that your home is set to the directory `/root`.

Task 3 :

The path to greatness is a lonely one. Fortunately, your superuser status gives you the ability to create a sidekick for yourself. First let's do this in a non-Puppet way. Type the following command:

```
useradd byte
```

Task 4 :

Now take a look at user byte using the `puppet resource` tool. Type the following command:

```
puppet resource user byte
```

Potent stuff. Note that byte's password attribute is set to `'!!!'`. This isn't a proper password at all! In fact, it's a special value indicating byte has no password whatsoever.

Task 5 :

Let's rectify byte's password situation by setting it to *puppetlabs*. Type the following command:

```
passwd byte
```

Now set the password to *puppetlabs* by typing it in and pressing Enter (Return) twice. You will not see anything displayed as you type the password.

Now if you take another look at byte using `puppet resource`, the value for byte's password attribute should now be set to a SHA1 hash of the password, something a little like this: `'1hNahKZqJ$9uL/RR2U.9ITZlKcMb0qJ.'`

Task 6 :

Now have a look at byte's home directory, which was set to `'/home/byte'` by default. Directories are a special kind of file, and so Puppet knows of them as File resources. The `title` of any file is, by default, the same as the path to that file. Let's find out more about the `tools` directory where our sidekick can store tools. Enter the command:

```
puppet resource file /home/byte/tools
```

Task 7 :

What? `ensure => 'absent'`, ? Values of the `ensure` attribute indicate the basic state of a resource. A value of `absent` means something doesn't exist at all. We need to make a directory for byte to store tools in:

```
mkdir /home/byte/tools
```

Now have another look at byte's tools directory:

```
puppet resource file /home/byte/tools
```

Quest Progress

Awesome! Have you noticed when you successfully finish a task, the 'completed tasks' in the lower right corner of your terminal increases? To check on your progress type the following command:

```
quest --progress
```

This shows your progress by displaying the tasks you've completed and tasks that still need completing.

Task 8 :

We want byte to be the owner of the tools directory. To do this, type the following commands:

```
chown -R byte:byte /home/byte/tools
```

Inspect the state of the directory one more time, to make sure everything is in order:

```
puppet resource file /home/byte/tools
```

The Resource Abstraction Layer

By now, we have seen some examples of how Puppet 'sees' resources on the system. A common pattern you might observe is that these are descriptions of *how* the resource in question should or does look. In subsequent quests, we will see how, instead of just inspecting resources, we can *declare* how specific resources *should look*, providing us the ability to model the state of these resources.

Puppet provides us this ability to describe resources of different types. Our job of defining how a system should be configured is reduced to one of creating a *high-level model* of the *desired state* of the system. We don't need to worry about how that model is realized.

Puppet takes the descriptions expressed by resource declarations and uses **providers** that are 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, this system of resource types and

the providers that implement them is called the **Resource Abstraction Layer**, or **RAL**.

You can describe the ideal state of a user resource. Puppet will choose a suitable provider to realize your definition - in the case of users, Puppet can use providers to manage user records in `/etc/passwd` files or `NetInfo`, or `LDAP`. Similarly, when you wish to install a package, you can stand back and watch Puppet figure out whether to use `yum` or `apt` for package management. This lets you ignore the implementation-related details of managing the resources, such as the names of the commands (is it `adduser` or `useradd`?), the arguments for the commands, file formats etc and lets you focus on the more important job of modeling the desired state for your systems.

By harnessing the power of the RAL, you can be confident of the potency of your Puppet skills wherever your journey takes you.

Review

Let's rehash what we learned in this quest. First, we learned two very important Puppet topics: the Resource Abstraction Layer and the anatomy of a resource. To dive deeper into these two important topics, we showed you how to use the `puppet describe` and `puppet resource` tools, which also leads us to a better understanding of Puppet's Language. These tools will be tremendously useful to you in the succeeding quests. Unfortunately we didn't get to write any Puppet code in this quest, but that's okay. We're going to start doing that in the Manifest Quest (the next quest)!

Manifests

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

Getting Started

As you saw in the Resources Quest, Puppet's resource declarations can be used to keep track of just about anything in this Learning VM. So far, you have made changes to the Learning VM without using Puppet. You looked at resource declarations using `puppet describe` and `puppet resource` only in order to track your effects. In this quest, you will learn to craft your own resource declarations and inscribe them in a special file called a **manifest**. When you're ready to get started, type the following command:

```
quest --start manifest
```

Let's See What Quests You've Completed

Before you get started, let's take a look at the quests you have completed. Type the following command:

```
quest --completed
```

This is useful in case you forget which Quests you've already done. In this case it shows that you have completed three quests: (1)Welcome Quest (2)Power of Puppet Quest and (3)Resources Quest.

Puppet Manifests

Manifests are files containing Puppet code. They are standard text files saved with the `.pp` file extension. The core of the Puppet language is the resource declaration as it describes a desired state for one resource. Puppet manifests contain resource declarations. Manifests, like the resource declarations they contain, are written in Puppet Language.

Don't Forget These Tools Too

You can use `puppet describe` and `puppet resource` for help using and understanding the `user` resource...and any other resource type you're curious about.

Let's get started by making sure you're in your home directory: `/root`. This is where you want to place newly created manifests.

```
cd /root
```

Text Editors

For the sake of simplicity and consistency, we use the text editor `nano` in our instructions, but feel free to use `vim` or another text editor of your choice.

