Download learnng vm contains PE from

https://learn.puppet.com/category/self-paced-training

<https://puppet.com/download-learning-vm>

Practice puppet below topics

* [Introduction](http://192.168.56.101/en_us/)
* [Setup](http://192.168.56.101/en_us/SETUP.html)
* Quests
* [Welcome](http://192.168.56.101/en_us/quests/welcome.html)
* [Hello Puppet](http://192.168.56.101/en_us/quests/hello_puppet.html)
* [Agent Run](http://192.168.56.101/en_us/quests/agent_run.html)
* [Manifests and Classes](http://192.168.56.101/en_us/quests/manifests_and_classes.html)
* [Package File Service](http://192.168.56.101/en_us/quests/package_file_service.html)
* [Variables and Templates](http://192.168.56.101/en_us/quests/variables_and_templates.html)
* [Class Parameters](http://192.168.56.101/en_us/quests/class_parameters.html)
* [Facts](http://192.168.56.101/en_us/quests/facts.html)
* [Conditional Statements](http://192.168.56.101/en_us/quests/conditional_statements.html)
* [The Forge](http://192.168.56.101/en_us/quests/the_forge.html)
* [Roles and Profiles](http://192.168.56.101/en_us/quests/roles_and_profiles.html)
* [Hiera](http://192.168.56.101/en_us/quests/hiera.html)
* [Defined Resource Types](http://192.168.56.101/en_us/quests/defined_resource_types.html)
* [Control Repository](http://192.168.56.101/en_us/quests/control_repository.html)
* [Puppetfile](http://192.168.56.101/en_us/quests/puppetfile.html)

Quest Guide [v2.3.0](http://192.168.56.101/en_us/releasenotes.html)

# Quest Guide for the Puppet Learning VM

## About the Learning VM and this Quest Guide

The Learning VM (Virtual Machine) is a self-contained learning environment that includes everything a new user needs to get started learning Puppet. Because learning Puppet requires making changes to a system's configuration, it's not wise to play with it directly on your own laptop or desktop. Not everyone who wants to learn Puppet, however, has easy access to a system where he or she can experiment freely. And even those who do will benefit from an environment with Puppet Enterprise pre-installed and a set of tools to help guide you through the basics of configuration management with Puppet.

This guide is the companion to the Learning VM. The content of the guide is paired with a quest command line tool on the VM that will provide live feedback as you progress through the list of tasks associated with each quest in this guide. By breaking each concept into a series of incremental and validated steps, we can ensure that you stay on track as you progress through the guide.

## Who should use the Learning VM?

This guide should be useful for any reader with an interest in configuration management, system administration, or related tasks. We have done our best to avoid any assumptions about a reader's familiarity with a specific operating system. While users familiar with a Linux command-line interface will already be familiar with most of the commands used in this guide, those more accustomed to a graphical user interface will find all necessary commands provided.

The Learning VM comes with Puppet Enterprise installed, and some of the content is specific to Puppet Enterprise. Users interested in the open source version of Puppet will nonetheless benefit from the majority of the content. Certain features, such as the graphical web console and Application Orchestration tool, are exclusive to Puppet Enterprise. Content related to the Puppet master-agent architecture, Puppet code, and module structure will be generally applicable to the open source version of Puppet, though there are some differences in file locations.

# Set up the Learning VM

## Get the VM

If you haven't already, [download the Learning VM](https://puppet.com/download-learning-vm).

## Virtualization setup

Get an up-to-date version of your preferred virtualization software. [VirtualBox](https://www.virtualbox.org/wiki/Downloads) is free and available for Linux, Mac, and Windows systems. VMware has several desktop virtualization applications, including [VMware Fusion](https://www.vmware.com/products/fusion/) for Mac and [VMware Workstation](https://www.vmware.com/products/workstation/) for Windows.

The Learning VM OVA file must be *imported* rather than opened directly. Launch your virtualization software and find an **Import** or **Import Appliance** option in the **File** menu. If you cannot locate an **Import** option, refer to your virtualization software's documentation.

If you have enough available memory on your host machine, increasing the memory allocation for the VM from the default 3GB to 4GB may improve performance and stability. Memory allocation settings are found by selecting the VM in the VirtualBox Manager window, opening the *Settings* dialog, and selecting the *System* section.

## Networking configuration

All the packages and modules needed to complete the Quest Guide are hosted locally on the VM itself, but you will need to set up networking to access the VM from your host system.

