Chapter 4. Understanding Puppet resources

We've already met three important types of Puppet resources: `package`, `file`, and `service`. In this lab, we'll learn more about these, plus other important resource types for managing users, groups, SSH keys, cron jobs, and arbitrary commands.

Files

We saw in [Lab 2], [*Creating your first manifests*] that Puppet can manage files on a node using the `file` resource, and we looked at an example which sets the contents of a file to a particular string using the `content` attribute. Here it is again (`file_hello.pp`):

```
file { '/tmp/hello.txt':
  content => "hello, world\n",
}
```

The path attribute

We've seen that every Puppet resource has a title (a quoted string followed by a colon). In the `file_hello` example, the title of the `file` resource is `'/tmp/hello.txt'`. It's easy to guess that Puppet is going to use this value as the path of the created file. In fact, `path` is one of the attributes you can specify for a `file`, but if you don't specify it, Puppet will use the title of the resource as the value of `path`.

Managing whole files

While it's useful to be able to set the contents of a file to a short text string, most files we're likely to want to manage will be too large to include directly in our Puppet manifests. Ideally, we would put a copy of the file in the Puppet

repo, and have Puppet simply copy it to the desired place in the filesystem. The `source` attribute does exactly that (`file_source.pp`):

```
file { '/etc/motd':
  source => '/examples/files/motd.txt',
}
```

To try this example with your Vagrant box, run the following commands:

```
sudo puppet apply /examples/file_source.pp
cat /etc/motd
The best software in the world only sucks. The worst software is significantly worse
than that.
-Luke Kanies
```

(From now on, I won't give you explicit instructions on how to run the examples; just apply them in the same way using `sudo puppet apply` as shown here. All the examples in this course are in the `examples/` directory of the GitHub repo, and I'll give you the name of the appropriate file for each example, such as `file_source.pp` .)

Note

Why do we have to run `sudo puppet apply` instead of just `puppet apply` ? Puppet has the permissions of the user who runs it, so if Puppet needs to modify a file owned by `root` , it must be run with `root` 's permissions (which is what `sudo` does). You will usually run Puppet as `root` because it needs those permissions to do things like installing packages, modifying config files owned by `root` , and so on.

The value of the `source` attribute can be a path to a file on the node, as here, or an HTTP URL, as in the following example (`file_http.pp`):

```
file { '/tmp/README.md':
  source => 'https://raw.githubusercontent.com/puppetlabs/puppet/master/README.md',
}
```

Although this is a handy feature, bear in mind that every time you add an external dependency like this to your Puppet manifest, you're adding a potential point of failure.

Note

Wherever you can, use a local copy of a file instead of having Puppet fetch it remotely every time. This particularly applies to software which needs to be built from a tarball downloaded from a website. If possible, download the tarball and serve it from a local webserver or file server. If this isn't practical, using a caching proxy server can help save time and bandwidth when you're building a large number of nodes.

Ownership

On Unix-like systems, files are associated with an **owner**, a **group**, and a set of **permissions** to read, write, or execute the file. Since we normally run Puppet with the permissions of the `root` user (via `sudo`), the files Puppet manages will be owned by that user:

```
ls -l /etc/motd
-rw-r--r-- 1 root root 109 Aug 31 04:03 /etc/motd
```

Often, this is just fine, but if we need the file to belong to another user (for example, if that user needs to be able to write to the file), we can express this by setting the `owner` attribute (`file_owner.pp`):

```
file { '/etc/owned_by_ubuntu':
  ensure => present,
  owner  => 'ubuntu',
}
ls -l /etc/owned_by_ubuntu
-rw-r--r-- 1 ubuntu root 0 Aug 31 04:48 /etc/owned_by_ubuntu
```

You can see that Puppet has created the file and its owner has been set to `ubuntu`. You can also set the group ownership of the file using the `group` attribute (`file_group.pp`):

```
file { '/etc/owned_by_ubuntu':
  ensure => present,
  owner  => 'ubuntu',
  group  => 'ubuntu',
}
ls -l /etc/owned_by_ubuntu
-rw-r--r-- 1 ubuntu ubuntu 0 Aug 31 04:48 /etc/owned_by_ubuntu
```

Note that this time we didn't specify either a `content` or `source` attribute for the file, but simply `ensure => present`. In this case, Puppet will create a file of zero size.

Permissions

Files on Unix-like systems have an associated **mode** which determines access permissions for the file. It governs read, write, and execute permissions for the file's owner, any user in the file's group, and other users. Puppet supports setting permissions on files using the `mode` attribute. This takes an octal value (base 8, indicated by a leading 0 digit), with each digit representing a field of 3 binary bits: the permissions for owner, group, and other, respectively. In the following example, we use the `mode` attribute to set a mode of `0644` ("read and write for the owner, read-only for the group, and read-only for other users") on a file (`file_mode.pp`):

```
file { '/etc/owned_by_ubuntu':
  ensure => present,
  owner  => 'ubuntu',
  mode   => '0644',
}
```

This will be quite familiar to experienced system administrators, as the octal values for file permissions are exactly the same as those understood by the Unix `chmod` command. For more information, run the command `man chmod`.