Task 1:

Create a manifest to remove user byte:

Unfortunately byte just doesn't seem to be working out as a sidekick. Let's create a manifest to get rid of byte. We will create a manifest, with some code in it. Type the following command, after you make sure you are in the `/root` directory as mentioned above:

```
nano byte.pp
```

Type the following instructions into Byte's manifest:

```
user { 'byte':  
  ensure => 'absent',  
}
```

Save the file and exit your text editor. We touched on this in the Resources Quests, but the `ensure => absent` attribute/value pair states that we are going to make sure user byte does not exist in the Learning VM.

Puppet Parser

What if we made an error when writing our Puppet code? The `puppet parser` tool is Puppet's version of a syntax checker. When provided with a file as an argument, this tool validates the syntax of the code in the file without acting on the definitions and declarations within. If no manifest files are provided, Puppet will validate the default `site.pp` manifest. If there are no syntax errors, Puppet will return nothing when this command is ran, otherwise Puppet will display the first syntax error encountered.

Task 2 :

Using the `puppet parser` tool, let's you check your manifest for any syntax errors. Type the following command:

```
puppet parser validate byte.pp
```

Again, if the parser returns nothing, continue on. If not, make the necessary changes and re-validate until the syntax checks out.



The 'puppet parser' tool can only ensure that the syntax of a manifest is well-formed. It cannot guarantee that your variables are correctly named, your logic is valid, or that your manifest does what you want it to.

Puppet Apply

Once you've created a manifest you will use the `puppet apply` tool to enforce it. The `puppet apply` tool enables you to apply individual manifests locally. In the real world, you may want an easier method to apply multiple definitions across multiple systems from a central source. We will get there when we talk about classes and modules in succeeding quests. For now, manifests and `puppet apply` aid in learning the Puppet language in small, iterative steps.

When you run `puppet apply` with a manifest file as the argument, a *catalog* is generated containing a list of all resources in the manifest, along with the desired state you specified. Puppet will check each resource in your environment against the resource declaration in the manifest. Puppet's **providers** will then do

everything necessary to bring the state of those resources in line with the resource declarations in your manifest.

Task 3 :

Once your `byte.pp` manifest is error free, we're going to simulate the change in the Learning VM without actually enforcing those changes. Let's see what happens:

```
puppet apply --noop byte.pp
```

In the returned output, Puppet tells us that it has not made the changes to the Learning VM, but if it had, this is what would be changed.

Task 4 :

Since the simulated change is what we want, let's enforce the change on the Learning VM.

```
puppet apply byte.pp
```

How is byte doing?

```
HINT: Use the puppet resource command discussed in the Resource Quest.
```

byte does not seem to be doing well. Actually, the user's gone. The `ensure => 'absent'` value clearly made short work of the user account.

Task 5 :

With Puppet manifests you can create as well as destroy. Lets create a new, stronger sidekick by adding user `gigabyte` to the Learning VM using Puppet. If you need help on how to do this, refer to the previous tasks you've just completed in this quest. One thing to note: `ensure => 'present'` will make sure GigaByte exists in the Learning VM.

The steps include creating a manifest file, and writing the minimal amount of Puppet code required to ensure that the user account is created. This task will be marked complete when the user exists on the system.

Review

This is a foundational quest you must understand in order to successfully use Puppet. As you saw when completing this quest, we've added two new tools to your toolbox: `puppet parser` and `puppet apply`. You always want to use `puppet parser` to check the syntax of your manifest before using `puppet apply` to enforce it. This quest contained a walkthrough of the "best practice" methods to creating, checking, applying your manifest. We've also created a simplified version below for your reference:

1. Open or create a manifest with the `.pp` extension
2. Add or edit your Puppet code
3. Use the `puppet parser` tool to check for syntax errors (*recommended*)
4. Simulate your manifest using `puppet apply --noop` (*recommended*)
5. Enfore your manifest using `puppet apply`
6. Check to make sure everything is working correctly (*recommended*)

On a final note, if you go back to the Power of Puppet quest, you will notice that the `init.pp` file containing the definition for `class lvmguide` is a manifest.

Variables

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest

Quest Objectives

- Learn how to make Puppet manifests more flexible using variables
- Learn how to interpolate variables in manifests
- Understand how facts can be used

Getting Started

Manifests contain instructions for automating tasks related to managing resources. Now it's time to learn how to make manifests more flexible. In this quest you will learn how to include variables, interpolate variables, and use Facter facts in your manifests in order to increase their portability and flexibility. When you're ready to get started, type the following command:

```
quest --start variables
```

Variables



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

-Confucius

Puppet can be used to manage configurations on a variety of different operating systems and platforms. The ability to write portable code that accomodates various platforms is a significant advantage in using Puppet. It is important that you learn to write manifests in such a way that they can function in different

contexts. Effective use of **variables** is one of the fundamental methods you will use to achieve this.

If you've used variables before in some other programming or scripting language, the concept should be familiar. Variables allow you to assign data to a variable name in your manifest and then use that name to reference that data elsewhere in your manifest. In Puppet's syntax, variable names are prefixed with a `$` (dollar sign). You can assign data to a variable name with the `=` operator. You can also use variables as the value for any resource attribute, or as the title of a resource. In short, once you have defined a variable with a value you can use it anywhere you would have used the value or data.

The following is a simple example of assigning a value, which in this case, is a string, to a variable.

```
$myvariable = "look, data!\n"
```

Also...

In addition to directly assigning data to a variable, you can assign the result of any expression or function that resolves to a normal data type to a variable. This variable will then refer to the result of that statement.



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.

Task 1:

Using Puppet, create the file `/root/pangrams/fox.txt` with the specified content.

Create a new manifest in your home directory.

```
nano ~/pangrams.pp
```

HINT: Refer to the Manifest Quest if you're stuck.