Your virtualization software provides several options for network configuration. If you are on a network that may have a proxy or restrictive firewall rules (ask your network administrator if you're not sure), we strongly suggest running the VM with an offline host-only network. This offline configuration requires more complex initial setup with VirtualBox software, but will help you avoid troubleshooting networking issues between your Puppet master and agent systems.

If you are using VirtualBox on a network without a proxy or restrictive firewall rules, you may find it simpler to use the bridged adapter described in the online configuration section below.

### Offline

For **VirtualBox:**

To use host-only networking on VirtualBox, you will need to create and configure a new network from the VirtualBox *preferences* panel. Note that this may be called *settings* on some systems. Be sure that you're looking at the preferences for VirtualBox itself, not the settings configurations for a specific VM.

Open the VirtualBox preferences panel and select the **network** section. Select **Host-only Networks**. Create a new network, and click the screwdriver icon to the side of the dialog to edit the network configuration. In the **Adapter** section, enter the following settings:

**IPv4 Address: 192.168.56.1**  
**IPv4 Network Mask: 255.255.255.0**

In the **DHCP Server** section, enter the following settings:

Check the **Enable Server** box  
Server Address: 192.168.56.1  
Server Mask: 255.255.255.0  
Lower Address Bound: 192.168.56.110  
Upper Address Bound: 192.168.56.200

Click **OK** to accept the adapter configuration changes, and again to exit the preferences dialog. Open the settings section for the Learning VM from the VirtualBox Manager window. Go to the **Network** section, select **Host-only Adapter**from the drop-down menu, and select the name of the host-only adapter you created from the **Name:** drop-down. Click **OK** to accept the setting change.

For **VMWare:**

Open the **Settings** dialog for the Learning VM and select the **Network** Section. Under the **Custom** heading, select the private network adapter option.

### Online

If you would like to run the Learning VM with internet access, set the *Network Adapter* to *Bridged*. Use an *Autodetect*setting if available, or accept the default Network Adapter name. (If you started the VM before making these changes, you may need to restart the VM before the settings will be applied correctly.)

Note that the Puppet module tool, yum, and RubyGems are configured to use local repositories, so you will not be able to access remote content without manually changing the settings for these tools. While we encourage exploration, we are not currently able to support issues beyond the scope of the Quest Guide.

## Log in

Start the VM. Rather than logging in directly, we suggest using the browser-based web terminal or SSH.

To access the web terminal, open your web browser and navigate to http://<VM's IP ADDRESS>:9091. Follow the instructions show on the splash page to log in.

On Mac systems, you can use the default Terminal application or a third-party application like iTerm. For Windows, we suggest the free SSH client [PuTTY](http://www.putty.org/). Use the credentials provided on the VM console splash page to authenticate.

Once you're logged in, continue on to the **Get started** section below to access the Quest Guide and begin the interactive lessons.

## Localization

The Learning VM's Quest Guide and Quest tool currently support English and Japanese localization. The default language is English. If you would like to use the Quest tool in Japanese, run the following command on the Learning VM:export LANG=ja\_JP.utf-8. Note that you must use SSH or the browser-based web terminal to see Japanese characters. Japanese characters will not display correctly on the default VirtualBox or VMware terminal.

## Get started

Once the VM is set up and you have connected, you're ready to get started on the interactive lessons in the Quest Guide. Access the Quest Guide by opening a web browser on your host system and entering the Learning VM's IP address in the address bar: http://<IP-ADDRESS>. (Note that you must use http, as https will connect you to the PE console interface.)

# Welcome

## Quest objectives

* Familiarize yourself with the Quest structure and tool

## Getting Started

Welcome to the Quest Guide for the Puppet Learning VM. This guide will be your companion as you make your way through the Learning VM's series of interactive lessons.

At this point, you should have the Learning VM set up and an open session in your browser-based terminal or SSH client. If not, please return to the Setup section for instructions on getting started.

Before we get into Puppet itself, this lesson provides with a brief introduction to the quest tool that you will use throughout the rest of this guide to track your progress.

Ready to get started? Run the following command on the VM to begin the "Welcome" quest.

quest begin welcome

## An introduction to the quest tool

One logging in to the Learning VM, you may have noticed a colored bar at the bottom of the terminal window. This status bar is provided by a tool called [tmux](http://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/). This status bar gives you quick access to your current *quest*, and your progress through the *tasks* included in that quest.

Go ahead and use the --help flag with the quest command to list the available subcommands.

Task 1:

quest --help

A couple of seconds after entering that command, you will see the status bar update to show 1 of 2 tasks complete. The small delay is the time it takes for the quest tool to complete its check against a list of task completion criteria and for tmux to run its periodic update of the status line.

