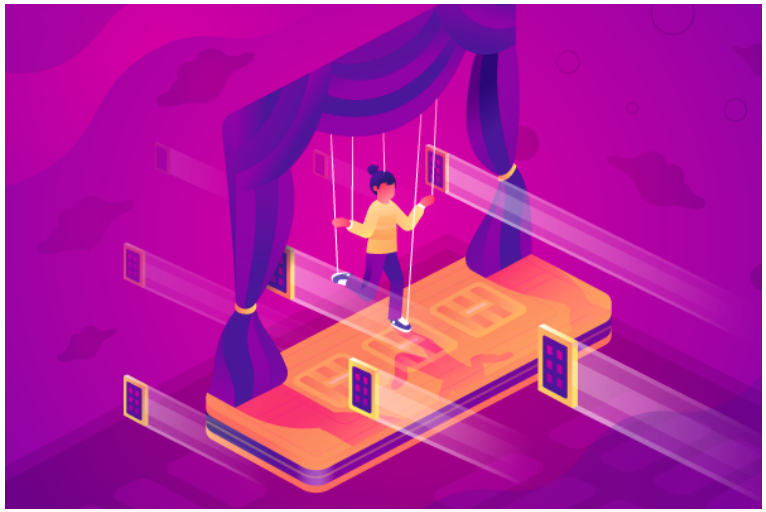
Study Guide



Puppet Professional Certification - PPT206 Study Guide

Elle Krout
elle@linuxacademy.com
July 31, 2019

Linux Academy

Contents

Idempotence

Before we delve into using Puppet, it might be best to clarify what Puppet is; and to clarify what Puppet is, we need to explore some core terms and concepts. The first of these is the trait of *idempotence*.

Idempotence is a property of an operation. In this case, it's something called a "Puppet agent run." If an operation is idempotent, it will enforce a change the first time it is run, but make no additional changes no matter how many times it's run afterward. So if we have an operation that installs `sysstat` the first time it's run, then does nothing the next time because `sysstat` has already been installed, it's idempotent.

To further clarify, look at a Bash script that installs `sysstat`:

```
#!/bin/bash

apt install sysstat -y
```

It is *not* idempotent. Although when we initially run the script, `sysstat` is installed, and it remains installed through each subsequent rerun. Because our server attempts to run `apt install sysstat -y` *every time* the script is run, it cannot be idempotent.

In contrast, what happens when we try to accomplish this same task via Puppet? Here we have an example Puppet *manifest*, or container of Puppet code, that also installs `systat`:

```
class sysstat {
  package { 'sysstat':
    ensure => installed,
  }
}
```

Don't worry about the details of this yet. All we need to know is that when I assign this manifest to a node, and perform a Puppet run with a `puppet agent -t`, the `sysstat` package is installed:

```
# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Retrieving locales
Info: Loading facts
Info: Caching catalog for ellejaclyn2c.mylabserver.com
Info: Applying configuration version '1554750720'
Notice: /Stage[main]/Sysstat/Package[sysstat]/ensure: created
Notice: Applied catalog in 21.95 seconds
```

While if I were to run it a second time, no changes would even attempt to take place:

```
# sudo puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Retrieving locales
Info: Loading facts
Info: Caching catalog for ellejaclyn2c.mylabserver.com
Info: Applying configuration version '1554750791'
Notice: Applied catalog in 15.94 seconds
```

See how the `Notice: /Stage[main]/Sysstat/Package[sysstat]/ensure: created` line is no longer there? This is because Puppet knew it did not need to attempt to install `sysstat` again. Our Bash script would try again and again, even though the output will tell us every time that `sysstat` has already been installed.

This characteristic of idempotency is the defining trait of a configuration manager tool, and Puppet is one of those. Configuration management tools let us describe our desired end state systems using a declarative language that is

then applied universally across the systems that need it. This eliminates configuration drift and allows us to ensure our systems are consistent throughout their lifecycles. This is done by abstracting away the commands we usually think of when we set up our servers (all the OS-specific commands, as well as any universal ones) and allowing us to write one single "description" per desired task. So even if we're installing Apache on three different operating system flavors, we're going to be doing so using the same description every time.

In Puppet, all of this is done through something called the "resource abstraction layer," and we'll talk about that next.

Wrap Up

We learned some core traits of configuration management systems, including:

- Configuration management systems are idempotent.
- An idempotent operation will make changes the first time it is run, but not any repeating times.
- In Puppet, our end state descriptions are idempotent.
- Ideally, this description can be used to manage servers regardless of OS or distribution.

Now that we have an understanding of these concepts, we can build off this to explore *how* Puppet achieves idempotency, and see what the language we will be using to actually configure our servers looks like.

Resource Abstraction

In our last section, we touched on how, as a configuration management tool, Puppet lets us describe our desired end state and then enforces those changes without us ever having to, say, write a Bash script. This is because we abstract away all the commands we usually run to configure our server in favor of using what Puppet calls *resources*, which are part of the aptly-named *resource abstraction layer* (RAL).

The RAL is what allows us to provide one module to configure something across all desired servers, regardless of operating system or distribution. After all, think of all the ways our servers can vary. We have different init systems, such as System V and systemd. We have different package managers, like `apt`, `yum`, `dnf`, and `zypper`. We even have different providers for things that sometimes seem similar or the same, like adding a user (which might just be `useradd` on many systems, but differs in Windows), and in instances where you're doing things like running an LDAP server.

So let's now consider the *resource* part of the RAL by looking at the manifest we saw before:

```
class sysstat {
  package { 'sysstat':
    ensure => installed,
  }
}
```

This code uses the `package` resource type. This was written in an Ubuntu 18.04 server, yet we aren't limited to using it only on Ubuntu. The package resource works as a wrapper to the actual package managers doing the work on the servers: the `yum`s and `apt`s, the `pacman`s and `gem`s. We can see a full list of supported package managers by using the `puppet describe` command:

```
# puppet describe package
```

```
...

Providers
---------
    aix, appdmg, apple, apt, aptitude, aptrpm, blastwave, dnf, dpkg, fink,
    freebsd, gem, hpux, macports, nim, openbsd, opkg, pacman, pip, pip3,
    pkg, pkgdmg, pkgin, pkgng, pkgutil, portage, ports, portupgrade,
    puppet_gem, rpm, rug, sun, sunfreeware, tdnf, up2date, urpmi, windows,
    yum, zypper
```

Puppet has abstracted all of that away. We don't need to know the nuances of any of these package managers, because we can rely instead on what this abstraction layer has set up for us. This is done through the use of *providers*, which is the underlying code that Puppet uses to ensure our end-state code can be implemented on the desired system. So when Puppet runs that `sysstat` installation code on Ubuntu 18.04, it will know to use the `apt` provider.

Of course, `package` is not the only resource type we have available to us. In Puppet 6, there are 11 of these core resource types, with additional ones that can be added as needed. On Puppet 5.5, which the oldest version of Puppet that is tested on, this number is 48. Of the removed 37 resource types, 13 become available once we install the `puppet-agent` package on the servers we want to configure, three are available on the Puppet Forge (the repository of additional Puppet code and modules) and the remaining 21 have been depreciated. The majority of those were Nagios-related types.

We'll get into detail about how to use these 24 included modules, and how to add a new one, in future sections. But until then, we now want to consider how Puppet actually accomplishes the configuration management issues it sets out to solve.

Wrap Up

In this section, we took a high-level look at how Puppet works in regards to configuration management, learning:

- What the resource abstraction layer is and how it simplifies our work

- How the resource abstraction layer works through the use of resource types
- What a resource type looks like
- How providers exist beneath our resource types to ensure our systems are configured using the correct tools

Now that we know how Puppet works on a basic level, however, we want to take a step deeper to look at how we'll set up our systems to work with Puppet and how Puppet actually goes about taking our end states and making them a reality.

Architecture

At it's most basic, Puppet works through the use of a "master-agent" setup. A primary server, called the Puppet master, stores configuration information that managed servers can request and receive to make changes based upon. These managed servers, of course, are the *agents* or *nodes*, and these are the servers that actually host our applications, web servers, monitoring setups, etc.

Regardless of whether you are using Puppet Open Source or Puppet Enterprise, every Puppet setup should contain a master and some agents. Although previous versions of Puppet permitted the use of standalone architectures, where a Puppet agent would store and compile its own configurations, PuppetLabs ceased support of this architecture with Puppet due to maintenance and security concerns. All you really need to know about it is that it used the *Puppet apply* application to compile and manage nodes, usually run as a `cron` job.

Master (Open Source and Enterprise)

A Puppet master, either the Enterprise and Open Source version, is comprised of at least two components: The Puppet Server and the certificate authority.

The Puppet Server is a Java Virtual Machine application that powers the *catalog converter*, which is what helps take our Puppet code and configure our managed nodes. We have a whole section about this next!

The certificate authority is an internal CA service that manages our master's certificates, as well as certificate requests from agents. Certificates are just `.pem` files stored in the `/etc/puppetlabs/puppet/ssl/certs` directory (although we could update this location at `/etc/puppetlabs/puppetserver/conf.d/ca.conf`).

We also have the option to include PuppetDB. This is added automatically in the Enterprise version, but we have to download it manually if we're using Open Source . PuppetDB collects Puppet-generated data and lets us use advanced Puppet features. And any other application that needs to use Puppet data can access it too.

Agents (Open Source and Enterprise)

Puppet agents are also the same regardless of version. An agent runs the Puppet agent application as a background service and also contains *Facter*, a system profiling library that reports facts about our system to our Puppet master that we can then use in our Puppet code. Both the agent application and Facter work off port 8140, and any Puppet masters also contain the Puppet agent application and Facter.

Master of Masters (Enterprise)

While all of the above is true for both Open Source and Enterprise Puppet, PE contains a number of additional features, as well as what is called a "Master of Masters."

For the setup we used in this course, we just had a single Puppet Enterprise master, which contained all the functionality of a "Master of Masters," even if there were no additional masters.

A Puppet Enterprise Master of Masters contains the Puppet Server and certification authority just as the Open Source version does, and always ships with PuppetDB. However, it also contains an RBAC and activity service, a node classifier service, an orchestration service, Code Manager, the console UI. It's also got a file sync client used to communicate with additional compile masters, in case that functionality is required.

We've already discussed the function of the Puppet Server, so let's consider the RBAC service. Role-based access control lets us manage user permissions, such as whether or not a user can change certain parameters around our agents. RBAC can also connect to OpenLDAP and Active Directory, should we need to import or export users and groups. Permissions are tied to the authentication tokens generated by PE, and the activity service logs changes to RBAC entities.

The Puppet node classifier service is built into the Enterprise console and allows us to assign *classes* to nodes, with a class being a named block of Puppet code. To clarify, the names are things we, the users, define. In our previous example, I used a class called `sysstat`.

The orchestration service is the toolset that lets us use Puppet Application Orchestration (deprecated in PE 2019.0) and the Puppet orchestrator. The Application Orchestrator lets us manage multi-tiered applications by setting dependencies

and ensuring all levels of the application are deployed in the correct order across nodes. The orchestrator lets us perform on-demand Puppet runs.

Additionally, Code Manager is paired with r10k and lets us manage and deploy our Puppet code across multiple masters, as well as manage environments and install modules. Code Manager is used alongside Git and automatically syncs our code so there are no discrepancies between masters.

The Console UI is website interface for Puppet Enterprise, where we can view our nodes in real time, access reports and data, manage our node classifications, and change our user access.

Finally, the file sync client works with Code Manager to ensure any Puppet code on our Master or Masters gets to the masters that work under it.

Compile Masters

The combination of the Puppet Server application and a file sync *server* makes up a *compile master*. Should we need a multi-master setup (PE supports up to 4,000 nodes, with each additional compile master increasing capacity by 1,500 to 3,000 nodes, depending on PuppetDB's remaining capacity) we would add a compile master, not a full Master of Masters.

Wrap Up

In this section, we learned the difference between Puppet Open Source and Puppet Enterprise, and viewed how PE builds off Puppet OS to provide a full-featured platform for managing our nodes. We learned also that:

- A Puppet master coordinates the desired end state of a server with the Puppet agent.
- The Puppet server is the component of the Puppet master that compiles our Puppet code for use.
- All nodes managed by Puppet run the Puppet agent application, including the Puppet master itself.
- The Enterprise version of Puppet uses a "Master of Masters," which contains additional tools that allow for orchestration, code management, role-based access control, and more.

- Puppet Enterprise compile masters run similar to Open Source Puppet, except that they also contain a file sync server that receives updated Puppet code from Code Manager.

We'll get more in-depth with each of these tools as we move along, starting in the our next section about catalog compilation.

The Lifecycle of a Puppet Run

Now that we understand the purpose of Puppet and how Puppet is set up to manage our servers, the question that remains is *how* does Puppet take our end states and actually process them to configure our servers? We know that the resource abstraction layer is what lets us use Puppet code for our descriptions, and that providers are the part of the underlying code that does the work translating resources, but there are still a lot of details here we're missing.

When we want to enforce our changes, or perform a *catalog compilation*, we can either wait for when a Puppet agent run occurs periodically or we can force it. We saw that earlier in this course when we viewed a `puppet agent -t`.

When running that `puppet agent -t` previously, we saw that it installed `sysstat`. This was because our *catalog* only contained a single *module*, with our catalog being the list of modules we want to run on a specific server, and with a *module* being a set of Puppet code written to perform a specific overall task (like installing and configuring Apache).

So what exactly happens when a catalog compilation occurs?

First, the node performs a handshake with the master, exchanging certificate information and creating a connection. It then requests a node object from the master.

Once this request is made, the master has to prepare the *node object*, which is a combination of information about a node: its facts, environment, parameters, and classes. The node object is prepared using the *node terminus*, which is a pluggable backend that hosts a key-value store. By default, the `plain` node terminus is used, which is just a blank node object. Other node termini exist, with the most common one being the `exec` node terminus. It will pull in *external node classifier* (ENC) data to its node object. ENC data is simply additional data telling a node which classes it should have. There is also an `ldap` node terminus, which pulls data from, you guessed it, LDAP.

Back on the agent, it receives this node object, and switches environments based on the data (if any). It then requests a catalog. When it does this, it provides the master with any facts recorded by Facter, and the environment associated with the node.

Next, Puppet looks for any variables in the catalog, and sets them using a combination of data provided from the node object, facts, and the certificate. These are set as top-scope variables that can be used by any modules. Any additional variables provided by the master are also set.

Now Puppet evaluates the *main manifest*, which is where we map our modules to our nodes. Puppet will parse this file, searching for any mappings that match the node's name. Any data here that can override our top-scope variables will do so. If Puppet cannot find any matching names data, the catalog converge will fail.

Any modules assigned to our node are now evaluated. Remember, modules are a collection of Puppet code related to one overall configuration, alongside any classes included in the modules themselves. Say a module requires we install a database and pulls in a MySQL Puppet module we didn't manually assign our node. This is where our providers come in. The evaluated modules are then added to the catalog.

Finally, any additional classes specified by the node object are evaluated and added to the catalog. With the catalog finished, it gets sent to the agent, which then enforces the changes dictated by the catalog itself. If enabled, the agent will also send a report to the master upon completion.

Wrap Up

We went through all the steps that occur during a Puppet run, including:

- The agent and master complete a handshake and create a connection.
- The agent provides the master with facts about its system from Factor.
- The master provides the agent with a node object, which it uses to update parameters such as environment, if needed.
- The master provides the agent with a compiled catalog.
- The agent applies the catalog and returns a report (if reporting is enabled).

And now that we know what Puppet's doing under the hood, we can start to get hands on and set up our environment.

Master of Masters

We can now finally get hands-on with Puppet! As mentioned in our previous lesson, we're using an Ubuntu 18.04 server as our Master of Masters. Go head and queue that up in Linux Academy's **Cloud Playground**. We have to set our server size to **large** in order to support Puppet Enterprise.

Speaking of PE's requirements, the system stats for our Master of Masters is going to depend heavily on how many nodes we're managing, and whether we're using a monolithic or multi-master install. The free version of Puppet Enterprise is locked at 10 nodes. But as we learned in our last lesson, the paid version can support up to 4,000 nodes. So, on a monolithic setup, these are our requirements:

Node volume	Cores	RAM	/opt/	/var/	EC2 Instance
< 10	2	6 GB	20 GB	N/A	m5.large
Up to 4,000	16-	32 GB -	100 GB	10 GB	c4.4xlarge

This changes if we're using compile masters, however. For those, the Master of Masters needs to meet the following requirements:

Node volume	Cores	RAM	/opt/	/var/	EC2 Instance
4,000 - 20,000	16	32 GB	150 GB	10 GB	c4.4xlarge

And each compile master requires:

Node volume	Cores	RAM	/opt/	/var/	EC2 Instance
1,500 - 3,000	4	8 GB	30 GB	2 GB	m5.large

Installing Puppet Enterprise

In this course, we're using Puppet Enterprise 2018.1.7, which is the latest release of the LTS version of Puppet, 2018.1. The Puppet Professional tests on version 2018.1 and later, so we'll be using a version that isn't outdated, but still falls in line with the exam. As mentioned previously, Puppet Enterprise 2019 had also been released, although it is a short-term cycle release.

Once your Cloud Playground server is ready, log in with SSH. We're going to get started by ensuring our host name is set up to work with Puppet within the restrictions of our Playground server. Open up `/etc/hosts` and add the full-qualified domain name to the loopback address:

```
$ sudo $EDITOR /etc/hosts

127.0.0.1 USERNAME#c.mylabserver.com localhost
```

Save and exit the file.

We now want to pull down the `.tar.gz` file that contains the Puppet installer:

```
$ wget --content-disposition 'https://pm.puppetlabs.com/puppet-enterprise/2018.1.7/puppet-enterprise-2018.1.7-ubuntu-18.04-amd64.tar.gz'
```

And expand the `tar` file:

```
$ tar -xf puppet-enterprise-2018.1.7-ubuntu-18.04-amd64.tar.gz
```

Move into our Puppet installer directory:

```
$ cd puppet-enterprise-2018.1.7-ubuntu-18.04-amd64/
```

And run the installer:

```
$ sudo ./puppet-enterprise-installer
```

At this point, we're presented with two options: using the text-based installer or the graphical installer, which will spin up a temporary web server at port 3000. If this was PE 2019, we'd also have the option to use an "express installer," which auto-populates the options based on our system's current setup.

Let's go ahead and select the text-mode installation for right now:

```
How to proceed? [1]: 1
```

We're taken to a Puppet Enterprise configuration file, where we're asked to set the administrator password and host name. This file is written in HOCON, a human-readable superset of JSON. Fill out the `"console_admin_password": ""` line to set a password of your choosing. We can leave the rest of the file as-is. But, there are additional options we can set here.

Specifically, if we wanted to separate our PuppetDB and web console from the master itself, we can use the `puppet_enterprise::puppet_master_host`, `puppet_enterprise::console_host`, and `puppet_enterprise::puppetdb_host` settings, which lets us define the fully-qualified domain names for the hosts we want to use for each component. These overall components, as well as Puppet orchestrator, also each have their own set of configuration parameters, which we can view **in the docs**. Using these, we can manually set our database options, PostgreSQL parameters, Code Manager and r10k settings, and general console and orchestrator parameters.

Since we're not using a split configuration, however, we can go ahead and save and exit.

We're now asked if we would like to continue installation using the `pe.conf` file we just created. This file is located at `conf.d/pe.conf` in the same file as the Puppet installer. To specify this or any `pe.conf` file at the time on installation, we can use the `-c` flag with our `puppet-enterprise-installer` command.

Type `n` to exit the installer. While there's nothing wrong with the file we just generated, let's go ahead and trigger a graphical install now to view the two different methods:

```
$ sudo ./puppet-enterprise-installer
```

This time, type `2` to choose the graphical-mode install. After an initial process where the PE repositories are added and packages downloaded, the temporary web server starts and we're asked to move to port 3000 on our server from our web browser. Note that we *must* use HTTPS here:

```
https://USERNAME#c.mylabserver.com:3000
```

From there, we're taken to the PE installer's landing page. Click **Let's get started** to begin.

If you notice, we're asked almost the same things as in our text-mode installer, except Puppet already assumes the FQDN is set properly because you're able to access this page. Instead, we're prompted to set any DNS aliases, with the default of `puppet` being pre-populated. This would have been automatically configured, had we followed through with our text-mode install.

Go ahead and set the administrator password now, then click **Continue**.

From here, we're asked to review the configuration. We're also provided with a link to the `pe.conf` file that this generates with the small print at the bottom. It should look something like:

```
########################################
#
# Puppet Enterprise 2018.1 installer config file
# Flavor: "monolithic"
# Created: 2019-04-15 15:20:31 UTC
# https://docs.puppet.com/pe/2018.1/install_pe_conf_param.html
#
########################################
{
```

```
  "console_admin_password": "PASSWORD",
  "puppet_enterprise::puppet_master_host": "%{::trusted.certname}",
  "pe_install::puppet_master_dnsaltnames": [
    "puppet"
  ]
}
```

After reviewing the settings, click **Continue** to start. Note that if you're using our Playground servers, you'll be met with a warning about the size of our `/opt/` directory, which requires more space in production environments. Since we're only managing two nodes alongside our Master of Masters, however, we can ignore this and **Deploy now**.

The Puppet Enterprise installation process begins. This should take around eight minutes, and we can view the process live by clicking on the **Log View**, if we want.

Once installation has finished, we can **Start using Puppet Enterprise**. Log in using the username **admin** and the password you just set.

We now have Puppet up and running!

Wrap Up

In this section, we grabbed a demo version of Puppet Enterprise 2018.1.7 for us to use with 10 nodes, learning:

- The system requirements for various Puppet setups
- How to use both the text-mode and graphical-mode installer
- What a `pe.conf` file is and how we can alter our Puppet Enterprise installation by providing onee
- How to log in to our Puppet console

Next, we have set up our agent nodes so our Puppet master can start moving the strings.

Restarting the Master of Masters

This section is specifically geared towards anyone running their setup for this course on our Cloud Playground. Anyone using an environment where they'll be turning their Master of Masters off while not studying may also find they benefit.

When restarting the server that hosts your Puppet Enterprise setup, you may find the server itself is accessible before the Puppet Enterprise Console is ready. This means, when you visit your `mylabserver.com` link, you'll be met with a broken page -- either as an error from your web browser, or an Nginx bad gateway error. Most of the time, you just have to wait for a couple of minutes. However, if there *was* an error during startup, you can manually restart your Puppet service by running the following series of commands, in order:

```
# sudo systemctl stop pe-puppetdb
# sudo systemctl stop pe-puppetserver
# sudo systemctl stop pe-console-services
# sudo systemctl start pe-puppetdb
# sudo systemctl start pe-puppetserver
# sudo systemctl start pe-console-services
```

Note that you may also have to wait a few minutes for everything to spin up when restarting this way, as well.

Puppet Agents

With our Master of Masters set up, which I will now refer to as simply the "master," all that's left to get our Puppet environment working is adding some nodes to manage. Of course, before we do that, we need some servers to work with. On the Cloud Playground (or your own environment), go ahead and spin up two additional servers: one CentOS 7, and one Ubuntu 18.04. Log in to each once they are up and running.

On both servers, update the `/etc/hosts` file so the FQDN on the master is mapped to its *private* IP:

```
PRIVATEIP   USERNAME@.mylabserver.com
```

Save and exit when done.

Ubuntu 18.04

We'll be working with the Ubuntu 18.04 agent first, because it has the same architecture as our master. To retrieve the command we'll use for installation, we have to log in to our Puppet Enterprise console, and go to the **Unsigned Certs** page.

From here, all we have to do is copy and paste the provided `curl` command onto our Ubuntu 18.04 server's terminal:

```
$ curl -k https://USERNAME#c.mylabserver.com:8140/packages/current/install.bash | sudo bash
```

Installation should take no more than two minutes. Once it's finished, we can refresh the console.

We're now presented with a certificate signing request. The node name of our server is listed, as is the fingerprint. To check the fingerprint against the server, we can run:

```
# sudo puppet agent --fingerprint
```

Assuming the fingerprint matches, click **Accept**.

CentOS 7

Since our CentOS 7 server uses a different distribution than our master, we have to do a little more footwork to get that one-command installation link. From the console, move to the **Classification** page. We're going to be getting a little hands-on with node classification features from here.

From this page, we can see we have one primary node group, called *All Nodes*, which, as you may have guessed, encompasses all our nodes. Underneath the *All Nodes* group, we have two subgroups: *All Environments* and *PE infrastructure*. We want to **expand** the **PE infrastructure** group. Click on the subgroup **PE Master**.

We're now taken to the classification page for our PE Master group. Don't worry about the Rules, Variables, or Activity pages for now. We'll get into those in our actual classification lessons. **Matching nodes** simply shows us what nodes are part of this group: our Puppet master.