Type the following Puppet code into the `pangrams.pp` manifest:

```
$pangram = 'The quick brown fox jumps over the lazy dog.'

file {'/root/pangrams':
  ensure => directory,
}

file {'/root/pangrams/fox.txt':
  ensure => file,
  content => $pangram,
}
```

Now that we have a manifest, let's test it on the VM.

Remember to validate the syntax of the file, and to simulate the change using the `-noop` flag before you use `puppet apply` to make the required change on the system.

Excellent! Take a look at the file to see that the contents have been set as you intended:

```
cat /root/pangrams/fox.txt
```

The file resource `/root/pangrams/fox.txt` is managed, and the content for the file is specified as the value of the `$pangram` variable.

Variable Interpolation

The extra effort required to assign variables starts to show its value when you begin to incorporate variables into your manifests in more complex ways.

Variable interpolation allows you to replace occurrences of the variable with the *value* of the variable. In practice this helps with creating a string, the content of which contains another string which is stored in a variable. To interpolate a variable in a string, the variable name is preceded by a `$` and wrapped in curly braces (`${var_name}`).

The braces allow `puppet parser` to distinguish between the variable and the string in which it is embedded. It is important to remember, 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.

```
"Variable interpolation is ${adjective}!"
```



A pangram is a sentence that uses every letter of the alphabet. A perfect pangram uses each letter only once.

 Task 2:

Create a file called `perfect_pangrams`. We will use variable substitution and interpolation in doing this.

Now you can use variable interpolation to do something more interesting. Go ahead and create a new manifest called `perfect_pangrams.pp`.

```
nano ~/perfect_pangrams.pp
```

HINT: Refer to the Manifest Quest if you're stuck



Wrapping a string without any interpolated variables in double quotes will still work, but it goes against conventions described in the Puppet Labs Style Guide.

Type the following Puppet code into the `perfect_pangrams.pp` manifest:

```
$perfect_pangram = 'Bortz waqf glyphs vex muck djin.'

$pgdir = '/root/pangrams'

file { $pgdir:
  ensure => directory,
}

file { "${pgdir}/perfect_pangrams":
  ensure => directory,
}

file { "${pgdir}/perfect_pangrams/bortz.txt":
  ensure => file,
  content => "A perfect pangram: \n${perfect_pangram}",
}
```

Once you have create the `perfect_pangrams.pp` file, enforce it using the appropriate `puppet apply` command, but not before you verify that the syntax is correct and have tried simulating it first. Refer to the Manifests quest if you need to refresh you memory on how to apply a manifest.

Here, the `$pgdir` variable resolves to `'/root/pangrams'`, and the interpolated string `"${pgdir}/perfect_pangrams"` resolves to `'/root/pangrams/perfect_pangrams'`. It is best to use variables in this way to avoid redundancy and allow for data separation in the directory and filepaths. If you wanted to work in another user's home directory, for example, you would only have to change the `$pgdir` variable, and would not need to change any of your resource declarations.

Have a look at the `bortz.txt` file:

```
cat /root/pangrams/perfect_pangrams/bortz.txt
```

You should see something like this, with your pangram variable inserted into the file's content string:

```
A perfect pangram:  
Bortz waqf glyphs vex muck djin.
```

What this perfect pangram actually means, however, is outside the scope of this lesson!

Facts



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

Puppet has a bunch of built-in, pre-assigned variables that you can use. Remember using the `Facter` tool when you first started? The `Facter` tool discovers information about your system and makes it available to Puppet as variables. Puppet's compiler accesses those facts when it's reading a manifest.

Remember running `facter ipaddress` told you your IP address? What if you wanted to turn `facter ipaddress` into a variable? It would look like this: `$::ipaddress` as a stand-alone variable, or like this: `${::ipaddress}` when interpolated in a string.

The `::` in the above indicates that we always want the top-scope variable, the global fact called `ipaddress`, as opposed to, say a variable called `ipaddress` you defined in a specific manifest.

In the [Conditions Quest](#), you will see how Puppet manifests can be designed to perform differently depending on facts available through `facter`. For now, let's play with some facts to get a feel for what's available.

Task 3 :

We will write a manifest that will interpolate `facter` variables into a string assigned to the `$message` variable. We can then use a `notify` resource to post a notification when the manifest is applied. We will also declare a file resource. We can use the same `$string` to assign our interpolated string to this file's content parameter.

Create a new manifest with your text editor.

```
nano ~/facts.pp
```

HINT: Refer to the Manifest Quest if you're stuck

Type the following Puppet code into the `facts.pp` manifest:

```
$string = "Hi, I'm a ${::osfamily} system and I have been up for  
${::uptime_seconds} seconds."  
  
notify { 'info':  
  message => $string,  
}  
  
file { '/root/message.txt':  
  ensure => file,  
  content => $string,  
}
```

Once you have created the `facts.pp` file, enforce it using the appropriate `puppet apply` command, after verifying that the syntax is correct.

You should see your message displayed along with Puppet's other notifications. You can also use the `cat` command or a text editor to have a look at the `message.txt` file with the same content.

```
cat /root/message.txt
```

As you can see, by incorporating facts and variables, and by using variable interpolation, you can add more functionality with more compact code. In the next quest we will discuss conditional statements that will provide for greater flexibility in using Puppet.

Review

In this quest you've learned how to take your Puppet manifests to the next level by using variables. There are even more levels to come, but this is a good start. We learned how to assign a value to a variable and then reference the variable by name whenever we need its content. We also learned how to interpolate variables, and how Facter facts are global variables available for you to use.

In addition to learning about variables, interpolating variables, and facts, you also gained more hands-on learning with constructing Puppet manifests using

Variables

Puppet's DSL. We hope you are becoming more familiar and confident with using and writing Puppet code as you are progressing.