Task 2:

You can use the status subcommand to get a more verbose list of tasks and their status. Give it a try now.

quest status

You'll see that Task 2: View the list of available quests is still incomplete. To complete this task, run the list subcommand:

quest list

Now check the status again to see it marked as complete:

quest status

(If you're curious about how these tasks work, take a look at the set of tests in the /usr/src/puppet-quest-guide/tests/ directory on the Learning VM. The source code for the quest tool itself is [here](https://github.com/puppetlabs/quest).)

## Review

In this quest, we introduced the concept of the quest and interactive task. You tried out the quest tool and reviewed the mechanics completing quests and tasks.

Now that you're familiar with the quest tool, you're ready to move on to the next quest: Hello Puppet.

# Hello Puppet

## Quest objectives

* Get started with the basics: What is Puppet and what does it do?
* Install the Puppet agent on a new system.
* Understand Puppet's resources and the resource abstraction layer.

## Get started

Any sufficiently advanced technology is indistinguishable from magic.

* Arthur C. Clarke

This quest kicks off your hands-on introduction to Puppet. We'll start by installing the Puppet agent and learning some of the core ideas involved in managing infrastructure with Puppet. Once the Puppet agent is installed on a new system, you will use the puppet resource tool to explore the state of that system. Through this tool, you will learn about *resources*, the basic units that Puppet uses to describe and manage a system.

Ready to get started? Run the following command on your Learning VM to begin this quest:

quest begin hello\_puppet

## What is Puppet?

Before getting into the details of how Puppet works and how to use it, let's discuss what Puppet is and why it's worth learning. When we talk about Puppet, we're often using the name as a shorthand for a collection of tools and services that work in concert to help you define how you want the systems in your infrastructure configured, and then automate the process of bringing those systems into your desired state and keeping them there. As you continue with this guide, we'll dive deeper into Puppet's individual components, but for now we'll introduce some of the key points that distinguish Puppet's approach from the other tools you might use to manage your systems.

Puppet lets you to define a desired state for all systems in your infrastructure. Once this state is defined, Puppet automates the process of getting your systems into that state and keeping them there.

This state for your systems is written in Puppet code, a domain specific language that lets you define how each server or device in your infrastructure should be configured. This **infrastructure as code** approach means that users concerned with version control, compliance, continuous integration and deployment, and testing can easily fold infrastructure into their development and release workflow.

Puppet code is a **declarative** language, which means that you describe only the desired state for your systems, not the steps needed to get there. Once you've specified in Puppet code those aspects of a system you want Puppet to manage, a Puppet agent service running in the background makes any changes needed to bring the system into compliance with the desired state.

Puppet's declarative approach requires that you think about the systems in your infrastructure differently than you would if you were using scripts, golden images, or runbooks. Puppet has several advantages that help you spend time focused on **what** your infrastructure can do for you rather than **how** you're going to get there.

**Puppet is portable.** Its declarative language gives you a single syntax for describing desired state across Windows and Unix-like operating systems, network devices, and containers. You don't have to switch languages and toolsets every time you start work on a new system. Learning Puppet gives you a skillset that can be carried across projects and roles.

**Puppet is centralized.** With Puppet's master-agent architecture, there's no need to connect to systems individually to make changes. Once the Puppet agent service is running on a system, it will periodically establish a secure connection to the Puppet master, fetch any Puppet code you've applied to it, and make the changes necessary to bring the system in line with the desired state you described. The fact that centralized control is built in from Puppet's foundations makes monitoring and compliance much easier. Puppet also offers orchestration features that allow you to coordinate changes across multiple systems involved in an application.

**Puppet is community driven.** Joining the [Puppet community](https://puppet.com/community) means you can benefit from the knowledge and code contributed by thousands of other users around the world. Ask questions on [Slack](https://slack.puppet.com/), [IRC](http://webchat.freenode.net/?channels=puppet), or our [Ask](http://ask.puppet.com/) Q&A site. Attend[Puppet User Groups (PUGs)](https://puppet.com/community/user-groups) to connect with other Puppet users in your area. Get a jump start on your Puppet infrastructure with [the Puppet Forge](http://192.168.56.101/en_us/quests/forge.puppet.com), a repository of modules maintained by Puppet and the Puppet community that give you everything you need to manage common applications and services. The Forge has a wide range of modules to help you manage everything from NTP and SQL Server to Minecraft. The base of well tested and reviewed code means that you can get started puppetizing key aspects of your infrastructure right out of the gate.