Configuration is the tab we're most interested in, however. This is where we define what bits and pieces of Puppet code we want applied to everything in this class. In this case, we want to add the code needed to configure a CentOS 7 agent.

At the **Add new class** input, search `pe_repo::platform`. The search results return a number of platforms that PuppetLabs maintains repositories for. Since we're on CentOS 7, we'll be looking for the EL 7 repositories: `pe_repo::platform::el_7_x86_64`.

Click **Add class** once you've found the correct class. Notice the blue bar that pops up at the bottom of our console. When we make any changes in our PE console, we have to confirm them by committing the changes. Think of it like using Git, our PE console has access to every bit of our infrastructure that it manages, and we need to be cautious about our changes. **Commit 1 change** when ready.

Now we need to force a Puppet run. We can do this by running `sudo puppet agent -t` via the terminal, but since we already have our console up, let's instead click on **Nodes** and select the Puppet master from the list.

Select **Run Puppet** at the top of the console. Leave all the options unchecked, then click **Run**. Wait for the run to finish.

Once it's done, click on **Unsigned certs** again and copy the `curl` command. Run it on the CentOS 7 server:

```
# curl -k https://rabbitheart1c.mylabserver.com:8140/packages/current/install.bash | sudo bash
```

As with Ubuntu, this should take no more than a few minutes.

When it finishes, view the certificate fingerprint:

```
# sudo puppet agent --fingerprint
```

Then return to the console. Refresh the page and compare the fingerprint against the one that is now listed on the Unsigned Cert page. **Accept** the CentOS 7 server.

Now that our nodes have been accepted into our architecture, the Puppet agent on each server will automatically request a catalog every 30 minutes.

Wrap Up

In this section, we learned how to:

- Install the Puppet agent on a server of the same architecture as the master
- Add additional architecture types for installation
- View the certificate fingerprint on a node
- Accept certificates from the PE console

We also learned that the agent process will query our master for a catalog every thirty minutes. In our next lesson, we'll be looking at how to further configure our Puppet setup depending on our needs.

Configuration: All Nodes

At this point, we have a usable Puppet infrastructure up and running. It can be tempting, now, to jump right into module creation; however, that would be leaving out one vital skill: The ability to configure our Puppet infrastructure for circumstances that are less ideal than spinning it up on some brand new servers outside of an existing infrastructure.

We talked a little about configuring the installation file itself, but we can also configure an existing Puppet infra, with over 100 configuration options. We won't be going through these individually, although a reference list is linked at the end of this section. Instead, we'll be looking at the different ways to configure Puppet overall, as well as the `puppet.conf` file and some core configurations.

Main Settings

We already know that when our Puppet Enterprise master starts up, a number of services start as well, including PuppetDB and the Enterprise console. Most of these have their own directories in `/etc/puppetlabs`. For example, should we need to view the configurations for console, we can find them in `/etc/puppetlabs/conf.d`. All configuration files are only loaded once, at service start, and if we make any changes, we'll need to restart the service.

However, most of our settings changes will not be in these files. Instead, when we configure Puppet, we'll be working primarily in two places: `/etc/puppetlabs/puppet/puppet.conf` and the general `/etc/puppetlabs/puppet/` directory. Let's actually look there now:

```
$ cd /etc/puppetlabs/puppet/
$ ls
auth.conf        fileserver.conf       puppet.conf    ssl
autosign.conf    hiera.yaml            puppetdb.conf
classifier.yaml  hiera.yaml.dpkg-dist  routes.yaml
```

The `puppet.conf` file has long been the primary Puppet configuration file, and the majority of additional files were separated either because they contain complex data structures, or because additional extensions cannot be added to the `puppet.conf` file.

Agent Settings

Let's go ahead and view what the current configuration file looks like on one of our nodes. Remember these settings will also work on our master, since it is an agent of itself:

```
# sudo $EDITOR /etc/puppetlabs/puppet/puppet.conf
```

Currently, we have two settings: `server` and `certname`. `server` provides the name of the Puppet master (on Open Source Puppet this defaults to `puppet`), while `certname` is the name of the server itself. Most often this is the FQDN, but if we wanted to set it to something custom, this is where it would be done.

Beyond this, however, there are some "core" settings Puppet specifically calls out to:

`environment`

One important value that isn't provided, but that we'll often have to set, is the `environment` variable. Let's actually go ahead and add this to our node:

```
environment = production
```

Since this is the default setting, it doesn't matter if we add it to our second node.

`sourceaddress`

We should be also aware of the `sourceaddress` option if we have a host with multiple addresses (multihomed). The address that the server should be using should be supplied as the value.

`runinterval`

To change how often the agent triggers a Puppet run, we can set the `runinterval` option to our desired time. The time can be formatted as seconds (`30` or `30s`), minutes (`30m`), hours (`1hr`), days (`1d`), and even years (`1y`).

`waitforcert`

Should the agent initiate a handshake and no response comes back, the agent will attempt to reconnect. To disable this behavior, we can set this to `false`.

`noop`

The remaining settings, including this one, change the behavior of the Puppet run. `noop`, specifically, sets it so the agent only tests if changes will actually be made of the server; it does not make the changes. This defaults to `false` and can actually be used within our Puppet code, as well.

`priority`

`priority` lets us set the "nice"ness of the agent service in the event we have concerns about it taking away system resources that will affect the server's functioning.

`report`

This lets us set whether or not the agent returns a report to the master. The default is `true`.

`tags`

When we write our Puppet code, we can use specify certain tags within portions of our code. When we use these in conjunction with `tags` in our configuration, we can ensure only the code that has the appropriate tags is run.

`trace`, `profile`, `graph`, and `show_diff`

These options all exist to help us debug the agent. `trace` specifically lets us print stack traces, `profile` enables experimental performance profiling, `graph` creates `.dot` graph files, and `show_diff` logs diffs when files are being replaced. These all default to `false` and must be enabled when needed.

`usecacheonfailure`

This setting dictates Puppet's behavior should a catalog fail. If it is set to `true`, Puppet will fall back to the last successful catalog.

`ignoreschedules`

Like `tags`, `ignoreshedules` reflects back to options we have when writing our Puppet code. Within our code, we can schedule when it is permitted to run. If we set this to `true`, these schedule settings will be ignored.

`prerun_command` and `postrun_command`

These settings let us set a command to run before and after each Puppet run. If this command fails the Puppet run will fail.

Wrap Up

This settings-heavy section took us through the basic structure of Puppet's configuration files as well as taught us:

- The default settings included in every `puppet.conf`
- How to assign our node to an environment
- How to overwrite certain configuration set in our Puppet code
- Ways to change the behavior of the Puppet agent during a Puppet run
- A set of debug commands

In the next session, we'll take a look at master-specific configuration. Then, it's back to some more practical tasks as we create an administrative user for ourselves to use and set up some RBAC rules.

References

- **Configuration Reference (Puppet 5.5)**
- **Configuration Reference (Latest)**

Configuration: Master

While we reviewed a number of settings in the previous lesson, one thing we didn't mention at all is how our settings are grouped. In our nodes' `puppet.conf` file, everything we looked at was added under `[main]`. On our Puppet master, however, we have two additional categories: `[agent]` and `[master]`.

`[agent]` stores configuration related to the Puppet agent. Anything we stored in `[main]` we can also store here. What's of interest in this lesson is the `[master]` group, with a few additional `[main]` settings.

`[main]`

While our `[main]` configuration still begins with the `certname` and `server` values, we have four more new settings listed here. These all fall under the category of "settings Puppet thinks most users will be familiar with," due to their fairly self-explanatory nature.

The `user` and `group` settings set the user and group that Puppet works under, including any services, tasks, and actions. `environment_timeout` sets how long the Puppet master will cache the data it loads from environments. The `0` means it's disabled.

`module_groups` is the one we have to be the most cautious of, because we're not supposed to change it. This is how Puppet pulls down everything it needs for our master. By setting a module group, we're telling Puppet to grab everything in these groups from the Puppet Forge and add it to our master.

> Curious about why Puppet has so many settings we don't really have to use? According to Puppet, "This is basically a historical accident. Due to the way Puppet's code is arranged, the settings system was always the easiest way to publish global constants that are dynamically initialized on startup. This means a lot of things have crept in there regardless of whether they needed to be configurable."
>
> ***"Important Config Settings" - Puppet 5.5 Documentation***

Additional Settings

Beyond the ones added by default, there are a few more settings that we place in the `[main]` section of our master's `puppet.conf` file when we need them:

`dns_alt_names`

This sets any alternative names for Puppet Server. This is also the setting used for round-robin DNS setups, and it has to be added *before* the certificate request from the masters is made. Added means in the `pe.conf` file or with the `puppet cert` command, which we'll address in future lessons.

`environmentpath`

`environmentpath` sets the location of directory environments. This defaults to `/etc/puppetlabs/code/environments`.

`basemodulepath`

It sets the location of Puppet modules available to all environments. The default is `/etc/puppet/code/modules`.

`[master]`

What we add to our `[master]` configuration depends quite a bit on our setup. Most of these are related either to Puppet extensions (additional Puppet features) and to certificate authority management.

But first, let's consider what's already in our `puppet.conf` file:

`node_terminus`

This setting determines which node terminus is used during the catalog compilation process. This is set to `plain` in Open Source deployments, and to `classifier` in PE. So when a Puppet run happens, our master will be checking with our Enterprise setup for external node classifications.

`storeconfigs` and `storeconfigs_backend`

`storeconfigs` sets whether or not Puppet will store data about each node's configuration, such as catalogs and facts, whereas the `storeconfigs_backend` set *where* this data will be stored.

`reports`

This sets which report handler to use, not whether or not to enable reports.

`always_retry_plugins`

When set to false, as with our `puppet.conf` file, any resource types and features are checked once. If they are not available, they are cached with a negative result and there will be no further attempts to load them.

`disable_i18n`

This setting turns off all translations of Puppet and module log messages.

Additional Settings

There are also a few extra settings not already in our file we need to be aware of:

`catalog_terminus`

If we have Puppet working with a lot of files on our nodes, this option lets us turn on an options static compiler that will sacrifice CPU cycles in favor of faster performance and reduced HTTPS traffic.

`ca`

This determines whether the master will act as a certificate authority. Remember, we can only have one.

`ca_ttl`

The `ca_ttl` option lets us set how long our certificates remain valid. The default for this is five years (`5y`).

`autosign`

This sets the location of the file where we can set up rules for autosigning certificates.

Wrap Up

In our configuration section, we took a look at our master's `pe.conf`, learning about:

- How Puppet groups its configuration settings

- The configuration settings included in a basic PE install
- Additional options we can use to further configure our Puppet master depending on our needs

We'll be getting hands-on again next, this time returning to the Puppet Enterprise console to take a look at our role-based access control options.

References

- **Configuration Reference (Puppet 5.5)**
- **Configuration Reference (Latest)**

RBAC

As the final bit of our Enterprise configuration, we're going to actually configure something! We've been logging in to our console as `admin` thus far, which is fine to start, but also generally not best practice from a security standpoint. As such, we'll be checking out how RBAC sets roles and create a new administrative user.

Add a Role

To use RBAC, we first need to log in to our PE console. Once we're done, go to **Access control**. This expands the list to show us three options. We want to go to **User roles**. We have four default roles here, but to understand how user roles work, let's add a role instead.

In this scenario, let's say we work somewhere that grants administrator access to servers on a tiered level, and `tier-1` admins can only perform the most basic of helpful tasks. We're going to give our user the **Name** `T1 Admin` with the description of `Can run basic admin tasks`. Click **Add role**.

Click on our newly-created role. We're taken to the **Member users** tab, where we can set which users have this role. We don't have any users yet, though, so let's look at the **Permissions** tab instead.

From here, we can determine what our `tier-1` admins can do. For now, let's give them the ability to perform `service` tasks on the production environment. Select `Tasks` from the **Add a permission** dropdown.

We only have one sub-option with this permission, so we're leaving that setting as-is. We do want to specify what kind of tasks however, so in the **Enter an instance** input area, search for `service`. We can further specify `service::linux` or `service::windows`, but let's leave it as the general `service` option.

Finally, we can set which nodes our newest admins can work on. We don't want them messing with any development environments (the people testing on them won't be too happy with admin interference), so instead we want to grant them the ability to manage services on our production environment. Essentially, we want to let our `tier-1` admins

perform the simplest troubleshooting task: turning it off and on again. Set **Permitted nodes** to `Node group`, then select `Production environment (production)`. **Add** the permission.

Add a User

Let's go ahead and add a user now. It won't be for our `T1 Admin` role, though. Instead, it will be for us. Switch to the **Users** page and go ahead and add your name to **Full Name**, then give yourself a **Login**. Click **Add local user**.

Click on your user. We have the option to add our email by selecting **Edit user metadata**. Otherwise, let's click **Generate password reset** to make ourselves a password.

Copy the link that appear, paste it into your browser, then create your password.

Add the User to a Role

We're not done yet, though! We can't add our user to a role from here. Instead, let's go back to the **User roles** page and select the **Administrators** role.

Select your user from the dropdown list, then click **Add user**. We're asked to confirm this change by clicking **Commit 1 change** at the bottom of the console. Do so.

We can now log out and log back in as our own user, with all the same power as being the `admin` user.

External Directories

Finally, RBAC offers us one more option we should at least look at: the ability to import in an external directory, such as LDAP. If we go to the **External directory** page, we can see that to set this up we would supply our directory information and credentials, and then set our querying information related to our existing setup.

Wrap Up

In tech, we frequently hear about the "principle of least privilege," where we ensure our users can only access exactly what they need in our systems. We learned how to implement this principle in our Puppet setup by:

- Creating a new user role through Puppet's RBAC system
- Creating a new Puppet user
- Adding a user to a role
- Viewing how to connect Puppet with an existing external directory

Puppet Server

Now that we have a fully working Puppet Enterprise setup, it's probably tempting to want to start immediately writing Puppet code. However, we need to take a step back again to look at Puppet Open Source. The Puppet Professional Exam tests on Puppet Open Source as well as Enterprise, and we can actually gain some additional insight into how our Enterprise setup works by looking at a simple Open Source setup.

We're only going to be creating one master and one node, and we can use a small- and micro-sized server for this setup. However, if you're out of Cloud Playground units or simply don't want to use two more, even temporarily, a Vagrantfile is provided down below, which uses VirtualBox to spin up an Ubuntu 18.04 server we can use as the master and a CentOS 7 server we can use as the agent.

In one of very first lessons, we took a look at a diagram where we saw our Puppet Enterprise Master of masters filled with various services and features it has to offer. But on an Open Source setup, this is pared down substantially, and the real focus is that of the Puppet Server itself. Puppet Open Source also uses substantially fewer resources. This is why we can install it on our smallest instance size with only a little bit of configuration. Puppet Open Source has the following requirements:

Nodes	Cores	RAM
<10	2	1 GB
<1000	4	4 GB

Now, we already know that the Puppet Server is a Java Virtual Machine living on our master host compiling our catalogs. When we install Puppet Open Source, this is essentially the *only* thing we're installing on our master. The certificate authority is included in the Open Source server package. The server and cert authority still work the same as

in our Enterprise version, however, so we can just go ahead and begin the installation process. Remember, we're using an Ubuntu 18.04 server as our master.

The Puppet Professional tests on Open Source Puppet 5.5 and above; as such, we'll be using the Puppet 5 platform and noting any differences between that the most recent release at the time of creation, Puppet 6.

To begin, let's update our `/etc/hosts` information and add our FQDN to the loopback address; we also want to add the alias `puppet`:

```
$ sudo $EDITOR /etc/hosts
```

Edit the `127.0.0.1` line:

```
127.0.0.1  USERNAME#c.mylabserver.com puppet localhost
```

Save and exit.

We now need to enable the Puppet 5 Platform repository:

```
$ wget https://apt.puppetlabs.com/puppet5-release-bionic.deb
$ sudo dpkg -i puppet5-release-bionic.deb
$ sudo apt update
```

From here, all we have to do install the Puppet Server is run:

```
$ sudo apt install puppetserver
```

The Puppet Server is not automatically started, however, giving us the time to make some configuration changes that will allow Puppet to run successfully on our resource-limited master. Specifically, we want to change the memory allocation, which is located in `/etc/default/puppetserver`, the server's init configuration file:

```
$ sudo $EDITOR /etc/default/puppetserver
```

Locate the `JAVA_ARGS` setting. By default, this is set so the Puppet Server can use 2 GB of RAM. Let's drop this down to 512 MB, the smallest size permitted:

```
JAVA_ARGS="-Xms512m -Xmx512m -Djruby.logger.class=com.puppetlabs.jruby_utils.jruby.Slf4jLogger"
```

Save and exit the file.

We can now start and enable the Puppet Server:

```
$ sudo systemctl start puppetserver
$ sudo systemctl enable puppetserver
```

Our Puppet master is now up and running! As with Enterprise, our Open Source master is also a master of itself, and we can also use `puppet agent` commands. However, the Puppet binaries are not automatically set up to let us work as a superuser. Let's try to view our master's fingerprint to see:

```
$ sudo puppet agent --fingerprint
sudo: puppet: command not found
```

To fix this, we can add a file in our `/etc/sudoers.d` directory adjusting our PATH.

```
$ sudo $EDITOR /etc/sudoers.d/extra
```

Add this line:

```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin:/opt/puppetlabs/bin
```

Notice that we're adding `/opt/puppetlabs/bin`. Tthis is the location of all our Puppet binaries. The binaries are also symlinked to `/bin`.

Save and exit the file, then refresh your session by running:

```
$ bash
```

We can now run Puppet's command line tools without having to become `root`.

Next, we'll see how we can install the Puppet agent on our CentOS 7 node, then see how to manage certificate requests from the command line.

Wrap Up

In this section, we installed the Open Source version of Puppet by:

- Adding the Puppet 5 repositories to our host
- Installing the Puppet Server
- Configuring the Puppet Server's startup parameters
- Ensuring the Puppet Server works and can restart automatically at startup

Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|

  config.vm.define "puppetserver" do |puppetserver|
    puppetserver.vm.box = "bento/ubuntu-18.04"
    puppetserver.vm.hostname = "puppetserver.mylabserver.com"
    puppetserver.vm.network "private_network", ip: "192.168.50.10"
  end

  config.vm.define "node" do |node|
    node.vm.box = "bento/centos-7.3"
    node.vm.hostname = "node1.mylabserver.com"
```

```
    node.vm.network "private_network", ip: "192.168.50.20"
  end

end
```

Puppet Agents

Unlike Puppet Enterprise, with Open Source Puppet we won't have a simple `curl` command provided to us. That said, the process is no more difficult than adding a repository and installing a package.

We'll be using a micro Playground server with CentOS 7. Let's jump right in by updating our `/etc/hosts` file so the private IP of our master is mapped to `puppet` and our master's FQDN:

```
$ sudo $EDITOR /etc/hosts
```

Add this line:

```
PRIVATE_IP_MASTER   puppet USERNAME#c.mylabserver.com
```

Save and exit.

Next, we want to add the Puppet 5 Platform repository:

```
$ sudo rpm -Uvh https://yum.puppet.com/puppet5-release-el-7.noarch.rpm
```

We can now install the `puppet-agent` package:

```
$ sudo yum install puppet-agent
```

Then get it up and running with:

```
$ sudo systemctl start puppet
$ sudo systemctl enable puppet
```

That's all there is to get the agent going!

However, we do still want to update our `/etc/sudoers.d` with a file setting our secure path:

```
$ sudo $EDITOR /etc/sudoers.d/extra
```

Add this line:

```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin:/opt/puppetlabs/bin
```

To confirm our changes worked, let's refresh our session and view the node's fingerprint:

```
$ bash
$ sudo puppet agent --fingerprint
```

Make note of this print! We're going to use it in the next section.

Wrap Up

This short and sweet bit showed us how to install the Puppet agent by:

- Adding the Puppet 5 Platform repository
- Installing the `puppet-agent` package
- Starting and enabling the service

Certificate Authority

Now that we have an agent running underneath our Puppet master, all that remains is accepting its certificate on the master. However, without the Puppet Enterprise Console, this means doing it on the command line.

In Puppet 5, the command line tool for managing certificates is in a bit of flux. Originally, Puppet used the `puppet cert` command to manage certificates. This has been deprecated in Puppet 5 though, and replaced with `pupperserver ca`. We can still use `puppet cert` commands if we wish (they do all still work) but we'll be focusing on `puppetserver ca` functionalities. Puppet 6 does not support the `puppet cert` command at all.

That said, if you're already familiar with `puppet cert`, know that most flags and options simply just carry over to `puppetserver ca`. So what was `puppet cert list --all` in previous versions is `puppetserver ca list --all` in 5 and above. All of these commands can also be used to manage certificates from our Puppet Enterprise master, should we not want to use the console.

Now let's actually run that `puppetserver ca list` command on our Open Source master:

```
$ sudo puppetserver ca list --all
```

Two fingerprints return: one for one master and one for our CentOS 7 node. Note the `+` beside the master server as well. This means the certificate for that server has been signed. Check that the fingerprint for the CentOS 7 node matches the one on the server.

Then, to sign our node's certificate, we just have to run:

```
$ sudo puppetserver ca sign --certname USERNAME#c.mylabserver.com
```

We could also use the `--all` flag instead of `certname`, which just accepts all pending certificate requests.

The `puppetserver ca` command provides us with more options than just `sign` and `list`, however, and we can perform a number of management tasks for our certificate authority with it:

puppetserver ca setup

We did not have to run this command when we initially set up our Open Source master. But should we need to configure our certificate authority, the `setup` command will configure a root and intermediate signing CA for Puppet. The root certificate is the self-signed certificate for our master, while the intermediate CA is used for signing incoming certificate requests.

When setting up a CA, we can also set aliases for our master with the `--subject-alt-names` flag. The `--ca-name` flag lets us set a common name for the signing cert, while `--certname` sets the name for the master certificate.

`puppetserver ca setup` can also use the two universal commands: `--help` and `--config`. `--help` simply outputs helpful information about the command, including flags and how to use it, while `--config` lets us define the location of the `puppet.conf` file. These can be used with any `ca` command.

puppetserver ca generate

This command generates a new certificate signed by the intermediate CA authority. Like with `setup`, `generate` is not a command we usually have to run. But in situations where we may have to pre-generate certificates for nodes, we can use the `--ca-client` option to create the certificate "offline", which we can then distribute to any node. Note that when we do this, the `puppetserver` service must be turned off.

We can also use `--subject-alt-names` with this command.

puppetserver ca import

If using an external certificate authority, we can use the `import` command to pull in our certificate information. To do this, we need to generate a cert chain from the external CA itself, then provide it with the `--crl-chain` flag. We also need to provide bundle (`--cert-bundle`) and private key (`--private-key`) information for the import to work.

`--certname` and `subject-alt-names` can also be used with this command.

`puppetserver ca revoke`

This revokes any in-use certificates. While we can specify more than one in a comma-separated list with `--certname`, we cannot revoke all our certificates with `--all`.

`puppetserver ca clean`

The `clean` command performs the same tasks as `revoke`, but also removes any files pertaining to the revoked certificate. This lets us do things like add a host with the same name as the previous cert, but with a different certificate.

As with revoke, we cannot use `--all` with this command.

Autosigning

What happens if we don't want to have to manually accept certificates every time they come from a known source? For example, if we wanted to automatically except everything using our Playground FQDN.

This is what certificate autosigning is for. We briefly touched on the `autosign` option in our `puppet.conf` page, noting that it can set the location of our autosigning configuration file or turn autosigning off entirely.

On our Enterprise master, the default `autosign.conf` file is simply a blank file in our configuration directory, `/etc/puppetlabs/puppet`. For Open Source, this file is not automatically created. We can create it, however:

```
$ sudo $EDITOR /etc/puppetlabs/puppet/autosign.conf
```

To add which servers we want to autosign, we just have to supply the FQDN in the file, with one per line. We can also use some globbing, although an asterisk can only occupy one or more subdomains. So we can't use `USERNAME*c.mylabserver.com`. Instead, if we want to accept any requesting coming in from our lab servers, we would use:

```
*.mylabserver.com
```

Save and exit the file.

Wrap Up

Useful with both Open Source and Enterprise versions of Puppet, the `puppetserver ca` command lets us:

- Create a certificate authority for creating connections between our master and nodes
- Import an existing certificate authority into Puppet
- Sign, revoke, and completely remove certificates

We also learned how to enable and use autosigning to accept certificate requests from known servers.

It is now safe to remove your Open Source Puppet servers; we will not use them again in this course.

The Puppet Forge

To get started managing our nodes' configurations, we need to start looking at Puppet modules. A *module* in Puppet is a collection of code that configures one specific item or task -- like Apache, Docker, MySQL, or NTP. We've hinted a little at this already -- talk of "classes" and "Puppet code" has be referenced but not expanded upon.

But we're not going to jump into the deep end yet! The Puppet Domain-Specific Language (DSL) has a lot of terms and moving parts. So to break down everything we have to work with, we're going to first take a look at what a well-formatted Puppet module looks like. And we're going to find one of those by looking at the Forge.

The Puppet Forge is a PuppetLabs-hosted repository of Puppet modules that are available for use. This directory contains both PuppetLabs-supplied and user-submitted modules and can be found at `forge.puppet.com`.

Go ahead and open the Puppet Forge in your browser. We're presented with the option to search for whatever module we might need, as well as provided with a list of some top and "partner" modules. What's most interesting to us, however, is the **New to modules?** section underneath this, which provides us with a key of tags we should be on the look out for while searching the Forge:

Supported modules are modules written by PuppetLabs themselves. These all support Puppet Enterprise and are "rigorously tested" for support in maintained Puppet versions.

Partner modules are also rigorously tested with Puppet Enterprise, but are not written by PuppetLabs. Instead, they are provided by a partner organization. Generally, there are modules written by the maintainers of what's being configured. In other words, Sensu provides the Sensu module, f5 the f5 module, and so forth.

Approved modules are user-submitted modules that meet the standards PuppetLabs has for modules. These are considered "reliable" and "well-written" but are not tested extensively like modules with the other two tabs.

Generally speaking, when pulling in a module it's best to look for one of these labels, especially if you're not confident in your ability to look through the module code itself and find any issues.

There are also two additional labels we can look for: *Tasks* and *PDK*. *Tasks* means that the module contains features that can be run outside of configuration management -- think things like running one-off commands. *PDK* means the module supports the Puppet Development Kit for validation and testing.

Now let's actually go search for a module. Specifically, we want to check out the `ntp` module, which provides us with a good overview of how the Puppet language works without being too overwhelming.

Of the NTP modules on the search results page, select the one written by PuppetLabs. We're taken to a page providing an overview of the module, including the versions of Puppet (Enterprise and Open Source) the module supports, the supported operating systems, and how to add the module to our Puppet setup. We're also provided with ratings information, including a "quality score" and "community rating."

The quality score is based on objective facts about the module. If we scroll down on the page and click on the **Score** tab, we can see this in more detail. The quality score is calculated through three overall metrics: code quality, puppet compatibility, and metadata quality. These metrics then record the number of errors, warnings, and notices the module has within these categories. We can additionally expand the compatibility and metadata sections to further see what is being tested.

There is also a section for the community rating, which asks five questions about the module itself relating to its docs, ease-of-use, need for user-changes, and whether or not you use the module in production.

Beyond this, we also have access to the README, which describes how to use the module, a reference of module classes, a changelog of any updates to the module, and information about any dependencies and compatibility. There's also a tab where we can view detailed licensing information.

But let's return to the top of this page. Notice the small link **Project URL**. If we click this we can actually see the module, because it takes us to the module's GitHub page. If we wanted to review the code before adding it to our server, this would be where we do it.

We also the option to submit any **Issues** we have with the module. This link takes us to a PuppetLab's public JIRA board, although note that the location of the issue tracker is specific to the module itself. For PuppetLabs modules, it takes us to the JIRA. For most others, the GitHub issues page is instead provided.

Adding a Module from the Forge

Let's go ahead and add this module to our Puppet setup. Had we already been working with Code Manager, we could do this by simply adding a single line to something called a Puppetfile, as evidenced by the provided code snippet. We're not that deep into Puppet though, so we'll be adding it using the `puppet module` command, which works with both OS and Enterprise versions of Puppet.

From your master's terminal, run:

```
$ sudo puppet module install puppetlabs-ntp --version 7.4.0
```

Let's now consider the output of this command:

```
Notice: Preparing to install into /etc/puppetlabs/code/environments/production/modules ...
Notice: Downloading from https://forgeapi.puppet.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/production/modules
└─┬ puppetlabs-ntp (v7.4.0)
  └── puppetlabs-stdlib (v5.2.0)
```

Unless we specify where we want the module to install, Puppet automatically places it in the production environment under the `$codedir`. By default, the `$codedir` is located at `/etc/puppetlabs/code`, with the production environment's modulepath specifically located at `/etc/puppetlabs/code/environments/production/modules`. We we can see that in the output itself.

We can also see that by running that `puppet module` command (versus doing something like pulling down the Git repo) any dependencies are also automatically added to the modulepath, as well.

Wrap Up

We took a look at how to use the Puppet Forge, learning:

- What each tag in the Forge entails for a module
- How to find information about a desired module
- How to download a module from the Forge

Next, we'll be taking this NTP module and using it to break down each component of a module, then take a look at the code itself to see how it reflects Puppet best practices.

Module Structure

In our last lesson, we installed the `ntp` module via the Puppet Forge. But what do we actually get when we added this module to our Puppet setup?

Let's go ahead and move into our module's working directory, then take a look at what we have:

```
$ cd /etc/puppetlabs/code/environments/production/modules/ntp
$ ls
```

There are a lot of files here! The majority of the text files are either self-explanatory or irrelevant to our goals. `CHANGELOG.mg` and `README.md` provide the same information as on the Puppet Forge, for example, while `CONTRIBUTING.md` only pertains to us if we want to make a PR to this module. This is something that's not within the scope of the Puppet Professional exam. Instead, we want to consider all the provided directories, as well as the `hiera.yaml` file.

Let's work in order as we break these down:

`data` and `hiera.yaml`

Hiera is Puppet's external data store, where we can keep configuration data in key-value pairs. The `hiera.yaml` file stores our Hiera configuration (as in, where we'll be storing our actual Hiera data, not the data itself). The Hiera data itself is instead stored in the `data` directory.

If we take a look at what we have in this directory:

```
$ ls data/
AIX-family.yaml        Fedora.yaml          RedHat-family.yaml  Solaris-10.yaml
Amazon.yaml            FreeBSD-family.yaml  SLES-10.yaml        Solaris-11.yaml
```

```
Archlinux-family.yaml  Gentoo-family.yaml  SLES-12.yaml  Suse-family.yaml
Debian-family.yaml     OpenSuSE.yaml       SLES-15.yaml  common.yaml
```

We can see that the `ntp` module specifically uses Hiera to store OS-specific information, as well as a `common.yaml` file, storing default values.

examples

Files stored in `examples` show us *how* to use this module. For `ntp`, this is an `init.pp` file that demonstrates how to assign the module to a node:

```
$ cat examples/init.pp

node default {

  notify { 'enduser-before': }
  notify { 'enduser-after': }

  class { 'ntp':
    require => Notify['enduser-before'],
    before  => Notify['enduser-after'],
  }

}
```

locales

This directory stores files concerning localizing the module for other languages. It isn't covered this in this course.

manifests

This is where we store our Puppet code! Let's hold off on looking in here until the next section.

readmes

This is the `README.md` file localized for other languages, and is also out of scope for this course.

spec

The `spec` folder is where we store our tests for integration testing. This is done through use of the RSpec test framework, and we'll deep-dive into what the directory structure and files in this folder means in another section.

templates

This directory stores exactly what it says on the bin: templates. There are used when we need to add any sort of file to our nodes whose options vary by node, purpose, operating system, etc. Let's consider the `ntp.conf.epp` file as an example:

```
$ cat templates/ntp.conf.epp
```

Notice the parts of the template encased in angle brackets. These are the parts of our file that have been templatized, and will be pulled in from Hiera. Notice how we can also use things like `if` statements, as well.

types

This directory stores aliases about any custom resource types we may use in the module. Remember, Puppet has a number of default resource types we can use, such as `package`, but we're not limited to these.

Additional Structures

There are also a few additional options that the `ntp` module *doesn't* use: - `tasks` stores any tasks we can create for use outside of the configuration management system. - `lib` stores any custom plugins, such as additional resource types and custom facts. - `files` is our static file data storage. - `facts.d` stores any additional facts about the server.

Also note that (as we can see with our `ntp` module) if we don't use a directory in a module for anything, we can simply remove it without any consequence. Our module's directory structure needs only to include the parts it actually uses and while there's no best practices consideration that state we should remove or keep unused directory structures either way, I tend to remove any to keep the directory clean and so anyone who looks at it can see exactly what the module contains.

Wrap Up

We went through the looked at how an entire module is structured, discovering:

- Where we store and configure our Hiera data
- Where to store files, static or otherwise, on our server
- The function of other directory structures, such as `examples`, `spec`, and `types`

In the next section, we'll move into the `manifests` directory and start breaking down some Puppet code.

Basics

Picking up where we left off, let's now venture into the `manifests` directory and take a look:

```
$ cd /etc/puppetlabs/code/environments/modules/ntp/manifests
$ ls
config.pp  init.pp  install.pp  service.pp
```

Here we have four files, all ending in `.pp`, which is a Puppet-specific file extension indicating that the file is a *manifest*.

Within a module, a manifest contains a named *class*. Let's look at the `install.pp` file as an example:

```
$ cat install.pp

class ntp::install {

  if $ntp::package_manage {

    package { $ntp::package_name:
      ensure => $ntp::package_ensure,
    }

  }

}
```

Everything contained in the `class ntp::install` block is our class. We talked briefly about resource types already. This particular class uses the `package` resource type to install the NTP package appropriate for the operating system of

our node. Additionally, the stanza of code that configures a single resource type is called a resource. So given the above code, this is a class:

```
class ntp::install {

  if $ntp::package_manage {

    package { $ntp::package_name:
      ensure => $ntp::package_ensure,
    }

  }

}
```

This is a resource:

```
package { $ntp::package_name:
  ensure => $ntp::package_ensure,
}
```

And this is a resource type:

```
package
```

We can also have more than one resource type in a class, as is the case in the `config.pp` file:

```
$ cat config.pp
```

This file is a little long, but as can see it contains multiple `file` resources all under the single `ntp::config` class. We're not limited to using the same resource type in a class either. This class could have just as easily contained `file` resource types alongside `notify` or `user`, or any other option provided to us.

Notice, too, the presence of `if` statements in these classes (among others, such as `case`), all wrapped around the individual resource types. This isn't something you see in the most basic of basic Puppet classes (like our example in the "Idempotence" section) but demonstrates the versatility we have when writing our Puppet code. Look back to that `install.pp` class again. The function of our `if` statement here is to check if we even want this class to run. The `$ntp::package_manage` value it's checking for is a value stored in our Hiera data. If the value for our `$ntp::package_manage` parameter is `true`, the resource type will do its job. But if it is `false`, the class will be applied and nothing will be run.

If we look back at our `config.pp` file again, we can see how the same concept is adapted to make sure certain pieces of code are only run on certain operating systems that meet certain parameters:

```
if ($facts['operatingsystem'] == 'SLES' and $facts['operatingsystemmajrelease'] == '12') or
    ($facts['operatingsystem'] == 'OpenSuSE' and $facts['operatingsystemmajrelease'] == '42') {
```

And to do things like make sure we aren't adding multiple templates for the same file, which would cause a conflict during compilation:

```
if $ntp::config_epp and $ntp::config_template {
```

Wrap Up

So at this point, we know that:

- A manifest is the `.pp` file that stores a defined part of our module, such as configuring package management.
- A class is a named container for our actual configuration descriptions.
- A resource is the actual configuration instructions, created through the use of resource types.
- And any piece of Puppet code we write can be manipulated with conditional statements.

Now we just need to learn the parameters by which we need to format and otherwise style this code when we write, then we can start working with modules of our own!

Style Guide

Now that we understand the parts of a module, we're going to delve into the details of its structure. Puppet has stringent requirements for how modules are written, and we need to pay attention to our code's actual setup.

The Overview

PuppetLabs stresses three primary points when it comes to writing clean Puppet code: readability matters, scoping and simplicity are key, and the module as a piece of software. All these principles really entail is that we need to consider how easy our code is to read. If, given the option between two styles (such as having listed items in a single line or separating them), we should always choose the one that's easier to read. We should also keep our code simple. If our `*.pp` file's purpose is to "install and start `nginx`," then `*.pp` should be two different manifests: one for installing, and one for starting.

Finally, remember that whatever code you write does not exist in the vacuum of right now. Like any piece of software, we will (or if not us, someone else) most likely have to return to it. So err on the side of making your module easy-to-understand even if someone has never seen it before.

The Basics

Let's open up our `service.pp` file for the `ntp` module:

```
$ cd /etc/puppetlabs/code/environments/production/modules/ntp/manifests
$ $EDITOR service.pp

# @summary
#   This class handles the ntp service.
#
```

```
# @api private
#
class ntp::service {

  if $ntp::service_manage == true {
    service { 'ntp':
      ensure     => $ntp::service_ensure,
      enable     => $ntp::service_enable,
      name       => $ntp::service_name,
      provider   => $ntp::service_provider,
      hasstatus  => true,
      hasrestart => true,
    }
  }

}
```

If using vim, run the following commands so spaces will be visible for this lesson:

:set list :set listchars-=space:.

Here we have a single class, hosting a single resource type, and nicely demonstrating how we should format our manifests at the simplest level.

Notice how comments begin with `#` symbols, how there's summary and API information included at the top of the file. This is all expected in any quality modules.

Notice, too, how the class definition is formatted, with a space after the class name (`ntp::service`) and the opening brace. Our resource declaration mimics this format as well, with a single space between the resource type and the

opening brace, as well as a space between the opening brace and the resource name (`ntp`). There should be no space between the name and the colon, and all resource names must be contained in quotes.

Then come our resource attributes. Tabbed in (with two spaces used as a soft tab for every level), each attribute is assigned a value via a hashrocket (`=>`). Hashrockets must all line up. Attribute lines must also always end in a comma.

Less obvious are some whitespace requirements: Specifically, we cannot end any line with trailing whitespace. We also need to ensure that when using the `ensure` attribute, it comes first in our attribute list.

Quotes

How we use our quotes also matters. Generally speaking, single quotes should always be used, as shown with the `'ntp'` resource name in our open `service.pp` file. However, if we were calling a variable here, quoting a string that already contained single quotes, or using an escape character, we would want to want to switch to double quotes. Additionally, if calling a variable, we want to encase that variable name in braces:

```
"${ntp-service}"
```

`init.pp` Requirements

Finally, let's exit the `service.pp` file and open up the `init.pp` file. We haven't taken a look at this yet, but here is where we pull in our other manifests and generally dictate the order our code should be run for this module:

```
$ $EDITOR init.pp
```

We're initially greeted with a long list of documentation, explaining the function of each parameter we have stored in Hiera. Then, we have an actual list of each parameter in the module, with its parameter type stated besides it.

This is followed by `contain` lines, which pull in the code from each of our classes, before we reach the `Class` definitions. The capital-C-Class line simple calls our class again by its name (from where they've been imported in via the `contain` line), then ensures they are run appropriately using chaining arrows. Chaining arrows should always come at the *start* of a line, and come in two kinds: `->` and `~>`.

The `->` is an ordering arrow, telling Puppet that the class on its right must be run after the class above it, whereas the `~>` is a notifying arrow, which is only applied if the prior resource makes some kind of change.

Wrap Up

We got down to the details in this section, looking at the requirements Puppet has for things like:

- Tabs or spaces (two spaces)
- Whitespace usage (no trailing spaces)
- Quotes (always single quotes, unless variables and strings with single quotes are involved)

We've also spent a lot of time *looking* at the `ntp` module we downloaded for the Forge, but we haven't actually run it yet. In the next section, we'll add the last piece of our configuration management setup by finding out how we can assign modules to nodes.

The Main Manifest

We've now spent quite a bit of time *looking* at the `ntp` module in effort to gain greater understanding of writing Puppet code. But what about actually using it?

To use our module, any module really, we need to map it to a node or nodes. To map a module to a node, we need to use either our main manifest or a Puppetfile. Guess which one we're using here?

The main manifest (did you guess correctly?) is a file or directory stored in each environment. By default, this is located at `$ENVIRONMENT/manifests` although the location can be changed via the `environment.conf` file stored in the environment's root directory. For example, if we're working with the production environment, we'd be updating the file at `/etc/puppetlabs/code/environments/production/environment.conf`.

If we take a look at this:

```
$ cd /etc/puppetlabs/code/environments/production/
$ cat environment.conf
```

We can see that right now the file just contains documentation with information about some default paths. And since the default main manifest works fine for our use case, we're just going to leave this file as-is.

Now let's move into our `manifests` directory and take a look:

```
$ cd manifests
$ ls
site.pp
```

Notice how just like within our `ntp` module, *this* `manifests` directory also contains a `.pp` file. The only difference is, instead of putting our actual code that does the configuring in our `.pp` files, we instead put our mappings.

Let's go ahead and open up the existing `site.pp` now:

```
$ sudo $EDITOR site.pp
```

We can see that the file contains a bit of documentation explaining its purpose, noting much of what we already described. It's also where we can store global object definitions, like if we wanted to store backups of all our file resources in a filebucket. We even already have a setting disabling just that:

```
File { backup => false }
```

But what's interesting to us is the `node default` section. Any class we place within this stanza will be applied to all nodes, which is exactly what we want for our `ntp` class. However, there are a few ways we can go about adding our class. The `ntp` module itself even lists two different ways.

If we go by our README, to add our `ntp` class, we would use:

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

If we go by our example, though, we would be adding:

```
node default {

  notify { 'enduser-before': }
  notify { 'enduser-after': }

  class { 'ntp':
    require => Notify['enduser-before'],
    before  => Notify['enduser-after'],
  }
```

```
}
```

Generally, *how* we call the class depends on if we need to do anything within the main manifest to the code itself. In the `include` example we're just running the `ntp` module without adding any additional parameters. In the `class` example, we add notification messages (that we would have to update to actually say something meaningful), which aren't actually part of the primary module. We could also do this in a way that lets us reconfigure which NTP servers we use:

```
class { 'ntp':
  servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
}
```

Or we can specify which interfaces as listen to, among other things:

```
class { 'ntp':
  servers  => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
  interfaces => ['127.0.0.1', '1.2.3.4']
}
```

Essentially, any existing attribute within the NTP module can be overwritten here if desired. We don't need to do any of that, though, so let's just use `include`:

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

Save and exit the file.

To get NTP working on our master, we can now run:

```
$ sudo puppet agent -t
```

This will apply all definitions in any files found in our main manifest directory. In our case, it's just the single `site.pp` file, but if we had others the files would be processed in alphabetical order, with all variables and definitions being carried over. So any variables assigned in `00_first.pp` would be valid through `99_last.pp`.

You may also force a Puppet run on any agents at this time, if desired. Remember, even if you don't manually run the agent, the NTP class will be added next time a default Puppet run occurs.

Wrap Up

In this section, we assigned our NTP module to our nodes through the use of our production environment's *main manifest* learning that:

- Each environment has its own manifest where we assign classes to nodes.
- There are multiple ways to assign classes.
- We can overwrite parts of our class in the main manifest if we need to.

GitHub and the PDK

We're almost ready to begin creating a module! But, we do have one more pit stop along the way. We must take some steps to actually prepare for module creation. In this case, we need to install the Puppet Development Kit and set up a GitHub repo for the module we'll be creating.

Let's get started by getting the PDK up and running. Add the Puppet 5 Platform repositories (the same ones from the Open Source lessons), then install the package:

```
$ wget https://apt.puppetlabs.com/puppet5-release-bionic.deb
$ sudo dpkg -i puppet5-release-bionic.deb
$ sudo apt update
$ sudo apt install pdk
```

While that installs, let's switch to our web browser and log in to GitHub. If you don't have a GitHub account, create one now.

> If you'd prefer to use GitLab or an alternative source control option, feel free! This course will assume the use of GitHub, however, so just be sure to make any changes when needed.

Once in your account, click **New** to begin the repo-creation process. We're going to creating an Apache module, since it doesn't require a lot of prerequisite knowledge about how Apache works. So when naming your module, do so accordingly. Since I have repos for various configuration management tools, I'll be naming mine `puppet-apache`. Set the repo to public, then create the repository.

Let's return to our terminal now. The PDK should have finished downloading, so we can create our module by moving into our production environment's `modules` directory and using the PDK:

```
$ cd /etc/puppetlabs/code/environments/production/modules
$ sudo pdk new module
```

We're now prompted with a series of questions about our module. Set the name to `apache`. Since we don't have a Puppet Forge user, fill in what you'd like here. We can update this later if we need. Provide your own name for who wrote the module, and leave the license as default. Finally, unselect the **Windows** option for supposed operating systems; ensure `RedHat based Linux` and `Debian based Linux` are left selected. Type `y` to continue.

A directory had been created in our working path. Access the directory and view its contents:

```
$ cd apache
$ ls
CHANGELOG.md  README.md     data      hiera.yaml     spec
Gemfile       Rakefile      examples  manifests      tasks
Gemfile.lock  appveyor.yml  files     metadata.json  templates
```

Notice how we have a lot of the same structures as the `ntp` module. Whether or not we need all these remains to be seen.

Let's go ahead and commit this to Git:

```
$ sudo git init
$ sudo git add .
$ sudo git commit -am "Module skeleton generated"
$ sudo git remote add origin https://github.com/elle-la/puppet-apache.git
$ sudo git push -u origin master
```

At this point we want to mention another difference between this version of Puppet and older ones: the use of the PDK for module creation. In prior versions of Puppet, the `puppet module generate` command was used for this process.

However, despite the PDK being the suggested method for creating modules in Puppet 5, I do want to note that **Puppet's public practice test** lists a question about the `puppet module generate` command being an answer. The command *does* still work, and asks a different series of questions than the PDK does:

```
$ sudo puppet module generate test-test
Warning: `puppet module generate` is deprecated and will be removed in a future release. This action
has been replaced by Puppet Development Kit. For more information visit https://puppet.com/docs/pdk/
latest/pdk.html.
   (location: /opt/puppetlabs/puppet/lib/ruby/vendor_ruby/puppet/face/module/generate.rb:142:in
'generate')
We need to create a metadata.json file for this module.  Please answer the
    following questions; if the question is not applicable to this module, feel free
    to leave it blank.

Puppet uses Semantic Versioning (semver.org) to version modules.
What version is this module?  [0.1.0]
-->

Who wrote this module?  [test]
-->

What license does this module code fall under?  [Apache-2.0]
-->

How would you describe this module in a single sentence?
-->

Where is this module's source code repository?
-->

Where can others go to learn more about this module?
```

```
-->

Where can others go to file issues about this module?
-->

----------------------------------------
{
  "name": "test-test",
  "version": "0.1.0",
  "author": "test",
  "summary": null,
  "license": "Apache-2.0",
  "source": "",
  "project_page": null,
  "issues_url": null,
  "dependencies": [
    {
      "name": "puppetlabs-stdlib",
      "version_requirement": ">= 1.0.0"
    }
  ],
  "data_provider": null
}
----------------------------------------

About to generate this metadata; continue? [n/Y]
--> y
```

I do suggest running the command to generate a module yourself, just to be familiar. Since the PDK is the recommended method of module creation, however, that is the method we have focused on in this course.

Wrap Up

In this section, we:

- Created our GitHub repo
- Generated the module skeleton for our future `apache` module, and
- Committed this skeleton to Git

The First Class

> Using vim? To help make writing your Puppet code easier, feel free to pull down this virmrc file:
>
> https://raw.githubusercontent.com/linuxacademy/content-ppt206-extra/master/.vimrc

To create a module in Puppet, we need to start somewhere. That somewhere is by writing our first class in our first manifest. Since we're working with Apache, we'll start at the beginning by setting up our class to install the appropriate package.

To begin, let's move into the `apache` module location generated in the previous lesson:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
```

We want to start by using the PDK to generate our manifest file and associated class skeleton. To do this, run:

```
$ sudo pdk new class install
```

Here, `install` is the name of our class and the name of the manifest generated. Note that when using the PDK to generate a class, we need to be in the primary directory for the class we're working with. In other words, if we ran the above command in our `../apache/manifests` directory, it would fail.

Let's now take a look at what the PDK generated:

```
$ sudo $EDITOR manifests/install.pp
```

This is what it should look like:

```
# @summary A short summary of the purpose of this class
#
# A description of what this class does
#
# @example
#   include apache::install
class apache::install {
}
```

Note how the class declaration has been added for us, along with some comments at the top we need to update. Let's go ahead and update the summary to add a short description, and remove the description line entirely. There's no need to expand on the function of this class past providing a simple summary:

```
# @summary
#   Installs the base Apache package.
#
```

We also removed the example, because we won't need to call this class on its own.

Next, let's work with the class itself. As noted, we already have the class definition added for us, so we now need to add the *resource* declaration. For this, we need to use the `package` resource type. A description and list of attributes for this resource can be found in the Puppet docs **here**.

