puppet labs

Learn Puppet:

Quest Guide for the Learning VM

Table of Contents:

Learning VM Setup

About the Learning Virtual Machine

The Learning Virtual Machine (VM) is a sandbox environment for you to play with and learn about Puppet Enterprise. The VM is powered by CentOS Linux and for your convenience, we've already pre-installed Puppet Enterprise (PE) onto the VM. To get started with learning about Puppet Enterprise, we first need to get the VM running.

The Learning VM comes in two flavors - a VMware version that is suitable for VMware Player or VMware Workstation on Linux and Windows based machines, and VMware Fusion for Mac, as well an Open Virtualization Format (OVF) file that is suitable for all virtualization players that support it. In this case we recommend Oracle's Virtualbox.

Getting started with the Learning VM

In order to get started, we need to open the file with the appropriate virtualization software. If you haven't already downloaded VMware Player, VMware Workstation, or Oracle Virtualbox, please see the linked below:

- VMWare Player
- VirtualBox

We will need to ssh to the Learning VM, and Putty is a free SSH client for Windows. If you are using a Windows computer, please also download Putty.

Once that is complete, please follow the instructions in the following sections.

VMware Setup

Before we get started, please ensure you have an up-to-date installation of your VMware virtualization software. Once you're certain everything is up to date, open the *.vmx* file you extracted from the VMware VM zip file, change the Network Adapter to use a Bridged connection, tweak the memory settings (we recommend increasing), and finally, power on the VM. If at any point, you are not sure or want to start from scratch, you can delete the files extracted from the zip archive, and start over again by extracting the files from the archive.

For the rest of this guide, the instructions are for VMware Fusion. However, this should assist you in using VMware Player or Workstation as well.

From the VMWare **File** menu, select **Open** and navigate to the .vmx file included in your Learning VM download. You can also drag and drop the .vmx file into the VMWare virtual machine library.

Don't launch the VM just yet; we'll want to adjust a couple of settings first. With the Learning VM selected in the VMWare library, open the **Settings** panel and click the **Network Adapter** icon.

Select **Autodetect** under the **Bridged Networking** heading as shown in the example.

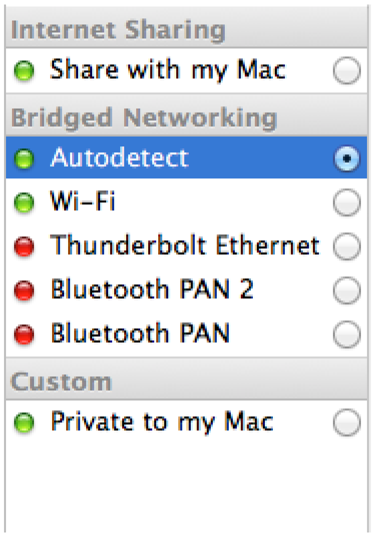


Figure 1

Next, we'll want to allocate some extra memory to the VM to ensure that it has the resources neccessary to run smoothly. Go back to the **Settings** panel and click the **Processors & Memory** icon. We suggest allocating at least two gigabytes of memory. Use the slider to set your memory allocation. Note that the Learning VM will likely still function with less memory allocated, but you may encounter performance issues.

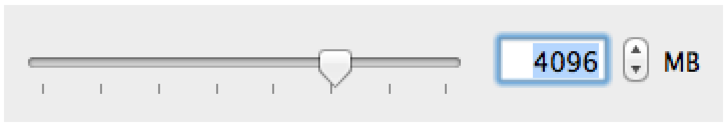


Figure 2

Now that your settings are configured, click the **Power On** button to boot up the VM.

```
NOTE: 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.
```

Once the VM is powered up, skip ahead to the Next Steps section below.

VirtualBox Setup

Be sure you have an up-to-date installation of VirtualBox. The Learning VM works best with VirtualBox 4.x. If you don't have VirtualBox 4.x, please get you free download of the latest version.

Choose “Import Appliance” from the File menu and select the .ovf file included with your download.

NOTE: **Do not** use the “New Virtual Machine Wizard” and select the included .vmdk file as the disk; machines created this way will kernel panic during boot.

Don't launch the VM just yet; we'll want to adjust a couple of settings first.

In the VirtualBox Manager panel, select **Network** to access networking options. Choose **Bridged Adapter** from the drop-down menu.