Directories

Creating or managing permissions on a **directory** is a common task, and Puppet uses the `file` resource to do this too. If the value of the `ensure` attribute is `directory`, the file will be a directory (`file_directory.pp`):

```
file { '/etc/config_dir':
  ensure => directory,
}
```

As with regular files, you can use the `owner`, `group`, and `mode` attributes to control access to directories.

Trees of files

We've already seen that Puppet can copy a single file to the node, but what about a whole directory of files, possibly including subdirectories (known as a **file tree**)? The `recurse` attribute will take care of this (`file_tree.pp`):

```
file { '/etc/config_dir':
  source  => '/examples/files/config_dir',
  recurse => true,
}
ls /etc/config_dir/
1  2  3
```

When `recurse` is `true` , Puppet will copy all the files and directories (and their subdirectories) in the source directory (`/examples/files/config_dir/` in this example) to the target directory (`/etc/config_dir/`).

Note

If the target directory already exists and has files in it, Puppet will not interfere with them, but you can change this behavior using the `purge` attribute. If this is `true` , Puppet will delete any files and directories in the target directory which are not present in the source directory. Use this attribute with care.

Symbolic links

Another common requirement for managing files is to create or modify a **symbolic link** (known as a **symlink**, for short). You can have Puppet do this by setting `ensure => link` on the `file` resource and specifying the `target` attribute (`file_symlink.pp`):

```
file { '/etc/this_is_a_link':
  ensure => link,
  target => '/etc/motd',
}
ls -l /etc/this_is_a_link
lrwxrwxrwx 1 root root 9 Aug 31 05:05 /etc/this_is_a_link -> /etc/motd
```

Packages

We've already seen how to install a package using the `package` resource, and this is all you need to do with most packages. However, the `package` resource has a few extra features which may be useful.

Uninstalling packages

The `ensure` attribute normally takes the value `installed` in order to install a package, but if you specify `absent` instead, Puppet will **remove** the package if it happens to be installed. Otherwise, it will take no action. The following example will remove the `apparmor` package if it's installed (`package_remove.pp`):

```
package { 'apparmor':
  ensure => absent,
}
```

By default, when Puppet removes packages, it leaves in place any files managed by the package. To purge all the files associated with the package, use `purged` instead of `absent` .

Installing specific versions

If there are multiple versions of a package available to the system's package manager, specifying `ensure => installed` will cause Puppet to install the default version (usually the latest). But, if you need a specific version, you can specify that version string as the value of `ensure`, and Puppet will install that version (`package_version.pp`):

```
package { 'openssl':
  ensure => '1.0.2g-1ubuntu4.8',
}
```

Note

It's a good idea to specify an exact version whenever you manage packages with Puppet, so that all the nodes will get the same version of a given package. Otherwise, if you use `ensure => installed`, they will just get whatever version was current at the time they were built, leading to a situation where different nodes have different package versions.

When a newer version of the package is released, and you decide it's time to upgrade to it, you can update the version string specified in the Puppet manifest and Puppet will upgrade the package everywhere.

Installing the latest version

On the other hand, if you specify `ensure => latest` for a package, Puppet will make sure that the latest available version is installed [*every time the manifest is applied*]. When a new version of the package becomes available, it will be installed automatically on the next Puppet run.

Note

This is not generally what you want when using a package repository that's not under your control (for example, the main Ubuntu repository). It means that packages will be upgraded at unexpected times, which may break your application (or at least result in unplanned downtime). A better strategy is to tell Puppet to install a specific version which you know works, and test upgrades in a controlled environment before rolling them out to production.

If you maintain your own package repository and control the release of new packages to it, `ensure => latest` can be a useful feature: Puppet will update a package as soon as you push a new version to the repo. If you are relying on upstream repositories, such as the Ubuntu repositories, it's better to manage the version number directly by specifying an explicit version as the value of `ensure`.

Installing Ruby gems

Although the `package` resource is most often used to install packages using the normal system package manager (in the case of Ubuntu, that's APT), it can install other kinds of packages as well. Library packages for the Ruby programming language are known as **gems**. Puppet can install Ruby gems for you using the `provider => gem` attribute (`package_gem.pp`):

```
package { 'ruby':
  ensure => installed,
}

package { 'puppet-lint':
  ensure   => installed,
  provider => gem,
}
```

`puppet-lint` is a Ruby gem and therefore we have to specify `provider => gem` for this package so that Puppet doesn't think it's a standard system package and try to install it via APT. Since the `gem` provider is not available unless Ruby is installed, we install the `ruby` package first, then the `puppet-lint` gem.