**Puppet connects you to the cutting edge.** Puppet provides a stable platform for bringing new technologies into production. Integrations with Docker, Kubernetes, Mesos, AWS, vSphere, and others help you engage with next generation tools in a way that's simple, reliable, and consistent.

## The Puppet agent

As we noted above, what we call Puppet is actually a variety of tools and services that work together to help you manage and coordinate the systems in your infrastructure. Though this ecosystem gives you a great degree of power and control, the complexity can leave a new user wondering where to start. So this was the first question we answered as we put together this guide: "Where do we begin?"

By introducing the Puppet agent and some of the command line tools included in the agent installation, we hope to strike the right balance between the big-picture view of Puppet and the the bottom-up fundamentals. You'll begin to understand the agent's role in the broader Puppet architecture, as well as the details of how it interacts with the system where it's installed.

The Puppet agent runs on every system that you want Puppet to manage. The agent serves as the bridge between the system where it's installed and the Puppet master server. The agent communicates in two directions: out to the master to see how its system should be configured, and then inward to native system tools to check the actual state of the system and to make changes to bring it in line with the desired state.

For now, we'll set aside the agent's interactions with the Puppet master and focus on the tools that Puppet uses to check and modify a system's state. When you install the Puppet agent you get a set of command-line tools that help you interact directly with a system in the same way the Puppet agent does. Using these tools will help you understand how the Puppet agent sees and modifies the state of the system where it's running.

## Agent installation

Though we're focusing on the local Puppet agent in this quest, we are still working within the context of the master-agent architecture set up on the Learning VM. The Learning VM itself has Puppet Enterprise pre-installed, including the Puppet master service. For each quest, the quest tool provides one or more agent systems that you explore and configure with Puppet.

For this quest, we've prepared a fresh system where you can install the Puppet agent and explore some of the tools it provides. The Puppet master hosts an agent install script that makes it easy to install the Puppet agent on any system that can access the master.

Task 1:

To get started, use ssh to connect to hello.puppet.vm by running the command below. When prompted, enter the password puppet.

ssh [learning@hello.puppet.vm](mailto:learning@hello.puppet.vm)

Now that you're connected to this system, copy and run the following command to load the agent installer from the master and run it on the agent system:

curl -k <https://learning.puppetlabs.vm:8140/packages/current/install.bash> | sudo bash

You will see text stream across the screen as the installation runs.

Note that if your agent is running on a different operating system than the Master, you will have to take some steps to ensure that the correct installation script available. In this case, our master and agent are both running CentOS 7, so we don't have to worry about this difference. You can find full documentation of the agent installation process, including specific instructions for Windows and other operating systems in the [installation documentation](https://docs.puppet.com/pe/latest/install_agents.html).

## Resources

As we noted above, the Puppet agent comes with a set of supporting tools you can use to explore your system from Puppet's perspective. Now that the agent is installed, let's use these tools to see what they can teach you about how Puppet works.

One of Puppet's core concepts is the *resource abstraction layer*. For Puppet, each aspect of the system you want to manage (such as a user, file, service, or package) is represented in Puppet code as a unit called a *resource*. The puppet resource tool lets you view and modify these resources directly.

Task 2:

Be sure you're still connected your agent system, and run the following command to ask Puppet to describe a file resource:

sudo puppet resource file /tmp/test

What you see is the Puppet code representation of a resource. In this case, the resource's type is file, and its path is/tmp/test:

file { '/tmp/test': ensure => 'absent',}

Let's break down this resource syntax into its components so we can see the anatomy of a resource a little more clearly.

type { 'title': parameter => 'value',}

The **type** is the kind of thing the resource describes. It can be a **core type** like a user, file, service, or package, or a **custom type** that you have implemented yourself or installed from a module. Custom types let you describe resources that might be specific to a service or application (for example, an Apache vhost or MySQL database) that Puppet doesn't know about out of the box. A resource's type points Puppet to the set of tools it will use to manage the resource on your system.

The body of a resource is a list of **parameter value pairs** that follow the pattern parameter => value. The parameters and possible values vary from type to type. They specify the state of the resource on the system. Documentation for resource parameters is provided in the [Resource Type Reference](https://docs.puppet.com/puppet/latest/reference/type.html).

The **resource title** is a unique name that Puppet uses to identify the resource internally. In the case of our file resource that you looked at above, the resource's title was the path of the file on the system: /tmp/test. Each resource type has a unique identifying feature that can be used as the resource title. A user resource uses the account name as a title, for example, and a package resource uses the name of the package you want to manage.

