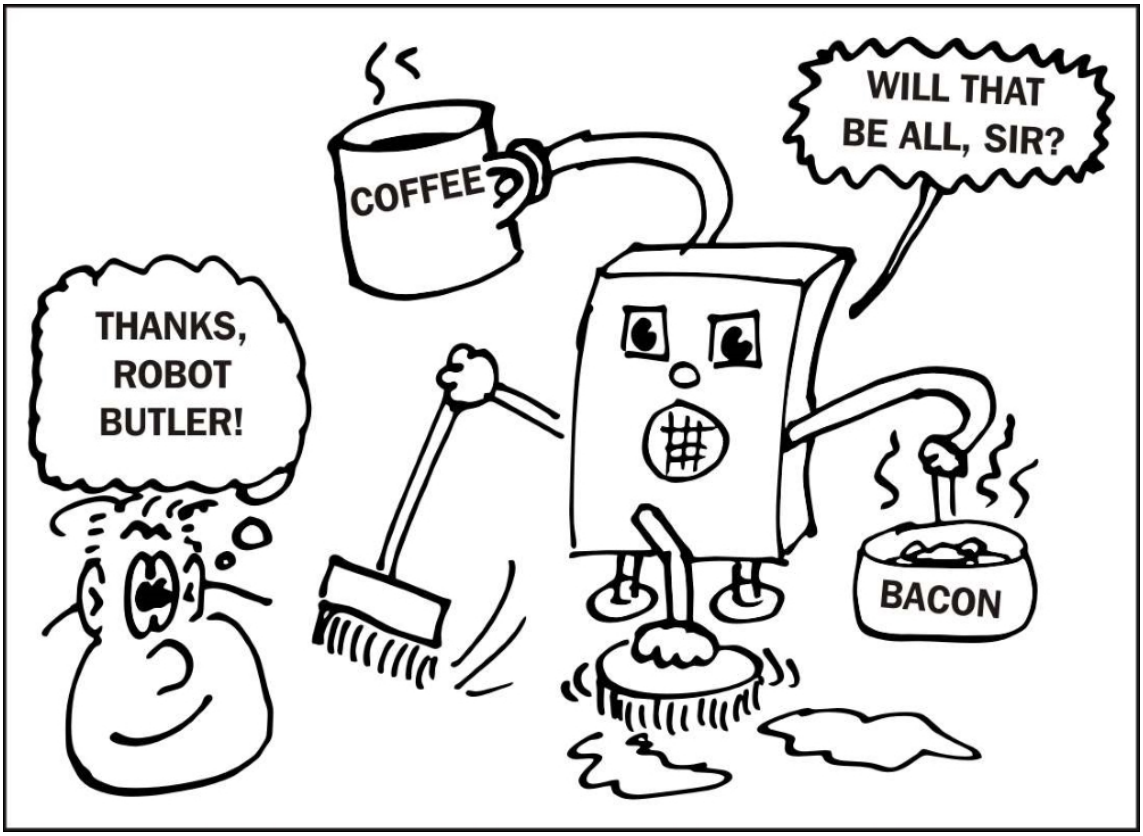
Lab 2. Creating your first manifests

[*Beginnings are such delicate times.*]

--[[*Frank Herbert, 'Dune'*]]{.attribution}

In this lab, you'll learn how to write your first manifest with Puppet, and how to put Puppet to work configuring a server. You'll also understand how Puppet compiles and applies a manifest. You'll see how to use Puppet to manage the contents of files, how to install packages, and how to control services.



Hello, Puppet -- your first Puppet manifest

The first example program in any programming language, by tradition, prints `hello, world`. Although we can do that easily in Puppet, let's do something a little more ambitious, and have Puppet create a file on the server containing that text.

On your Vagrant box, run the following command:

```
sudo puppet apply /examples/file_hello.pp
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.07 seconds
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/ensure: defined content as
```

```
'{md5}22c3683b094136c3398391ae71b20f04'
Notice: Applied catalog in 0.01 seconds
```

We can ignore the output from Puppet for the moment, but if all has gone well, we should be able to run the following command:

```
cat /tmp/hello.txt
hello, world
```

Understanding the code

Let's look at the example code to see what's going on (run `cat /example/file_hello.pp`, or open the file in a text editor):

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

The code term `file` begins a **resource declaration** for a `file` resource. A **resource** is some bit of configuration that you want Puppet to manage: for example, a file, user account, or package. A resource declaration follows this pattern:

```
RESOURCE_TYPE { TITLE:
  ATTRIBUTE => VALUE,
  ...
}
```

Resource declarations will make up almost all of your Puppet manifests, so it's important to understand exactly how they work:

- `RESOURCE_TYPE` indicates the type of resource you're declaring; in this case, it's a `file`.
- `TITLE` is the name that Puppet uses to identify the resource internally. Every resource must have a unique title. With `file` resources, it's usual for this to be the full path to the file: in this case, `/tmp/hello`.