The `puppet-lint` tool, by the way, is a good thing to have installed. It will check your Puppet manifests for common style errors and make sure they comply with the official Puppet style guide. Try it now:

```
puppet-lint /examples/lint_test.pp
WARNING: indentation of => is not properly aligned (expected in column 11, but found
it in column 10) on line 2
```

In this example, `puppet-lint` is warning you that the `=>` arrows are not lined up vertically, which the style guide says they should be:

```
file { '/tmp/lint.txt':
  ensure => file,
  content => "puppet-lint is your friend\n",
}
```

When `puppet-lint` produces no output, the file is free of lint errors.

Installing gems in Puppet's context

Puppet itself is written at least partly in Ruby, and makes use of several Ruby gems. To avoid any conflicts with the version of Ruby and gems which the node might need for other applications, Puppet packages its own version of Ruby and associated gems under the `/opt/puppetlabs/` directory. This means you can install (or remove) whichever system version of Ruby you like and Puppet will not be affected.

However, if you need to install a gem to extend Puppet's capabilities in some way, then doing it with a `package` resource and `provider => gem` won't work. That is, the gem will be installed, but only in the system Ruby context, and it won't be visible to Puppet.

Fortunately, the `puppet_gem` provider is available for exactly this purpose. When you use this provider, the gem will be installed in Puppet's context (and, naturally, won't be visible in the system context). The following example demonstrates how to use this provider (`package_puppet_gem.pp`):

```
package { 'r10k':
  ensure   => installed,
  provider => puppet_gem,
}
```

Note

To see the gems installed in Puppet's context, use Puppet's own version of the `gem` command with the following path:

`/opt/puppetlabs/puppet/bin/gem list`

Using ensure_packages

To avoid potential package conflicts between different parts of your Puppet code or between your code and third-party modules, the Puppet standard library provides a useful wrapper for the `package` resource, called `ensure_packages()` . We'll cover this in detail in [Lab 7], [*Mastering modules*].

Services

Although services are implemented in a number of varied and complicated ways at the operating system level, Puppet does a good job of abstracting away most of this with the `service` resource and exposing just the two attributes of services which you most commonly need to manage: whether they're running (`ensure`) and whether they start at boot time (`enable`). We covered the use of these in [Lab 2], [*Creating your first manifests*], and most of the time, you won't need to know any more about `service` resources.

However, you'll occasionally encounter services which don't play well with Puppet, for a variety of reasons. Sometimes, Puppet is unable to detect that the service is already running and keeps trying to start it. Other times, Puppet may not be able to properly restart the service when a dependent resource changes. There are a few useful attributes for `service` resources which can help resolve these problems.

The hasstatus attribute

When a `service` resource has the attribute `ensure => running` attribute, Puppet needs to be able to check whether the service is, in fact, running. The way it does this depends on the underlying operating system. On Ubuntu 16 and later, for example, it runs `systemctl is-active SERVICE` . If the service is packaged to work with `systemd` , that should be just fine, but in many cases, particularly with older software, it may not respond properly.

If you find that Puppet keeps attempting to start the service on every Puppet run, even though the service is running, it may be that Puppet's default service status detection isn't working. In this case, you can specify the `hasstatus => false` attribute for the service (`service_hasstatus.pp`):

```
service { 'ntp':
  ensure    => running,
  enable    => true,
  hasstatus => false,
}
```

When `hasstatus` is false, Puppet knows not to try to check the service status using the default system service management command, and instead, will look in the process table for a running process which matches the name of the service. If it finds one, it will infer that the service is running and take no further action.

The pattern attribute

Sometimes, when using `hasstatus => false` , the service name as defined in Puppet doesn't actually appear in the process table, because the command that provides the service has a different name. If this is the case, you can tell Puppet exactly what to look for using the `pattern` attribute.

If `hasstatus` is `false` and `pattern` is specified, Puppet will search for the value of `pattern` in the process table to determine whether or not the service is running. To find the pattern you need, you can use the `ps` command to see the list of running processes:

```
ps ax
```

Find the process you're interested in and pick a string which will match only the name of that process. For example, if it's `ntpd` , you might specify the `pattern` attribute as `ntpd` (`service_pattern.pp`):

```
service { 'ntp':
  ensure    => running,
  enable    => true,
```

```
  hasstatus => false,
  pattern   => 'ntpd',
}
```

The hasrestart and restart attributes

When a service is notified (for example, if a `file` resource uses the `notify` attribute to tell the service its config file has changed, a common pattern which we looked at in [Lab 2], [*Creating your first manifests*]), Puppet's default behavior is to stop the service, then start it again. This usually works, but many services implement a `restart` command in their management scripts. If this is available, it's usually a good idea to use it: it may be faster or safer than stopping and starting the service. Some services take a while to shut down properly when stopped, for example, and Puppet may not wait long enough before trying to restart them, so that you end up with the service not running at all.