There's a caveat to resource titles, however, that can trip up new users. You can set a file resource's title to something other than the file's path if you also set a path parameter for that resource. For example:

file { 'my\_file': ensure => 'present', path => '/tmp/test'}

A file resource's path will default to the title if the path parameter is not explicilty set. The same pattern applies for other resources, such as a user's account name or package name as we mentioned above. This parameter that defaults to the title is called a **namevar**. You can find more information about the **namevar** for different resource types in the [Resource Type Reference](https://docs.puppet.com/puppet/latest/reference/type.html).

Now that you're more familiar with the resource syntax, let's take another look at that file resource.

file { '/tmp/test': ensure => 'absent',}

Notice that it has a single parameter value pair: ensure => 'absent'. The ensure parameter tells us the basic state of a resource. For the *file* type, this parameter will tell us if the file exists on the system and if it does, whether it's a normal file, a directory, or a link.

Task 3:

Puppet is telling us that this file doesn't exist. Let's see what happens when you use the touch command to create a new empty file at that path. Run:

touch /tmp/test

Now use the puppet resource tool to see how this change is represented in Puppet's resource syntax:

sudo puppet resource file /tmp/test

Now that the file exists on the system, Puppet has more to say about it. It shows the ensure and contentparameters and their values, plus information about the file's owner, when it was created, and when it was last modified.

file { '/tmp/test': ensure => 'file', content => '{md5}d41d8cd98f00b204e9800998ecf8427e', ...}

The value of the content parameter might not be what you expected. When the puppet resource tool interprets a file in this resource declaration syntax, it converts the content to an MD5 hash. This hashing lets Puppet quickly compare the actual content of a file on your system against the expected content to see if any change is necessary. This kind of shortcut means that the way Puppet shows an existing resource on the system and the way that resource would be fully defined in Puppet code's resource syntax can be a little different.

Task 4:

Let's use the puppet resource tool to add some content to our file.

Running puppet resource with a parameter=value argument tells Puppet to modify the resource on the system to match the value you set. (Note that while this is a great way to test out changes in a development environment, it's not a good way to manage production infrastructure. Don't worry, though, we'll get to that soon enough.) We can use this to set the content of your file resource. Run the following command:

sudo puppet resource file /tmp/test content='Hello Puppet!'

Puppet will display some output as it checks the hash of the existing content against the new content you provided. When it sees that the hashes don't match, it sets the file's content to the value of the command's content parameter.

Look at the file to see the modified content by running:

cat /tmp/test

### Types and Providers

This translation back and forth between the state of the system and Puppet's resource syntax is the heart of Puppet's resource abstraction layer. To get a better understanding of how this works, let's take a look at another resource type, the package.

Task 5:

As an example, we'll look at the package for the Apache webserver httpd. Run:

sudo puppet resource package httpd

Because this package doesn't exist on the system, Puppet shows you the ensure => purged parameter value pair. This purged value is similar to the absent value you saw earlier for the file resource, but in the case of the package resource, indicates that both the package itself and any configuration files installed by the package manager are both absent.

package { 'httpd': ensure => 'purged',}

As we mentioned above, each resource **type** has a set of **providers**. A **provider** is the translation layer that sits between Puppet's resource representation and the native system tools it uses to discover and modify the underlying system state. Each resource type generally has a variety of different providers.

These providers can seem invisible when everything is working correctly, but it's important to understand how they interact with the underlying tools.

The quickest way to see the inner workings of a provider is to break it. Tell Puppet to install a nonexistent package named bogus-package by running:

sudo puppet resource package bogus-package ensure=present

The error message tells you that yum wasn't able to find the specified package, and lists the command that Puppet's yum provider tried to run when it saw that the package wasn't already installed:

Error: Execution of '/bin/yum -d 0 -e 0 -y install bogus-package' returned 1:Error Nothing to do

Puppet selects a default provider based on the agent's operating system and whether the commands associated with that provider are available. You can override this default by setting a resource's provider parameter.

Try installing the same fake package again, this time with the gem provider:

sudo puppet resource package bogus-package ensure=present provider=gem

You'll see a similar error with a failed gem command instead of the yum command:

Error: Execution of '/bin/gem install --no-rdoc --no-ri bogus-package'returned 2: ERROR: Could not find a valid gem 'bogus-package' (>= 0) in any repositoryError: /Package[bogus-package]/ensure: change from absent to present failed: Execution of '/bin/gem install --no-rdoc --no-ri bogus-package' returned 2: ERROR: Could not find a valid gem 'bogus-package' (>= 0) in any repository

Now that you know what's happening in the background, try installing a real package with the default provider:

sudo puppet resource package httpd ensure=present

This time, Puppet installs the package. The value of the ensure parameter shows you the specific version of the installed package:

package { 'httpd': ensure => '2.4.6-45.el7.centos',}

When you don't specify a version of the package to install, Puppet defaults to installing the latest available version and displays this version number as the value of the ensure attribute.