The purpose of the `package` resource type is fairly straightforward: It helps us manage packages on our servers. We do this by using the following attributes to describe the package we're working with:

```
name
provider
ensure
adminfile
allow_virtual
allowcdrom
```

```
category
configfiles
description
flavor
install_options
instance
package_settings
platform
reinstall_on_refresh
responsefile
root
source
status
uninstall_options
vendor
```

Of course, we won't need to use all of these when actually writing our class. For this class, we actually only need to provide data for two attributes: `name` and `ensure`. However, for more involved package management scenarios, we have a robust set of options to work with.

Let's go ahead and write our resource declaration:

```
class apache::install {
  package { 'httpd':
  }
}
```

Note the resource declaration name here: `httpd`. This is the name of the package used to install Apache on CentOS 7. For now, we're going to focus on just one distro and expand out in our next video. If the resource declaration name shares the name of the package we're downloading, we don't need to provide the `name` attribute in our provided parameters. All we need to do to install the `httpd` package with this class is add the `ensure` option:

```
class apache::install {
  package { 'httpd':
    ensure => present,
  }
}
```

`ensure` is how we determine what state we want our package to be in. Options include `present`, `absent`, `purged`, `held`, `installed`, and `latest`.

Save and exit the file. To make sure everything is working, let's add an `init.pp` file:

```
$ sudo pdk new class apache
```

Then `include` our class. We'll refactor this file later; for now, we just want to make sure we can test our `apache` module:

```
$ sudo $EDITOR manifests/init.pp
```

Add the `install` class:

```
class apache {
  include apache::install
}
```

Save and exit.

Finally, let's add our module to our main manifest so we can test it on our CentOS 7 node. Drop down to the main environment directory:

```
$ cd ../..
```

And open up the main manifest:

```
$ sudo vim manifests/site.pp
```

At the bottom, add a node definition for your CentOS 7 node, then `include` the entire `apache` module:

```
node USERNAME#c.mylabserver.com {

  include apache

}
```

Save and exit.

Log on to the CentOS 7 node to test. We can perform a "dry run" of our module by using the `--noop` flag:

```
$ sudo puppet agent -t --noop
```

Notice how it tracks which changes would be made:

```
Notice: /Stage[main]/Apache::Install/Package[httpd]/ensure: current_value 'purged', should be
'present' (noop)
Notice: Class[Apache::Install]: Would have triggered 'refresh' from 1 event
Notice: Stage[main]: Would have triggered 'refresh' from 1 event
```

Go ahead perform an actual Puppet run when you're confident in the changes:

```
$ sudo puppet agent -t
```

If everything runs successfully, move back into the `apache` directory on the master, and commit to git:

```
$ cd modules/apache
$ sudo git add .
```

```
$ sudo git commit -am "Add install class with CentOS 7 support"
$ sudo git push origin master
```

Wrap Up

This sections illustrated how to:

- Use the PDK to generate a class and manifest
- Write effective comments at the manifest level
- Create a simple class that performs a basic task in Puppet

params.pp

At this point, you may immediately notice a problem with our existing installation class. It only works on CentOS 7. But our infrastructure contains an Ubuntu 18.04 server too. For our module to really work across all distributions, we'll need to use one of two options for defining class parameters: `params.pp` or Hiera.

The `params.pp` pattern started as a hack using Puppet's class inheritance behavior. To use `params`, we set a separate class where we do nothing but assign variables, then have our other classes inherit the values from this class. With the release of Hiera 5, we've been given an option to do this same thing through the Hiera data store, but both methods are considered valid and we need to know each to pass the exam.

Now, let's consider our `install` class:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
$ sudo $EDITOR manifests/install.pp

class apache::install {
  package { 'httpd':
    ensure => present,
  }
}
```

There are two values here we would ideally replace with variables: our `name` and `ensure` attributes. Remember that `name` in this case is the `httpd` in the resource declaration. But first let's consider *where* these values would be different. The `name` attribute depends on the operating system of the host, but `ensure` is more ambiguous. Whether or not we want Apache installed is going to be dependent on the role of the server itself. We'll be adding a variable for `ensure` a little differently than in our OS-specific cases.

Go ahead and exit the `install` class, if you have it open. We're instead going to start by adding that `params.pp` file and opening it up:

```
$ sudo pdk new class params
$ sudo $EDITOR manifests/params.pp
```

Let's update the comments:

```
# @summary
#   Operating system-related variables.
#
```

To actually define our variables, we'll have to use conditionals alongside facts pulled from Facter, but we can start by adding any variables that are not dependent on operating system:

```
class apache::params {
  $install_ensure = present,
}
```

Note the name we use for the variable. When assigning variables, we want to make sure we can tell both *where* the variable is being used (in our `install` class) and *which* attribute it's supplying (`ensure`).

To add our OS-specific variable, we need to craft a conditional statement. But *how* we craft this statement is really up to us. Both Puppet's docs and supported modules use `case` or `if` statements for this, with `case` statements being used in simpler setups, and `if` statements being used when the logic is more advanced. We'll be sticking with a `case` statement for now.

There are two ways we can call this statement. We can either use `$facts` to reference Facter and then include the `['os']['family']` parameters, or we can just call the fact with `$::osfamily` directly. Note though that `osfamily` is now just `family` in Puppet 6 and above.

Option 1:

```
class apache::params {
  $install_ensure = 'present'

  case $facts['os']['family'] {
  }
}
```

Option 2:

```
class apache::params {
  $install_ensure = 'present'

  case $::osfamily {
  }
}
```

I'll be using Option 2 from here on out.

Let's define our two supported operating systems:

```
class apache::params {
  $install_ensure = 'present'

  case $::osfamily {
    'RedHat': {
    }
    'Debian': {
    }
  }
}
```

We now want to add any OS-specific variables. For us, that's our `name` value in the `install` class, so we'll call it `install_name`:

```
class apache::params {
  $install_ensure = 'present'

  case $::osfamily {
    'RedHat': {
      $install_name = 'httpd'
    }
    'Debian': {
      $install_name = 'apache2'
    }
  }
}
```

Now we can go ahead and save and exit, then reopen the `install` class:

```
$ sudo $EDITOR manifests/install.pp
```

To call our parameters within any classes, we use the `inherits` keyword. This will make our `params` class the parent scope for our `install` class, meaning that the `install` call will use all of the `params` class's variables and resource defaults. That said, we still need to manually call and define each variable in our `install` class, assigning them the value found in the `params.pp` file. To do this, we add our inheritance data in our class definition, after the class name but before the opening brace. It should look like this:

```
class apache::install (
  $install_name   = $apache::params::install_name,
  $install_ensure = $apache::params::install_ensure,
) inherits apache::params {

We can then call the variables in our code:
```

```
class apache::install (
  $install_name   = $apache::params::install_name,
  $install_ensure = $apache::params::install_ensure,
) inherits apache::params {
  package { "${install_name}":
    ensure => $install_ensure,
  }
}
```

Save and exit the file. To test our changes, we want to perform a Puppet run on both the CentOS 7 and Ubuntu 18.04 nodes. So let's first update our main manifest:

```
$ sudo $EDITOR ../../manifests/site.pp

node rabbitheart3c.mylabserver.com {

  include apache

}
```

Then run the `puppet agent -t` command on each node:

```
$ sudo puppet agent -t
```

When we're content that everything is working, return to the master and update the GitHub repository:

```
$ sudo git add .
$ sudo git commit -am "Add install class with CentOS 7 support"
$ sudo git push origin master
```

Wrap Up

In this section, we explored one method of writing Puppet code so it works across multiple distributions, learning:

- The function of a `params` class and how it works
- How to use `case` statements in our Puppet code
- How to use class inheritance to use our `params` variables in regular classes

Hiera

While the `params.pp` file works well and is, according to Puppet, "not going anywhere." Working with a lot of variables and operating systems can be a bit tedious. As such, we're actually going to refactor our existing `params.pp` file so that we do everything in Hiera instead. Again, we *do* need to know how `params` to pass the exam, and you very may well choose to use this option in your own modules. We're just going to be refactoring to Hiera for the rest of this section.

To get started, we need to set up our Hiera configuration so it expects not just default Hiera data, but data sorted into various files by OS family. But which Hiera configuration do we place this in? Because on thing about Hiera is that it *does* use a Hierarchy-style system. We can set Hiera data on multiple levels and each of these levels has it's own settings file. We have entire-infra-level Hiera settings at `/etc/puppetlabs/puppet/hiera.yaml`, environment-level settings in the environment's main directory, and then module-level Hiera data within our module itself. Since we're going to start our focus by refactoring our module's `params`, you might be able to guess the configuration we'll be starting with: the module's.

And this is actually how it's done in the NTP module we downloaded, so if we want to view an example:

```
$ cat /etc/puppetlabs/code/environments/production/modules/ntp/hiera.yaml

---
version: 5

defaults:
  datadir: 'data'
  data_hash: 'yaml_data'

hierarchy:
  - name: 'Full Version'
    path: '%{facts.os.name}-%{facts.os.release.full}.yaml'
```

```
  - name: 'Major Version'
    path: '%{facts.os.name}-%{facts.os.release.major}.yaml'

  - name: 'Distribution Name'
    path: '%{facts.os.name}.yaml'

  - name: 'Operating System Family'
    path: '%{facts.os.family}-family.yaml'

- name: 'common'
    path: 'common.yaml'
```

We'll be using a similar setup to the `Operating System Family` configuration, where we will supply Hiera data related to our OS families in individual files under `data`. But before we do that, I want to highlight why Hiera is one of the nicer options for defining OS-specific variables. Notice how we have section for full version of a distro, major versions, general distros, as well as OS families. If we were breaking things down like this in a `params.pp` file, we would be need a series of `case` and `if` statements to capture all the logic. Instead, in Hiera, this file is read so the `common` scope variables are at the bottom of the hierarchy, so to speak. And they're replaces with any of the same variables provided in the hierarchy above them. So a "full version" variable will also take precedence over a major version variable, which overwrites a distro variable, which replaces any OS family variables. Then we're finally down to the common options.

Let's go ahead and set up this configuration for our `apache` module:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
$ sudo $EDITOR hiera.yaml
```

This is what it should look like:

```
---
version: 5
```

```
defaults:  # Used for any hierarchy level that omits these keys.
  datadir: data         # This path is relative to hiera.yaml's directory.
  data_hash: yaml_data  # Use the built-in YAML backend.

hierarchy:
  - name: 'Operating System Family'
    path: '%{facts.os.family}-family.yaml'

  - name: 'common'
    path: 'common.yaml'
```

Note how we leave the `common.yaml` configuration. This is where we'll store our default values for things like our `install_ensure` setting, which is independent of OS family.

Save and exit the file.

Now let's add a `RedHat-family.yaml` file:

```
$ sudo $EDITOR data/RedHat-family.yaml
```

And add our single OS-dependent `install` variable in it:

```
---
apache::install_name: 'httpd'
```

Save and exit, then do the same for Debian-based servers with `Debian-family.yaml`:

```
---
apache::install_name: 'apache2'
```

Save and exit again.

Finally, we want add the `install_ensure` variables to Hiera, as well. But since that isn't distro-dependant, we can just add it to the `common.yaml` file, which will store our default:

```
$ sudo $EDITOR data/common.yaml

---
apache::install_ensure: present
```

We may also want to add a default value for the `install_name` variable here. Generally this would be the most common package name for the module, which is a bit difficult to determine for Apache. That said, since we have *more* Ubuntu 18.04 servers than we do CentOS 7, it makes the most sense in our use case to include an `apache2` default:

```
---
apache::install_ensure: present
apache::install_name: 'apache2'
```

Now we need to update the `install.pp` manifest itself. Open the file:

```
$ sudo $EDITOR manifests/install.pp
```

And replace the current variables with the name of the module (`apache`), followed by two colons and the variable name (`apache::install_name`, `apache::install_ensure`):

```
class apache::install (
  $install_name   = $apache::params::install_name,
  $install_ensure = $apache::params::install_ensure,
) inherits apache::params {
  package { "${apache::install_name}":
    ensure => $apache::install_ensure,
  }
}
```

And remove the class inheritance setup from our class declaration:

```
class apache::install {
  package { "${apache::install_name}":
    ensure => $apache::install_ensure,
  }
}
```

That said, we do still need to declare our variables for the module, we're just not doing it here. Instead of adding our parameters to each individual class, we want to make sure anyone using our module can see all the variables we use from the start, in our `init.pp`. We can't do this with the `params.pp` method, but it makes things much cleaner when using Hiera:

```
$ sudo $EDITOR manifests/init.pp
```

To do this, we do something similar to how we used inheritance with our params, but we don't need to use the `inherits` option. Instead, we can simply define our variables and what type of data they expect to accept:

```
class apache  (
  String $install_name,
  String $install_ensure,
) {
  include apache::install
}
```

To confirm everything is working, use `puppet parser validate` to check your work, then run a `puppet agent -t` on both nodes.

Commit to GitHub when ready:

```
$ sudo git add .
$ sudo git commit -am "Refactored module for Hiera"
$ sudo git push origin master
```

Wrap Up

In this section, we learned:

- The three ways we can store Hiera data
- How to change a `hiera.yaml` file to use facts to set data
- How to refactor a module with existing `params` to that it uses Hiera instead

Files

At this point, we have the basics of module authoring down. We can create simple classes, use a `params.pp` file to parameterize those classes, and now we also know how to use Hiera to do the same thing. But there will be times where we can't get away with just writing a class. Sometimes we want to be able to take other data and move that data onto our nodes, like when we manage a configuration file. For this, we use the `file` resource type, as well as the `file` directory, to store any static content we want moved to any of our nodes.

For our Apache module, we'll be using `file` to get our Apache configs where they need to go. In an ideal world, this would be done by creating a template and individually parameterizing each value in the config, and making sure the appropriate values are used on CentOS versus Ubuntu and vice-verse. But since we don't want to spend an hour rewriting a repetitive configuration file, and because the default configs between CentOS and Apache are so different, we're going to use our configurations to demonstrate how non-templated files work. Then we'll use our virtual hosts file to depict templating.

To get started, let's pull down the files for our config:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache

$ sudo curl https://raw.githubusercontent.com/linuxacademy/content-ppt206-extra/master/apache2.conf -
o files/Debian.conf
$ sudo curl https://raw.githubusercontent.com/linuxacademy/content-ppt206-extra/master/httpd.conf -o
files/RedHat.conf
```

Now, the easy part: whenever we manage files on Puppet, we want to make sure whoever accesses those files on the node itself knows that are not to be touched. So, let's make a note at the top of each file that these are maintained by config management:

```
$ sudo $EDITOR files/Debian.conf

# This file is managed by Puppet; update module to make changes.
#
```

Do the same for the `RedHat.conf` file:

```
$ sudo vim files/RedHat.conf

# This file is managed by Puppet; update module to make changes.
#
```

Now to craft our associated class:

```
$ sudo pdk new class config
$ sudo $EDITOR manifests/config.pp
```

Let's update the comments at the top:

```
# @summary
#   Manages configuration files for Apache
#
```

Next, we want to use the **file resource type** to craft our resource declaration. `file`, like `package`, has a number of attributes to consider, although the ones we want to concern ourselves with are `path`, `ensure`, `source`, `mode`, `owner`, and `group`.

Write the resource definition:

```
class apache::config {
  file { 'apache_config':
```

```
  }
}
```

Note that we're not using the namevar the same way that we did previously. While we can use the namevar for `file`, it's taken from the `path` attribute. We're going to add this manually, alongside our other attributes:

```
class apache::config {
  file { 'apache_config':
    ensure => $apache::config_ensure,
    path   => $apache::config_path,
    source => "puppet:///modules/apache/${osfamily}.conf",
    mode   => '0644',
    owner  => 'root',
    group  => 'root',
  }
}
```

Notice how some of our attributes use variables and some do not. While there would be nothing *wrong* with having `source`, `mode`, `user`, and `group` taking variables, since these values are unchanging across distros (with some exception), we can hardcode these values in. They're not something we want to set on a role, or node, basis. That said, the `source` attribute works in opposition of this. Here, we instead leverage Facter so we can call our different configuration file sources straight in our file's name without needing to set any parameters. We could also provide an array here, if you wanted to have different versions or hosts determine the configuration. Whichever location is first on the array takes precedence, but if there is no match, it fails over to the next location on the list until it finds one that has an existing file.

The `source` attribute is also interesting in that is uses the Puppet URI setting. The use of `puppet:` signifies that the file is saved to a Puppet mount point, and we can access it without a needing the full directory path. Note how we can leave out the `file` directory when specifying the location. Puppet already knows to look there.

We could also leave out the `mode` line entirely if we wanted. The default value for any file managed by Puppet is `0644`. It's a personal preference whether that is something you would want to keep in practice. I find it's nice to have a visual reference, especially since not everyone is going to know off-hand what Puppet's default values are.

Let's now add the Hiera variables to our data. `config_path` will be distro-dependent, but we can add `config_ensure` to our `common` file and nothing else. Let's start with our distro-related variable.

Open the `RedHat-family.yaml` file and set the variables:

```
$ sudo $EDITOR data/RedHat-family.yaml

apache::config_path: '/etc/httpd/conf/httpd.conf'
```

Next, set the variable for Debian-based servers:

```
$ sudo $EDITOR data/Debian-family.yaml:

apache::config_path: '/etc/apache2/apache2.conf'
```

Finally, change the `common.yaml` file:

```
$ sudo $EDITOR data/common.yaml

apache::config_ensure: 'file'
apache::config_path: '/etc/apache2/apache2.conf'
```

With that done, all that's left is to update our `init.pp` file and do a little refactoring. First, we can update our parameters:

```
class apache  (
  String $install_name,
  String $install_ensure,
```

```
  String $config_ensure,
  String $config_path,
) {
```

Then we want to replace our simple `include` with a setup that will ensure our classes are run in order. We do this by switching the `include` to `contain`. `contain` allows the `apache` class itself to create a "container" around any contained code, ensuring none of it is run before the actual container itself. This lets us then form relationships between our classes with our chaining arrows:

```
) {
  contain apache::install
  contain apache::config

  Class['::apache::install']
  -> Class['::apache::config']
}
```

Save and exit the file, then test the module with both the `puppet parser validate` command and by forcing Puppet runs on both nodes. When finished, commit to git:

```
$ sudo git add .
$ sudo git commit -am "Added config class to manage configuration files"
$ sudo git push origin master
```

Wrap Up

We got hands-on with the `file` resource, learning:

- The basic attributes of the resource type
- About Puppet URIs
- How to call facts within our classes

Metaparameters

Anyone who has ever made changes to a service's configuration file can probably see a small issue with our last class. Generally, when making changes to any config file, we want to restart the service for those changes to take affect. But Puppet has no way to knowing that a file we're changing needs a restart to take place unless we tell it to. So for this, we want to look at our *metaparameters*.

Metaparameters are attributes that can be used for any resource type. There are thirteen of these, although you'll find you only frequently reference three or four of them. That said, we need to be aware of all thirteen for our exam, so we'll start by taking a general look at each one, then get hands-on with a couple of them specifically.

Before we begin, however, let's pull in the following code as our `service.pp` manifest:

```
# @summary
#   Allows for the Apache service to restart when triggered
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
  }
}
```

And update our Hiera data as shown:

```
# data/RedHat-family.yaml
apache::service_name: 'httpd'

# data/Debian-family.yaml
```

```
apache::service_name: 'apache2'

# data/common.yaml
apache::service_name: 'apache2'
apache::service_ensure: 'running'
apache::service_enable: true
```

alias

`alias` lets us set an alternative namevar when the current one is long or otherwise unwieldy to use. We've actually already done the shorthand version of this! In our `config` class, we name the class `apache_config` despite it's namevar being defined as the `$apache::config_path` variable in the class itself:

```
class apache::config {
  file { 'apache_config':
    ensure => $apache::config_ensure,
    path   => $apache::config_path,
    source => "puppet:///modules/apache/${osfamily}.conf",
    mode   => '0644',
    owner  => 'root',
    group  => 'root',
  }
}
```

We could have achieved this same effect by using the `alias` metaparameter, like so:

```
class apache::config {
  file { "puppet:///modules/apache/${osfamily}.conf":
    ensure => $apache::config_ensure,
    alias  => 'apache_config',
    path   => $apache::config_path,
```

```
    mode   => '0644',
    owner  => 'root',
    group  => 'root',
  }
}
```

This makes it easier for us to reference our classes by name, especially when that name would otherwise include a variable.

audit

`audit` resource auditing. When using the `audit` parameter, we supply it with the name of an attribute (or an array of attributes). Then, whenever that attribute is used in a Puppet run, Puppet will track whether that attribute contributed a change and what that change was.

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    audit      => ['name', 'ensure', 'hasrestart']
  }
}
```

before

While our `init.pp` chaining arrows are a wonderful, visual way to order our modules, we also have the option to use the `before` metaparater. This attribute lets us provide a class or array of classes that cannot be run before our resource, ensuring our classes are being run in the appropriate order.

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    before     => File["$apache::vhosts_path"]
  }
}
```

`consume` and `export`

`consume` and `export` are specifically useful for orchestration and when using Puppet to deploy applications that work across multiple servers. It allows us to provide data from other resources to use with the current class. For example, if we wanted to supply the `user` and `password` data provided in a `mysql` resource to an `apache` class, we would include the `export` metaparameter in the `mysql` resource and the `contain` in the `apache` resource:

```
mysql::db { $name:
  user      => $wp_db_user,
  password  => $wp_db_password,
  export    => Sql[$name],
}

wordpress::instance::app { $name:
  install_dir => "/var/www/${name}",
  consume     => Sql[$name],
}
```

`loglevel`

This sets the log level for the resource. Log levels include: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`/`verbose`, `debug`

`noop`

`noop` ensures the class will be tested but not applied, by default, just as when we ran the `puppet agent -t --noop` command.

`notify`

This expresses that the defined resource or class depends on this class to run. It also ensures the dependent resource is only run when there are changes to the notifying resource. We can also see this with the use of the `~>` chaining arrow.

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    notify     => File['vhosts_file']
  }
}
```

`require`

This works almost as the reverse of `notify`. Instead of supplying the metaparameter to the resource that needs to be come *before* other resources, `require` is added to resources that come *after* the define resource or resources.

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    require    => File['apache_config']
  }
}
```

schedule

`schedule` sets a time *when* Puppet is permitted to manage a resource. To use this, we must first define a schedule with the `schedule` resource type, then use the metapameter to set the frequency:

```
  schedule { 'everyday':
    period => daily,
    range  => "2-4"
  }

  class apache::service {
    service { "${apache::service_name}":
      ensure     => $apache::service_ensure,
      enable     => $apache::service_enable,
      hasrestart => true,
      schedule => 'everyday'
    }
  }
```

This is especially useful if you don't want changes to occur during high-traffic times.

stage

By default, we run our classes that are run using the `main` stage. We can set other stages to order our resources in blocks using the `stage` resource type, then calling it in a class:

```
stage { 'pre':
  before => Stage['main'],
}

class { 'apt-updates':
  stage => 'pre',
}
```

subscribe

Any resources a class is subscribed to are run before the resource with the `subscribe` attribute, and they only run when changes to the subscribed resource occur.

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    subscribe  => File["${apache::config_path}"]
  }
}
```

tag

Let's us supply a list of tags for our resource:

```
class apache::service {
  service { "${apache::service_name}":
    ensure     => $apache::service_ensure,
    enable     => $apache::service_enable,
    hasrestart => true,
    tag        => ['config', 'web', 'basemodules']
  }
}
```

Ordering Our `service` Class

So, now we want to make sure our actual `service` class is set up to restart whenever there are changes to the `config` class. This is already set up in part in the code itself. The `hasrestart` value informs the service that it can restart when appropriate. We just need to tell it what class is going to trigger this restart.
Of the above parameters, we can use five of them to do just that. It's really up to us how we want to define these relationships. We've actually already determined how we're defining our ordering, though: With chaining arrows, which act as alternatives for any of the ordering parameters. So let's go ahead and update our `init.pp` instead:

```
$ sudo $EDITOR manifests/init.pp

class apache  (
  String $install_name,
  String $install_ensure,
  String $config_ensure,
  String $config_path,
  String $service_name,
  Enum["running", "stopped"] $service_ensure,
  Boolean $service_enable,
) {
  contain apache::install
  contain apache::config
```

```
  contain apache::service

  Class['::apache::install']
  -> Class['::apache::config']
  ~> Class['::apache::service']
}
```

When we're content with your changes, test the server on the two nodes with `puppet agent -t`, and, of course, add these changes to GitHub:

```
$ sudo git add .
$ sudo git commit -am "Added a service class"
$ sudo git push origin master
```

Wrap Up

In this section , we went through our list of available metaparameters, including:

- Ordering parameters that let us define the way in which our code runs
- Auditing and debugging parameters, like `audit` and `loglevel`
- Parameters that will make application orchestration and overall management easier, such as `consume`, `stage`, and `export`
- And generally helpful metaparamters, like `tag` and `alias`

Templating

Unlike NTP, Apache isn't a service that can really do it's job with only a package and a configuration file. If we really want a functioning Apache formula, we need to include a virtual hosts configuration. And ideally, we need a way to include more than one if we want a module that can work across multiple use cases. This means a couple of things: 1. We need a virtual hosts template where we can feed in Hiera data. 2. We need to create a "class" that can be run multiple times with different data each time.