If you specify `hasrestart => true` for a service, then Puppet will try to send a `restart` command to it, using whatever service management command is appropriate for the current platform (`systemctl` , for example, on Ubuntu). The following example shows the use of `hasrestart` (`service_hasrestart.pp`):

```
service { 'ntp':
  ensure     => running,
  enable     => true,
  hasrestart => true,
}
```

To further complicate things, the default system service `restart` command may not work, or you may need to take certain special actions when the service is restarted (disabling monitoring notifications, for example). You can specify any `restart` command you like for the service using the `restart` attribute (`service_custom_restart.pp`):

```
service { 'ntp':
  ensure  => running,
  enable  => true,
  restart => '/bin/echo Restarting >>/tmp/debug.log && systemctl restart ntp',
}
```

In this example, the `restart` command writes a message to a log file before restarting the service in the usual way, but it could, of course, do anything you need it to. Note that the `restart` command is only used when Puppet restarts the service (generally because it was notified by a change to some config file). It's not used when starting the service from a stopped state. If Puppet finds the service has stopped and needs to start it, it will use the normal system service start command.

In the extremely rare event that the service cannot be stopped or started using the default service management command, Puppet also provides the `stop` and `start` attributes so that you can specify custom commands to stop and start the service, just the same way as with the `restart` attribute. If you need to use either of these, though, it's probably safe to say that you're having a bad day.

Users

A user on Unix-like systems does not necessarily correspond to a human person who logs in and types commands, although it sometimes does. A user is simply a named entity that can own files and run commands with certain permissions and that may or may not have permission to read or modify other users' files. It's very common, for

sound security reasons, to run each service on a system with its own user account. This simply means that the service runs with the identity and permissions of that user.

For example, a web server will often run as the `www-data` user, which exists solely to own files the web server needs to read and write. This limits the danger of a security breach via the web server, because the attacker would only have the `www-data` user's permissions, which are very limited, rather than the `root` user's, which can modify any aspect of the system. It is generally a bad idea to run services exposed to the public Internet as the `root` user. The service user should have only the minimum permissions it needs to operate the service.

Given this, an important part of system configuration involves creating and managing users, and Puppet's `user` resource provides a model for doing just that. Just as we saw with packages and services, the details of implementation and the commands used to manage users vary widely from one operating system to another, but Puppet provides an abstraction which hides those details behind a common set of attributes for users.

Creating users

The following example shows a typical `user` and `group` declaration in Puppet (`user.pp`):

```
group { 'devs':
  ensure => present,
  gid    => 3000,
}

user { 'hsing-hui':
  ensure => present,
  uid    => '3001',
  home   => '/home/hsing-hui',
  shell  => '/bin/bash',
  groups => ['devs'],
}
```

The user resource

The title of the resource is the username (login name) of the user; in this example, `hsing-hui` . The `ensure => present` attribute says that the user should exist on the system.

The `uid` attribute needs a little more explanation. On Unix-like systems, each user has an individual numerical id, known as the **uid**. The text name associated with the user is merely a convenience for those (mere humans, for example) who prefer strings to numbers. Access permissions are in fact based on the uid and not the username.

Note

Why set the `uid` attribute? Often, when creating users manually, we don't specify a uid, so the system will assign one automatically. The problem with this is that if you create the same user (`hsing-hui` , for example) on three different nodes, you may end up with three different uids. This would be fine as long as you have never shared files between nodes, or copied data from one place to another. But in fact, this happens all the time, so it's important to make sure that a given user's uid is the same across all the nodes in your infrastructure. That's why we specify the `uid` attribute in the Puppet manifest.

The `home` attribute sets the user's home directory (this will be the current working directory when the user logs in, if she does log in, and also the default working directory for cron jobs that run as the user).

The `shell` attribute specifies the command-line shell to run when the user logs in interactively. For humans, this will generally be a user shell, such as `/bin/bash` or `/bin/sh` . For service users, such as `www-data` , the shell

should be set to `/usr/sbin/nologin` (on Ubuntu systems), which does not allow interactive access, and prints a message saying `This account is currently not available`. All users who do not need to log in interactively should have the `nologin` shell.

If the user needs to be a member of certain groups, you can pass the `groups` attribute an array of the group names (just `devs` in this example).

Although Puppet supports a `password` attribute for `user` resources, I don't advise you to use it. Service users don't need passwords, and interactive users should be logging in with SSH keys. In fact, you should configure SSH to disable password logins altogether (set `PasswordAuthentication no` in `sshd_config`).

The group resource

The title of the resource is the name of the group (`devs`). You need not specify a `gid` attribute but, for the same reasons as the `uid` attribute, it's a good idea to do so.

Managing SSH keys

I like to have as few interactive logins as possible on production nodes, because it reduces the attack surface. Fortunately, with configuration management, it should rarely be necessary to actually log in to a node. The most common reasons for needing an interactive login are for system maintenance and troubleshooting, and for deployment. In both cases there should be a single account named for this specific purpose (for example, `admin` or `deploy`), and it should be configured with the SSH keys of any users or systems that need to log in to it.