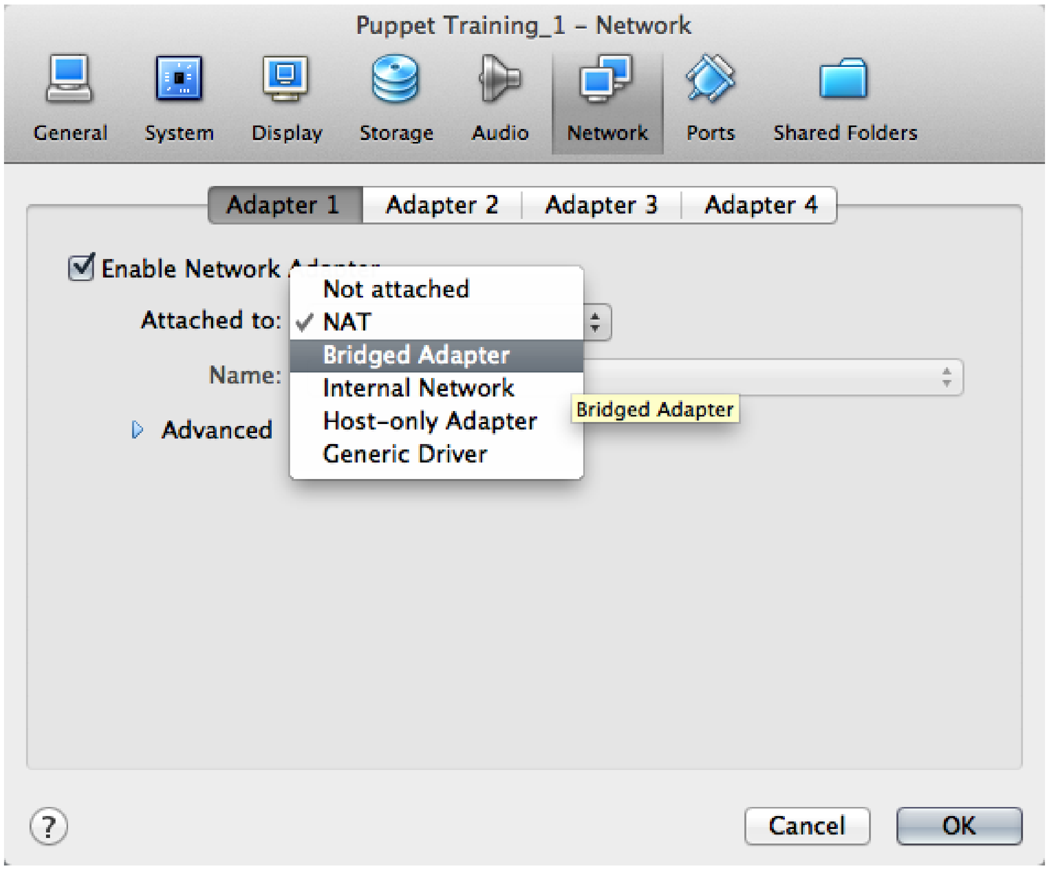


Figure 3

For everything to work smoothly, we suggest allocating at least two gigabytes of memory to the VM. In the VirtualBox Manager panel, click **System** to access the system options and use the slider to set your memory allocation. Note that the Learning VM will likely still function with less memory allocated, but you may encounter performance issues.

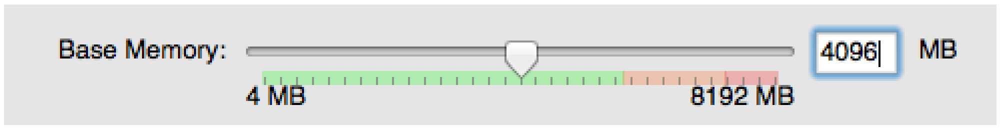

Base Memory: 4096 MB

Figure 4

Now that everything is configured, click the **Start** button in the upper left to boot up the VM.

Note: Refer to [VirtualBox documentation] for additional information as required.

Once the VM is powered up in VirtualBox, follow the Next Steps below:

Next Steps

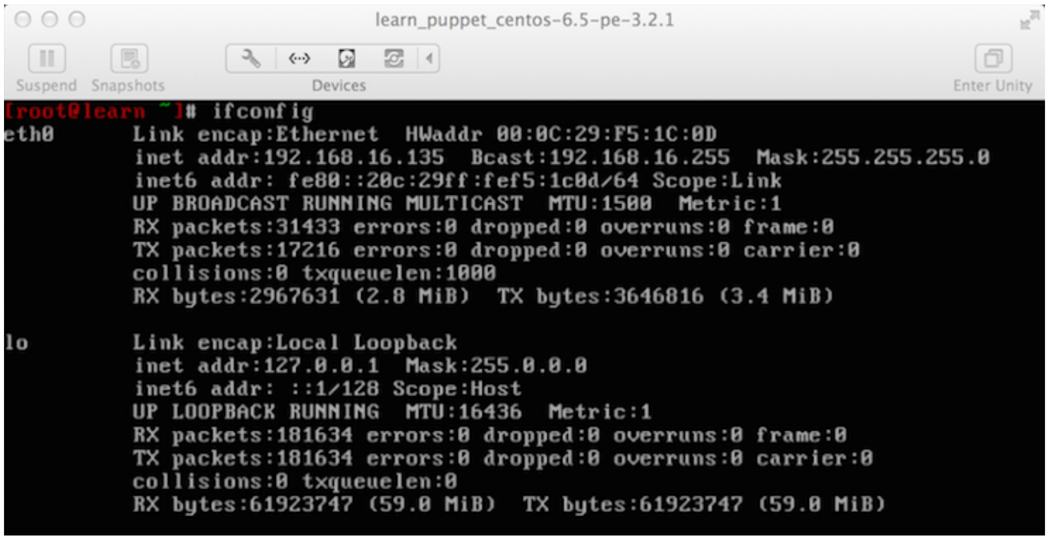
Once you have the VM running, log into it using the following credentials:

- Login Username: root
- Password: puppet

Once you are logged in, please make a note of the IP address. You'll need this to access the VM via SSH. If you forget the IP address, or if it changes, you can access it again by entering the following command:

```
    ifconfig
```

The IP address will be listed as the `inet addr` for the `eth0` interface. For example:



```
[root@learn ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:F5:1C:0D
          inet addr:192.168.16.135  Bcast:192.168.16.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fef5:1c0d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31433 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17216 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2967631 (2.8 MiB)  TX bytes:3646816 (3.4 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:181634 errors:0 dropped:0 overruns:0 frame:0
          TX packets:181634 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:61923747 (59.0 MiB)  TX bytes:61923747 (59.0 MiB)
```

Figure 5

The IP address for the VM in the above example is: **192.168.16.135**. Please note that the IP address for your VM will be different.

One more thing before we are done. To ensure you have the best experience, and the quest progress tracking works as expected, you need to **logout from the VM**.

Type the command:

```
exit
```

and press enter, in the VM.

Now you are ready to learn more about Puppet using the installation of Puppet Enterprise on the VM. Please continue following the rest of the Quest Guide. We hope you have fun learning Puppet!