The remainder of this block of code is a list of attributes that describe how the resource should be configured. The attributes available depend on the type of the resource. For a file, you can set attributes such as `content`, `owner`, `group`, and `mode`, but one attribute that every resource supports is `ensure`.

Again, the possible values for `ensure` are specific to the type of resource. In this case, we use `file` to indicate that we want a regular file, as opposed to a directory or symlink:

```
ensure  => file,
```

Next, to put some text in the file, we specify the `content` attribute:

```
content => "hello, world\n",
```

The `content` attribute sets the contents of a file to a string value you provide. Here, the contents of the file are declared to be `hello, world`, followed by a newline character (in Puppet strings, we write the newline character as `\n`).

Note that `content` specifies the entire content of the file; the string you provide will replace anything already in the file, rather than be appended to it.

Modifying existing files

What happens if the file already exists when Puppet runs and it contains something else? Will Puppet change it?

```
sudo sh -c 'echo "goodbye, world" >/tmp/hello.txt'
cat /tmp/hello.txt
goodbye, world
sudo puppet apply /examples/file_hello.pp
cat /tmp/hello.txt
hello, world
```

The answer is yes. If any attribute of the file, including its contents, doesn't match the manifest, Puppet will change it so that it does.

This can lead to some surprising results if you manually edit a file managed by Puppet. If you make changes to a file without also changing the Puppet manifest to match, Puppet will overwrite the file the next time it runs, and your changes will be lost.

So it's a good idea to add a comment to files that Puppet is managing: something like the following:

```
# This file is managed by Puppet - any manual edits will be lost
```

Add this to Puppet's copy of the file when you first deploy it, and it will remind you and others not to make manual changes.

Dry-running Puppet

Because you can't necessarily tell in advance what applying a Puppet manifest will change on the system, it's a good idea to do a dry run first. Adding the `--noop` flag to `puppet apply` will show you what Puppet would have done, without actually changing anything:

```
sudo sh -c 'echo "goodbye, world" >/tmp/hello.txt'
sudo puppet apply --noop /examples/file_hello.pp
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.04 seconds
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/content: current_value {md5}7678...,
should be {md5}22c3... (noop)
```

Puppet decides whether or not a `file` resource needs updating, based on its MD5 hash sum. In the previous example, Puppet reports that the current value of the hash sum for `/tmp/hello.txt` is `7678...`, whereas according to the manifest, it should be `22c3...`. Accordingly, the file will be changed on the next Puppet run.

If you want to see what change Puppet would actually make to the file, you can use the `--show_diff` option:

```
sudo puppet apply --noop --show_diff /examples/file_hello.pp
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.04 seconds
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/content:
--- /tmp/hello.txt      2017-02-13 02:27:13.186261355 -0800
+++ /tmp/puppet-file20170213-3671-2yynjt        2017-02-13 02:30:26.561834755 -0800
@@ -1 +1 @@
-goodbye, world
+hello, world
```

These options are very useful when you want to make sure that your Puppet manifest will affect only the things you're expecting it to---or, sometimes, when you want to check if something has been changed outside Puppet without actually undoing the change.

How Puppet applies the manifest

Here's how your manifest is processed. First, Puppet reads the manifest and the list of resources it contains (in this case, there's just one resource), and compiles these into a catalog (an internal representation of the desired state of the node).

Puppet then works through the catalog, applying each resource in turn:

1. First, it checks if the resource exists on the server. If not, Puppet creates it. In the example, we've declared that the file `/tmp/hello.txt` should exist. The first time you run `sudo puppet apply`, this won't be the case, so Puppet will create the file for you.
2. Then, for each resource, it checks the value of each attribute in the catalog against what actually exists on the server. In our example, there's just one attribute: `content`. We've specified that the content of the file should be `hello, world\n`. If the file is empty or contains something else, Puppet will overwrite the file with what the catalog says it should contain.

In this case, the file will be empty the first time you apply the catalog, so Puppet will write the string `hello, world\n` into it.

We'll go on to examine the `file` resource in much more detail in later chapters.