Puppet provides the `ssh_authorized_key` resource to control the SSH keys associated with a user account. The following example shows how to use `ssh_authorized_key` to add an SSH key (mine, in this instance) to the `ubuntu` user on our Vagrant VM (`ssh_authorized_key.pp`):

```
ssh_authorized_key { 'john@bitfieldconsulting.com':
  user => 'ubuntu',
  type => 'ssh-rsa',
  key  =>
'AAAAB3NzaC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/2Tf2pl7kHdbsi

}
```

The title of the resource is the SSH key comment, which reminds us who the key belongs to. The `user` attribute specifies the user account which this key should be authorized for. The `type` attribute identifies the SSH key type, usually `ssh-rsa` or `ssh-dss`. Finally, the `key` attribute sets the key itself. When this manifest is applied, it adds the following to the `ubuntu` user's `authorized_keys` file:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/2Tf2pl7kHdbsi7
 john@bitfieldconsulting.com
```

A user account can have multiple SSH keys associated with it, and anyone holding one of the corresponding private keys and its passphrase will be able to log in as that user.

Removing users

If you need to have Puppet remove user accounts (for example, as part of an employee leaving process), it's not enough to simply remove the `user` resource from the Puppet manifest. Puppet will ignore any users on the system that it doesn't know about, and it certainly will not remove anything it finds on the system that isn't mentioned in the

Puppet manifest; that would be extremely undesirable (almost everything would be removed). So we need to retain the `user` declaration for a while, but set the `ensure` attribute to `absent` (`user_remove.pp`):

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

Once Puppet has run everywhere, you can remove the `user` resource if you like, but it does no harm to simply leave it in place, and in fact, it's a good idea to do this, unless you can verify manually that the user has been deleted from every affected system.

Note

If you need to prevent a user logging in, but want to retain the account and any files owned by the user, for archival or compliance purposes, you can set their `shell` to `/usr/sbin/nologin` . You can also remove any `ssh_authorized_key` resources associated with their account, and set the `purge_ssh_keys` attribute to `true` on the `user` resource. This will remove any authorized keys for the user that are not managed by Puppet.

Cron resources

Cron is the mechanism on Unix-like systems which runs scheduled jobs, sometimes known as batch jobs, at specified times or intervals. For example, system housekeeping tasks, such as log rotation or checking for security updates, are run from cron. The details of what to run and when to run it are kept in a specially formatted file called `crontab` (short for **cron table**).

Puppet provides the `cron` resource for managing scheduled jobs, and we saw an example of this in the `run-puppet` manifest we developed in [Lab 3], [*Managing your Puppet code with Git*] (`run-puppet.pp`):

```
cron { 'run-puppet':
  command => '/usr/local/bin/run-puppet',
  hour    => '*',
  minute  => '*/15',
}
```

The title `run-puppet` identifies the cron job (Puppet writes a comment to the `crontab` file containing this name to distinguish it from other manually-configured cron jobs). The `command` attribute specifies the command for cron to run, and the `hour` and `minute` specify the time (`*/15` is a cron syntax, meaning "every 15 minutes").

Note

For more information about cron and the possible ways to specify the times of scheduled jobs, run the command `man 5 crontab` .

Attributes of the cron resource

The `cron` resource has a few other useful attributes which are shown in the following example (`cron.pp`):

```
cron { 'cron example':
  command     => '/bin/date +%F',
  user        => 'ubuntu',
  environment => ['MAILTO=admin@example.com', 'PATH=/bin'],
  hour        => '0',
  minute      => '0',
```

```
  weekday     => ['Saturday', 'Sunday'],
}
```

The `user` attribute specifies who should run the cron job (if none is specified, the job runs as `root`). If the `environment` attribute is given, it sets any environment variables the cron job might need. A common use for this is to email any output from the cron job to a specified email address, using the `MAILTO` variable.

As before, the `hour` and `minute` attributes set the time for the job to run, while you can use the `weekday` attribute to specify a particular day, or days, of the week. (The `monthday` attribute works the same way, and can take any range or array of values between 1-31 to specify the day of the month.)

Note

One important point about cron scheduling is that the default value for any schedule attribute is `*`, which means [*all allowed values]. For example, if you do not specify an `hour` attribute, the cron job will be scheduled with an `hour` of `, meaning that it will run every hour. This is generally not what you want. If you do want it to run every hour, specify* hour => " `in your manifest, but otherwise, specify the particular hour it should run at. The same goes for` minute `. Accidentally leaving out the` minute` attribute and having a job run sixty times an hour can have amusing consequences, to say the least.

Randomizing cron jobs

If you run a cron job on many nodes, it's a good idea to make sure the job doesn't run everywhere at the same time. Puppet provides a built-in function `fqdn_rand()` to help with this; it provides a random number up to a specified maximum value, which will be different on each node, because the random number generator is seeded with the node's hostname.

If you have several such jobs to run, you can also supply a further seed value to the `fqdn_rand()` function, which can be any string and which will ensure that the value is different for each job (`fqdn_rand.pp`):