Welcome

Quest Objectives

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

Getting Started

In this quest we'll introduce you to the Learning VM and give you the information you need to get you started on your learning adventure. When you're ready to get started, type the following command:

```
quest --start welcome
```

The Puppet Enterprise Learning Virtual Machine

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

Welcome to the Quest Guide for the Puppet Enterprise Learning Virtual Machine (VM). This guide is your companion to learning Puppet using the Learning VM. You should have started up the VM by now, and have an IP Address for the VM.

Use the IP address to SSH to the VM. We do this for convenience and it's essential for you to get the most out of the Learning VM and this Quest Guide.

If you are logged in directly to the Virtual Machine, please logout from it by typing the following command in the VM's terminal and pressing Enter:

```
exit
```

We will now connect to the VM over SSH.

To SSH to the VM, on a Linux system or a Mac, you can open a Terminal application and run the following command:

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

where `<ip-address>` will be replaced by the IP address for your Learning VM that you noted down when setting up your VM.

If you are using a Windows computer, please use an SSH client. We recommend Putty.

Here are the credentials to log in to the Learning VM via SSH:

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

Once you're logged in, feel free to take a look around. You will see the Learning VM is fairly typical of a Unix-based operating system. You should be aware though, that some services are running in the background, including the SSH service you're using to access this Learning VM from your own terminal.

We should give you a heads up; since you're logged in to the `root` account, which is garnished by the `uid => 0`, you carry the mark of a *Superuser*. Your account gives you the ability to change just about anything you would like in this Learning VM, just as you would if you were tasked with administrating a machine.

By following this Quest Guide, you will learn how Puppet allows you to use these privileges easily and effectively.

What is Puppet?

So what is Puppet, and why should you care? At a high level, Puppet manages your machines' configurations. You describe your machine configurations in an easy-to-read declarative language, known as the Puppet DSL, and Puppet will bring your systems into the desired state and keep them there.

Puppet Enterprise is a complete configuration management platform, with an optimized set of components proven to work well together. It combines Puppet (including a preconfigured production-grade puppet master stack), a web console for analyzing reports and controlling your infrastructure, powerful orchestration features, cloud provisioning tools, and professional support.

It may seem a lot easier to "just run a command" to effect a change in configuration, or perhaps writing a script that executes a series of commands looks like a more effective way to manage the configuration of a system. This is true, as long as you're only concerned about a single change, or changes to a single system. The true power of Puppet is that it allows you to describe all the details of the configuration for multiple machines in a composable manner, and allows you to manage the configuration of multiple machines (think hundreds or thousands) without having to write complicated scripts that are hard to comprehend; or maintaining an inventory of all your systems, and logging in to each system in turn to run the required commands or scripts. Puppet automates the process of configuring your systems and keeping them configured exactly as you need them to be.

But a journey of a thousand miles starts with a single step. This Learning VM will get you started by means of examples that help you configure the VM. While doing the exercises, imagine the possibilities in using what you learn to manage hundreds or thousands of systems in an effortless, intuitive manner!

Task 1 :

Before we dig any deeper, let's check and see what version of Puppet Enterprise we 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 3.2.1)

This indicates that Puppet Enterprise 3.2.1 is installed on the Learning VM, which leverages Puppet version 3.4.3. Puppet Enterprise includes more than 40 open source projects, including Puppet, MCollective, PuppetDB, Hiera, and others that we’ve integrated, certified, performance-tuned, and security-hardened to make it a complete solution suitable for automating mission-critical enterprise infrastructure. In addition, it includes several capabilities found only in Puppet Enterprise, including event inspection, supported modules, role-based access control, certification management and VMware cloud provisioning.

What is a Quest?

Up to this point we've introduced you to the Learning VM and Puppet. We'll continue to dive into greater detail about Puppet in future quests. Wait a minute!

What's a quest? That's a great question! A **Quest** is a structured tutorial consisting of a number of interactive tasks that will help you learn about a topic related to Puppet.

Each Quest includes a number of **Tasks** that give you a hands-on opportunity to apply what you have learned. You have already finished a task by now, since the first task was to execute the `puppet -V` command earlier. But how do you keep track of everything as you progress? What if you forget what quest you are on? These are all great questions and that's why we specifically created a 'Quest Tool' for this Learning VM to help you when you're in need.

The Quest Tool

To monitor your status as you progress through these Quests, we've created a quest tool you can use in the Learning VM. However, this quest tool is not part of Puppet itself. We have included this tool in the Learning VM to provide you with real-time feedback as you progress through the many Quests and Tasks on your journey to learn Puppet.

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
```

Task 3 :

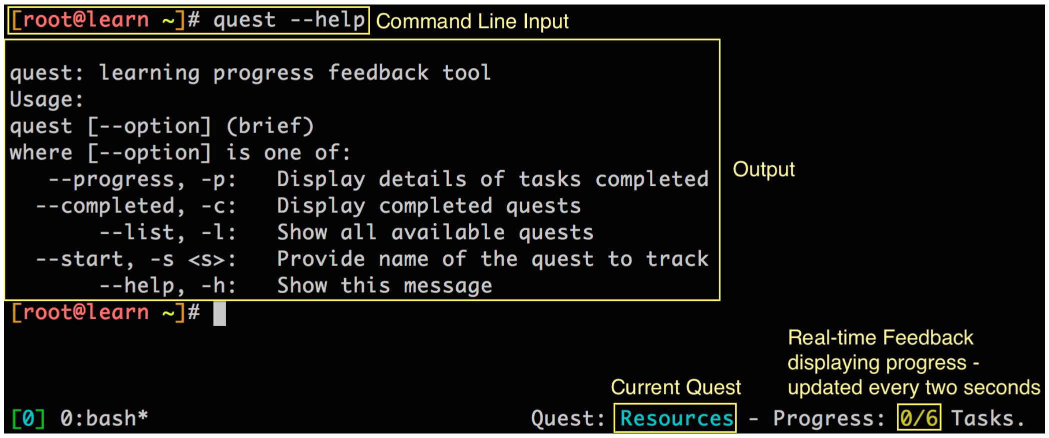
Let's find out how much progress you have made thus far! Execute the following command:

```
quest --progress
```

Using the quest tool is entirely optional, but we have also integrated it into the first few quests to help you out if needed.

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

In addition to the quest commandline tool, we have also integrated real-time feedback into the VM, which is displayed in the bottom-right corner of the terminal, as seen in Figure 1.