Creating a file of your own

Create your own manifest file (you can name it anything you like, so long as the file extension is `.pp`). Use a `file` resource to create a file on the server with any contents you like. Apply the manifest with Puppet and check that the file is created and contains the text you specified.

Edit the file directly and change the contents, then re-apply Puppet and check that it changes the file back to what the manifest says it should contain.

Managing packages

Another key resource type in Puppet is the **package**. A major part of configuring servers by hand involves installing packages, so we will also be using packages a lot in Puppet manifests. Although every operating system has its own package format, and different formats vary quite a lot in their capabilities, Puppet represents all these possibilities with a single `package` type. If you specify in your Puppet manifest that a given package should be installed, Puppet will use the appropriate package manager commands to install it on whatever platform it's running on.

As you've seen, all resource declarations in Puppet follow this form:

```
RESOURCE_TYPE { TITLE:
  ATTRIBUTE => VALUE,
  ...
}
```

`package` resources are no different. The `RESOURCE_TYPE` is `package`, and the only attribute you usually need to specify is `ensure`, and the only value it usually needs to take is `installed`:

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

Try this example:

```
sudo puppet apply /examples/package.pp
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.52 seconds
Notice: /Stage[main]/Main/Package[cowsay]/ensure: created
Notice: Applied catalog in 29.53 seconds
```

Let's see whether `cowsay` is installed:

```
cowsay Puppet rules!
 _________________
< Puppet rules! >
 -----------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Now that's a useful package!

How Puppet applies the manifest

The title of the `package` resource is `cowsay`, so Puppet knows that we're talking about a package named `cowsay`.

The `ensure` attribute governs the installation state of packages: unsurprisingly, `installed` tells Puppet that the package should be installed.

As we saw in the earlier example, Puppet processes this manifest by examining each resource in turn and checking its attributes on the server against those specified in the manifest. In this case, Puppet will look for the `cowsay` package to see whether it's installed. It is not, but the manifest says it should be, so Puppet carries out all the necessary actions to make []{#id43 .indexterm}reality match the manifest, which here means installing the package.

Note

It's still early on in the course, but you can already do a great deal with Puppet! If you can install packages and manage the contents of files, you can get a very long way towards setting up any kind of server configuration you might need. If you were to stop reading right here (which would be a shame, but we're all busy people), you would still be able to use Puppet to automate a large part of the configuration work you will encounter. But Puppet can do much more.

Exercise

Create a manifest that uses the `package` resource to install any software you find useful for managing servers. Here are some suggestions: `tmux`, `sysdig`, `atop`, `htop`, and `dstat`.

Querying resources with the puppet resource

If you want to see what version of a package Puppet thinks you have installed, you can use the `puppet resource` tool:

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

`puppet resource TYPE TITLE` will output a Puppet manifest representing the current state of the named resource on the system. If you leave out `TITLE`, you'll get a manifest for all the resources of the type `TYPE`. For example, if you run `puppet resource package`, you'll see the Puppet code for all the packages installed on the system.

Note

`puppet resource` even has an interactive configuration feature. To use it, run the following command:

```
puppet resource -e package openssl
```

If you run this, Puppet will generate a manifest for the current state of the resource, and open it in an editor. If you now make changes and save it, Puppet will apply that manifest to make changes to the system. This is a fun little feature, but it would be rather time-consuming to do your entire configuration this way.

Services

The third most important Puppet resource type is the **service**: a long-running process that either does some continuous kind of work, or waits for requests and then acts on them. For example, on most systems, the `sshd` process runs all the time and listens for SSH login attempts.

Puppet models services with the `service` resource type. The `service` resources look like the following example (you can find this in `service.pp` in the `/examples/` directory. From now on, I'll just give the filename of each example, as they are all in the same directory):

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

The `ensure` parameter governs whether the service should be running or not. If its value is `running`, then as you might expect, Puppet will start the service if it is not running. If you set `ensure` to `stopped`, Puppet will stop the service if it is running.

Services may also be set to start when the system boots, using the `enable` parameter. If `enable` is set to `true`, the service will start at boot. If, on the other hand, `enable` is set to `false`, it will not. Generally speaking, unless there's a good reason not to, all services should be set to start at boot.

Getting help on resources with puppet describe

If you're struggling to remember all the different attributes of all the different resources, Puppet has a built-in help feature that will remind you. Run the following command, for example:

```
puppet describe service
```

This will give a description of the `service` resource, along with a complete list of attributes and allowed values. This works for all built-in resource types as well as many provided by third-party modules. To see a list of all the available resource types, run the following command:

```
puppet describe --list
```

The package-file-service pattern

It's very common for a given piece of software to require these three Puppet resource types: the `package` resource installs the software, the `file` resource deploys one or more configuration files required for the software, and the `service` resource runs the software itself.

Here's an example using the MySQL database server (`package_file_service.pp`):

```
package { 'mysql-server':
  ensure => installed,
  notify => Service['mysql'],
}

file { '/etc/mysql/mysql.cnf':
  source => '/examples/files/mysql.cnf',
  notify => Service['mysql'],
}

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

The `package` resource makes sure the `mysql-server` package is installed.