Looking back to the Power of Puppet Quest, can you identify where and how variables are used in the `lvmguide` class?

Conditional Statements

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest
- Variables Quest

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. This, in conjunction with Facter facts, will enable you to write Puppet code that accomodates different platforms, operating systems, and functional requirements.

To start this quest enter the following command:

```
quest --start conditionals
```

Conditions

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

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data pertaining to the systems. This enables you to write code to perform as desired on

a variety of operating systems and under differing system conditions. Pretty neat, don't you think?

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

The following is an example of an `if` statement you might use to raise a warning when a class is included on an unsupported system:

```
if $::is_virtual == 'true' {
  # Our NTP module is not supported on virtual machines:
  warning( 'Tried to include class ntp on virtual machine.' )
}
elsif $::operatingsystem == 'Darwin' {
  # Our NTP module is not supported on Darwin:
  warning( 'This NTP module does not yet work on Darwin.' )
}
else {
  # Normal node, include the class.
  include ntp
}
```

In addition to the `==` operator, which tests for equality, there is also a regular expression match operator `=~`. The `==` operator is not case sensitive. In the above example, if you had:

```
if $::is_virtual == 'TRUE' {  
  # Our NTP module is not supported on virtual machines:  
  warning( 'Tried to include class ntp on virtual machine.' )  
}  
elsif $::operatingsystem == 'darwin' {  
  # Our NTP module is not supported on Darwin:  
  warning( 'This NTP module does not yet work on Darwin.' )  
}  
else {  
  # Normal node, include the class.  
  include ntp  
}
```

... the behavior would remain unchanged.

The Warning Function

The `warning()` function will not affect the execution of the rest of the manifest, but if you were running Puppet in the usual Master-Agent setup, it would log a message on the server at the 'warn' level.

The regular expression operator `==~` helps you test whether a string matches a pattern you specify. For example, in the following, we capture the digits that follow `www` in the hostname, such as `www01` or `www12` and store them in the `$1` variable for use in the `notice()` function.

```
if $::hostname ==~ /^www(\d+)\./ {  
  notice("Welcome to web server number $1")  
}
```

Task 1:

Just as we have done in the Variables Quest, let's create a manifest and add a simple conditional statement. The file should report on how long the VM has been up and running.

```
nano ~/conditionals.pp
```

Enter the following code into your `conditionals.pp` manifest:

```

if $::uptime_hours < 2 {
    $myuptime = "Uptime is less than two hours.\n"
}
elsif $::uptime_hours < 5 {
    $myuptime = "Uptime is less than five hours.\n"
}
else {
    $myuptime = "Uptime is greater than four hours.\n"
}
file {'/root/conditionals.txt':
    ensure => present,
    content => $myuptime,
}

```

Use the `puppet parser` tool to check your syntax, then simulate the change in `--noop` mode without enforcing it. If the noop looks good, enforce the `conditionals.pp` manifest using the `puppet apply` tool.

Have a look at the `conditionals.txt` file using the `cat` command.

Task 2:

Use the command `factor uptime_hours` to check the uptime yourself. The notice you saw when you applied your manifest should describe the uptime returned from the Factor tool.

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.

Task 3:

Create a `case.pp` manifest with the following conditional statement and `file` resource declaration.

```
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") }
}

file {'/root/case.txt':
  ensure => present,
  content => "Apache package name: ${apache_pkg}\n"
}
```

When you've validated your syntax and run a `--noop`, apply the manifest:

```
puppet apply case.pp
```

Use the `cat` command to inspect the `case.txt` file. Because the Learning VM is running CentOS, you will see that the selected Apache package name is 'httpd'.

For the sake of simplicity, we've output the result of the case statement to a file, but keep in mind that instead of using the result of the case statement like the one above to define the contents of a file, you could use it as the title of a `package` resource declaration, as shown below:

```
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. It is a paradigm you will encounter frequently in published Puppet modules.

Also note that Puppet will choose the appropriate *provider* for the package depending on the operating system, without you having to mention it. On Debian-based systems, for example, it may use `apt` and on RedHat systems, it will use `yum`.

The 'selector' Statement

Selector statements are very 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.

Task 4:

By writing a Puppet manifest that uses a selector, create a file `/root/architecture.txt` that lists whether the VM is a 64-bit or a 32-bit machine.

To accomplish this, create a file in the root directory, called `architecture.pp`:

```
nano architecture.pp
```

We know that i386 machines have a 32-bit architecture, and x86_64 machines have a 64-bit architecture. Let's set the content of the file based on this fact:

```
file { ['/root/architecture.txt'] :
  ensure => file,
  content => $::architecture ? {
    'i386' => "This machine has a 32-bit architecture.\n",
    'x86_64' => "This machine has a 64-bit architecture.\n",
  }
}
```

See what we did here? Instead of having the selector return a value and saving it in a variable, as we did in the previous example with `$rootgroup`, we use it to specify the value of the `content` attribute in-line.

Once you have created the manifest, check the syntax and apply it.

Inspect the contents of the `/root/architecture.txt` file to ensure that the content is what you expect.

Before you move on

We have discussed some intense information in the Variables Quest and this Quest. The information contained in all the quests to this point has guided you towards creating flexible manifests. Should you not understand any of the topics previously discussed, we highly encourage you to revisit those quests before moving on to the Resource Ordering Quest.

Resource Ordering

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest
- Variables Quest
- Conditions Quest

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
```

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. We have referred to the catalog vaguely in the previous sections. The **catalog** is a compilation of all the resources that will be applied to a given system, and the relationships between those resources. In building the catalog, unless we *explicitly* specify the relationship between the resources, Puppet will manage them in its own order.