```
[root@learn ~]# quest --help

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
[root@learn ~]#

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

Command Line Input
Output
Real-time Feedback displaying progress - updated every two seconds
Current Quest

Figure 1

Review

In this introductory quest we provided a very high level explanation of what Puppet is, what a quest is, and how to use the quest tool. As you progressed through this quest, you learned about the mechanics of successfully completing a quest by means of completing the associated tasks. We hope you have a general understanding of how to complete a quest and what is in store for you on your learning journey.

The Power of Puppet

Prerequisites

- Welcome Quest

Quest Objectives

- Use a module from the Forge, and a pre-written module to setup a website serving the quest guide.
- Use the Puppet Enterprise (PE) Console to manage the Learning VM's configuration

Getting Started

In this quest you will be introduced to the power of Puppet using the Puppet Enterprise (PE) Console and the Puppet Forge. The objective of this quest is to set up a functional website locally that will serve you the entire Quest Guide as an alternative to the PDF version. Don't worry though, both the website and PDF have the same content and will help guide you on your journey to learn Puppet.

We are not going to be writing any code in this quest to accomplish the objectives. Instead, we will be using a module from the Forge, as well as another that has been written for you on the Learning VM. This will serve as an example of how you can use the code that you will learn to write in the succeeding sections. We hope this gives you a high-level overview of what it is like to use Puppet Enterprise to manage systems, while leveraging code developed by the community that is shared on the Forge. Some of the technical terms in this quest may not be familiar, but we promise we will explain all of it by the end of the guide!

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

```
quest --start power
```

The Puppet Forge

The Puppet Forge is a public repository of modules written by members of the Puppet community. We can leverage the content from the Forge to simplify the process of managing your systems. We will go into greater detail about the Forge in the Puppet Module Tool Quest.

But what is a Module, you ask? Simply put, a module is a self-contained bundle of code and data. We will learn about Modules in greater detail in the Module Quest.

Puppet Enterprise Supported Modules

Remember how we indicated in the Welcome Quest that Puppet Enterprise comes with several benefits, in addition to Puppet? Puppet Enterprise supported modules are modules that are rigorously tested with Puppet Enterprise and are also supported (maintained) by Puppet Labs.

Task 1 :

Install the puppetlabs-apache module.

For us to get started with setting up our Quest Guide website, we need to download and install the apache module from the Forge. In your terminal, type the following command:

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

This tells Puppet to download and install the `puppetlabs-apache` module from the Forge onto the Learning VM. Modules are installed in the directory specified as Puppet's *modulepath*. For Puppet Enterprise, this defaults to `/etc/puppetlabs/puppet/modules/`.

Offline? No problem.

We've cached the required modules on the Learning VM. If you don't have internet access, run the following terminal commands instead of using the `puppet module` tool:

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

tar zxvf /usr/src/forge/puppetlabs-apache-0.8.1.tar.gz -C .

mv puppetlabs-apache-* apache
```

Great job! You've just installed your first module from the Forge. Pretty easy don't you think? We also went ahead and created the `lvmguide` module we will need. You will find this module in the `modulepath` directory as well.

The lvmguide and apache modules

The `lvmguide` module includes a *class* of the same name, which configures the Learning VM to act as a webserver serving the Quest Guide as a static html website. If you're wondering what a class is, it is a named block of Puppet code.

It does this by using the `apache` module you just installed to:

- install the Apache httpd server provided by the `httpd` package
- create a new VirtualHost that will act as the Quest Guide website
- configure httpd to serve the Quest Guide website
- start and manage the httpd service

Finally, the `lvmguide` class places the files for the website in the appropriate directory.

Putting the modules to use

As we mentioned previously, a class is a named block of Puppet code. The `lvmguide` class contains the definition that will configure a machine to serve the quest guide. In order to configure the Learning VM to serve you the Quest Guide website, we need to assign the `lvmguide` class to it, a process known as *Classification* (don't worry about the terminology at this point. We'll talk more about this in the Modules Quest). To do this we will use the Puppet Enterprise Console.

The Puppet Enterprise Console

The PE Console is Puppet Enterprise's web-based Graphical User Interface (GUI) that lets you automate your infrastructure with little or no coding necessary. You can use it to:

- Manage node requests to join the Puppet deployment
- Assign Puppet classes to nodes and groups
- View reports and activity graphs
- Trigger Puppet runs on demand
- Browse and compare resources on your nodes
- View inventory data
- Invoke orchestration actions on your nodes
- Manage console users and their access privileges

For this quest, we will use the PE Console to classify the VM with the `lvmguide` class.

The Facter Tool

Puppet Enterprise is often used in concert with other tools to help you administer your systems. Facter is one such tool. You can use the Facter tool to help you obtain facts about your system. Facter is Puppet's cross-platform system profiling library. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

Facter is used and described in later quests, but the sooner you get familar with Facter, the better. Go ahead and get to know your system a little better using the `facter` command.

Task 2 :

Find the ipaddress of the VM using Facter:

To access the PE Console we need to find your IP address. Luckily, Facter makes it easy to find this. In your terminal, type the following command:

Facter is one of the many prepackaged tools that Puppet Enterprise ships with.