We're going to address that first point right now.

First, let's move into our `apache` directory:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
```

Next, we want to create our file template. We can create templates in two languages: the embedded Puppet language (EPP), or the embedded Ruby (ERB). For our virtual hosts template, we'll be using EPP.

Generally, before starting to write a template, it helps to have a reference for the file itself. The base vhosts template I'm using as inspiration is this:

```
<VirtualHost *:80>
    ServerName subdomain.mylabserver.com
    ServerAlias subdomain
    ServerAdmin admin@mylabserver.com
    DocumentRoot /var/www/subdomain/html
</VirtualHost>
```

We want to template this so it can be used multiple times on any one server, or on multiple servers. We're also specifically going to focus on subdomains here, although there's nothing stopping you from challenging yourself to refactor the code in the next two sections to work for subdomains and primary domains.

Let's create and open the file where we'll be storing our template:

```
$ sudo vim templates/vhosts.conf.epp
```

Notice the `epp` suffix. This lets Puppet know we're using an embedded Puppet template.

Let's start just by copying in our base in its entirety:

```
<VirtualHost *:80>
    ServerName subdomain.mylabserver.com
    ServerAlias subdomain
    ServerAdmin admin@mylabserver.com
    DocumentRoot /var/www/subdomain/html
</VirtualHost>
```

When we template files with embedded Puppet, we use various `<% %>`-style tags to signify where we're either importing in information such as variables, adding comments, or writing any store of statement. These are usually `if` statements.

Let's start by adding in placeholders for our variables. We want to give users the option to define port, subdomain, the admin's email (optional), and the document root where we'll host our website files. To set variables we use the `<%= $VARIABLENAME %>` syntax:

```
<VirtualHost *:<%= $port %>>
    ServerName <%= $subdomain %>.mylabserver.com
    ServerAlias <%= $subdomain %>
    ServerAdmin <%= $admin %>
```

```
    DocumentRoot <%= $docroot %>
</VirtualHost>
```

We can also use the `<%= %>` syntax to call Facter facts. Which is exactly what we'll do to replace the `mylabserver.com` part of our file:

```
<VirtualHost *:<%= $port %>>
    ServerName <%= $subdomain %>.<%= $facts[fqdn] %>
    ServerAlias <%= $subdomain %>
    ServerAdmin <%= $admin %>
    DocumentRoot <%= $docroot %>
</VirtualHost>
```

Note that we have to use the `fact[fqdn]` syntax here. Using something like `$::fqdn` will cause an error.

We're not quite done yet, however! We want to make it so the `ServerAdmin` setting is optional, which means crafting an `if` statement about it. To use actual code in our EPP file, we encase it in plain `<% %>` brackets:

```
<% if $admin =~ String[1] { -%>
    ServerAdmin <%= $admin %>
<% } -%>
```

The `String[1]` data type tells Puppet that we want to include this part of the template only if the length of our `admin` string is greater than one character long.

We also want to pay attention to the dash (`-`) at the start of the closing tag. When we end our templating tags with `-%>` we are telling Puppet to strip any tailing whitespace from the provided data. We also have the option to open the `<%-`, which strips any indentation from the start of the line.

Additionally, we have the option to add comments to our template with the `<%#` opening tag. Let's add one to the top of our file:

```
<%# Virtual hosts template -%>
```

Finally, we want to take a look at how we can add parameters to our template. We'll be calling our parameters our associated manifest in the next lesson, so we won't be doing it here for our template, but if we did need to include a parameters list, it would look like the following:

```
<%- | Integer $port,
      String  $sitename,
      String  $admin,
      String  $docroot
| -%>
```

Notice the use of pipes (`|`) to mark the boundaries of the parameter stanza.

We can now save and exit the file. We'll work on implementing this in the next lesson, so no parser tests or GitHub pushes yet!

Wrap Up

In this section, we covered how to write a template, including how to:

- Add variables to our template
- Use `if` statements and other conditionals to increase our template's flexibility
- Write comments to guide end-users, and
- Define parameters at the top of the file

Defined Types

We've already mentioned that we want to be able to run our virtual hosts template as many times as we need to configure all our desired subdomains, but what we don't know if that to do this we won't be creating a class. Instead, we'll be using what is known as a *defined resource type* (or just plain "defined type").

Defined types are pieces of Puppet code that can be run over and over again in our module. That essentially act as new individual resource types we can use as often as we need.

Let's start by generating our defined type's manifest file:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
$ sudo pdk new defined_type vhosts
```

And updating our comments:

```
# @summary
#   Generates virtual hosts files based on template
#
# @example
#   apache::vhosts { 'namevar': }
```

This time we want to leave the example in there. But, we'll substitute it for what we have to feed to our main manifest later.

We now want to pull in our variables from your template at the top of the file, similar to how we added variables for our `params` and `init` classes. We'll be using these within our resource type definitions as well:

```
define apache::vhosts (
  Integer $port,
  String[1] $sitename,
  String $admin,
  String[1] $docroot,
) {
```

Notice how we can specify string length here, just as we could in our template.

We now want to make sure that whatever `docroot` we specify exists. For this, we also use `file`, but instead set the `ensure` option to `directory`. We also want to add Hiera data to set our directory owner and group:

```
  file { "${docroot}":
    ensure => 'directory',
    owner  => $apache::vhosts_owner,
    group  => $apache::vhosts_group,
  }
```

We now want to add in a *second* file resource here, this time one that will create the `subdomain.conf` file in the specified virtual hosts location for our nodes. Since this is something that differs between distros, we'll create a Hiera variable for the location where we place our vhosts configs, then use the `subdomain` variable to name the file itself:

```
  file { "${apache::vhosts_dir}/${subdomain}.conf":
```

Let's add attributes to set ownership and mode:

```
  file { "${apache::vhosts_dir}/${vh_sitename}.conf":
    ensure  => 'file',
    owner   => $apache::vhosts_owner,
    group   => $apache::vhosts_group,
    mode    => '0644',
```

And now we want to call our template. We use the `content` attribute for this, which can take raw text, the `template()` values for ERB, or the `epp()` value for EPP templates. We'll be using EPP:

```
    content => epp('apache/vhosts.conf.epp')
```

But we aren't done yet! We need to pass in our defined type's variables (the ones we already have set in the template itself) so our template can access them. This is done by calling each variable as if it were its own attribute in the `content` line itself:

```
    content => epp('apache/vhosts.conf.epp', {'port' => $port, 'subdomain' => $subdomain, 'admin' =>
$admin, 'docroot' => $docroot}),
```

Finally, we want to make sure this resource triggers a restart for the Apache service when changes are made. Let's use `notify` for this, making our full document look like:

```
define apache::vhosts (
  Integer $port,
  String $subdomain,
  String $admin,
  String[1] $docroot,
) {
  file { "${docroot}":
    ensure => 'directory',
    owner  => $apache::vhosts_owner,
    group  => $apache::vhosts_group,
  }
  file { "${apache::vhosts_dir}/${subdomain}.conf":
    ensure  => 'file',
    owner   => $apache::vhosts_owner,
    group   => $apache::vhosts_group,
    mode    => '0644',
    content => epp('apache/vhosts.conf.epp', {'port' => $port, 'subdomain' => $subdomain, 'admin' =>
```

```
$admin, 'docroot' => $docroot}),
    notify  => Service["${apache::service_name}"],
  }
}
```

Let's now add our Hiera data, all of which is OS family-specific:

```
# data/RedHat-family.yaml
apache::vhosts_dir: '/etc/httpd/conf.d'
apache::vhosts_owner: 'apache'
apache::vhosts_group: 'apache'

# data/Debian-family.yaml
apache::vhosts_dir: '/etc/apache2/sites-available'
apache::vhosts_owner: 'www-data'
apache::vhosts_group: 'www-data'

# data/common.yaml
apache::vhosts_dir: '/etc/apache2/sites-available'
apache::vhosts_owner: 'www-data'
apache::vhosts_group: 'www-data'
```

We also want to add these to our `init.pp`, although we don't need to add the defined type to the `init.pp` class itself:

```
# manifests/init.pp
class apache  (
  String $install_name,
  String $install_ensure,
  String $config_ensure,
  String $config_path,
  String $service_name,
  Enum["running", "stopped"] $service_ensure,
```

```
  Boolean $service_enable,
  String[1] $vhosts_dir,
  String[1] $vhosts_owner,
  String[1] $vhosts_group,
) {
```

Now to actually use our defined type! To call our define, we add it to our main manifest, using it similar to a class, only without the `class` name. Let's initially test our code in CentOS:

```
$ sudo $EDITOR ../../manifests/site.pp

node USERNAME#c.mylabserver.com {

  include apache

  apache::vhosts { 'puppet_project':
    port      => 80,
    subdomain => 'puppetproject',
    admin     => 'admin@mylabserver.com',
    docroot   => '/var/www/html/puppetproject',
  }

}
```

Next, run the `puppet parser` against the `vhosts` manifest, then test the code on the CentOS 7 node with a `puppet agent -t`.

Once you've confirmed everything works, we can further play with the features of the defined type. For our Ubuntu server, we're going to declare our resource type twice, once leaving out the admin address:

```
node USERNAME#c.mylabserver.com {
```

```
  include apache

  apache::vhosts { 'puppet_project':
    port      => 80,
    subdomain => 'puppetproject',
    admin     => 'admin@mylabserver.com',
    docroot   => '/var/www/html/puppetproject',
  }

}

  apache::vhosts { 'puppet_project_dev':
    port      => 8081,
    subdomain => 'puppetproject-dev',
    admin     => '',
    docroot   => '/var/www/html/puppetproject-dev',
  }

}
```

Save and exit. We also want to take this time to update our docstring in the `vhosts.pp` manifest:

```
# @example
# apache::vhosts { 'namevar':
#   port      => INT,
#   subdomain => STR,
#   admin     => STR,
#   docroot   => STR,
```

Then test on the Ubuntu server with `puppet agent -t`. When you're content the module it working, remember to commit to GitHub!

```
$ sudo git add .
$ sudo git commit -am "Added virtual hosts defined type"
$ sudo git push origin master
```

Wrap Up

In this section, we looked at an alternate way of using Puppet resource types by creating a defined type, which lets us:

- Repeatedly use the resource as many times as we need
- Set our own variables to use as resource attributes

At this point, we have a pretty decent start to an Apache module! We'll be looking at testing our end states for this module next, but if you still want to practice module creation I suggest looking into ways to expand the module to remove default site configurations and enable any sites.

Unit Testing

When we create modules using the Puppet Development Kit, we aren't just creating the `.pp` manifest files. You may have noticed this whenever we ran `pdk new [class/defined_type]`, a `spec` file was also generated somewhere in the `spec` directory.

This is because the PDK doesn't just set us up to write well-written modules. It also sets us up to confirm that our modules work. Right now, we're relying on the responses from the `puppet parser` and any `puppet agent -t` return data to confirm our modules are doing what we say. But how do we know Puppet isn't just assuming everything is working as expected? Well, we use unit testing.

Unit testing is the process of creating individual software tests that determine if our code is functioning as expected. So if we have a class that sets the permissions of a file to `0644`, the unit test for that class will actually check that this happened.

The PDK ships with both `rspec-puppet` and `rspec-puppet-facts`, which leverage the Ruby-based test suite, RSpec. RSpec, and unit testing as a whole, is especially useful for catching errors that aren't technically wrong. This would be something like if we set the wrong permissions on a file. Untested, that might not be something we notice for months, but the chances we'll input the wrong permissions in both the module or the spec test are slim, so it will pick up issues we might otherwise miss.

Now, let's get started by looking at what the PDK provided us for tests.

First, we want to move into our `apache` module directory:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
```

Then we want to review the `spec` directory that the PDK generated for us:

```
$ cd spec
$ ls
classes             defines   functions  spec_helper.rb
default_facts.yml   fixtures  hosts
```

`classes` and `defines` are the locations where we store our class tests and defined type tests, respectively. `default_facts.yml` is used if we want to test against certain facts, and any data provided there will override the actual facts a server is using. `fixtures` stores any dependency modules, which is essentially just a copy of our `apache` module as a whole right now. `functions` and `hosts` store additional unit tests as well, especially those related to functions and hosts. These are written in the same way class and defined type tests will be written. Finally, the `spec_helper.rb` file contains any "helper" code for running our tests. This includes things like adding the `require` lines for `rspec-puppet-facts`, ensuring our directory structure works as intended, and all our facts are sorted.

Now let's take a look at what the PDK generated for us for our class tests. Although we can separate our tests our into the different files the PDK created for us, we'll be working directly with the `apache_spec.rb` file as we learn the basics.

```
$ cat classes/apache_spec.rb
```

We're presented with:

```
require 'spec_helper'

describe 'apache' do
  on_supported_os.each do |os, os_facts|
context "on #{os}" do
  let(:facts) { os_facts }

  it { is_expected.to compile }
end
  end
end
```

Right now, this just tests if the class can be compiled into a catalog (`it { is_expected.to compile }`, with the `context "on #{os}" do` portion allowing us to loop through all supported distros to perform the check. So let's see what happens if we run this test:

```
$ sudo pdk test unit --tests=classes/apache_spec.rb

pdk (INFO): Using Ruby 2.5.3
pdk (INFO): Using Puppet 6.4.0
[✔] Preparing to run the unit tests.
[✔] Running unit tests.
Run options: exclude {:bolt=>true}
  Evaluated 6 tests in 8.628672553 seconds: 0 failures, 0 pending.
```

Puppet confirms our tests run successfully. But right now it's only confirming that our module can compile. We want more than that.

Now, let's go ahead and open our `apache_spec.rb` file and add more tests:

```
$ sudo vim classes/apache_spec.rb
```

We're going to test three additional things: that all our classes are contained, that our appropriate package is installed, and that the appropriate service is started and enabled. This won't cover our entire module, but will be enough for us to cover any structures you need to aware of for the exam. The PDK and RSpec testing is not a huge focus of the exam, we just need to know how it works and be able to read a unit test file.

So, right now our unit test looks like this:

```
require 'spec_helper'

describe 'apache' do
  on_supported_os.each do |os, os_facts|
    context "on #{os}" do
```

```
      let(:facts) { os_facts }

      it { is_expected.to compile }
    end
  end
end
```

This is specifically structured to use `rspec-puppet-facts`, which allows us to iterate through all of the OSes our module supports. Without it we would have to write separate files for each operating system, or OS version, and every use case we would otherwise consider.

These lines, in particular:

```
  on_supported_os.each do |os, os_facts|
    context "on #{os}" do
      let(:facts) { os_facts }
```

Create a loop using facts. For each operating system, the loop will pull in another operating system-related fact (`let(:facts) { os_facts }`), then runs any tests within the function specifically against a host with that OS. For OS-specific variables, we will have to use `case` or `if` statements, but we'll still be managing much less code than if we wrote individual `context` stanzas for each OS.

Now let's consider:

```
it { is_expected.to compile }
```

This is how all our actual tests will be structured at the most basic level. We'll add as we consider more advanced use-cases, but every test line will essentially look like:

```
it { DOES SOMETHING IN A CERTAIN WAY }
```

Most RSpec tests can be read as a sentence, answering the question "What is it expected to do?"

So, let's go ahead and expand out the current test, focusing first on tests we can run across all operating systems: the tests that check that we have all our classes contained. To do this, we'll write three lines using the `contain_class` function, instead of `compile`. You'll notice as we progress we use the `contain_*` format a lot. See a full list of options in **RSpec-Puppet's docs**.

```
require 'spec_helper'

describe 'apache' do
  on_supported_os.each do |os, os_facts|
    context "on #{os}" do
      let(:facts) { os_facts }

      it { is_expected.to contain_class('apache::install') }
      it { is_expected.to contain_class('apache::config') }
      it { is_expected.to contain_class('apache::service') }
      it { is_expected.to compile }
    end
  end
end
```

Save and exit. Let's see if our tests work!

```
$ sudo pdk test unit --tests=spec/classes/apache_spec.rb
```

Success! That said, notice the returned data. If our tests succeeds, it will let us know there were no failures. But what does it look like if our tests fail? There are two things we want to look our for here. Let's reopen our `apache_spec.rb` file:

```
$ sudo vim classes/apache_spec.rb
```

And add a stray curly bracket somewhere. Save and exit, then rerun the unit test:

```
$ sudo pdk test unit --tests=spec/classes/apache_spec.rb
pdk (INFO): Using Ruby 2.5.3
pdk (INFO): Using Puppet 6.4.0
[✔] Preparing to run the unit tests.
[✖] Running unit tests.
```

Notice how it quits at `[✖] Running unit tests.` with no errors. This means we messed up the *syntax* of our file is an unexpected way.

On the other hand, if something simply isn't working, the errors will resemble:

```
$ sudo pdk test unit --tests=spec/classes/install_spec.rb
pdk (INFO): Using Ruby 2.5.3
pdk (INFO): Using Puppet 6.4.0
[✔] Preparing to run the unit tests.
[✖] Running unit tests.
Run options: exclude {:bolt=>true}
  Evaluated 12 tests in 7.41826528 seconds: 12 failures, 0 pending.
failed: rspec: ./spec/classes/install_spec.rb:11: error during compilation: Evaluation Error: Error
while evaluating a Resource Statement, Class[Apache::Install]: has no parameter named
'install_name' (line: 2, column: 1) on node rabbitheart1c.mylabserver.com
  apache::install on debian-8-x86_64 should compile into a catalogue without dependency cycles
```

So if there's an actual issue with our code or with our class, we can just follow the feedback to troubleshoot the issue. In this case, it's because the `apache::install` class doesn't have our Hiera data defined, since we do that in our `init.pp`.

Wrap Up

In this section, we learned the function of unit tests, and that:

- Puppet uses the RSpec test framework to perform unit tests

- The PDK installed and configured the test framework for us
- How to write basic unit tests working with the `rspec-puppet-facts` gem

Working Across OSes

At this point in our test case, we've tested for everything that works across all distros. All that's left is to consider things that we need to address on an OS-family by OS-family basis. To do this, we're going to craft an `if` statement:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache/spec
$ sudo vim classes/apache_spec.rb

describe 'apache' do
  on_supported_os.each do |os, os_facts|
    context "on #{os}" do
      let(:facts) { os_facts }

      it { is_expected.to contain_class('apache::install') }
      it { is_expected.to contain_class('apache::config') }
      it { is_expected.to contain_class('apache::service') }
      it { is_expected.to compile }

      case os_facts[:osfamily]
      when 'Debian'
      when 'RedHat'
      end
    end
  end
end
```

We then want to supply each case with tests that check that the Apache package is present, and that the Apache service is running, enabled, and expected to restart given certain parameters. We'll start with the package check and focus first on Debian. At it's most basic, we can check for the package with:

```
it { is_expected.to contain_package('apache2') }
```

But we specially want to check that our `ensure` variable is set correctly, so we expand our test with the `.with` function:

```
it { is_expected.to contain_package('apache2').with(ensure: 'present') }
```

Notice how this still mostly works as a sentence, albeit kind of a stilted one once we get to our parameters.

We also want to mimic this same structure with the `service` test, only this time we're going to provide multiple parameters:

```
it { is_expected.to contain_service('apache2').with(ensure: 'running', enable: true, hasrestart:
true) }
```

Giving us this resulting code for Debian:

```
      when 'Debian'
        it { is_expected.to contain_package('apache2').with(ensure: 'present') }
        it { is_expected.to contain_service('apache2').with(ensure: 'running', enable: true,
hasrestart: true) }
```

Let's repeat this for RedHat:

```
      when 'RedHat'
        it { is_expected.to contain_package('httpd').with(ensure: 'present') }
        it { is_expected.to contain_service('httpd').with(ensure: 'running', enable: true,
hasrestart: true) }
```

Finally, we do want to address one use case we don't have: how to pull in parameters when using a `params.pp` file with your module. For this, we would just call our parameters using a `:params` line:

```
let :params do
  {
    install_name: 'apache2',
    install_ensure: 'present',
  }
end
```

Save and exit the file. Let's run our tests one more time:

```
$ sudo pdk test unit --tests=spec/classes/apache_spec.rb
```

And another success!

Wrap Up

In this section, we expanded our unit tests for different operating systems, learning how to:

- Use `case` statements account for different variables on different OSes
- Test for specific parameters using `with`
- Add a `:params` section to define any variables that use the `param` pattern

Looking for an extra challenge? Continue writing the unit tests for this module, specifically focusing on the `config` class. You'll use the same processes as we did here, just with the `contain_file` option.

Facts Basics

At this point in the course, Facter isn't anything new to us. We've used facts rather substantially to manipulate our Puppet module across various distos. But we haven't taken the time to really *look* at Facter itself and how our facts are structured, created, and set up for us to use.

So as we know, we can use Facter to query for basic facts about system. To receive a full list of these facts, all we have to do is run:

```
$ facter
```

We're then promptly met with a long list of information about our host. The information is then structured in two ways: as either a flat or a structured fact. Flat facts look like our `timezone` or `virutal` facts, a one-to-one key-value pair:

```
  timezone => UTC
  virtual => kvm
```

On the other hand, structured facts are organized in an array, and contain a number of key-value pairs that fall underneath an overall key. This is how the `os` fact that we're previously worked with works:

```
$ facter os

{
  architecture => "amd64",
  distro => {
    codename => "bionic",
    description => "Ubuntu 18.04.2 LTS",
    id => "Ubuntu",
    release => {
```

```
      full => "18.04",
      major => "18.04"
    }
  },
  family => "Debian",
  hardware => "x86_64",
  name => "Ubuntu",
  release => {
    full => "18.04",
    major => "18.04"
  },
  selinux => {
    enabled => false
  }
}
```

But we can go even further here. We can also classify our facts by *how* they collect information, giving us core facts, custom facts, and external facts.

Core facts are the ones we just listed above. They ship with Facter and we can use them without doing any special work. Custom facts are ones generated via a Ruby script that produced the needed values. Finally, external facts are provided via static data or through and executable script. This is *not* a Ruby script we include in one of our load paths, but something like a Bash or Python script.

Regardless of which kind of fact we write, all non-core facts are treated like plugins, and the preferred way of supplying them is through modules. Specifically, custom facts are stored in `MODULE/lib/facter`, while external facts live in `MODULE/fact.d`. We can also work from the `/opt/puppetlabs/facter` and `/etc/puppetlabs/facter` directories.

Now let's get started creating some new facts.

Wrap Up

In this quick Facter overview, we learned:

- How to query for facts
- How flat versus structured facts looked
- The different types of facts and where to find them

External Facts

When it comes to providing Puppet with more than just our core facts, we already know we have a couple of options to work with, but for now we're going to be focusing on external facts.

We touched on how we have a few options for where we host external facts: preferably within a module, but also with the option to place them in some general configuration directories. Currently, there are four places we can store facts that Puppet will check on its own:

- `MODULEDIR/MODULE/fact.d`
- `/opt/puppetlabs/facter/facts.d/`
- `/etc/puppetlabs/facter/facts.d/`
- `/etc/facter/facts.d/`

None of these directories are created on their own, so if we'll be adding external facts, that is the best place to start. Let's first add a fact to our `apache` module that notes which web servers we're using.