```
cron { 'run daily backup':
  command => '/usr/local/bin/backup',
  minute  => '0',
  hour    => fqdn_rand(24, 'run daily backup'),
}

cron { 'run daily backup sync':
  command => '/usr/local/bin/backup_sync',
  minute  => '0',
  hour    => fqdn_rand(24, 'run daily backup sync'),
}
```

Because we gave a different string as the second argument to `fqdn_rand` for each cron job, it will return a different random value for each `hour` attribute.

The range of values returned by `fqdn_rand()` includes 0, but does not include the maximum value you specify. So, in the previous example, the values for `hour` will be between 0 and 23, inclusive.

Removing cron jobs

Just as with `user` resources, or any type of resource, removing the resource declaration from your Puppet manifest does not remove the corresponding configuration from the node. In order to do that you need to specify `ensure`

`=> absent` on the resource.

Exec resources

While the other resource types we've seen so far (`file` , `package` , `service` , `user` , `ssh_authorized_key` , and `cron`) have modeled some concrete piece of state on the node, such as a file, the `exec` resource is a little different. An `exec` allows you to run any arbitrary command on the node. This might create or modify state, or it might not; anything you can run from the command line, you can run via an `exec` resource.

Automating manual interaction

The most common use for an `exec` resource is to simulate manual interaction on the command line. Some older software is not packaged for modern operating systems, and needs to be compiled and installed from source, which requires you to run certain commands. The authors of some software have also not realized, or don't care, that users may be trying to install their product automatically and have install scripts which prompt for user input. This can require the use of `exec` resources to work around the problem.

Attributes of the exec resource

The following example shows an `exec` resource for building and installing an imaginary piece of software (`exec.pp`):

```
exec { 'install-cat-picture-generator':
  cwd     => '/tmp/cat-picture-generator',
  command => '/tmp/cat-picture/generator/configure && /usr/bin/make install',
  creates => '/usr/local/bin/cat-picture-generator',
}
```

The title of the resource can be anything you like, though, as usual with Puppet resources it must be unique. I tend to name `exec` resources after the problem they're trying to solve, as in this example.

The `cwd` attribute sets the working directory where the command will be run (**current working directory**). When installing software, this is generally the software source directory.

The `command` attribute gives the command to run. This must be the full path to the command, but you can chain several commands together using the shell `&&` operator. This executes the next command only if the previous one succeeded, so in the example, if the `configure` command completes successfully, Puppet will go on to run `make install` , otherwise, it will stop with an error.

Note

If you apply this example, Puppet will give you an error like the following:

```
Error: /Stage[main]/Main/Exec[install-cat-picture-generator]/returns: change from
notrun to 0 failed: Could not find command '/tmp/cat-picture/generator/configure'
```

This is expected because the specified command does not, in fact, exist. In your own manifests, you may see this error if you give the wrong path to a command, or if the package that provides the command hasn't been installed yet.

The `creates` attribute specifies a file which should exist after the command has been run. If this file is present, Puppet will not run the command again. This is very useful because without a `creates` attribute, an `exec`

resource will run every time Puppet runs, which is generally not what you want. The `creates` attribute tells Puppet, in effect, "Run the `exec` only if this file doesn't exist."

Let's see how this works, imagining that this `exec` is being run for the first time. We assume that the `/tmp/cat-picture/` directory exists and contains the source of the `cat-picture-generator` application.

1. Puppet checks the `creates` attribute and sees that the `/usr/local/bin/cat-picture-generator` file is not present; therefore, the `exec` resource must be run.
2. Puppet runs the `/tmp/cat-picture-generator/configure && /usr/bin/make install` command. As a side effect of these commands, the `/usr/local/bin/cat-picture-generator` file is created.
3. Next time Puppet runs, it again checks the `creates` attribute. This time `/usr/local/bin/cat-picture-generator` exists, so Puppet does nothing.

This `exec` resource will never be applied again so long as the file specified in the `creates` attribute exists. You can test this by deleting the file and applying Puppet again. The `exec` resource will be triggered and the file recreated.

Note

Make sure that your `exec` resources always include a `creates` attribute (or a similar control attribute, such as `onlyif` or `unless`, which we'll look at later in this lab). Without this, the `exec` command will be run every time Puppet runs, which is almost certainly not what you want.

Note that building and installing software from source is not a recommended practice for production systems. It's better to build the software on a dedicated build server (perhaps using Puppet code similar to this example), create a system package for it, and then use Puppet to install that package on production nodes.

The user attribute

If you don't specify a `user` attribute for an `exec` resource, Puppet will run the command as the `root` user. This is often appropriate for installing system software or making changes to the system configuration, but if you need the command to run as a particular user, specify the `user` attribute, as in the following example (`exec_user.pp`):

```
exec { 'say-hello':
  command => '/bin/echo Hello, this is `whoami` >/tmp/hello-ubuntu.txt',
  user    => 'ubuntu',
  creates => '/tmp/hello-ubuntu.txt',
}
```

This will run the specified command as the `ubuntu` user. The `whoami` command returns the name of the user running it, so when you apply this manifest, the file `/tmp/hello-ubuntu.txt` will be created with the following contents:

```
Hello, this is ubuntu
```

As with the earlier example, the `creates` attribute prevents Puppet from running this command more than once.

The onlyif and unless attributes

Suppose you only want an `exec` resource to be applied under certain conditions. For example, a command which processes incoming data files only needs to run if there are data files waiting to be processed. In this case, it's no good adding a `creates` attribute; we want the existence of a certain file to trigger the `exec`, not prevent it.

The `onlyif` attribute is a good way to solve this problem. It specifies a command for Puppet to run, and the exit status from this command determines whether or not the `exec` will be applied. On Unix-like systems, commands generally return an exit status of zero to indicate success and a non-zero value for failure. The following example shows how to use `onlyif` in this way (`exec_onlyif.pp`):

```
exec { 'process-incoming-cat-pictures':
  command => '/usr/local/bin/cat-picture-generator --import /tmp/incoming/*',
  onlyif  => '/bin/ls /tmp/incoming/*',
}
```

The exact command isn't important here, but let's assume it's something that we would only want to run if there are any files in the `/tmp/incoming/` directory.

The `onlyif` attribute specifies the check command which Puppet should run first, to determine whether or not the `exec` resource needs to be applied. If there is nothing in the `/tmp/incoming/` directory, then `ls /tmp/incoming/*` will return a non-zero exit status. Puppet interprets this as failure, so does not apply the `exec` resource.

On the other hand, if there are files in the `/tmp/incoming/` directory, the `ls` command will return success. This tells Puppet the `exec` resource must be applied, so it proceeds to run the `/usr/local/bin/cat-picture-generator` command (and we can assume this command deletes the incoming files after processing).

You can think of the `onlyif` attribute as telling Puppet, "Run the `exec` resource [*only if*] this command succeeds."

The `unless` attribute is exactly the same as `onlyif` but with the opposite sense. If you specify a command to the `unless` attribute, the `exec` will always be run unless the command returns a zero exit status. You can think of `unless` as telling Puppet, "Run the `exec` resource [*unless*] this command succeeds."

When you apply your manifest, if you see an `exec` resource running every time which shouldn't be, check whether it specifies a `creates`, `unless`, or `onlyif` attribute. If it specifies the `creates` attribute, it may be looking for the wrong file; if the `unless` or `onlyif` command is specified, it may not be returning what you expect. You can see what command is being run and what output it generates by running `sudo puppet apply` with the `-d` (debug) flag:

```
sudo puppet apply -d exec_onlyif.pp
Debug: Exec[process-incoming-cat-pictures](provider=posix): Executing check '/bin/ls
/tmp/incoming/*'
Debug: Executing: '/bin/ls /tmp/incoming/*'
Debug: /Stage[main]/Main/Exec[process-incoming-cat-pictures]/onlyif: /tmp/incoming/foo
```

The refreshonly attribute

It's quite common to use `exec` resources for one-off commands, such as rebuilding a database, or setting a system-tunable parameter. These generally only need to be triggered once, when a package is installed, or occasionally, when a config file is updated. If an `exec` resource needs to run only when some other Puppet resource is changed, we can use the `refreshonly` attribute to do this.

If `refreshonly` is `true`, the `exec` will never be applied unless another resource triggers it with `notify`. In the following example, Puppet manages the `/etc/aliases` file (which maps local usernames to email addresses), and a change to this file triggers the execution of the command `newaliases`, which rebuilds the system alias database (`exec_refreshonly.pp`):

```
file { '/etc/aliases':
  content => 'root: john@bitfieldconsulting.com',
  notify  => Exec['newaliases'],
}