```
facter ipaddress
```

Please write down the IP address for the VM as you will need it to access the PE Console. Next, in your browser's address bar, type in: **https://**, and press Return (Enter) to load the PE Console.

You can see all the facts by running `facter -p`

Awesome! You are at the doorstep of the PE Console. Enter the login information below to gain access:

Username: puppet@example.com
Password: learningpuppet

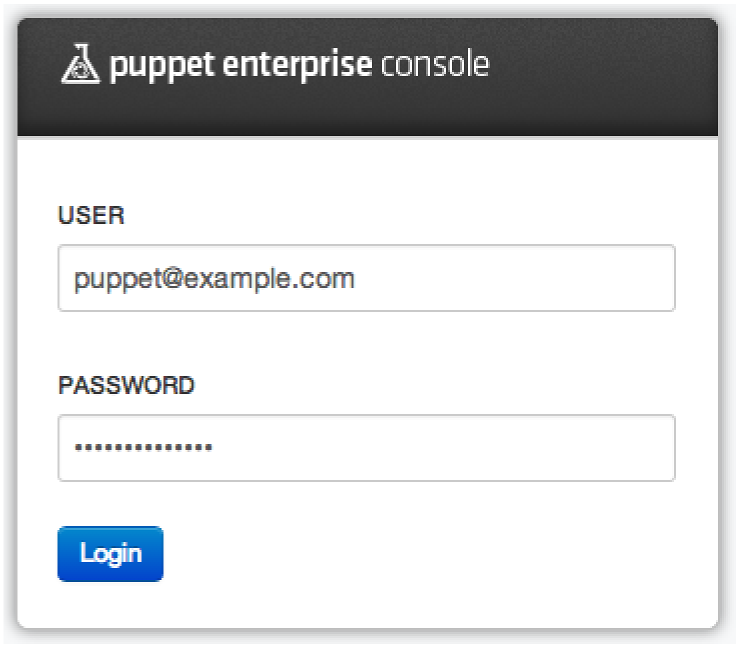


Figure 1

Security Restriction

If your browser gives you a security notice, go ahead and accept the exception. The notification appears because we are using a self-signed certificate.

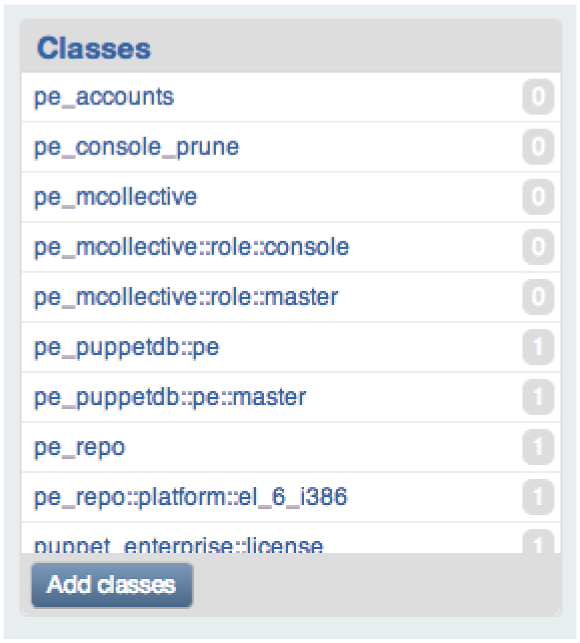
You're in! Now that you have access to the PE Console, the next step will be to classify the "learn.localdomain" node with the `lvmguide` class.

Using the PE Console for Classification

In order to classify the `learn.localdomain` node (the Learning VM) with the `lvmguide` class, we need to add the class to the list of classes available to the PE Console to classify nodes with. Let's add it.

Adding a Class to the PE Console

STEP #1: Click the "Add Classes" button
You may need to scroll to the bottom of the page. The button is located in the lower left hand corner of the screen in the Classes panel.

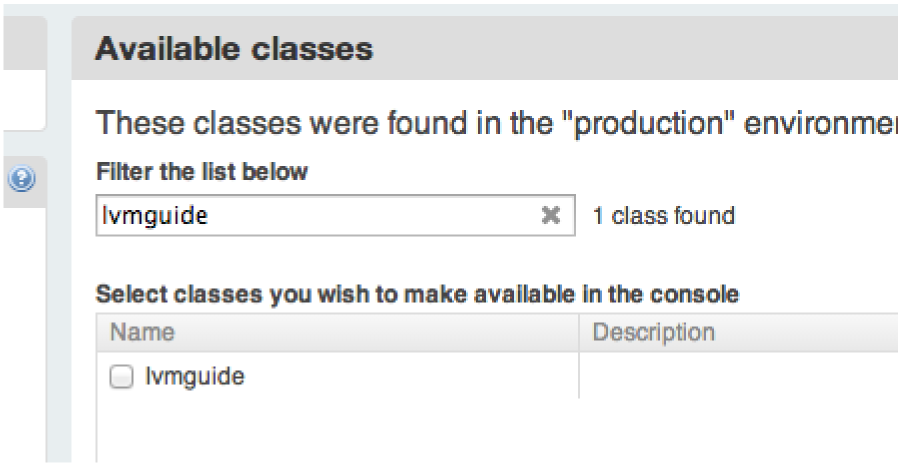


Classes

pe_accounts	0
pe_console_prune	0
pe_mcollective	0
pe_mcollective::role::console	0
pe_mcollective::role::master	0
pe_puppetdb::pe	1
pe_puppetdb::pe::master	1
pe_repo	1
pe_repo::platform::el_6_i386	1
puppet_enterprise::license	1

Figure 2

STEP #2: Type *lvmguide* in the "Filter the list below" input box



Available classes

These classes were found in the "production" environme

Filter the list below

lvmguide

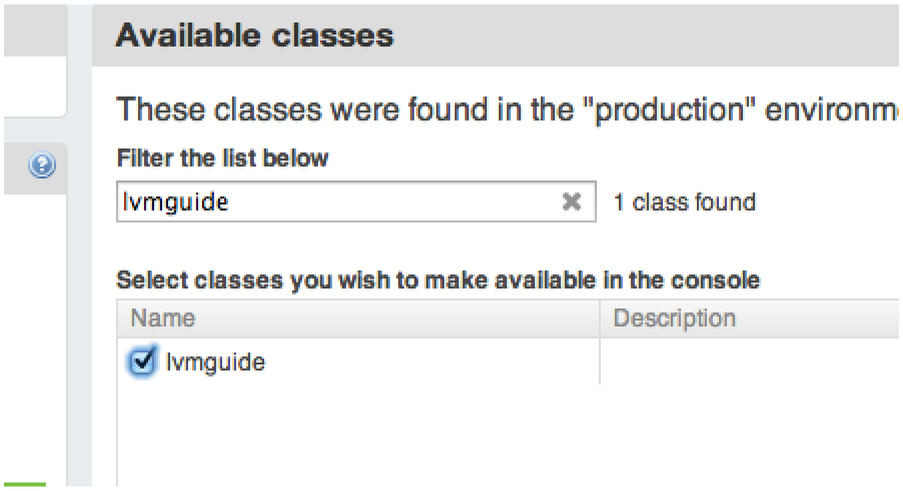
1 class found

Select classes you wish to make available in the console

Name	Description
lvmguide	

Figure 3

STEP #3: Select the checkbox associated with *lvmguide* class that appears in your results once you filter


Available classes

These classes were found in the "production" environm

Filter the list below

lvmguide

1 class found

Select classes you wish to make available in the console

Name	Description
✓ lvmguide	

Figure 4

STEP #4: Click the "Add selected classes" button to add the *lvmguide* class to the list of classes that can be assigned to node in the PE Console.

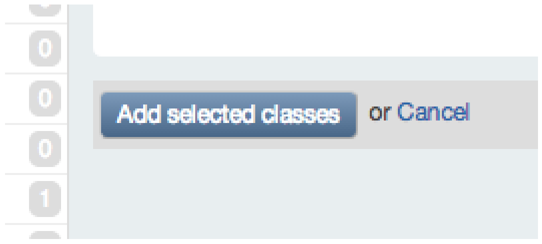


Figure 5

Verification