The config file for MySQL is `/etc/mysql/mysql.cnf` , and we use a `file` resource to copy this file from the Puppet repo so that we can control MySQL settings.

Finally, the `service` resource ensures that the `mysql` service is running.

Notifying a linked resource

You might have noticed a new attribute, called `notify` , in the `file` resource in the previous example:

```
file { '/etc/mysql/mysql.cnf':
  source => '/examples/files/mysql.cnf',
  notify => Service['mysql'],
}
```

What does this do? Imagine you've made a change to the `mysql.cnf` file and applied this change with Puppet. The updated file will be written to a disk, but because the `mysql` service is already running, it has no way of knowing that its config file has changed. Therefore, your changes will not actually take effect until the service is restarted. However, Puppet can do this for you if you specify the `notify` attribute on the `file` resource. The value of `notify` is the resource to notify about the change, and what that involves depends on the type of resource that's being notified. When it's a service, the default action is to restart the service.

Usually, with the package-file-service pattern, the file notifies the service, so whenever Puppet changes the contents of the file, it will restart the notified service to pick up the new configuration. If there are several files that affect the service, they should all notify the service, and Puppet is smart enough to only restart the service once, however many dependent resources are changed.

The name of the resource to notify is specified as the resource type, capitalized, followed by the resource title, which is quoted and within square brackets: `Service['mysql']`.

Resource ordering with require

In the package-file-service example, we declared three resources: the `mysql-server` package, the `/etc/mysql/mysql.cnf` file, and the `mysql` service. If you think about it, they need to be applied in that order. Without the `mysql-server` package installed, there will be no `/etc/mysql/` directory to put the `mysql.cnf` file in. Without the package or the config file, the `mysql` service won't be able to run.

A perfectly reasonable question to ask is, "Does Puppet apply resources in the same order in which they're declared in the manifest?" The answer is usually yes, unless you explicitly specify a different order, using the `require` attribute.

All resources support the `require` attribute, and its value is the name of another resource declared somewhere in the manifest, specified in the same way as when using `notify`. Here's the package-file-service example again, this time with the resource ordering specified explicitly using `require` (`package_file_service_require.pp`):

```
package { 'mysql-server':
  ensure => installed,
}

file { '/etc/mysql/mysql.cnf':
  source  => '/examples/files/mysql.cnf',
  notify  => Service['mysql'],
  require => Package['mysql-server'],
}

service { 'mysql':
  ensure  => running,
  enable  => true,
  require => [Package['mysql-server'], File['/etc/mysql/mysql.cnf']],
}
```

You can see that the `mysql.cnf` resource requires the `mysql-server` package. The `mysql` service requires both the other resources, listed as an array within square brackets.

When resources are already in the right order, you don't need to use `require`, as Puppet will apply the resources in the order you declare them. However, it can be useful to specify an ordering explicitly, for the benefit of those reading the code, especially when there are lots of resources in a manifest file.

In older versions of Puppet, resources were applied in a more or less arbitrary order, so it was much more important to express dependencies using `require`. Nowadays, you won't need to use it very much, and you'll mostly come across it in legacy code.

Summary

In this lab, we've seen how a manifest is made up of Puppet resources. You've learned how to use Puppet's `file` resource to create and modify files, how to install packages using the `package` resource, and how to manage services with the `service` resource. We've looked at the common package-file-service pattern and seen how to use the `notify` attribute on a resource to send a message to another resource indicating that its configuration has been updated. We've covered the use of the `require` attribute to make dependencies between resources explicit, when necessary.

You've also learned to use `puppet resource` to inspect the current state of the system according to Puppet, and `puppet describe` to get command-line help on all Puppet resources. To check what Puppet would change on the system without actually changing it, we've introduced the `--noop` and `--show_diff` options to `puppet apply`.

In the next lab, we'll see how to use the version control tool Git to keep track of your manifests, we'll get an introduction to fundamental Git concepts, such as the repo and the commit, and you'll learn how to distribute your code to each of the servers you're going to manage with Puppet.