For the most part, Puppet specifies relationships between resources in the appropriate manner while building the catalog. For example, if you say that user `gigabyte` should exist, and the directory `/home/gigabyte/bin` should be present and be owned by user `gigabyte`, then Puppet will specify a relationship between the two - that the user should be managed before the directory. These are implicit (shall we call them obvious?) relationships.

Sometimes, however, you will need to ensure 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. One might ask as to why there is not an implicit relationship in this case. The answer is that, often times, 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 us to specify the relationship between them.

When you need a group of resources to be managed in a specific order, you must explicitly state the dependency relationships between these resources within the resource declarations.

Relationship Metaparameters

Metaparameters follow the familiar `attribute => value` syntax. 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. Since this is the first time we've mentioned arrays - here's an example of an array:

```
$my_first_array = ['one', 'two', 'three']
```



A metaparameter is a resource attribute that can be specified for `_any_` type of resource, rather than a specific type.

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.

Task 1:

Create a puppet manifest to manage the `/etc/ssh/sshd_config` file

Create the file `/root/sshd.pp` using a text editor, with the following content in it.

```
file { '/etc/ssh/sshd_config':  
  ensure => file,  
  mode   => 600,  
  source => '/root/examples/sshd_config',  
}
```

What we have done above is to say that Puppet should ensure that the file `/etc/ssh/sshd_config` exists, and that the contents of the file should be sourced from the file `/root/examples/sshd_config`. The `source` attribute also allows us to use a different URI to specify the file, something we will discuss 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 2 :

Disable GSSAPIAuthentication for the SSH service

Edit the `/root/examples/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:

```
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.

Now, let's apply the change. Remember to check syntax, and do a dry-run using the `--noop` flag first, before using `puppet apply /root/sshd.pp` to apply your changes.

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.pp` file and add the `openssh-server` package to it.

Task 3 :

Manage the package for the SSH server

Type the following code in above your file resource in file `/root/sshd.pp`

```
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.

Let's do a Quick 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.

Classes

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest
- Variables Quest
- Conditions Quest
- Ordering Quest

Quest Objectives

- 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

So we've mentioned the term *class* in previous quests. In this quest we cover the use of classes within a Puppet manifest to group resource declarations (and everything we've learned up to this point) into reusable blocks of Puppet code. When you're ready to get started, type the following command:

```
quest --start classes
```

This is just an example

We've written this quest to help you learn the functionality and purpose of classes. To keep things simple, we will write code to both define classes and include them within a single manifest. Keep in mind however, that in practice you will always define your classes in a separate manifest. In the Modules Quest we will show you the proper way to define classes and declare classes.

Defining Classes

In Puppet's language **classes** are named blocks of Puppet code. Once you have defined a class, you can invoke it by name. Puppet will manage all the resources that are contained in the class definition once the class is invoked. Please remember that classes in Puppet are not related to classes in Object Oriented Programming. In Puppet, classes serve as named containers for blocks of Puppet code.

Let's dive right in, and look at an example of a class definition. We have created a class definition for you. Look at the contents of the file `/root/examples/modules1-ntp1.pp`. Open it using `nano` or your favorite text editor.

The file should contain the following code:

```
class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file     = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file     = 'ntp.conf.debian'
    }
  }

  package { ['ntp']:
    ensure => installed,
  }

  file { ['ntp.conf']:
    path     => '/etc/ntp.conf',
    ensure   => file,
    require  => Package['ntp'],
    source   => "/root/examples/answers/${conf_file}"
  }

  service { ['ntp']:
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }
}
```

That's a class definition. As you can see, there is a `case` statement, and three resources, all contained within the following pair of curly braces:

```
class ntp {  
  
}
```

The conditional (case statement) sets up the value for the `$service_name` and `$conf_file` variables appropriately for a set of operating systems. Three resources - a package, a file, and a service are defined, which use the variables to provide flexibility.

Now what would happen if we applied this manifest, containing the class definition?

Task 1:

Apply the manifest containing the `ntp` class definition:

```
puppet apply /root/examples/modules1-ntp1.pp
```

That's funny. Nothing happened, and nothing changed on the system!

This is because the class in the `modules1-ntp1.pp` manifest is only being defined and not declared. When you applied the manifest, it is as if Puppet went, "Ok! Got it. When you ask for class `ntp`, I am to know that it refers to everything in the definition." You have to *declare* the class in order to make changes and manage the resources specified in the definition. Declared? What's that? We will discuss that next.

Declaring Classes

In the previous section, we saw an example of a class definition and learned that a class is a collection of resources. The question that still needs answering is, how can we use the class definition? How can we tell Puppet to use the definition as part of configuring a system?

The simplest way to direct Puppet to apply a class definition on a system is by using the `include` directive. For example, to invoke class `ntp` you would have to say:

```
include ntp
```

in a Puppet Manifest, and apply that manifest.

Now you might wonder how Puppet knows *where* to find the definition for the class. Fair question. The answer involves Modules, the subject of our next lesson. For now, since we want to try applying the definition for class `ntp`, let's put the line `include ntp` right after the class definition.

We have already done that for you, open the file `/root/examples/modules1-ntp2.pp`:

```
nano /root/examples/modules1-ntp2.pp
```

You should see the line:

```
include ntp
```

as the very last line of the file.

Task 2:

Declare class `ntp`

Go ahead and now apply the manifest `/root/examples/modules1-ntp2.pp`.

```
HINT: Use the puppet apply tool. Refer to the Manifests Quest.
```

Great! This time Puppet actually managed the resources in the definition of class `ntp`.

Again, please do not ever do this above example in real life, since you *always* want to separate the definition from the declaration. This is just an example to show you the functionality and benefit of classes. In the Modules Quest we will show you the proper way to define classes and declare classes separately.