You should see the class appear in the list of classes on the left, and also see a verification message at the top of the PE Console.

Classifying a node

Now that the `lvmguide` is available in the PE Console, let's classify the node `learn.localdomain` with this class.

STEP #1: 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.

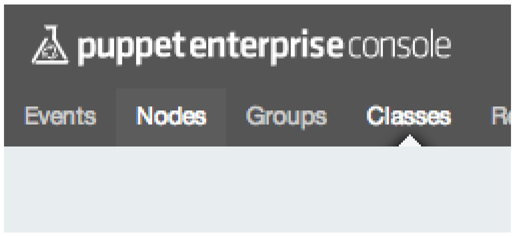


Figure 6

STEP #2: Click on the *learn.localdomain* node hyperlink (should be the only one listed since we are only managing the Learning VM with Puppet Enterprise). The list of nodes should be near the center of the PE Console.

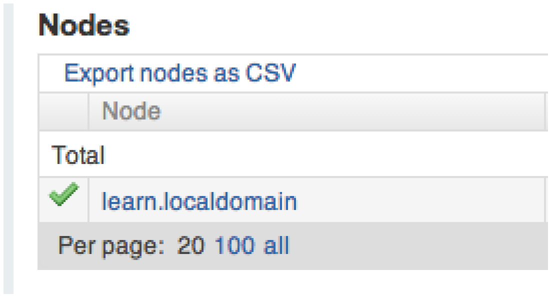


Figure 7

STEP #3: Click the "Edit" button located in the top-right corner of the screen.

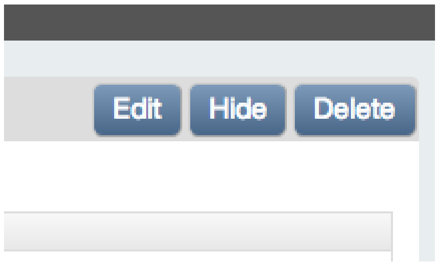
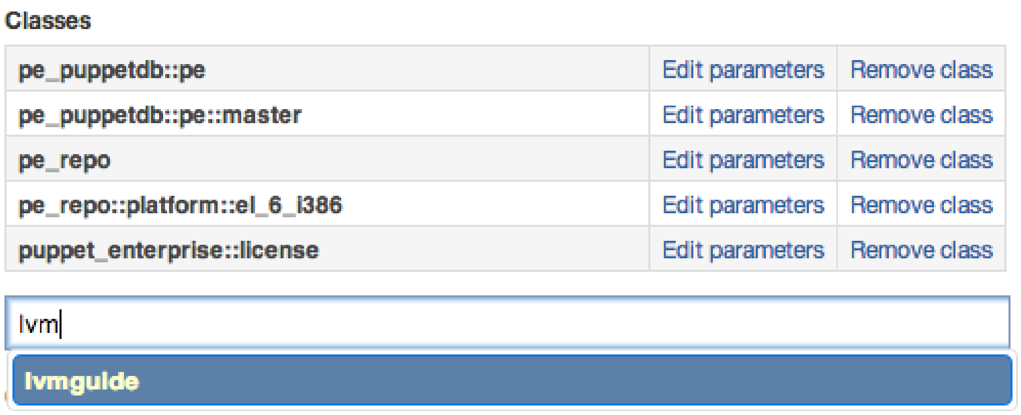


Figure 8

STEP #4: In the Classes section, type *lvmguide* in the "add a class" input box



Classes

pe_puppetdb::pe	Edit parameters	Remove class
pe_puppetdb::pe::master	Edit parameters	Remove class
pe_repo	Edit parameters	Remove class
pe_repo::platform::el_6_i386	Edit parameters	Remove class
puppet_enterprise::license	Edit parameters	Remove class

Figure 9

Editing Class parameters

If you click on *Edit Parameters* for the lvmguide class you can see the defaults. There is nothing for you to do here for now, but it's good to notice it so you will know where to look if you need to edit the parameters for a class in the future.

STEP #5: Click the "Update" button at the bottom.

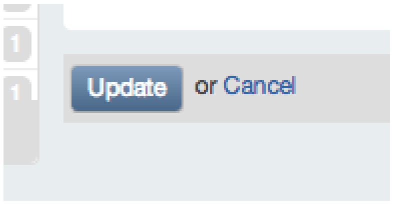


Figure 10

Excellent! We just finished classifying the `learn.localdomain` node with the `lvmguide` class.

Let's Run Puppet

Now that we have classified the `learn.localdomain` node with the `lvmguide` class, we need to tell puppet to configure the machine appropriately. The Puppet `agent` daemon does this by default by running in the background on any nodes you manage with Puppet. Every 30 minutes, the Puppet agent daemon requests a *catalog* from the Puppet Master, and applies the retrieved catalog. Since we want to see the configuration changes (as specified in the `lvmguide` class) applied immediately, we can ask the agent to fetch and enforce the definition for the “learn.localdomain” node.

Task 3 :

Apply the configuration changes.

Run puppet agent:

To fetch and apply the catalog we will manually trigger a Puppet run. Puppet will check the classification you set in the PE Console and automatically run through all the steps needed to implement the `lvmguide` class. In your terminal, type the following command:

```
puppet agent --test
```

After a brief delay, you will see text scroll by in your terminal indicating that some changes were made to the configuration of the VM and the website was set up. **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.

Ensure the website serving the Quest Guide is running!

There are no commands to run to make this happen - no packages to install, no files to edit, and no services to start. All the heavy lifting was done by the lvmguide and apache modules you installed.

Let's check out the Quest Guide! In your browsers address bar, type the following URL: `http://<ip-address>`.

> Note that `https://<ip-address>` will load the PE Console, while `http://<ip-address>` will load the Quest Guide as a website.

The website for the quest guide will remain accessible for as long as the VM's IP address remains the same. If you were to move your computer or laptop to a different network, or if you suspended your laptop and resumed work on the Learning VM after a while, the website may not be accessible. You still have the PDF to follow along with.

In case any of the above issues happen, and you end up with a stale IP address, do the following to get a new IP address.

Refresh your DHCP lease:

```
service network restart
```

Find your IP address:

```
facter ipaddress
```

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

Exploring the lvmguide Class

Let's take a high level look at what the `lvmguide` class does for us. In your terminal you're going to want to `cd` into the modules directory:

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

Next, using `vim`, `nano`, or a text editor of your choice, open up the `init.pp` manifest. Type the following command in you terminal:

```
nano lvmguide/manifests/init.pp
```

> We will learn more detailed information about each of these concepts in future quests. However, this is a leading example of how a few lines of Puppet code can automate a relatively complex task.

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

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

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

  file { '/var/www/html/lvmguide':
    ensure  => directory,
    owner   => $::apache::params::user,
    group   => $::apache::params::group,
    source  => 'puppet:///modules/lvmguide/html',
    recurse => true,
    require => Class['apache'],
  }

}
```

In the above code sample, we see that the `class lvmguide`:

1. Takes two parameters: `$document_root` and `$port`

2. Declares the class `apache` with the `default_vhost` parameter set to `false`. That's the equivalent of saying "I need apache to be setup without the default VirtualHost."
3. Declares an `apache::vhost` for the quest guide with the title `learning.puppetlabs.vm`, and with `$port` and `$docroot` set to the class parameters. Now this is the same as saying "Please set up a VirtualHost website running on port 80, serving the "learning.puppetlabs.vm" website with the files from the /var/www/html/lvmguide' directory
4. Sets the document root to `/var/www/html/lvmguide` (which is the default value for the $document_root parameter
5. Finally, the class ensures that the directory `/var/www/html/lvmguide` exists and that its contents are managed recursively. We notice that the source for the contents of that directory is also specified

The details may not be extremely clear at this point in your learning journey, but will hopefully get clearer as you complete more quests. The rest of the Quest Guide will help you get to the point where you can write your own classes and modules. Our hope is that the code for the `lvmguide` class listed above appears to be friendly and elegant. We often talk about Puppet's Domain Specific Language (DSL) as **self-documenting code**.

The real power here is that you now have the class `lvmguide` that can be applied to any number of nodes as long as you manage them with Puppet. Since we have `lvmguide` as a module in a directory by itself, you can share that directory with someone else, and ask them to place it in their Puppet master's modulepath, where they too can also get the Quest Guide website up and running on any number of nodes. With a few lines of code, we installed and configured the Apache httpd web server to serve us the Quest Guide website, by leveraging a freely available, shared, module from the forge - `puppetlabs-apache`.

Review

Great job on completing the quest! To summarize, we learned how to leverage exising modules on the Forge and use the PE Console to set up our quest guide website locally. We also learned how to classify a node with a class. We will continue to reference the block of Puppet code for the `lvmguide` module throughout the remaining quests in order to understand what we did.

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. You would need to 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 as well as can 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 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* 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: `'$1$hNahKZqJ$9ul/RR2U.9ITZlKcMbOqJ.'`

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 like. In subsequent quests, we will see how, instead of just inspecting existing resource, we can *declare* how specific resource *should look like*, providing us the ability to model the state of these resources.