Basic Facts

Move into the module's directory:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
```

Then create a `facts.d` directory:

```
$ sudo mkdir facts.d
```

When we add an external fact, we have a few options for our format: we can write an "executable" fact (which is any kind of non-Ruby script our server can execute), or a plain fact in a structured data file. These files should be written in YAML, JSON, or plain text (`.txt`), and simply contain key-value pairs, although YAML and JSON both support setting multi-level facts.

Since we're already familiar with YAML, let's use this as our file type:

```
$ sudo vim facts.d/web_server.yaml
```

Then add our key-value pair:

```
---
web_server: Apache

If we expect to later expand our `web` facts, we could also set this up as:

---
web:
  server: Apache
```

Save and exit.

Executable Facts

Now, let's consider writing an external fact that relies on a script. For this, we want to create a fact that will tell us how many users are on the system. To do this, we're going to just use a Bash script that will evaluate the following command:

```
$ who | wc -l
```

That said, it's not enough for us to just send the output of that command to our STDOUT. Any evaluated facts need to follow the `key=value` format for their output for the fact to work.

Since this fact isn't module-specific, we're going to start by creating a new `facter/fact.d` directory in `/etc/puppetlabs`:

```
$ sudo mkdir -p /etc/puppetlabs/facter/facts.d
$ cd /etc/puppetlabs/facter/facts.d/
```

Now, let's write our Bash script:

```
$ sudo vim user_count.sh
```

Let's add our interpreter information:

```
#!/bin/bash
```

Then assign the output of our `who | wc -l` command to a variable:

```
count=$(who | wc -l)
```

And use the `echo` command to output our desired fact:

```
echo "user_count=$count"
```

Save and exit, then ensure the script is executable:

```
$ sudo chmod -x user_count.sh
```

Test the script:

```
$ ./user_count.sh
```

Viewing Our Facts

To see our `user_count` script, all we have to do is query Facter:

```
$ sudo facter user_count
```

To view our any facts added via module, we need to add the `-p` flag:

```
$ sudo facter -p web_server
```

Wrap Up

We created two external facts in this section, learning how to:

- Create simple facts based on key-value pairs and more structured data
- Write scripts whose output works as an external fact
- Query for our new facts

Custom Facts

Beyond writing external facts, we also have the option to write *custom* facts, which are facts written and evaluated in Ruby.

These custom facts are not stored like our external facts, but instead work the same as plugins do. These facts can be placed in any directory that Ruby reads as part of its load path, as long as it is in a file named `facter`. In our instance, we'll be adding our custom fact to our `apache` module.

Let's move into the module's directory:

```
$ cd /etc/puppetlabs/code/environments/production/modules/apache
```

Ruby looks for the presence of a `lib` directory for plugins. Let's add this, along with the `facter` subdirectory:

```
$ sudo mkdir -p lib/facter
```

Now we can start crafting our custom fact. Specifically, we'll be adding a fact that outputs how many processes the Apache service is using, which means we'll have to consider the differences between Red Hat and Debian-based servers.

Let's begin by creating and opening our `apache_processes.rb` file:

```
$ sudo vim lib/facter/apache_processes.rb
```

Next, we want to define the name of the fact. This can either be a completely new fact, or we can change the behavior of an existing fact by providing that fact's name:

```
Facter.add(:apache_processes) do
end
```

As we can see, the name of our fact in this instance is `apache_processes`.

Now we want to provide Facter with the code it needs to evaluate to get the results of our fact. This is done with a `setcode do` stanza:

```
Facter.add(:apache_processes) do
  setcode do
  end
end
```

Simpler facts can substitute this for a single line: `setcode 'CMD'`. Puppet suggests using this for simple, short commands.

We can now write the evaluation statement for the fact itself. To find how many Apache processes we have running, however, we need to consider our distro. So let's go ahead and craft a case statement:

```
Facter.add(:apache_processes) do
  setcode do
    osfamily = Facter.value(:osfamily)
    when 'Debian'
    when 'RedHat'
    end
  end
end
```

Then add our code. To run a Linux command through our custom fact, we use the `Facter::Code::Execution.execute` function:

```
Facter.add(:apache_processes) do
  setcode do
    osfamily = Facter.value(:osfamily)
    case osfamily
    when 'Debian'
      Facter::Core::Execution.execute('pgrep apache2 | wc -l')
    when 'RedHat'
      Facter::Core::Execution.execute('pgrep httpd | wc -l')
    end
  end
end
```

Finally, since this fact only works on Linux-based OSs, we can use the `confine` option to ensure the fact is only added to Linux-based servers, not Windows:

```
Facter.add(:apache_processes) do
  setcode do
    confine :kernel => "Linux"
    osfamily = Facter.value(:osfamily)
    case osfamily
    when 'Debian'
      Facter::Core::Execution.execute('pgrep apache2 | wc -l')
    when 'RedHat'
      Facter::Core::Execution.execute('pgrep httpd | wc -l')
    end
  end
end
```

Now let's save and exit the file. To deploy this fact on any server (that has the Linux kernel), we can use our `puppet agent -t` command. Notice, however, that this fact is added to *all* servers, not just the ones that use the Apache module. This is true of any module-specific facts, and there's no way around this.

For example, we can query for our fact on our master, which doesn't use the Apache module at all:

```
$ sudo puppet agent -t
$ sudo facter -p apache_processes
0
```

Notice that it return with a `0`.

In contrast, should we run this on our CentOS or Ubuntu nodes, we'll get a different response:

```
$ sudo puppet agent -t
$ sudo facter -p apache_processes
6
```

Finally, one thing we did not address for the sake of ease up until this point is that the `-p` flag with Facter is going to be removed in future releases. Instead, PuppetLabs suggests we use the `puppet facts find` command to output all Facter values. We cannot query our facts on a single-fact basis with this, however. To achieve a similar result as `facter -p`, I suggest using `grep`:

```
$ sudo puppet facts | grep apache_processes
```

Wrap Up

In this section, we learned the components of a custom module by:

- Creating a new fact based on our Apache module
- Ensure the fact only works on Linux
- Using Ruby to evaluate a Linux command and provide that result to Facter

Profiles

At this point, our Apache module has classes, defined types, facts, and unit tests. But when we configure servers, especially mass amount of servers, we generally don't just install Apache and call is a day. We also generally frequently apply the same sorts of modules to the same sorts of servers. We can suspect our web servers have Apache and PHP, and maybe we have a few full LAMP stacks with MySQL thrown in, too. Well, instead of having to apply our modules individually to our servers, we can create *profiles* and *roles*, which are wrapper classes that let us configure related swaths of infrastructure.

Profiles contain a number of relate modules. They're called "component modules" in this setup, due to how they configure single components of infrastructure. Profiles should be set up to configure some kind of "layer" in your technology stack.

Roles, meanwhile are collections of profiles that configure a complete system.

In this lesson, we're going to be setting up three profiles: a `base` profile that contains the modules we want on *all* our servers, an `apache` profile that includes our module, a vhosts definition (that also pulls in PHP), and a `mysql` module that configures MySQL and creates a database.

But before we do any of that, we need to pull down the component modules we need to complete this. Let's move into our `modules` directory:

```
$ cd /etc/puppetlabs/code/environments/production/modules/
```

And grab the `motd`, `php`, and `mysql` modules:

```
$ sudo puppet module install puppetlabs-motd --version 3.0.0
$ sudo puppet module install puppet-php --version 6.0.2
$ puppet module install puppetlabs-mysql --version 9.0.0
```

Now we want to create the location where we store our profiles. Since a profile is just a layer of indirection, it's simply just another class. We just have to use the PDK to create our `profile` location:

```
$ sudo pdk new module profile
$ cd profile
```

Now let's start by creating our `base` profile:

```
$ sudo pdk new class base
$ sudo vim manifests/base.pp
```

The class we create is essentially going to function in the same way we have our `node` definitions now. We'll `include` our NTP module, and add a message of the day through the `motd` module:

```
# @summary
#  Core modules for all servers
#
# @example
#   include profile::base
class profile::base {
  include ::ntp

  class { '::motd':
    content => "This host is managed by Puppet!\n",
  }

}
```

Note how we update our comments. Notice, too, that to include our `ntp` class we add a double colon to the front of our class name. This allows us to back up from the `profile` scope and include our `ntp` class. Think of it like dropping down a directory on the command line. Finally, by using `class` to call our `motd` module, we can overwrite any default values

with ones that will work best for our own hosts. You'll find yourself using this most frequently when using generalized modules that have to work for a lot of hosts, such as Puppet Forge modules.

Save and exit the `base` class.

Now let's create our `apache` profile:

```
$ sudo pdk new class apache
$ sudo vim manifests/apache.pp
```

For this, we can start by including our base `apache` module:

```
# @summary Basic Apache configuration
#
# @example
#   include profile::apache
class profile::apache {
  include ::apache
}
```

And then we also want to add our new `php` module:

```
  class { '::php':
    pear => true,
  }
```

Save and exit the file.

Finally, let's repeat this process so we'll have a working `mysql-server` profile for our next step:

```
$ sudo pdk new class mysql-server
$ sudo vim manifests/mysql-server.pp
```

```
# @summary
#   Manages MySQL
#
# @example
#   include profile::mysql
class profile::mysql {
  class { '::mysql::server':
    root_password           => 'passwordhash'
    remove_default_accounts => true,
  }
}
```

Next, we're going to combine our profiles into a single role and check out our resulting LAMP stack!

Wrap Up

We've now created a layer of indirection between our component modules and what will eventually be assigned to our servers, learning:

- The purpose of roles and profiles
- How to create a profile
- How scoping works

Roles

If profiles set up specific areas of our server, then roles exist to configure an entire server based on its purpose in your infrastructure. Roles should *only* contain profiles, nothing else, making this an exceptionally quick and easy lesson.

Roles, like profiles, are stored as manifests under an overall `role` module we have to create. Let's do this now:

```
$ cd /etc/puppetlabs/code/environments/production/modules/
$ sudo pdk new module role
```

Use the same defaults as we have when creating prior modules; be sure to deselect "Windows" when given the option.

Create the `lamp` role:

```
$ cd role
$ sudo pdk new class lamp
```

Now, to create our role, all we have to do is `include` the relevant profiles:

```
$ sudo vim manifests/lamp.pp

# @summary
#   Configures a full LAMP stack
#
# @example
#   include role::lamp
class role::lamp {
  include profile::base
  include profile::apache
```

```
  include profile::mysql
}
```

Save and exit the file.

Let's test this out by updating our `node` definitions:

```
$ sudo vim ../../manifests/site.pp

node USERNAME#C.mylabserver.com {
  include role::lamp
}

node USERNAME#C.mylabserver.com {
  include role::lamp
}
```

Save and exit.

Now, run perform a Puppet run on both servers!

> At the time of writing, there was an issue with the MySQL module on our Ubuntu hosts, in which I had to initially run a `sudo apt --fix-broken install`. If you encounter any issues, run this same command, then rerun the `puppet agent -t` command. It should work properly the second time.

Wrap Up

In this little section, we learned:

- Where to add roles

- That roles should only include profiles
- How to apply roles to our servers

Groups and Classification

While much of our course up until this point has been relevant to both Open Source Puppet and Puppet Enterprise, we're going to begin to work exclusively in the PE realm for much of the remaining time here.

Despite this, you'll find we're already a little bit familiar with the concept of classification, as in we've already done it. Classification is the practice of configuring our nodes by mapping our classes, parameters, variables, and Hiera data to our nodes. And up until now we've done with exclusively with our main manifest. Yet with Puppet Enterprise, we have some better options.

To start, ensure you are logged in to the PE Console, then click on **Classification** on the left menu.

We've seen this page before! Here we have our list of existing node groups, all preconfigured by Puppet when we installed it on our server. In PE, we won't be classifying our nodes by adding information to our main manifest. Instead, we'll be classifying things on the basis of node group. So let's use these existing groups to explore how classification works.

Right off the bat, we can see that our two most general node groups are "All Environments" and "PE Infrastructure," and the "All Environments" group has a defining feature. It's marked with an **Env group** tag. This is because there are two kinds of groups we have access to: *classification node groups* and *environment node groups*. It goes without saying which one our "All Environments" group is a part of.

The difference between these two node groups is simply that while we assign our normal classification data of classes, parameters, and variables to our classification groups, we use environment groups to only assign *environments* to nodes. This is something we'll touch on in greater depth in a bit, but it's important to understand the distinction now.

Let's go ahead and expand one of these groups, specifically our **PE Infrastructure** option. These overall node group is used to configure our Puppet Enterprise install, since Puppet *does* use itself to set up Puppet Enterprise. So underneath our overall infrastructure grouping, we have everything broken down further into subgroups or child groups. These

groups configure specific components of our PE setup, such as the master, our cert authority, PuppetDB, and the very console we're looking at now.

Parent and child groups use a concept called *inheritance* to apply our modules on the nodes in our groups, and helps us refine our modules so they are work under certain circumstances. Think of it a little like how we overrode certain settings in our profiles and roles, but on a group or environment level.

When it comes to group inheritance, the process is similar to Hiera inheritance or any other scope we've seen in Puppet: the children inherent all classifications defined in the parent. If we want a child group to use different parameters, then we simply overwrite them in the child class. Again, this is much like we did in our profiles. That said, we don't overwrite our classes as a whole. If the parent class uses the `mysql` class, then the child class will always use it as well.

So now let's actually take a look at one of these existing groups before we do anything ourselves. We've spent quite a bit of time on our Puppet master, so let's choose this option specifically. But first, let's click it's parent group, **PE Infrastructure**.

Immediately, we can see that no actual nodes are assigned to this overall group. So instead, let's look at our **Configuration** page. Here we can see that the overall `puppet_enterprise` class is applied to *all* nodes under this parent group. We can also see where Puppet provided our master's FQDN as a parameter override for many of the values in this class.

Now, let's step back and look at the **PE Master** group. This has our master assigned to it, and when we look under **Configuration**, we can see a number of related classes assign to this node. We even have a little experience assigning classes to this node already. In our installation videos, we added the `pe_repo::platform::el_7_x86_64` class, which let us use our single-command installer on CentOS. We can even see *when* we did this, under the **Activity** tab.

Wrap Up

Puppet Enterprise provides us with an additional group-based classification system. Regarding this system, we learned:

- The different between an environment group and a classification group

- How inheritance works across groups
- How to navigate groups from within our PE console

Adding Groups

Let's now go ahead and actually create a group, or rather series of groups. When it comes to creating groups for our infrastructure, we need to think about the best way to break things down. Generally, when it comes to creating groups, we want to think of our high-level business requirements, then consider how this reflects the actual infrastructure we use within our organization.

For example, let's say we run a web application, and one of our primary goals is ensuring our web app is up and running. It's safe to say, then, web servers are an important part of our infrastructure. Which means we most likely want an overall *web* or *web servers* group. This group would most likely contain our Apache or Nginx module, and anything we need for the overall configuration of a web server at the most basic of levels.

We then would then want to include child classes for more specific use-cases. Perhaps we need dev servers under a *web dev* group, or we have some specific hosts that need a full LAMP stacks (although these would probably be our web dev servers, realistically). Maybe we would want to take this a step further and have another experimental group under our LAMP stack where we override our MySQL parameters and we install MariaDB instead. Regardless, when it comes to adding groups the goal should be to think of the highest-level you can work at first, then go from there.

So let's go ahead and actually add some groups now. We'll pulling from our example and start by creating a general *web* group, then configure a *lamp* group underneath that for any hosts that require the full stack.

To create a group, we need to log in to our Puppet Enterprise console and return to our **Classification** page. Click **Add group...**.

This expands a menu there we're given the option to define our parent group, provide the group name and environment, as well as write a description of the environment itself. We also have the option to check if the group is an environment group.

For our overall web server group, we specifically want to leave the **Parent name** as *All Nodes*, then provide the **Group Name** of *Web Server*. Note that there are no real capitalization or style conventions to follow for the name, but make

sure it's plain text and alphanumeric. Finally, set the **Environment** to *production*. Do *not* check the environment group option. Add a description only if you want.

Click **Add** when done.

Now let's repeat this process to add our *LAMP* group. This time, set the **Parent name** to *Web Server* and the **Group name** to *LAMP*. The rest of the settings should remain the same.

If we scroll to the bottom of our list of groups, we can now see our Web Server group at the bottom. Click on it to expand the group and display our LAMP child group.

Let's now click on the **LAMP** group itself. On the **Rules** tab we have the option to either define fact-based rules to add servers to our group, or we can just supply the certname under the **Pin specific nodes to the group** option.

Adding Rules

To add hosts to our group via rules, all we need to do is define a fact, define an operator, then set the value of that operator. To see this in action, let's see how we can add our non-master hosts to this specific group.

For this, let's first select our *fqdn* fact from the dropdown. We now want to set our operator. These operators are the same basic operators we see in math and programming: `=` for exact matches, `~` for close matches, `!=` to match anything *except* the provided value, and a number of additional options such as greater-than or less-than.

Now, in an ideal world, our hosts would most likely have hostnames related to our purpose, like `web1.mylabserver.com`, `web2.mylabserver.com`, and so on and so forth. If this were the convention we were using, we could set our operator to `~` and then provide `web` as our value. We can even try this our with the username our current hostnames are based on. For example, my servers use the `ellejaclyn` username, so if I type `ellejaclyn` for my value, I can see that I have three node matches.

That's not how Linux Academy's playground servers are set up, though, unfortunately. So instead let's change our operator to `!=` and define our master's hostname. Click **Add rule**, then **Commit 1 change**.

Alternatively, we also have the option of adding host-level facts to each of our non-master nodes defining a role fact, then simply using that role to assign our nodes. But this is outside the scope of the course. That said, we've already covered how to add custom facts, so if you prefer this way or just want the extra challenge, feel free to add a `role` fact. You may also choose to simply just pin each host by certname.

Finally, let's finish out this lesson by logging into our master, and removing our node definitions from our main manifest:

```
$ sudo vim /etc/puppetlabs/code/environments/production/manifests/site.pp
```

Simply delete both node definitions for our non-master hosts, and clear out any data you have in the `default` definition.

Wrap Up

We've now created two groups, for us to work with (a parent and a child) and learned how:

- To add either a classification or environment group
- Assign nodes to groups using the certname
- Assign nodes to groups using fact-based rules

Defining Data

Although we have our groups created, and our nodes assigned to our group, we still haven't actually assigned any classes to our groups yet. To do this, we want to return to the **Classification** page of our PE console and first select our **Web Server** group.

For our group classification setup, we're going to be abandoning our roles and profiles in favor of assigning our component modules directly. This which is a perfectly valid way of classifying our nodes, and will also let us demonstrate some of the more interesting classification features we have open to us. To do this, let's first click on the **Configuration** tab.

We can now begin to add our classes. For our overall web server configuration, we're going to be specifically adding our `apache` module, and that same PHP module we used in our overall `apache` profile.

To add a class, simply search for it in the **Add new class** box, then click **Add class**. Make sure the classes you add are `apache` and `php` specifically, not any the subclasses they contain. Notice, however, that it truly is only our *classes*. Should we search for our `apache::vhosts` defined type, it is not available to us. Should we want to access it and assign it to our groups in this method, we would need to create an additional wrapper class for it, then add that class to our `init.pp` like we normally would.

Adding Parameters

If we now look at our added classes, we can see we have the option to add different parameters to each. Let's click on the **Parameter name** dropdown for `apache`. Here, any of the parameters we provide to our `init.pp` file are available for us to change. Let's do this by changing our `install_ensure` option to `latest`. Click **Add parameter**.

Now, let's commit the changes we made by clicking **Commit 2 changes**.

We also want to update our *LAMP* group, so let's now select this group from our **Classification** page. Right now, our Web Server group installs Apache and PHP. This means that our LAMP group does, too. But we still need to add MySQL, so let's use the **Add new class** ability to add our `mysql::server` class. Click **Add class** when done.

But what if we want our LAMP stack to use different Apache parameters than the ones we have for our Web Server group? Well, this is where the **Data** section of our configuration comes in. Simply search for the `apache` module, select your desired variable, and override the parameter that way.

Let's do this by setting our `install_ensure` variable back to `present`. Save the parameter, then commit the changes.

Adding Variables

Using the parameters option for each of our classes is not the only way to provide variables. If we wanted to set any general variables for our overall group itself, instead of working with them on a class-by-class basis, we can select the **Variables** tab, then provide any variable we use as a key with whatever value we want.

For example, this would be an ideal place to set the information for our virtual hosts file, and by adding something like `servername` to our global variables, we could also reference it in other modules that may need that same information.

Note, however, that our variables cannot use the `::` scope indication, so we can't specify module-specific variables. These should be done by adding parameters to classes, regardless.

Wrap Up

In this lesson, we assigned classes to our groups, learning:

- How to set class-specific parameters for each group
- How to set parameters for classes assigned to the parent group
- How these parameters work alongside group inheritance
- How to set top-level variables that can be used across the group

Basic Environment Management

Up until this point, we have been exclusively working in the `production` environment, but in actual practice any good Puppet setup, any good *code* setup really, does not work exclusively on the `production` environment. As such, knowing how to use multiple environments in Puppet is integral in using the platform.

So the most basic question we can ask here is: What is an environment? It might help to think of an environment almost almost as a git branch. It lets us segment our Puppet setup into different versions of the same modules, to test on different sets of servers. This lets us test our code and our infrastructure setup as a whole. If our infrastructure is segmented wholly between teams, we also have the option to use environments to mimic this divided infrastructure setup.

So how do we add an environment? All we have to do is add a new directory into our `/etc/puppetlabs/code/environments` path, and ensure that the new environment contains both `manifests` and `modules` directories. Let's go ahead and create an environment called `staging`:

```
$ cd /etc/puppetlabs/code/environment
$ sudo mkdir -p staging/manifests staging/modules
```

We also have the option to add a `hiera.yaml` and `data` file if we wish to use environment-level Hiera data, and an `environment.conf` file should we want to update any configurations. We only have four settings for this file, and we can see a commented description of these in our `production` environment's `environment.conf`:

```
$ cat production/environment.conf

# modulepath = ./modules:$basemodulepath
# manifest = (default_manifest from puppet.conf, which defaults to ./manifests)
# config_version = (no script; Puppet will use the time the catalog was compiled)
# environment_timeout = (environment_timeout from puppet.conf, which defaults to 0)
```

```
    # Note: unless you have a specific reason, we recommend only setting
    # environment_timeout in puppet.conf.
```

Essentially, this lets us change our `manifests` and `modules` directories if we so choose, define an executable script that will generate a `config_version` (or configuration version), or override the default environmental timeout. For the most part we will not have to update this file. These values tend to instead be set globally in our `puppet.conf` file.

As for assigning nodes to various environments, we have a number of options for this. We can assign an `environment` fact to each node, we can define a group in our PE Console, we can simply assign our nodes in the environment's main manifest (as we did earlier in this course), or we can take a more full-featured option, which is what we'll be looking at for the rest of this course.

Puppet offers us the option to use r10k and Code Manager to manage our Puppet code and its associated environments as a whole. Code Manager also allows us to sync our Puppet code across masters in any multi-master setups. r10k is the backend to Code Manager, but can be used on its own to manage our environments, as well.

Wrap Up

In this section, we learned some environment basics, including:

- How to create an environment
- Environment use cases
- How to change our environment settings

Using a Control Repo

Before we set up Code Manager itself, we want to set up what is known as a *control repo*. A control repo is a Git repository that lets us use version control to track, maintain, and deploy our Puppet code. As we update the control repository, our respective environments will be updated, too.

At minimum, a control repo needs to contain a `production` branch, a Puppetfile, and an `environment.conf` file with a modified `modulepath` that lets us set environment-specific modules and settings. And while we can create a fresh repository and add these files ourselves, PuppetLabs suggests that most users copy the example control repository they provide, so we will do exactly that.

Create a Key Pair

First, we need to create a key pair that we can use along with our control repo; since this is stored in the `puppetserver` configuration directory, we'll need to drop become `root`:

```
$ sudo -i
# ssh-keygen -t rsa -b 2048 -P '' -f /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa
```

Note that we must *not* include a password for this key.

Set the key's ownership to the `pe-puppet` user and group:

```
# chown pe-puppet:pe-puppet /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa
```

Set Up Git and GitHub

Now we want to create our Git repository. We'll be using GitHub in this course, but if you have a preferred Git host, feel free to use that.

Log in to GitHub, then click **New** to create a new repository. Name the repository something relevant. I'll be going with `control-repo`. Click **Create Repository**.

Now we want to associate our relevant SSH key. Click on the **Settings** tab, then click **Deploy keys**. On the command line, retrieve your public key details:

```
# cat /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa.pub
```

Add it to the **Key** text box on the Deploy keys page. Give the key a title, check off **Allow write access**, then select **Add key**.

Copy down the control repo. Ideally, we would use a separate workstation for this, but for the sake of simplicity, let's store it in our `cloud_user`'s home directory on our master.

```
# exit
$ git clone https://github.com/puppetlabs/control-repo.git
```

If we look in our new repo, we can see it contains all the required files (including a `Puppetfile`, which we'll discuss later in this section): the `environment.conf` and `hiera.yaml` configuration files, our usual `data` and `manifests` directories, related helper scripts (`scripts`), and what is called a `site-modules` directory that lets us store our roles and profiles in a more structured system.