A detailed look at the `lvmguide` class

In the Power of Puppet Quest, we used a class called `lvmguide` to help us set up the website version of this Quest Guide. The `lvmguide` class gives us a nice illustration of structuring a class definition. We've included the code from the `lvmguide` class declaration below for easy reference as we talk about defining classes. Don't worry if a few things remain unclear at this point. For now, we're going to focus primarily on how class definitions work.

```
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,
  }
}
```

In this example we've **defined** a class called `lvmguide`. The first line of the class definition begins with the word `class`, followed by the name of the class we're defining: in this case, `lvmguide`.

```
class lvmguide (
```

Notice that instead of the usual curly bracket, there is an open parenthesis at the end of this first line, and it isn't until after the closing parenthesis that we see the opening curly bracket.

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

The variable declarations contained in these parentheses are called **class parameters**.

Class parameters allow you to pass a set of parameters to a class. In this case, the parameters are `$document_root` and `$port`. The values assigned to these parameters in the class definition are the **default values**, which will be used whenever values for the parameters are not passed in.

The first item you see inside the curly braces is... another class! One of the advantages of keeping your classes modular is that you can easily pull together all the classes you need to achieve a particular purpose.

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

Notice how the code looks similar to how you might describe a user, file or package resource. It looks like a *declaration*. It is, indeed, a *class declaration*. This is an alternative to using `include` to invoke existing class definitions. In this case, we wanted to set up an apache server to host our Quest Guide content as a website. Instead of trying to reinvent the wheel, we are able to pull in the existing `apache` class from the `apache` module we downloaded from the Forge.

If we had wanted to include the `apache` class with its default parameter settings, we could have used the `include apache` syntax. Turns out that just like the `lvmguide` class, the `apache` class is defined to accept parameters. Since we wanted to set the `default_vhost` parameter, we used the resource-like class declaration syntax. This allows us to set `default_vhost` to `false`.

Our final code block in the class definition is a resource declarations:

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

First, we declare an `apache::vhost` resource type, and pass along values from our class parameters to its `port` and `docroot` attributes. The `apache::vhost` resource type is defined in, and provided by the `apache` module that we installed, and helps manage the configuration of Apache2 Virtual Hosts.

As in the above example, class definitions give you a concise way to group other classes and resource declarations into re-usable blocks of Puppet code. You can then selectively assign these classes to different machines across your Puppetized network in order to easily configure those machines to fulfill the defined function. Now that the `lvmguide` class is defined, enabling the Quest Guide website on other machines would be as easy as assigning that class in the PE Console.

Review

We learned about classes, and how to define them. We also learned two ways to invoke classes - using the `include` keyword, and declaring classes using a syntax

Classes

similar to resource declarations. Classes are a whole lot more useful once we understand what modules are, and we will learn about modules in the next quest.

Modules

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest
- Variables Quest
- Conditions Quest
- Ordering Quest
- Classes Quest

Quest Objectives

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

Getting Started

So far we've seen some simple examples of Puppet code, but what makes Puppet such an adaptable tool? When it comes to getting things done, it's all about the **module**. A Puppet module is a self-contained bundle of code and data, organized around a particular purpose. The code would encompass manifests, the data would include all the source files, templates etc that you need to accomplish the purpose of the module. Everything we have learned until this point and more can come together as part of a module. When you're ready, type the following command:

```
quest --start modules
```

What's a Module?

If *resources* and *classes* are the atoms and molecules of Puppet, we might consider *modules* our amoebas: the first self-contained organisms of the Puppet world.

Thus far, in order to gain insight into Puppet's language features and syntax, we have been writing one-off manifests, perhaps using a source file for the contents of some configuration file (as we did for the SSH daemon configuration) etc. We have to remember, however that Puppet is designed to help you manage *lots* of systems (not just one system) from a central point - the master.

Although it is possible to list all the resources (users, files, package, services, cron tasks etc) that you need to manage on a system in one huge manifest, and then apply that manifest on the system, this is not scalable. Neither is it composable, or flexible enough to be reused. Classes were our first stop on the path to learning how to create 'blocks' - building blocks that we can use to describe the configuration of the systems we seek to manage.

There is still a missing part of the puzzle. When you ask Puppet to, say, `include ssh` on a particular node, or, as we did in the Power of Puppet quest, *classify* a node (learn.localdomain) with a class (lvmguide), how does Puppet know *where* to find the definition for the class, in order to ensure that the specified configuration is realized?

The answer is, we agree to put the class definitions in a standard location on the file system, by placing the manifest containing the class definition in a specific directory in a module.

Simply put, a Module is a directory with a specific structure - a means for us to package everything needed to achieve a certain goal. Once we agree to stick to the standard way of doing this, a significant benefit is the ability to *share* our work, such that others who seek to achieve the same goal can re-use our work. The [Forge](#) is the central location where you can find modules that have been developed by others.

In our initial Quest, we were able to use an Apache module from the Forge. This let us easily install and configure an Apache webserver to host the website version of this Quest Guide. The vast majority of the necessary code had already been written, tested, documented, and published to the Puppet Forge.

Modules provide a structure to make these collections of pre-written Puppet code, well, *modular*. In order to enable Puppet to access the classes and types defined in a module's manifests, modules give us a standard pattern for the organization of and naming of Puppet manifests and other resources.

Module Path

All modules are located in a directory specified by the *modulepath* variable in Puppet's configuration file. On the Learning VM, Puppet's configuration file can be found at `/etc/puppetlabs/puppet/puppet.conf`.

Task 1:

Find the modulepath on the Learning VM.