Puppet provides us this ability to describe resources of different types of resources. Each type is a high-level model of the resource. Our job in 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 is achieved.

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 implentation details with 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 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 incase 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 suceeding 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 6 :

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 walkthough 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
- Mainfest 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"
```

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.

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.

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 look at the file to see that the contents have been set as you intended:

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

The string assigned to the `$pangram` variable was passed into your file resource's `content` attribute, which in turn told Puppet what the content of the `/tmp/pangram.txt` file should exist.

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

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 occurences 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}!"`

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? You guessed it. 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."

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

Puppet's DSL. We hope you are becoming more familar 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
- Mainfest Quest
- Classes 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 the 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:
  warn( 'Tried to include class ntp on virtual machine.' )
}
elsif $operatingsystem == 'Darwin' {
  # Our NTP module is not supported on Darwin:
  warn( 'This NTP module does not yet work on Darwin.' )
}
else {
  # Normal node, include the class.
  include ntp
}
```

The Warn Function

The `warn()` 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.

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 `facter uptime_hours` to check the uptime yourself. The notice you saw when you applied your manifest should describe the uptime returned from the Facter 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.

- Basic cases are compared with the `==` operator (which is case-insensitive).
- Regular expression cases are compared with the `=~` operator (which is case-sensitive).
- The 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 in an actual practice, instead of using the result of the case statement like the one above to define the contents of a file, you could use it directly in a `package` resource declaration like the following:

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

This would allow you to always install and manage the right Apache package for a machine's operating system. This kind of careful accounting for different the conditions under which a manifest might run 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 have 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) keyword. In the block surrounded by curly braces are 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 `warn()`, it is up to you to make sure your code handles unexpected conditions gracefully. You wouldn't want Puppet to forge ahead with 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 have guided you in 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
- Mainfest Quest
- Varibales 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 reading 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 give a list of things to do, and it was left to decide the most efficient way to get those 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 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.

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

- `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. Notify will generate a refresh even whenever the 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.

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 and 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 that we have created the manifest, applying the manifest will ensure that the `/etc/ssh/sshd_config` file will have the exact same content as the `/root/examples/sshd_config` 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 it's configuration file, but all that would mean nothing if the package that install the SSH server 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_config 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 to contents of a file in a directory that does not exists, 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 specify 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
- Mainfest 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 define classes and declare classes separately.

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 defintion 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 to 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 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',
){
 class { 'apache':
   default_vhost => false,
 }
 apache::vhost { 'learning.puppetlabs.vm':
   port    => $port,
   docroot => $document_root,
 }
 file { '/var/www/html/lvmguide':
   ensure  => directory,
   owner   => $::apache::params::user,
   group   => $::apache::params::group,
   source  => 'puppet:///modules/lvmguide/html',
   recurse => true,
   require => Class['apache'],
  }
}
```

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 paranthesis 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 the 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 two code blocks in the class definition are resource declarations:

```
apache::vhost { 'learning.puppetlabs.vm':
  port    => $port,
  docroot => $document_root,
}
file { '/var/www/html/lvmguide':
  ensure  => directory,
  owner   => $::apache::params::user,
  group   => $::apache::params::group,
  source  => 'puppet:///modules/lvmguide/html',
  recurse => true,
  require => Class['apache'],
}
```

First, we declare a `apache::vhost` resource type, and pass along values from our class parameters to its `port` and `docroot` attributes.

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 way to invoke classes - using the `include` keyword, and declaring classes using a syntax similar to resource declarations. Classes are 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
- Mainfest 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.

Until now, we've been writing one-off manifests to demonstrate individual Puppet topics, but in actual practice, every manifest should be contained within a module.

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.

Simple 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 this 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, as also the `lvmguide` module that is use to set up the quest guide website. 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. Whatever other manifests it contains, however, the `init.pp` manifest defines the module's main class, which must have same name as the module itself, in this case, `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 use 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

Once you starting writing and using modules, you can start creating 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 resuse 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
- Mainfest Quest
- Varibales 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 this 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 more in 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 for that module. To do so from the command line, type the following command:

```
puppet module search puppetlabs-mysql
```

You can also search for just `mysql` to see all the modules that have `mysql` in their name or description. Modules on the Forge are always named as *authorname-*

modulename. The `puppetlabs-mysql` module is a module authored and curated primarily by the puppetlabs organization.

Task 3 :

Install a module

There's something on the Forge that exists! Let use it. Let's install `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
```

You will that the module is downloaded and installed. You will also see that in addition to the `puppetlabs-mysql` module, all the modules it depends on were also downloaded and installed.

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 witht the appropriate values for the parameters.

If you would like to further inspect the `puppetlabs-mysql` module Puppet code, you need to want 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` module 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 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 configrations with Puppet, by providing us the ability to re-use the work of the Puppet 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. You can reach us at learningvm@puppetlabs.com. We look forward to hearing from you.

Glossary of Puppet Vocabulary

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

attribute

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

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

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

agent

(or **agent node**)

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

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

catalog

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

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

class

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

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

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

classify

(or **node classification**)

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

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

declare

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

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

Contrast with "define."

define

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

define (noun)

(or **definition**)

An older term for a defined resource type.

define (keyword)

The language keyword used to create a defined type.

defined resource type

(or **defined type**)

See "type (defined)."

ENC

See external node classifier.

environment

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

expression

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

external node classifier

(or **ENC**)

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

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

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

fact

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

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

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

Facter

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

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

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

filebucket

A repository in which Puppet stores file backups when it has to replace files. A filebucket can be either local (and owned by the node being mangaed) 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 Heira 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. See ["Relationships and Ordering"][lang*puppet*relationships] in the Puppet language reference for more details.

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. (`~>`) See ["Relationships and Ordering"][lang*puppet*relationships] in the Puppet language reference for more details.

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.