```
$ cd control-repo
$ ls
```

Let's go ahead and replace the existing origin for this directory to our new Git repo:

```
$ git remote remove origin
$ git remote add origin GIT_REPO_URL
```

And push our data to GitHub:

```
$ git push origin production
```

Wrap Up

We prepared our `control-repo` in this section, ensuring we:

- Created a Git repo
- Cloned the suggested PuppetLab's `control-repository`
- Set our new repo as the remote to our own repo

Code Manager

With our `control-repo` in place, all that's left is for us to set up is Code Manager itself. To finish up our setup and get Code Manager working alongside our repo, we want to switch to our Puppet Enterprise console. Log in, then navigate to the **Classification** page.

Expand the **PE Infrastructure** group, then select the **PE Master** group. Switch to the **Configuration** tab.

We want to look for the `puppet_enterprise::profile::master` class and add some additional parameters: Let's set `code_manager_auto_configure` to `true`, `r10k_remote` to the *SSH* option for our GitHub repo (it should begin with `git@github.com:`), and the `r10k_private_key` to `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`.

Commit the changes, then navigates to **Nodes**, select your Puppet master, and perform a Puppet run to enforce these changes.

Create a Deploy User

We also want to create a user for Code Manager to use. Navigate to **Access control** > **Users** and create a new user. I gave mine a name of `Code Manager` and a login of `codemanager`. **Add local user**, then select the user and generate a password.

Switch to the **User roles** page and select **Code Deployers**. Assign our new user to this role.

Finally, we need to finish out our user creation through the `puppet-access` command line tool. From the Puppet master, run:

```
$ puppet-access login --lifetime 365d
```

And log in as the `codemanager` user.

We can then test that Code Manager is working by running a:

```
$ puppet-code deploy --dry-run
```

Wrap Up

In this section, we set up Code Manager by:

- Updating our Puppet Master group; specifically, the `puppet_enterprise::profile::master` class
- Adding a new Code Manager user
- Testing our Code Manager setup

The Puppetfile

With Code Manager working, we can now begin defining our environments and classifying our nodes within these environments. Unlike in previous lessons where we did this with the main manifest or through the PE console **Classification** page, we'll be working with what is known as a Puppetfile.

If we look in our `control-repo` we can see that one of these has already been created in our default `production` branch. In this file, we'll define which modules we want to run by providing the name of the module and the repository the module can be found at. This is why we pushed our `apache` module to GitHub.

But first, let's see how we can work with a new environment. With Code Manager, we can manage a new environment by simply adding a new branch to our repository. Remember the `staging` directory we created earlier in our `environments` folder? Let's go ahead and work with that:

```
$ git checkout -b staging
```

Now, open `Puppetfile`:

```
$ vim Puppetfile
```

The top of the file initially defines the location of the Puppet Forge. This lets us easily add Forge modules to our nodes without needing to define the specific GitHub repo. In fact, let's do this with the `ntp` module. To add a Forge-based module, all we have to do is list the name of the module and desired version, like so:

```
mod 'puppetlabs-ntp', '8.0.0'
```

We're also provided with examples of how to add any non-Forge modules hosted on Git repositories. We'll just with the same `mod MODULENAME` format, then define the Git location. If we're using a non-master branch, we'll also need to define the branch name. Let's add our `apache` module:

```
mod 'apache',
  :git => 'YOUR_GIT_INFORMATION'
```

Save and exit.

We now want to push our changes to our overall `control-repo`:

```
$ git commit -am "Added NTP and Apache modules"
$ git push origin staging
```

Note how we push to the `staging` branch.

We can now apply this to our nodes assigned to this environment. But the problem is that right now there *are* no nodes in this environment. Let's switch to our Puppet Enterprise Console and log in.

Move to the **Classification**, then add a new *environment* group, with the **Environment** set to `staging`. Add it to the **All Environments** parent group.

Click on the new group once added. Assign either of your nodes. I'll be adding my CentOS host. **Commit** the changes.

We can now return to our command line and deploy our environment. We use the `--wait` flag so we can view any additional feedback and ensure the process worked:

```
$ puppet-code deploy staging --wait
```

Success! But what happens to our existing environment setup when we do this? Well, let's go check that out:

```
$ cd /etc/puppetlabs/code
```

What's that? It won't let us `cd` into our `code` directory as our `cloud_user` anymore? Since we enabled Code Manager, Puppet automatically changed the permissions to our `environment` directories, locking them down for even superusers. Let's switch to `root`:

```
$ sudo -i
```

And view our `staging` environment:

```
$ cd /etc/puppetlabs/code/environments/staging
```

If we now look in our `modules` directory, we can see the two modules from our Puppetfile have been added:

```
$ ls modules
apache ntp
```

And what of our `production` environment? Well, since we haven't deployed any code to it, it's not even there. We now only have `staging`.

Wrap Up

We learned how to use Code Manager to actually setup our hosts, through the use of a Puppetfile, which lets us:

- Define the modules we want to use
- Set particular information about these modules, such as specifying a specific path or branch
- Push changes to our modules through the use of a `puppet-code deploy`

Orchestration Overview

The Puppet orchestrator allows us two benefits: the ability to perform Puppet runs on defined nodes, node groups, or PQL queries, and the ability to run Puppet tasks, which are one-off commands we can execute on our target machines. This gives us the ability to manage the rollout of our hosts, instead of relying on the every-thirty-minute configuration changes commonly used.

The Puppet orchestrator is comprised of two protocols: Puppet Execution Protocol (PXP) and Puppet Communications Protocol (PCP). The PXP is a message format used by `pe-orchestration` services that requests that the executed task be run on the remote host. This is also what receives any responses regarding that task.

Meanwhile, the PCP is the underlying protocol the PXP itself uses. This describes how PXP messages get routed to an agent and then back to the orchestrator. It's run via the JVM-based service run as part of `pe-orchestration-services` on our master of masters.

We'll get into the details on how this works when we dive into the individual "Puppet Run" and "Task" sections, but for now let's consider what we need to complete tasks with the orchestrator.

Puppet orchestrator itself is a Puppet Enterprise product, and already came configured with our PE install. That said, we have to enable it for our users, and we also have the option to adjust our orchestrator configurations for our setup as a whole, as well as for specific users.

Allowing Orchestrator Access

To provide access to orchestrator, we need to make sure our user has access to the appropriate permissions through our RBAC system. Since the user we created has full admnistrator privileges, this isn't a problem, but other users will need the following role access:

Type	Permission
Puppet agent	Run Puppet on agent node
Job orchestrator	Start, stop, and view jobs
Tasks	Run tasks
Nodes	View node data from PuppetDB

Should we wish to use the `puppet-job` command from the CLI as well, we need to enable our `puppet-access` token. This is no different than when we performed the same task for Code Manager:

```
$ puppet-access login --lifetime 7d
```

Configuration Options

As orchestrator is managed by PE, it's not a service whose global configuration is one we want to manage. Instead, we can create user-specific configurations by creating the `~/.puppetlabs/client-tools/orchestrator.conf` file.

The `.puppetlabs` directory was created when we generated our first access token, but we need to add the `client-tools` directory underneath this:

```
$ mkdir .puppetlabs/client-tools
```

Then create our configuration file for orchestrator:

```
$ vim .puppetlabs/client-tools/orchestrator.conf
```

We need to format this file with JSON, and we have five settings we can change: `service-url`, `environment`, `cacert`, `token-file`, `color`, and `noop`:

```
{
  "options" : {
    "service-url": "https://PUPPET MASTER HOSTNAME:8143",
    "environment": "production",
    "cacert": "/etc/puppetlabs/puppet/ssl/certs/ca.pem",
    "token-file": "~/.puppetlabs/token",
    "color": true,
    "noop": false
  }
}
```

Since we don't need to set any of these for our own user, however, let's just leave the `color` parameter:

```
{
  "options" : {
    "color": true
  }
}
```

Save and exit. We're now prepared to use orchestrator!

Wrap Up

In this section, we learned:

- The function of orchestrator within our Puppet infrastructure
- How to enable orchestrator for a user through RBAC and the `puppet-access` command
- How to set user-specific configuration options

On-Demand Puppet Runs

Previously, when we needed to trigger a Puppet run on any of our nodes, we used a `puppet agent -t`, but Puppet Enterprise provides us with with alternative options for this, allowing us instead to use either the PE Console or `puppet-job` command line tool to orchestrate on-demand Puppet runs on defined sets of nodes.

Using the Console

Let's begin by trying it out on our PE console. Log in as your user, then navigate to the **Puppet** link, under **Run** on the left menu. This allows us to set up the circumstances of our Puppet run by defining the desired environment to use, the run mode (such as `noop` for a test run), and the list of nodes you wish to run.

The list of nodes, specifically, is defined in the **Inventory** section, and we have the option of supplying a **Node list**, **PQL query**, or **Node group**.

Using a **Node list** or **Node group** is the simplest option. We can search for our nodes or group, select it, then apply the Puppet run. The **PQL query** option is a little more advanced, although we're provided with a list of common queries that can help us work through common orchestration use-cases. Do we need to update every server that uses a specific package? Then select **Nodes with a specific resource**. Want to target everything using a specific class? Select **Nodes with a specific class**, and so on and so forth.

In fact, let's actually go ahead and try this. Select **Nodes assigned to a specific environment**). By default, this is set up to perform a Puppet run on `production` hosts. Let's update this to `staging`:

```
nodes[certname] { catalog_environment = "staging" }
```

Click **Submit query**. Our two nodes come up in the node list. If everything looks good to you, click **Run job**.

We're then taken to a processing page for our job. As the nodes complete their Puppet run, they will be reported as either succeeded, failed, or skipped.

The Process

So what exactly happened when we triggered these Puppet runs through orchestrator? Let's go through it step-by-step:

1. The orchestrator requests classification information about the nodes on which the job is run.
2. The orchestrator requests the environment graph from the Puppet Server.
3. The orchestrator create the job ID and begins to poll nodes.
4. The orchestrator queries PuppetDB for the agent version.
5. The orchestrator triggers the PCP broken, which starts to perform any actions on the targeted nodes.
6. The PXP agent returns its results to the PCP broker.
7. The orchestrator receives the results and requests run reports from PuppetDB.

The CLI

Of course, we don't have to restrict ourselves to using the PE console. Let's switch to our command line (either on our master or our workstation), wherever you set your token in the previous lesson.

We can perform all the same things we just did, only with the `puppet-job` command. To trigger a run, we would specifically use `puppet-job run --QUERY_TYPE QUERY`. So to run that same PQL query we just used, we would run:

```
puppet-job run --query 'nodes[certname] { catalog_environment = "staging" }'
```

This will report the same output we saw on our console. We can also view this report from the Console under the **Jobs** page if desired.

Wrap Up

In this lesson, we took a look at using orchestrator for on-demand Puppet runs, learning:

- How to trigger a run through the CLI and console
- How to use node names, node groups, and PQL queries to define which nodes we're working with
- The underlying events that occur when we request an orchestrator run

Using Tasks

Puppet's orchestrator can do more than just perform simple coordinated Puppet runs. One of the primary features of orchestrator is that it allows us to run ad-hoc commands across our nodes in a similar manner to how we performed Puppet runs. Let's go ahead and try with now with one of our default tasks.

Using the Console

Log in to the Puppet Enterprise console, then move to the **Task** page from the left menu.

We're first prompted to select a task. Here we have a few defaults shipped with PE: `facter_task`, `package`, `puppet_conf`, `puppet_enterprise::dhparam`, `service`, `service::linux`, and `service::windows`. Task names should be self-explanatory. `facter_task` lets us query for facts, `package` manages packages, and so on.

Let's try to query for our `os` fact. Select `facter_task` from the menu, and provide an additional description if you wish.

Next, we need to supply the task parameters. These are pre-defined options we can set, some of which are required for the task to run. Select `fact` for our Facter task, then set the value of the task to `os`. Click **Add Parameter**.

We can choose the schedule this task for the future, but leave it set to **Now**.

Finally, we can select our nodes in the same manner as we did last lesson. Let's just use the **Node list** option, selecting both our non-master hosts. Select **Run job** when ready.

As before, we are taken to a page that outputs our results for our job. Notice how we receive the output we would normally get when querying facts on our command line on our console.

The Process

So what happens when we run a task? The process deviates slightly from performing simple on-demand Puppet runs:

1. The PE client sends the task command.
2. The orchestrator checks if the user is authenticated.
3. The orchestrator fetched the node target from PuppetDB and returns the nodes.
4. The orchestrator requests data from Puppet Server.
5. Puppet Server returns task metadata, file URIs, and file SHAs.
6. The orchestrator validates the task command and then sends the job ID to the client.
7. The orchestrator sends task parameters and file information to the PXP agent.
8. The PXP agent sends an initial response back to the orchestrator and checks the SHA against the local cache.
9. The PXP agent requests a task file from Puppet Server.
10. Puppet Server returns the task file to the PXP agent.
11. The task runs.
12. The PXP agent return the results to the orchestrator.
13. The client requests events from the orchestrator.
14. The orchestrator returns the results to the client.

The CLI

Additionally, we can run tasks from the CLI. For this, we don't use the `puppet-job` command but instead `puppet task`. The format of the command should resemble: `puppet task run TASK_NAME PARAMETER=VALUE --TARGET_TYPE TARGETS`.

Let's do ahead and try to restart `httpd` on our CentOS node:

```
$ puppet task run service action=restart name=httpd --nodes USERNAME#c.mylabserver.com
```

As before, the output on our CLI reflects that on our console, and we can view the job on our console if so desired.

Wrap Up

In this section, we learned how to run ad-hoc commands on our nodes via tasks, including:

- How to run tasks on the CLI
- How to tun tasks on the console
- The underlying process that happens when we run tasks

Writing Tasks

Currently, our list of tasks is limited. Though Puppet ships with some useful tasks, it is in no way an exhaustive list. So, how do we add new tasks? Well, tasks are added through modules, so when we add a module with task support, those tasks are automatically added for us to use. None of our currently-used Puppet Forge modules have tasks, and using those tasks would in no way differ from using the pre-installed tasks, so let's instead look into writing tasks ourselves.

We can write tasks in any programming language and should be stored in the `tasks` directory in our module path. Put how can we edit our module now that Code Manager has taken over our environment directory? Well, it's a good thing we've been using GitHub. Go ahead and pull down your `apache` module, either to your workstation or simply use the `cloud_user`'s home directory on your master if you wish.

```
$ git clone https://github.com/USERNAME/MODULE.git apache
$ cd apache
```

Let's add our `tasks` directory:

```
$ mkdir tasks
```

While we can use any language to create our tasks, the default is Bash, so if we use the PDK to create our task file, it outputs a `.sh` file:

```
$ pdk new task ctl
pdk (INFO): Creating '/home/cloud_user/apache/tasks/ctl.sh' from template.
pdk (INFO): Creating '/home/cloud_user/apache/tasks/ctl.json' from template.
```

We're going to be creating a simple enough script that will run the `apachectl` command with defined parameters. The goal here isn't to learn advanced script writing, but to understand the context within Puppet, so we're going to keep this simple with regards to the script itself. Notice too that when we created our task, a `.json` file was included along

with our script. This is where we're going to store the metaparameters that define how our script is used, as well as our parameters themselves.

When we name our tasks we should follow the usual Puppet naming structure: lowercase letters, digits, and underscores are allowed, and the name of the task will be referred to as `modulename::taskname` -- or `apache::ctl` in this case.

Now let's open up our script:

```
$ vim tasks/ctl.sh
```

Our hashbang has already been provided. This is mandatory for *all* tasks, regardless of language.

Now we can write our Bash script. But no matter how simple or advanced our scripts are, there are some additional standards we can reference when writing our tasks. The primary one is that when we define a variable, we want to use as a user-supplied parameter, we need to include the `PT_` prefix when providing the variable.

For example, our goal here is to let users use some `apachectl` commands. So our script is simply going to be:

```
apachectl $PT_command
```

Now let's save and exit and take a look at our JSON file. Most of the work that goes into getting our script to work with Puppet can be found in this file, not the script itself. Right now, our JSON file contains placeholder metadata we need to update:

```
$ vim tasks/ctl.json

{
  "puppet_task_version": 1,
  "supports_noop": false,
  "description": "A short description of this task",
  "parameters": {
  }
```

```
}

So let's go ahead and update this:

{
  "puppet_task_version": 1,
  "supports_noop": false,
  "description": "Allows us to run apachectl commands through Puppet",
  "parameters": {
  }
}
```

Now we want to supply our parameters. We just have one, `PT_command`, but when we reference it in our parameters, we need to leave out the `PT_` prefix:

```
{
  "puppet_task_version": 1,
  "supports_noop": false,
  "description": "Allows us to run apachectl commands through Puppet",
  "parameters": {
    "command": {
    }
  }
}
```

Now we want to provide both a description and a type, with the type working similar to how we assigned type in our `init.pp` files. That said, we need to be incredibly cautious about how we reference our type, lest we allow some dangerous security problems into our system. So this means instead of defining our type as a string, we want to use `Enum` and provide specific options:

```
{
  "puppet_task_version": 1,
```

```
  "supports_noop": false,
  "description": "Allows us to run apachectl commands through Puppet",
  "parameters": {
    "command": {
      "description": "apachectl command to run",
      "type": "Enum[start,stop,restart,status,graceful,graceful-stop]"
    }
  }
}
```

Save and exit, then push these changes to GitHub:

```
$ git add .
$ git commit -m "Added apachectl task"
$ get push origin master
```

As our final task, we want to apply this to our Puppet setup. However, *where* we add the module matters. Right now, we only have our `apache` module assigned to staging. But to add tasks, the parent module needs to be on the production environment.

So let's go ahead and move back to our `control-repo`:

```
$ cd ../control-repo/
```

And switch to the `prodction` branch:

```
$ git branch production
```

Add the module to the `Puppetfile`:

```
$ vim Puppetfile
```

```
mod 'apache',
  :git => 'https://github.com/USERNAME/MODULE'
```

Save and exit. We can now deploy our environment:

```
$ puppet-code deploy production --wait
```

Test the Task

To test our task, let's switch back to our PE console, then move to the **Tasks** page. `apache::ctl` is now available in the dropdown. Select it, then change the `command` parameter to `restart` and run it against both hosts.

Success! Our task is working and we now have a sense of how to create and use our own Puppet tasks.

Wrap Up

In this section we created a new Puppet task, learning:

- How to add custom tasks to our Puppet setup
- How to supply parameters to custom tasks
- How the JSON metadata file is used in conjunction with the task script itself

Bolt Overview

PuppetLab's Bolt is an open source orchestration tool that allows us to make changes on our nodes without needing to install any additional tools or services on the nodes we are managing. In this way, Bolt can be comparable to configuration management system Ansible, which allows users to make configuration changes over SSH.

We can install Bolt on Windows, Mac, or Linux, and if we wished we could add it directly to our workstation computer. I'll still be working through my Puppet master for this lesson, but you can choose to work through your usual workstation if desired.

Installing Bolt is simple enough. Since we already added the Puppet 5 repository when we installed the PDK, all we have to do is run:

```
$ sudo apt-get install puppet-bolt
```

The package name is the same on CentOS, as well. Working on Windows or Mac? **Just go to Bolt's homepage to download the package** and double click to install.

With Bolt installed, we can now run one-off commands and scripts, use Puppet tasks and plans, and even perform a `puppet apply` of any specified manifests. As you may imagine, this makes Bolt especially useful for things like bootstrapping your nodes to work under Puppet.

As previously mentioned, Bolt uses SSH to access our hosts. So we do need to perform a preliminary login to ensure our key exchange is working properly for passwordless SSH. For example, if we were to try to run a command on either of our agent nodes right now:

```
$ bolt command run 'hostname -f' --nodes USERNAME#c.mylabserver.com -p
```

We would receive an error:

```
Host key verification failed...
```

So let's log into our remote machine for a moment, then attempt to run the command again:

```
$ ssh cloud_user@USERNAME#c.mylabserver.com
$ exit
$ bolt command run 'hostname -f' --nodes USERNAME#c.mylabserver.com -p
```

This time Bolt returns the expected value.

Notice, too, the use of `--nodes`, which may look familiar to our orchestrator commands from the prior section. This is because it works the same, although we can also replace `--nodes` for `--targets`. Additionally, we have the `--query` or `-q` option to perform PuppetDB queries to define our targets.

The `--noop` and `--params` options also work that same as in our `puppet-job` or `puppet tasks` commands.

We can also select what user we wish to work as with `--user`, or we can escalate our privileges with `--run-as`.

We've also been using the `-p` flag so we can log in as our `cloud_user`, but we can simplify this process by using SSH keys instead:

```
$ ssh-keygen
$ ssh-copy-id cloud_user@USERNAME#c.mylabserver.com
```

We can now run passwordless commands:

```
$ bolt command run 'hostname -f' --nodes USERNAME#c.mylabserver.com
```

So now that we have Bolt up and running, let's get ready to look at how we can use Bolt to perform more open-ended changes -- especially compared to the default orchestrator.

Wrap Up

In this section, we covered the basics of Bolt, learning:

- That Bolt is Puppet's open source (and slightly more advanced) orchestration tool
- That Bolt works off SSH
- How to install Bolt on our system and ensure it can connect to our hosts

Basic Commands

You have already noticed a benefit to Bolt that our PE orchestrator lacks: We can run ad-hoc commands of any kind and are not limited to only using tasks to perform actions to our hosts. So while we should create tasks (and later, plans) for often-used actions, we also have the option to easily perform any command we need through Bolt without any additional prerequisites (past having Bolt installed). And since, unlike orchestrator, Bolt works over SSH and our targets do *not* need to have Puppet installed, it is also a popular tool to do things like bootstrap nodes into Puppet or perform any preliminary tasks.

So, let's start with the basics. We already saw that we can run things like:

```
$ bolt command run 'hostname -f' --nodes rabbitheart2c.mylabserver.com
```

However, we should note that when using any sort of redirection in our Bolt commands, we need to pass in the shell prompt before writing the command. So if we wanted to pass some information into a file, we would run:

```
$ bolt command run "bash -c 'echo hello > /tmp/hello'" --nodes rabbitheart2c.mylabserver.com
```

We also have the option to run scripts on nodes. This is done the same way as above, but instead of supplying a command we would supply the location of a script:

```
$ bolt command run ../script.sh --nodes rabbitheart2c.mylabserver.com
```

Note that the script must contain the appropriate interpreter declaration at the top of the file.

Finally, we can also use Bolt to manage files. Let's go ahead and pass in our `.vimrc` config:

```
$ bolt file upload .vimrc ~/.vimrc --nodes rabbitheart2c.mylabserver.com
```

With the basics out of the way, we can now look at working with more robust Puppet tasks.

Wrap Up

In this section, we covered basic Bolt commands, such as:

- Running ad-hoc commands of any kind
- Using a script to make changes on our targets
- Copying a file to our targets

Running Tasks

Tasks in Bolt are no different from tasks using orchestrator, and the tasks we're running are the, same. This means everything we learned in our previous section on tasks is the same for Bolt, except we'll be using the `bolt` command instead of `puppet tasks`.

To run a task in Bolt, we use the `bolt tasks run` command, and we can pass in parameters with the `parameter=value` format. Again, this is similar to using orchestrator on its own. That said, if we try to run our `apache::ctl` module it will fail:

```
$ bolt task run apache::ctl command=restart --run-as root --sudo-password --nodes
rabbitheart2c.mylabserver.com
Could not find a task named "apache::ctl". For a list of available tasks, run "bolt task show"
```

This is due to an issue with Bolt's module path:

```
$ bolt task show
...
MODULEPATH:
/home/cloud_user/.puppetlabs/bolt/modules:/home/cloud_user/.puppetlabs/bolt/site-modules:/home/
cloud_user/.puppetlabs/bolt/site
```

To solve this, we can update the modulepath, but first let's create an overall `modules` directory in our home directory:

```
$ mkdir modules
$ mv apache modules/
```

Bolt's configuration can be altered by creating a `bolt.yaml` file in `.puppetlabs/bolt/`:

```
$ vim .puppetlabs/bolt/bolt.yaml
modulepath: "/home/cloud_user/.puppetlabs/bolt/modules:/home/cloud_user/.puppetlabs/bolt/site-
modules:/home/cloud_user/.puppetlabs/bolt/site:/home/cloud_user/modules"
```

Let's go ahead and run our `apache::ctl` task now:

```
$ bolt task run apache::ctl command=restart --run-as root --sudo-password --nodes
rabbitheart2c.mylabserver.com
```

Notice, too, how we need to ensure we're working as `root` for this task, supplying our escalation password with `--sudo-password`. By leaving it blank, we allow a prompt to appear when running the command, letting us type it in.

This time, success! We can now add, and run, tasks in Bolt.

Wrap Up

In this short section, we:

- Ran a task in Bolt
- Added new tasks in Bolt
- Updated our `bolt.yaml` configuration file

Using Plans

While tasks are wonderful for common tasks that are simple to run, there are times when we want to be able to write something more robust than just a simple script. Plans allow us to combine tasks with other logic so that we can complete more complex operations through Bolt. Right now, we have a few prepackaged plans at our disposal:

```
$ bolt plan show
aggregate::count
aggregate::nodes
canary
facts
facts::info
puppetdb_fact
reboot
```

And to run a plan, we would just use the `bolt plan run` command, feeding in parameter when appropriate:

```
$ bolt plan run facts --nodes rabbitheart2c.mylabserver.com
```

We can also, of course, create our own plans. To do this, we need to add a `plans` directory to our module

```
$ cd modules/apache/
$ mkdir plans
```

We're going to create a plan to *remove* Apache from our hosts, using a combination of commands, tasks, and parameters. Plans follow the same naming structures of classes and tasks, although there is currently no PDK command to generate them. We'll have to create our plan the old-fashioned way (from scratch), in our text editor.