exec { 'newaliases':
  command     => '/usr/bin/newaliases',
  refreshonly => true,
}
```

When this manifest is applied for the first time, the `/etc/aliases` resource causes a change to the file's contents, so Puppet sends a `notify` message to the `exec` resource. This causes the `newaliases` command to be run. If you apply the manifest again, you will see that the `aliases` file is not changed, so the `exec` is not run.

Note

While the `refreshonly` attribute is occasionally extremely useful, over-use of it can make your Puppet manifests hard to understand and debug, and it can also be rather fragile. Felix Frank makes this point in a blog post, [*Friends Don't Let Friends Use Refreshonly*]:

"With the `exec` resource type considered the last ditch, its `refreshonly` parameter should be seen as especially outrageous. To make an `exec` resource fit into Puppet's model better, you should use [the `creates`, `onlyif`, or `unless`] parameters instead." Refer to:

http://ffrank.github.io/misc/2015/05/26/friends-don't-let-friends-use-refreshonly/

Note that you don't need to use the `refreshonly` attribute in order to make the `exec` resource notifiable by other resources. Any resource can notify an `exec` resource in order to make it run; however, if you don't want it to run [*unless*] it's notified, use `refreshonly`.

Note

By the way, if you actually want to manage email aliases on a node, use Puppet's built-in `mailalias` resource. The previous example is just to demonstrate the use of `refreshonly`.

The logoutput attribute

When Puppet runs shell commands via an `exec` resource, the output is normally hidden from us. However, if the command doesn't seem to be working properly, it can be very useful to see what output it produced, as this usually tells us why it didn't work.

The `logoutput` attribute determines whether Puppet will log the output of the `exec` command along with the usual informative Puppet output. It can take three values: `true`, `false`, or `on_failure`.

If `logoutput` is set to `on_failure` (which is the default), Puppet will only log command output when the command fails (that is, returns a non-zero exit status). If you never want to see command output, set it to `false`.

Sometimes, however, the command returns a successful exit status but does not appear to do anything. Setting `logoutput` to `true` will force Puppet to log the command output regardless of exit status, which should help

you figure out what's going on.

The timeout attribute

Sometimes, commands can take a long time to run, or never terminate at all. By default, Puppet allows an `exec` command to run for 300 seconds, at which point Puppet will terminate it if it has not finished. If you need to allow a little longer for the command to complete, you can use the `timeout` attribute to set this. The value is the maximum execution time for the command in seconds.

Setting a `timeout` value of `0` disables the automatic timeout altogether and allows the command to run forever. This should be the last resort, as a command which blocks or hangs could stop Puppet's automatic runs altogether if no timeout is set. To find a suitable value for `timeout`, try running the command a few times and choose a value which is perhaps twice as long as a typical run. This should avoid failures caused by slow network conditions, for example, but not block Puppet from running altogether.

How not to misuse exec resources

The `exec` resource can do anything to the system that you could do from the command line. As you can imagine, such a powerful tool can be misused. In theory, Puppet is a declarative language: the manifest specifies the way things should be, and it is up to Puppet to take the necessary actions to make them so. Manifests are therefore what computer scientists call **idempotent**: the system is always in the same state after the catalog has been applied, and however many times you apply it, it will always be in that state.

The `exec` resource rather spoils this theoretical picture, by allowing Puppet manifests to have side-effects. Since your `exec` command can do anything, it could, for example, create a new 1 GB file on disk with a random name, and since this will happen every time Puppet runs, you could rapidly run out of disk space. It's best to avoid commands with side-effects like this. In general, there's no way to know from within Puppet exactly what changes to a system were caused by an `exec` resource.

Commands run via `exec` are also sometimes used to bypass Puppet's existing resources. For example, if the `user` resource doesn't do quite what you want for some reason, you could create a user by running the `adduser` command directly from an `exec`. This is also a bad idea, since by doing this you lose the declarative and cross-platform nature of Puppet's built-in resources. `exec` resources potentially change the state of the node in a way that's invisible to Puppet's catalog.

Note

In general, if you need to manage a concrete aspect of system state which isn't supported by Puppet's built-in resource types, you should think about creating a custom resource type and provider to do what you want. This extends Puppet to add a new resource type, which you can then use to model the state of that resource in your manifests. Creating custom types and providers is an advanced topic and not covered in this course, but if you want to know more, consult the Puppet documentation:

https://docs.puppet.com/guides/custom_types.html

You should also think twice before running complex commands via `exec`, especially commands which use loops or conditionals. It's a better idea to put any complicated logic in a shell script (or, even better, in a real programming language), which you can then deploy and run with Puppet (avoiding, as we've said, unnecessary side-effects).

Note

As a matter of good Puppet style, every `exec` resource should have at least one of `creates`, `onlyif`, `unless`, or `refreshonly` specified, to stop it from being applied on every Puppet run. If you find yourself using `exec` just to run a command every time Puppet runs, make it a cron job instead.

Summary

We've explored Puppet's `file` resource in detail, covering file sources, ownership, permissions, directories, symbolic links, and file trees. We've learned how to manage packages by installing specific versions, or the latest version, and how to uninstall packages. We've covered Ruby gems, both in the system context and Puppet's internal context. Along the way, we met the very useful `puppet-lint` tool.

We have looked at `service` resources, including the `hasstatus`, `pattern`, `hasrestart`, `restart`, `stop`, and `start` attributes. We've learned how to create users and groups, manage home directories, shells, UIDs, and SSH authorized keys. We saw how to schedule, manage, and remove cron jobs.

Finally, we've learned all about the powerful `exec` resource, including how to run arbitrary commands, and how to run commands only under certain conditions, or only if a specific file is not present. We've seen how to use the `refreshonly` attribute to trigger an `exec` resource when other resources are updated, and we've explored the useful `logoutput` and `timeout` attributes of `exec` resources.

In the next lab, we'll find out how to represent data and variables in Puppet manifests, including strings, numbers, Booleans, arrays, and hashes. We'll learn how to use variables and conditional expressions to determine which resources are applied, and we'll also learn about Puppet's `facts` hash and how to use it to get information about the system.