Now that you've had a chance to explore this system with the newly installed Puppet agent, exit to return to the Puppet master before you continue to the next quest.

exit

## Review

In this quest, you learned what Puppet is and some of the advantages of managing your infrastructure with Puppet's declarative domain-specific language and master-agent architecture.

You installed the Puppet agent on a new system using an install script hosted on the Puppet master. Once the agent and suite of supporting tools were installed, you learned the fundamentals of Puppet's **resource abstraction layer**, including **resources**, resource **types**, and the **providers** that translate between your Puppet code and the native system tools.

Now that you've learned some of the core concepts behind Puppet, you're ready to use these ideas in a more realistic workflow. The puppet resource command is a great way to explore a system or test Puppet code, but it's not the tool you'll be using to automate your configuration management.

In the next quest, you'll learn how to save your Puppet code to a file called a **manifest** and organize it into a **module**within your Puppet master's **codedir**. This structure lets Puppet keep track of where to find all the resources it needs to manage your infrastructure.

And you'll see how the Puppet agent communicates with your Puppet master to fetch a compiled list of resources called a **catalog** based on Puppet code kept on the master.

# Agent run

## Quest objectives

* Understand the process of a Puppet agent run.
* Learn how to list and sign agent certificates.
* Use the *site.pp* manifest to classify a node.

## Get started

Now, here you see, it takes all the running you can do, to keep in the same place.

* Lewis Carroll

The puppet resource command you explored in the previous quest let you see a system the way Puppet does—through the resource abstraction layer. While exploring and manipulating resources through Puppet's command line tools are useful, the real value of the resource abstraction layer is to provide a single common language for your Puppet master to manage all the systems in your infrastructure.

In this quest, we'll walk through a Puppet agent run to demonstrate how the Puppet agent communicates with the Puppet master server. We'll then write some Puppet code to a manifest on the Puppet master to begin defining a desired state for our agent system.

Ready to get started? Run the following command on your Learning VM:

quest begin agent\_run

## The agent/master architecture

As we mentioned in the previous quest, Puppet is typically used in what's called an agent/master (client/server) architecture.

In this architecture, each managed node in your infrastructure runs a *Puppet agent* service. One or more servers (depending on the size and complexity of your infrastructure) act as *Puppet master(s)* and run the Puppet server service to handle communication with your agents. In a default *monolithic* master installation, the Puppet master server also hosts several supporting services such as the PE console services and PuppetDB. In larger deployments these services may be distributed across other servers to improve performance and provide redundancy.

By default, the Puppet agent service initiates a Puppet run every half-hour. This periodic run ensures that your system stays in the desired state you described in your Puppet code. Any configuration drift that occurs between runs is remediated the next time the agent runs.

We've disabled these automatic runs on the Learning VM's agent systems. Instead, you'll manually trigger runs to get more control and visibility as you learn how Puppet works.

F,{6d3f6333-ddfd-4720-903b-2956ecf7b3d5}{60},13,7.3125

The Puppet agent begins a Puppet run by sending a *catalog request* to the Puppet master. This request includes some information about the agent system provided by a tool called *Facter*.

The Puppet master uses this information from Facter along with your Puppet code to compile a catalog that tells the agent exactly how the resources on its system should be configured.

Your Puppet master keeps a copy of the Puppet codebase you've created to define the desired state for systems in your infrastructure. This Puppet code is primarily made up of resource declarations like the ones you saw in the previous quest. While your Puppet code's final purpose is to define resources, it also includes some language features such as variables, classes, and conditionals that can give you control over which resources you want on a system and how their parameters are set. The master parses this code to create a *catalog*. The catalog is the final list of system resources that define the desired state for an agent node.

The Puppet master sends this catalog back to the Puppet agent, which then uses its *providers* to check if the desired state of each resource defined in the catalog matches the actual state of the resource on the system. If any differences are found, the providers help Puppet implement whatever changes are necessary to bring the actual state of the system into line with the desired state defined in the catalog.

Finally, the Puppet agent generates a report that includes information about unchanged resources, successful changes, and any errors it may have encountered during the run. It sends this report back to the Puppet master, which stores it in PuppetDB and makes it available via the PE console's web GUI.

## Certificates

There's one more thing to note before we can move on to demonstrating a Puppet agent run.

