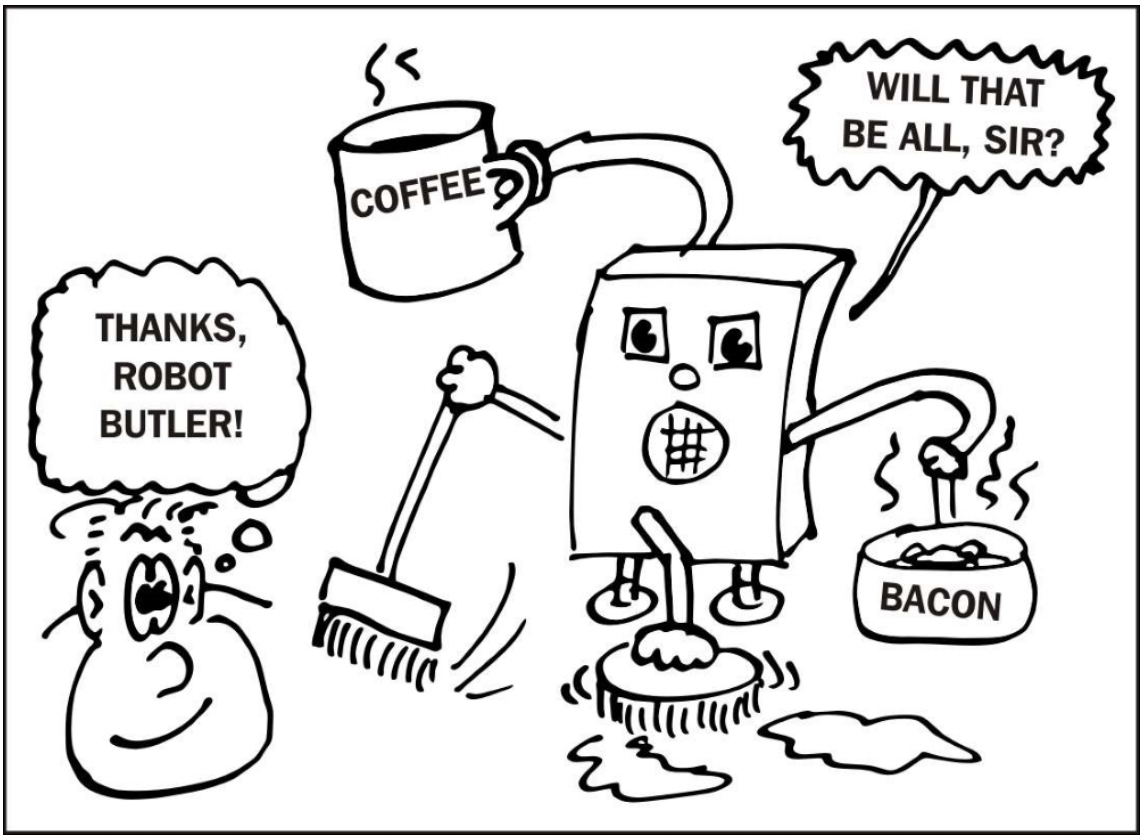
Lab 1. Creating your first manifests

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.