```
$ vim plans/remove.yaml
```

Plans are comprised of three structures: `steps`, `parameters`, and `returns`, with `returns` being optional.

`steps` is comprised of an array of actions to complete, in order, known as "step objects." We can use five different kind of step objects: `command`, `task`, `script`, `file` (which uses `source` and `destination` keys), and `plan`. Each configures a "step" involving an action for which the step is named -- i.e., the `command` step object runs a command, `task` runs a task, `script` a script, etc.

Let's start by creating a `task` step object that will cause `apachectl` to perform a graceful shutdown of our Apache services:

```
steps:
  - task: apache::ctl
    target: rabbitheart2c.mylabserver.com
    description: "Shuts down Apache service"
    parameters:
      command: graceful-stop
```

We're now going to use `command` to create a copy of our log files before we remove our package:

```
  - command: "cp /etc/httpd/logs/access_log /tmp/apache_access"
    target: rabbitheart2c.mylabserver.com
    description: "Create snapshop of apache log files"
```

Finally, remove the package:

```
  - task: package
    target: rabbitheart2c.mylabserver.com
    description: "Uninstall Apache"
    parameters:
      action: uninstall
      name: httpd
```

Let's save and exit.

Now we can test this with:

```
$ bolt plan run apache::remove --run-as root --sudo-password
```

Note the use of escalated privileges.

We can also write plans using the usual Puppet DSL, although you'll find that while code written for plans has a lot of the usual trappings of Puppet code. It's not quite the same as what we're used to. We can see this in practice by converting our current script:

```
$ bolt plan convert plans/remove.yaml
# WARNING: This is an autogenerated plan. It may not behave as expected.
plan apache::remove() {
  run_task('apache::ctl', 'rabbitheart2c.mylabserver.com', "Shuts down Apache service", {'command'
=> 'graceful-stop'})
  run_command("cp /etc/httpd/logs/access_log /tmp/apache_access", 'rabbitheart2c.mylabserver.com',
"Create snapshop of apache log files")
  run_task('package', 'rabbitheart2c.mylabserver.com', "Uninstall Apache", {'action' => 'uninstall',
'name' => 'httpd'})
}
```

Notice how to run various tasks, commands, etc. we use the `run_` prefix, then the name of our step object. We then feed our attributes into the parentheses and provide our parameters similar to how we define parameters in resources and classes. They're encased in curly brackets and mapped via hash rockets.

There are some benefits to using the Puppet language over YAML, however. Namely, we can include logic, such as `if` or `case` statement, just as we can in any other piece of Puppet code. If you're looking for a way to expand on this lesson, a good way would be to add logic to our above plan so we can remove Apache from Debian-based hosts as well as our CentOS host.

Wrap Up

We worked with plans in this section, and covered how to:

- Write plans in YAML and the Puppet DSL
- Convert plans
- Run plans on our hosts

PuppetDB Overview

PuppetDB is PostgreSQL-backed database that collects data concerning our nodes. Included with Enterprise and available for Open Source Puppet, PuppetDB also enables some advanced Puppet features, such as the ability to use exported resources.

We already have PuppetDB installed. But if we wanted to add it manually, we could do so from the Puppet 5 repository we'd already enabled, through the `puppetlabs-puppetdb` module or from source. We also have the option to split our installation so PuppetDB works on a separate server.

Configuration

Post-installation, PuppetDB's configuration files can be found in the `/etc/puppetlabs/puppetdb/conf.d` directory. The files inside are stored as `.ini` files and portioned in different `[sections]`, similar to many of the other Puppet configuration files we're worked on in this course.

Note, however, that if you installed PuppetDB through the Puppet Forge module, these changes should be made in Puppet itself, not directly to the configuration file. This is also true for us using Puppet Enterprise. If we wanted to make changes to our PuppetDB, we would do so though the PE console, under the `PE PuppetDB` group.

Should we navigate to this group now, we can expand the `Parameter name` menu to view the primary configuration options we should familiarize ourselves with. For the most part, these are all self-explanatory. We can provide specific database information, change our RBAC setup for PuppetDB, define the certname, change our port setup, and we can even update our `java_args` expression for startup.

CLI

If we were working in an Open Source Puppet setup, our options for working with PuppetDB would be somewhat limited. While we would be able to use exported resources (which we'll discuss in a bit), when it comes to the CLI we have a single option: We can check our node status:

```
$ sudo puppet node status USERNAME#c.mylabserver.com

Currently active
Last catalog: 2019-06-25T16:21:37.061Z
Last facts: 2019-06-25T16:21:36.539Z
```

That said, since we're using Puppet Enterprise we also have access to the `puppet query` and `puppet db` commands.

`puppet query`

The `puppet query` command allows us to run Puppet queries on the CLI. We'll be using this extensively in a bit, but for now let's just get a sense of how it works by running a familiar query (note that you may need to renew your access token):

```
puppet query "nodes[certname] { catalog_environment = 'staging' }"
```

`puppet db`

`puppet db` provides us with two abilities: We can export our current database with `puppet db export FILENAME.tgz`, or import an existing database with `puppet db import FILENAME.tgz`.

Additionally, we can use one of three anonymization profiles for creating exports:

- `full`: This will anonymize all data including node names, resource types, resource titles, parameter names, values, any log messages, file names, and file lines. The structure of the data will remain unaffected.
- `moderate`: The recommended profile, it anonymizes node names, resource titles, parameter values, log messages, file names, report logs messages, and `time`-based metric names.
- `low`: This anonymizes node names, parameter values, and log messages.

Metrics

Finally, we can view the metrics for our PuppetDB server through an SSH tunnel. Since the performance of our database is integral to its function, we should know how to access its metrics page.

To do this, we need to set up an SSH tunnel on our *workstation*:

```
$ ssh -L 8080:localhost:8080 cloud_user@PUPPETDBHOST
```

Note that for us, our `PUPPETDBHOST` is also our master.

Wrap Up

In this lesson, we learned the basics of PuppetDB, including:

- How to install PuppetDB
- PuppetDB configuration options
- The basics of the PuppetDB CLI
- How to view our PuppetDB metrics

Exported Resources

PuppetDB doesn't just allow for the querying of data however. It also provides us with the ability to store resource information that we can then use in other modules, manifests, and the like.

The best way to get a sense of how this works is to write it ourselves, so let's create a `host` record we can reference that lets us build an `/etc/hosts` file based on our nodes' IP address and FQDN for any web servers.

Move into our faux-workstation, where we stored our Apache module:

```
$ cd modules/apache
```

We're going to update our `config` class to include an exported `host` definition. Note that the `host` resource type is depreciated in Puppet 6.0 and above, but is available for use in our version of Puppet. However, it does provide the clearest example of how exported resources work. `host` creates an entry in `/etc/hosts` using the provided parameters.

Open up the `config.pp` manifests:

```
$ vim manifests.config.pp
```

Let's add the `host` resource type as though we weren't exporting it first:

```
  host { "${hostname}":
    host_aliases => "$fqdn",
    ip           => "$ipaddress",
  }
```

Then, to signify we want this information stored in PuppetDB, we prepend the resource type with `@@`:

```
  @@host { "${hostname}":
    host_aliases => "$fqdn",
    ip           => "$ipaddress",
  }
```

Save and exit, then commit the changes to git:

```
$ git add .
$ git commit -m "Added host exported resource"
$ git push origin master
```

We then want to force a Puppet run on our two non-master hosts. We can do this through the command line via orchestrator if we wish:

```
$ puppet job run --nodes NODELIST
```

Finally to confirm our resource has been exported, run:

```
$  puppet query "resources[certname, type, title, parameters] { exported = true }"
```

We now want to create a `base` class that will create host records based on the information pulled from this exported resource. Let's drop into our `modules` directory: `$ cd ..`

Then we'll create a `base` module, filling out the PDK form as normal:

```
$ pdk new module base
$ cd base
```

Let's just use the `init.pp` to import our hostname information:

```
$ pdk new class base
$ vim manifests/init.pp
```

We can now reference our exported resource with `Host <<||>>`:

```
class base {
  Host <<||>>
}
```

We'll want to add this GitHub, as well. Create a new GitHub repo using the instructions from the beginning of the "Module Authoring" section, then add your `base` module to Git:

```
$ git init
$ git add .
$ git commit -m "Base host module"
$ git remote add origin https://github.com/USERNAME/REPO.git
$ git push origin master
```

Now add this to our `staging` environment:

```
$ cd
$ cd control-repo
```

Make sure you are on the `staging` branch:

```
$ git branch
```

Add the new module to your `Puppetfile`:

```
$ vim Puppetfile

mod 'base',
  :git => 'https://github.com/USERNAME/REPO.git'
```

Save and exit, then commit the change:

```
$ git add .
$ git commit -m "Added base module to Puppetfile"
$ git push origin staging
```

Deploy the changes:

```
$ puppet code deploy staging --wait
```

We'll want to access our **Classification** page on the PE Console now and select our **dev-staging** group. Add the **base** class to the configuration, commit the changes, and perform a Puppet run on each host.

Wrap Up

In this section, we learned about exported resources, including:

- How to export fact data in a resource
- Where that data is stored and how to view it
- How to import the exported resource

PQL Basics

While we can connect to our PuppetDB database and run SQL-based queries on it as we would any database, PuppetDB also has its own query language that we should be aware of. Any query written in the Puppet Query Language follows the same basic structure:

```
<entity> [<projection>] { <filter> <modifiers> }
```

`entity`

Of these, only the `entity` aspect is required for the query to run. An entity is the context the query executes on. This defines the results you will receive once the query is run. An entity essentials defines *what* you are querying for, such as node information, environments, facts, or resources. **A full list can be found here**.

`projection`

The `projection` allows us to define a subset of our overall query that we wish to return. For example, we might want to parse our `node` entity using our facts or environment. We also saw this when we searched for our exported resource:

```
resources[certname, type, title, parameters] { exported = true }
```

Instead to retrieving all our `resources` entity information, we filtered by the name of the node, the resource type and title, and included any set parameters. Had we left our projections out, we would have instead received a full list of information, including the environment, any tags, and more:

```
$ puppet query "resources { exported = true }"
```

Projections can also use functions instead of queries. Functions include `count()`, `avg(<field>)`, `sum(<field>)`, `min(<field>)`, and `max(<field>)`. Some of these require an additional field value to work.

`filter` and `modifier`

Next, we have the ability to further `filter` our results. This portion of the query is stored in curly brackets, and can reference entity fields, facts, and environment information. Filters are often further limited by the provided modifier. So if we wanted to filter our nodes the start with a particular certname (say, your Linux Academy username), the filter would be `certname` and the modifier would be your username. Here's what mine looks like:

```
$ puppet query 'nodes[certname] { certname ~ "^ellejaclyn" }'
```

Notice how we use a conditional operator to map our modifier to the filter. Also notice the use of regex.

Additional Sorting

Finally, we can further manipulate our Puppet query results by either grouping or paging or results. When we group our results, we condense all rows with the defined value into a single row. For example, if we wanted to see a simple list of fact names, we can use `group by`:

```
$ puppet query "facts[name] { group by name }"
```

We can also use SQL-like paging clauses: `limit`, `offset`, and `order by`. So if we wanted to, so, receive a list of reports in the order they were received, we would run:

```
$ puppet query "reports {certname = MASTER order by receive_time }"
```

Wrap Up

In this section, we discovered the components of a Puppet query, including the:

- Entity
- Projection
- Filter
- Modifier

Building Puppet Queries

For the most part, we've kept our queries simple for the sake of explaining its various components. Yet we can build out our queries to work in the most specific of circumstances. Let's first begin with a few basics, however.

Say, for example, we just wanted to filter our PuppetDB data via a fact:

```
$ puppet query "inventory { facts.os.name = 'Ubuntu' }"
```

Note that `inventory` is alternative endpoint for the `facts` entity that lets us better filter our facts.

But this isn't necessarily enough to filter our results down to what we want. What if we wanted to only query for facts on Ubuntu hosts with the `apache2` service? Well, we can use `and` to further built our query:

```
$ puppet query "inventory { facts.os.name = 'Ubuntu' and resouces { type = 'Service' and title =
'apache2' } }"
```

We can use `and` as much as we need, as well, chaining together all sorts of queries.

Of course, we can also use `or` in place of `and` to manipulate our query. Let's look for Ubuntu *or* Red Hat-family hosts:

```
$ puppet query "inventory { facts.os.name = 'Ubuntu' or facts.s.family = 'RedHat' }"
```

Finally, let's close out the section by looking for Ubuntu or CentOS hosts on the `staging` environment containing either the `httpd` or `apache2` service:

```
$ puppet query "inventory { facts.os.name = 'Ubuntu' or facts.s.name = 'CentOS' and resources { type
= 'Service' and title = 'apache2' or title = 'httpd' } and environment = 'staging' }"
```

Wrap Up

This section was all about using the components of the previous lesson to build practical Puppet queries!

The Node Graph

Although it may seem like you've spent countless hours in Puppet by the time you've reached this section, if you've ever seen a full configuration management setup you'll know that the amount of modules we've touched in this course is comparatively small. As our infrastructures grow, we'll want to be be able to easily see how our modules are affecting change.

With Puppet Enterprise, this is where our **node graph** comes in. A node graph is just that: a graphical representation of the changes Puppet made on our node, based on its most recent catalog compilation.

To look at a node graph, we simply have to log in to the Puppet Enterprise console, select a node from the **Nodes** page, and click on the **Node graph** link below the certname of the host.

We're presented with a flow graph of our overall setup for that single node. We can see each individual module class that was applied, with the color of the dot indicating whether it was unchanged, changed, or in a failed state. Underneath each module's overall class, we see the individual classes or resource types, breaking each configuration section down to its smallest part. Should we click on any of these items, we can also further see a list of tags and dependencies for that individual item.

We can also perform some simple filter tasks on our node graph itself. It allows us to search by resource, stage, class, tag, and status and centers, and highlights the desired area of the map as the filter is applied.

That said, the node graph is not the only option we have for exploring our nodes and node changes. For a more detailed look, we'll want to consider our reports.

The Reports Table

Each time a Puppet run is performed on our hosts, a report is taken of the run (unless it's otherwise disabled in the configuration). To view these reports, we need our Puppet Enterprise console. Specifically, we need to move to the **Reports** page.

Here we have a chronological list of all logs and events for our system. We can filter these by run status or simply view them on their own, but each overall report informs us of the node the report is from, and how many failed, changed, unchanged, skipped, and corrective changes were made.

By selecting one of these reports (simply on the report time), we can see a list of resources events that occurred on the host. Here we can break our Puppet runs down to individual resources, allowing us to track where our modules are failing (or otherwise).

For more specific details, especially during changes or failures, we can select the **Log** tab. This provides us with a formatted log of the Puppet run. Warnings are color coded in yellow, and failures highlighted in red. If you recall the feedback we got when we performed `puppet agent -t` runs on our hosts earlier in this course, it's the same thing but presented on our PE console.

Finally, we can also view the **Metrics** of the run. This informs us of basic facts about the run, such as environment, time taken, when the run began, amount of resources managed, and more.

Overall, you'll likely find yourself most often turning to the **Logs** page when there's any troubleshooting to be done. While the **Events** and **Metrics** pages provide important context and data, for common troubleshooting we'll discover the most helpful information in our logs.

Filtering Reports

But let's return to the overall **Reports** page itself. By default, it displays *all* reports in chronological order, or we can filter via the run status. However, by using the **Filter by fact value** option, we can further refine our reports. This can be especially useful when tracking trends in run status.

Still working from the **Reports** page, select **Filter by fact value**. We have two options for our defined rules here: We can match *all* rules or *any* rules Essentially, when we match *all* rules, we can place an `and` between our rules:

```
osfamily = Debian  AND
pe_server_version ~ 2018.1
```

The above express would match only our master.

Alternatively, if we switch to the *any* node filter, we would read this same set of rules as:

```
osfamily = Debian OR
pe_server_version ~ 2018.1
```

Which would provide us with results for both our Ubuntu node and Ubuntu master.

Exporting Data

Should we track down an interesting trend, or simply need a record of information, we can also download the .CSVs of our reports. Underneath the filter area, we have the option to **Export data**. This will download a local copy of the provided report table, with all the filters we set.

Installation

We're going to finish out this guide by spending some time on troubleshooting strategies for situations we may come across while using Puppet. And since it makes sense to start the beginning, we'll initially take a look at troubleshooting a Puppet install.

Most of the time, Puppet install issues fall into one of two camps: communication issues and infrastructure support issues. How we begin to troubleshoot will primarily depend on the way the issue is presenting itself.

The PE Installer

If using the Puppet Enterprise installer (any version), you'll most often notice a problem when it attempts to connect to its localhost using the FQDN we provided to Puppet. Most often this can be fixed by either updating the `/etc/hosts` file to ensure the FQDN is appropriately mapped or by checking for any firewall interference. If there *is* firewall interference, you may want to disable the firewall entirely and implement a firewall module via Puppet, or you can just simply open the correct port numbers.

Open Source Installation

If you're having trouble connecting to an Open Source installation, you probably won't notice the issue until you attempt to start the Puppet Server service itself. In this case, you'll still want to check the firewall and `/etc/hosts` file, but you'll also want to take a look at the Puppet configuration itself: `/etc/puppetlabs/puppet/puppet.conf`

Here, you may find you need to manually add the `certname` parameter and set it to your FQDN. This should can be set in both the `[main]` and `[master]` sections of the file.

Time Out Issues

If communication between the host and itself is not the issue, it might be the size of your host. If you're working outside of the recommended settings for the Puppet master, you'll need to update the `/etc/default/puppetserver` file and change the memory settings to support a smaller setting:

```
JAVA_ARGS="-Xms2g -Xmx2g -Djruby.logger.class=com.puppetlabs.jruby_utils.jruby.Slf4jLogger"
```

You can set the `2g` portions of the above line to be as small as `512m` if needed.

A Final Note

Should your installation go wrong, you can rerun the Puppet installer an unlimited amount of times with no negative repercussions. If you have the time and you've yet to pull in any nodes, it might be worth it to simply rerun the install and watch the logs if you're having trouble pinpointing the problem.

Wrap Up

In this section, we learned how to troubleshoot a failed install by:

- Checking our DNS information
- Reviewing our firewall settings
- Changing the memory expectations upon startup

Communication

While we discussed the effect DNS issues could have on the installation process, there are a number of other places were communication between components can cause issue in a Puppet setup. Specifically, this is the communication between the Puppet agent on our hosts and our master(s). For this, there are some basic things we want to confirm to ensure our components can talk to one another.

DNS and Firewall Issues

As with the Puppet master itself in the previous section, the *first* thing we want to check is that there are no issues with DNS or a firewall between our Puppet master and any agents. If we're uncertain that the connection is working, run:

```
$ telnet MASTER_FQDN 8140
```

If this fails, perform the same checks as the previous lesson on both master and agented node. You will also want to make sure the Puppet Server service has been started on the master (`pe-puppetserver` or `puppetserver`, depending on Puppet version).

Finally, another quick check we can make is to confirm that the time is synced between both hosts. We can check this manually or, for an easier fix, we might want to simply add NTP to all our hosts.

Certification Settings

If your DNS and firewall settings look okay and the issue still persists, there are a few other things we want to consider, such as our certificate settings for our agent nodes.

The easiest check we can make is to ensure the node's certification is accepted on the appropriate master. We can confirm this by running `puppetserver ca list --all` on the master and checking that our cert is signed for the problem node.

If the problem still persists, the issue might lie in the certificate itself. Either the certification has invalid dates, or our node shares a name with a previous node that has since been removed from the infrastructure overall, but not from Puppet. In both of these instances, we'll want to reissue our node's certificate.

On the master:

```
$ puppet cert clean NODE_NAME
```

On the problem node:

```
$ rm -r $(puppet agent --configprint ssldir)
$ puppet agent -t
```

On the master:

```
$ puppetserver ca sign NODE NAME
```

Wrap Up

In this section, we reviewed common causes for issues between our nodes and master, including:

- DNS and firewall issues
- Time sync problems
- Certificate troubleshooting

Code Manager

Code Manager is one of the more complicated components in our Puppet setup, and when it comes to troubleshooting any issues, we'll have to take a less straight-forward approach to troubleshooting.

Logs, Statuses, and Communication

Should Code Manager fail, the first and easiest task we can perform is to run the `puppet code status` command. This provides us with an overview of our Code Manager deployment, and indicates the overall status of Code Manager and the file service it uses. A working Code Manager deployment will display the state as `running`, but a failure will generally look like the command not working at all. For example, a response of `Connection timed out (os error 110)` indicates that Code Manager is struggling to connect to our Puppet Server. Here, we would want to perform our usual DNS checks, as well as ensure the Puppet Server service has been started.

The `puppet code status` command may also return information about an invalid configuration. In this instance, we either want to check our config at `/etc/puppetlabs/puppetserver/conf.d/code-manager.conf` (for manual setups), or view our class settings in the PE Enterprise Console for Code Manager.

If we're still unsure of there the issue lies, however, we may want to look into our Code Manager logs. These logs are found at `/var/log/puppetlabs/puppetserver/puppetserver.log`, and contain not just Code Manager logs but information for our whole Puppet Server setup. We'll want to look for logs for two specific endpoints: `/v1/deploys` or, if we're using a webhook along with Code Manager, `/v1/webhook`.

Finally, we can test the communication between our Code Manager and the control repo through the `puppet code deploy --dry-run` command. If there are issues with our Git repo, we should receive an `Unable to determine current branches for Git source` warning.

Puppetfile Troubleshooting

Our issue may not be with Code Manager itself, however. If our issue is that Code Manager appears to be working, but our end results aren't what we expect, we'll want to follow a different troubleshooting tactic.

First, we'll want to check our `Puppetfile`. We can check our syntax via the `r10k puppetfile check` command, but beyond that we'll also want to check our Puppetfile's sources. Look for version changes for Forge modules, and ensure that any referenced Git repos still exist. We can further verify this by copying our control repo to a temporary directory and running:

```
$ sudo -H -u pe-puppet bash -c \
  '/opt/puppetlabs/puppet/bin/r10k puppetfile install'
```

This will install all modules in the Puppetfile and note any errors that occur during the test deploy.

Wrap Up

In this section, we looked at both communication and Puppetfile-related issues we can encounter when running Code Manager, including:

- How to test the connection between Code Manager and the Puppet Server
- Where to find our Code Manager logs
- How to check our Puppetfile and control repo for issues

PostgreSQL

Puppet uses PostgreSQL as its database backend, and if we're experiencing issues with the database it can often be traced to the database using too much space or too much memory. Both of these can be solved through either an easy configuration change, or by implementing a split configuration in which the database is housed on a separate host from the Puppet Server.

If we find we're running out of space on our host due to our database, we can enable the `autovacuum` setting, which performs routine maintenance on our database, recovering disk space and updating any related statistics.

If we're running out of RAM, we first want to confirm it by reviewing the `/var/log/pe-postgresql/pgstartup.log` for the following error:

```
FATAL: could not create shared memory segment: No space left on device
DETAIL: Failed system call was shmget(key=5432001, size=34427584512,03600).
```

Once confirmed, we would want to update our kernel settings to use about 50% of our current total RAM for our `shmmax` setting. Our `shmall` setting should be `shmmax` divided by the page size, which can be found by running `getconf PAGE_SIZE`.

```
$ sysctl -w kernel.shmmax=RAM
$ sysctl -w kernel.shmall=RAM
```

Additionally, we may find we're experiencing issues due to conflicting port usage. In this instance, we would want to change our database port via the `puppet_enterprise::puppetdb_port` setting.

Wrap Up

In this section, we reviewed some common database issues, including how to:

- Fix space issues
- Update our RAM settings

High Availability

Although our demo setup includes only a single master, we also want to take the time to understand how to troubleshoot a multi-master environment. There are a few things we can check, should a master fail to communicate:

Latency

If your masters communicate over a high latency connection, you may simply need to run the command a second time.

Redundant Settings

When using multi-master you can set either the `server` for a split environment, or the `server_list` for a round robin setup. If both are set the setup will fail.

Empty Groups

With a multi-master setup, each master needs to have at least one node assigned to it. If there is an empty node group, any jobs for that group will fail.

Wrap Up

In this section, we covered the basics of high availability troubleshooting, including:

- How to handle latency
- How to look for duplicate settings
- How to ensure there are no empty node groups