All communications between an agent and master happen over SSL. Before the master communicates with an agent, it needs a way to validate that the agent node is authentic. This prevents unauthorized connections from spoofing an agent node to access sensitive data that might be included in a catalog. While Puppet does provide [options for encrypting data](https://puppet.com/blog/encrypt-your-data-using-hiera-eyaml) within a catalog, it's best to control which systems can make a catalog request in the first place.

Puppet requires any system contacting the Puppet master to authenticate with a signed certificate. The first time a Puppet agent contacts the Puppet master, it will submit a *certificate signing request (CSR)*. A Puppet administrator can then validate that the system sending the CSR should be allowed to request catalogs from the master before deciding to sign the certificate. (You can read more about the details of Puppet's cryptographic security and the certification system on the [docs page](https://docs.puppet.com/background/ssl/index.html))

Task 1:

Start by connecting to your agent system. When you began this quest, the system you used in the last quest was destroyed and a new one was created. This new system has the Puppet agent pre-installed, so there's no need to repeat the installation process. Go ahead and connect to the new agent system with the same credentials you used in the last quest:

**username: learning**  
**password: puppet**

ssh [learning@agent.puppet.vm](mailto:learning@agent.puppet.vm)

First, try to trigger a Puppet agent run without your agent system's certificate signed. The agent will attempt to contact the Puppet master, but its request will be rejected.

sudo puppet agent -t

You'll see a notification like the following:

Exiting; no certificate found and waitforcert is disabled

No problem: you just have to sign the certificate. For now, we'll show you how to do it from the command line. If you prefer a GUI, the PE console includes [tools for certificate management](https://docs.puppet.com/pe/latest/console_cert_mgmt.html).

Task 2:

First, exit your SSH session to return to the your Puppet master:

exit

Use the puppet cert tool to list unsigned certificates:

puppet cert list

Sign the cert for agent.puppet.vm:

puppet cert sign agent.puppet.vm

Task 3:

With that taken care of, your Puppet agent is authorized to make catalog requests.

## Triggering a Puppet run

As noted above, the default for the Puppet agent service is to initiate a Puppet run every thirty minutes. Because it would be hard to clearly demonstrate Puppet with these scheduled background runs, we've disabled the Puppet agent service on your agent system. Instead, you can use the puppet agent -t command to manually trigger a run.

Go ahead and connect to your agent node:

ssh [learning@agent.puppet.vm](mailto:learning@agent.puppet.vm)

Trigger an agent run. Now that the agent's certificate is signed, it will receive a catalog from the Puppet master.

sudo puppet agent -t

While you haven't yet told Puppet to manage any resources on the system, you'll see a lot of text scroll by. Most of what you see is a process called [pluginsync](https://docs.puppet.com/puppet/latest/plugins_in_modules.html#auto-download-of-agent-side-plugins-pluginsync). During pluginsync, any extensions installed on the master (such as custom facts, resource types, or providers) are copied to the Puppet agent before the Puppet run continues. This ensures that the agent has all the tools it needs to correctly apply the catalog.

This pluginsync process adds a lot of clutter, but we'll focus on three lines that look like the following. If you'd like to see these lines with less clutter, trigger another agent run.

Info: Loading factsInfo: Caching catalog for agent.puppet.vmInfo: Applying configuration version '1464919481'

This output shows you one side of the conversation between the agent and master we discussed at the beginning of this quest.

You see that the Puppet agent loads the facts it needs to share the details of the system where it's running with the Puppet master.

Next, you see when the agent has received a catalog because it tells you when it caches a copy of the new catalog. (The Puppet agent can be configured to fail over to this cached catalog if it is unable to connect to the master.)

Finally, the Puppet agent applies the catalog. Normally after this step, you would see a list of all the changes made by the agent. In this case, however, the Puppet master didn't find any Puppet code to apply to your agent node, and it didn't make any changes (other than those involved in pluginsync) during this run.

## Classification

To make something more interesting happen, you'll have to specify a desired state for some resources on the agent.puppet.vm system.

Task 4:

Remember, the Puppet code you use to describe how you want a node to be configured is kept on the Puppet master. End your SSH session on the agent.puppet.vm agent node to return to the Puppet master:

exit

Before diving in and writing some Puppet code, let's take a moment to go over the catalog compilation process from the Puppet master's perspective. This will help you understand exactly what you're doing as you write code to apply to your agent.

When the Puppet server service on the Puppet master receives a catalog request with a valid certificate, it begins a process called *node classification* to determine what Puppet code will be compiled to generate a catalog for the agent making the request.

There are three different ways to handle node classification.