If you're ever wondering where your modulepath is, you can find it by running the `puppet agent` command with the `--configprint` flag and specifying `modulepath`:

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

What the returned value tells us is that Puppet will look in the directories `/etc/puppetlabs/puppet/modules` and then in `/opt/puppet/share/puppet/modules` to find the modules in use on the system. Classes and types defined by modules in these directories will be available to Puppet.

Module Structure

The skeleton of a module is a pre-defined structure of directories that Puppet already knows how to traverse to find the module's manifests, templates, configuration files, plugins, and anything else necessary for the module to function. Of these, we have encountered manifests and files that serve as the source for configuration files. We will learn about the rest in due course.

Remember, `/etc/puppetlabs/puppet/modules` is in our modulepath. Use the `ls` command to see what's in that directory:

```
ls /etc/puppetlabs/puppet/modules
```

There's the `apache` module we installed before. There is also the `lvmguid` module that was used to set up the quest guide website, that was already in place when we started. Use the `tree` command to take a look at the basic directory structure of the module. (To get a nice picture, we can use a few flags with the `tree` command to limit our output to directories, and limit the depth to two directories.)

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

You should see a list of directories, like so:

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

Once you get down past this basic directory structure, however, the `apache` module begins to look quite complex. To keep things simple, we can create our own first module to work with.

Task 2 :

First, be sure to change your working directory to the modulepath. We need our module to be in this directory if we want Puppet to be able to find it.

```
cd /etc/puppetlabs/puppet/modules
```

You have created some users in the *Resources* and *Manifests* quests, so this resource type should be fairly familiar. Let's make a `users` module that will help us manage users on the Learning VM.

The top directory of any module will always be the name of that module. Use the `mkdir` command to create your module directory:

```
mkdir users
```

Task 3 :

Now we need two more directories, one for our manifests, which must be called `manifests`, and one for our tests, which must be called (you guessed it!) `tests`. As you will see shortly, tests allow you to easily apply and test classes defined in your module without having to deal with higher level configuration tasks like node classification.

Go ahead and use the `mkdir` command to create `users/manifests` and `users/tests` directories within your new module.

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

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

```
users
├─ manifests
└─ tests

2 directories, 0 files
```

Task 4 :

Create a manifest defining class users:

The manifests directory can contain any number of the `.pp` manifest files that form the bread-and-butter of your module. If there is a class with the same name as the module, the definition for that class should be in a file name `init.pp`. In our case, we need a class called `users` in our module with the same name. The definition for class users should be in a file called `init.pp` in the `manifests` directory. This is necessitated by the mechanism by which Puppet locates class definitions when you name a class. If for example, you had a file called `settings.pp` in the manifests directory, you will have to refer to it as class `users::settings` for Puppet to be able to find the class definition contained in it. By placing the definition for class `users` in `init.pp`, we can refer to that class definition by the name of the module - `users`.

Go ahead and create the `init.pp` manifest in the `users/manifests` directory. (We're assuming you're still working from the `/etc/puppetlabs/puppet/modules`. The full path would be `/etc/puppetlabs/puppet/modules/users/manifests.init.pp`)

```
nano users/manifests/init.pp
```

Now in that file, add the following code:

```
class users {
  user { 'alice':
    ensure => present,
  }
}
```

We have defined a class with just the one resource in it. A resource of type `user` with title `alice`.

Use the `puppet parser` tool to validate your manifest:

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

For now, we're not going to apply anything. This manifest *defines* the `users` class, but so far, we haven't *declared* it. That is, we've described what the `users` class is, but we haven't told Puppet to actually do anything with it.

Declaring Classes from Modules

Remember when we talked about *declaring* classes in the Classes Quest? We said we would discuss more on the correct way to use classes in the Modules Quest. Once a class is *defined* in a module, there are actually several ways to *declare* it. As you've already seen, you can declare classes by putting `include [class name]` in your main manifest, just as we did in the Classes Quest.

The `include` function declares a class, if it hasn't already been declared somewhere else. If a class has already been declared, `include` will notice that and do nothing.

This lets you safely declare a class in several places. If some class depends on something in another class, it can declare that class without worrying whether it's also being declared elsewhere.

Task 5:

Write a test for our new class:

In order to test our `users` class, we will create a new manifest in the `tests` directory that declares it. Create a manifest called `init.pp` in the `users/tests` directory.

```
nano users/tests/init.pp
```

All we're going to do here is *declare* our `users` class with the `include` directive.

```
include users
```

Try applying the new manifest with the `--noop` flag first. If everything looks good, apply the `users/tests/init.pp` manifest without the `--noop` flag to take your `users` class for a test drive, and see how it works out when applied.

Use the `puppet resource` tool to see if `user alice` has been successfully created.

So what happened here? Even though the `users` class was in a different manifest, we were able to *declare* our test manifest. Because our module is in Puppet's *modulepath*, Puppet was able to find the correct class and apply it.

You just created your first Puppet module!!

Classification

When you use a *Node Classifier*, such as the Puppet Enterprise Console, you can *classify* nodes - which is to say that you can state which classes apply to which nodes, using the node classifier. This is exactly what we did when we used the PE Console to classify our Learning VM node with the `lvmguide` class in the Power of Puppet quest. In order to be able to classify a node thus, you *must* ensure all of the following:

1. There is a module (a directory) with the same name as the class in the modulepath on the Puppet master
2. The module has a file called `init.pp` in the `manifests` directory contained within it
3. This `init.pp` file contains the definition of the class

With modules, you can create composable configurations for systems. For example, let's say that you have a module called 'ssh' which provides class `ssh`, another called 'apache' and a third called 'mysql'. Using these three modules, and the classes provided by them, you can now classify a node to be configured with any combination of the three classes. You can have a server that has mysql and ssh managed on it (a database server), another with apache and ssh managed on it (a webserver), and a server with only ssh configured on it. The possibilities are endless. With well-written, community-vetted, even Puppet Supported Modules from the Forge, you can be off composing and managing configuration for your systems in no time. You can also write your *own* modules that use classes from these Forge modules, as we did with the `lvmguide` class, and reuse them too.