1. The site.pp manifest is a special file on the master where you can write node definitions. This is the method we'll be using now and in several of the following quests. It gives you the most direct view of how node classification works.
2. The PE console includes a GUI node classifier that makes it easy to manage node groups and classification without editing code directly. Though this is a very efficient way to manage node classification, you'll understand it best after you're familiar with some of the underlying Puppet concepts.
3. Finally, if you want to customize node classification, you can create your own [external node classifier](https://docs.puppet.com/guides/external_nodes.html). An external node classifier can be any executable that takes the name of a node as an argument and returns a YAML file describing the Puppet code to be applied to that node. This is an advanced topic and is not covered in this guide.

## The site.pp manifest

When a Puppet agent contacts the Puppet master, the master checks for any node definitions in the site.pp manifest that match the agent system's name. In the Puppet world, the term "node" is used to mean any system or device in your infrastructure, so a node definition defines how Puppet should manage a given system.

It will help to understand what a node definition looks like with an example. Go ahead and open your site.pp manifest:

vim /etc/puppetlabs/code/environments/production/manifests/site.pp

Scroll past the comments and default node definition to the bottom of the file. (In Vim, you can type G to jump the the bottom of a file) This is where you'll create a new node definition for the agent.puppet.vm system. The outline of the node definition should look like this:

node 'agent.puppet.vm' {}

Normally you would include one or more class declarations in this node block. A class defines a group of related resources, allowing them to be declared as a single unit. Using classes in your node definitions keeps them simple and well organized and encourages the reuse of code. We'll cover the details of classes in the next quest. For now, however, we'll take a shortcut and write a resource declaration directly into your node definition. In this case, use a resource type called notify that will display a message in the output of your Puppet run without making any changes to the system.

Add the following notify resource to your node definition. (You'll probably learn Puppet code syntax more quickly if you type out your code manually, but if you prefer to paste content into Vim, you can hit ESC to enter command mode and type :set paste to disable the automatic formatting. Press i to return to insert mode before pasting your text.)

node 'agent.puppet.vm' { notify { 'Hello Puppet!': }}

Remember, use ESC then :wq to save your file and exit Vim.

You may notice that this resource declaration doesn't include any parameters. The only feature of this notifyresource we care about is the message it displays. If it's not set explicitly by the message parameter, this message will default to the resource title. This lets us save some time by leaving out the parameter value pairs and using the title to define the message we want the resource to display. (This parameter that uses the resource title as its default is called a **namevar**. You can read more about the role of the namevar in the [Puppet docs](https://docs.puppet.com/puppet/latest/lang_resources.html))

Now that you have a concrete example of a node declaration, let's return to our review of the agent run process from the master's perspective. When the agent contacts the master, the master finds a matching node definition in the site.ppmanifest and uses the Puppet code contained in that node definition to compile a catalog.

The master sends that compiled catalog to the agent, which then applies the catalog to the system where it's running. In this case, the catalog only includes a notify resource, so the agent will display the specified message as it applies the catalog, but no changes will be made to the system.

Task 5:

Now that you have some Puppet code for the master to parse and return to the agent, trigger another Puppet run.

SSH to your agent node:

ssh [learning@agent.puppet.vm](mailto:learning@agent.puppet.vm)

Use the puppet agent tool to trigger a Puppet run:

sudo puppet agent -t

The output will include something like this:

Notice: Hello Puppet!Notice: /Stage[main]/Main/Node[agent.puppet.vm]/Notify[Hello Puppet!]/message: defined 'message' as 'Hello Puppet!'Notice: Applied catalog in 0.45 seconds

Now disconnect from your agent node.

exit

## Review

We began this quest with a discussion of Puppet's *agent/master architecture* and the communication between the Puppet master and an agent. The agent begins this process by sending a *catalog request* to the master. The master first checks to see if the agent has a valid *certificate*. If the certificate is valid, the master consults several methods of *classification* to begin the process of catalog compilation. In this quest, we used a *node definition* in the site.ppmanifest to classify our node. The master then compiles a catalog that it returns to the agent. The agent checks if the current state of its system matches the desired state described in the catalog and makes any changes necessary to bring it in line. For the sake of simplicity in this quest, we used a notify resource to display a message rather than make any changes to the system. Once the agent has applied the catalog (or failed to apply the catalog if an error occurs), it sends a report of the run's results back to the master, which stores it in the PuppetDB.

Now that we've introduced the resource abstraction layer, the agent/master communication involved in a Puppet run, and a simple example of classification with the site.pp manifest, you've seen the foundations that the rest of Puppet is built on.