Review

1. We identified what the features of a Puppet Module are, and understood how it is useful
2. We wrote our first module!
3. We learned how modules (and the classes within them) can be used to create composable configurations by using a node classifier such as the PE Console.

The Forge and the Puppet Module Tool

Prerequisites

- Welcome Quest
- Power of Puppet Quest
- Resources Quest
- Manifest Quest
- Variables Quest
- Conditions Quest
- Ordering Quest
- Classes Quest
- Modules Quest

Quest Objectives

- Confidently use the `puppet module` tool in association with Forge modules

Getting Started

In the previous Modules Quest we primarily learned about the structure of a module and how to create a module. Next we want to answer questions like:

- How do I install a module?
- How do I use a module?
- What about upgrading an existing module?
- How can I do all this and more from the command line?

To complete this quest, the Learning VM will need to be connected to the Internet.

When you're ready to take your Puppet knowledge to the next level, type the following command:

```
quest --start forge
```

The `puppet module` tool is one of the most important tools in expanding your use of Puppet. The `puppet module` tool allows you to create, install, search, (and so much more) for modules on the Forge. We'll also discuss the Puppet Forge in greater detail below.

The `puppet module` tool has subcommands that make finding, installing, and managing modules from the Forge much easier from the command line. It can also generate empty modules, and prepare locally developed modules for release on the Forge.

Actions

- `list` - List installed modules.
- `search` - Search the Puppet Forge for a module.
- `install` - Install a module from the Puppet Forge or a release archive.
- `upgrade` - Upgrade a puppet module.
- `uninstall` - Uninstall a puppet module.
- `build` - Build a module release package.
- `changes` - Show modified files of an installed module.
- `generate` - Generate boilerplate for a new module.

Task 1:

List all the installed modules

Let's see what modules are already installed on the Learning VM and where they're located. To do this, we want Puppet to show it to us in a tree format. Go ahead and type the following command:

```
puppet module list --tree
```

Task 2:

Search the forge for a module

Wow! you have a lot installed. That's great. Let's install one more - the `puppetlabs-mysql` module. You should search the Forge modules that help you configure mysql. To do so from the command line, type the following command:

```
puppet module search mysql
```

The results list all the modules that have `mysql` in their name or description. Modules on the Forge are always named as *authorname-modulename*. You will notice that one of the modules listed in the search results is `puppetlabs-mysql`. The `puppetlabs-mysql` module is a module authored and curated primarily by the puppetlabs organization.

Task 3 :

Install a module

Let's install the `puppetlabs-mysql` module. In fact, let us specify a specific version of the module. Run the following command:

```
puppet module install puppetlabs-mysql --version 2.2.2 --ignore-requirements
```

You will see that the module is downloaded and installed. With the `--ignore-requirements` flag, we specify that there is no need to verify that all of its requirements are met.

By default, modules are installed in the `modulepath`.

Task 4 :

Upgrade an installed module

Okay, now let's go ahead and upgrade the `mysql` module to the latest version:

```
puppet module upgrade puppetlabs-mysql
```

Great, if you needed to uninstall a module, you can do so by running the following command:

```
puppet module uninstall puppetlabs-mysql
```

The Puppet Forge

The [Puppet Forge](#) is a public repository of modules written by members of the puppet community for Puppet. These modules will provide you with classes, new resource types, and other helpful tools such as functions, to manage the various aspects of your infrastructure. This reduces your task from describing the classes using Puppet's DSL to using existing descriptions with the appropriate values for the parameters.

If you would like to further inspect the `puppetlabs-mysql` module Puppet code, you need to `cd` to the path then open the `init.pp` manifest.

```
cd /etc/puppetlabs/puppet/modules/mysql/manifests
nano init.pp
```

However, there is a much easier way to inspect this module by visiting the [page for the puppetlabs-mysql module on the Forge](#).

The documentation on the page provides insight into how to use the provided class definitions in the module to accomplish tasks. If we wanted to install `mysql` with the default options, the module documentation suggests we can do it as follows:

```
include '::mysql::server'
```

It's as simple as that! So if we wanted our machine to have the `mysql` server installed on it, all we need to do is ensure that the above **class declaration** is in some manifest that applies it to our node.

Puppet Enterprise Supported Modules

[Puppet Enterprise Supported Modules](#) make it easy to ensure common services can be set up, configured and managed easily with Puppet Enterprise. These are modules that are tested with Puppet Enterprise, and officially supported under the umbrella of Puppet Enterprise support. They will be maintained, with bug and security patches as needed, and are vetted to work on multiple platforms. The list of modules is growing, and includes modules to manage, among others, NTP, Firewall, the Windows registry, APT, MySQL, Apache, and many others. Here's a [List of all supported modules at the Forge](#).

Review

We familiarized ourselves with the Puppet Module tool, which allows us to download and manage modules from the Puppet Forge. Once a module is installed, we have access to all the definitions and tools provided by the installed module. This allows us to accelerate the process of managing system configurations with Puppet, by providing us the ability to re-use the work of the Puppet community. Once you become familiar with Puppet, you might even contribute to the Forge yourself, sharing your work with the community!

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.

Please report issues at the [Learning VM issue tracker](#).

You can also 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, comparison 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/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](#).