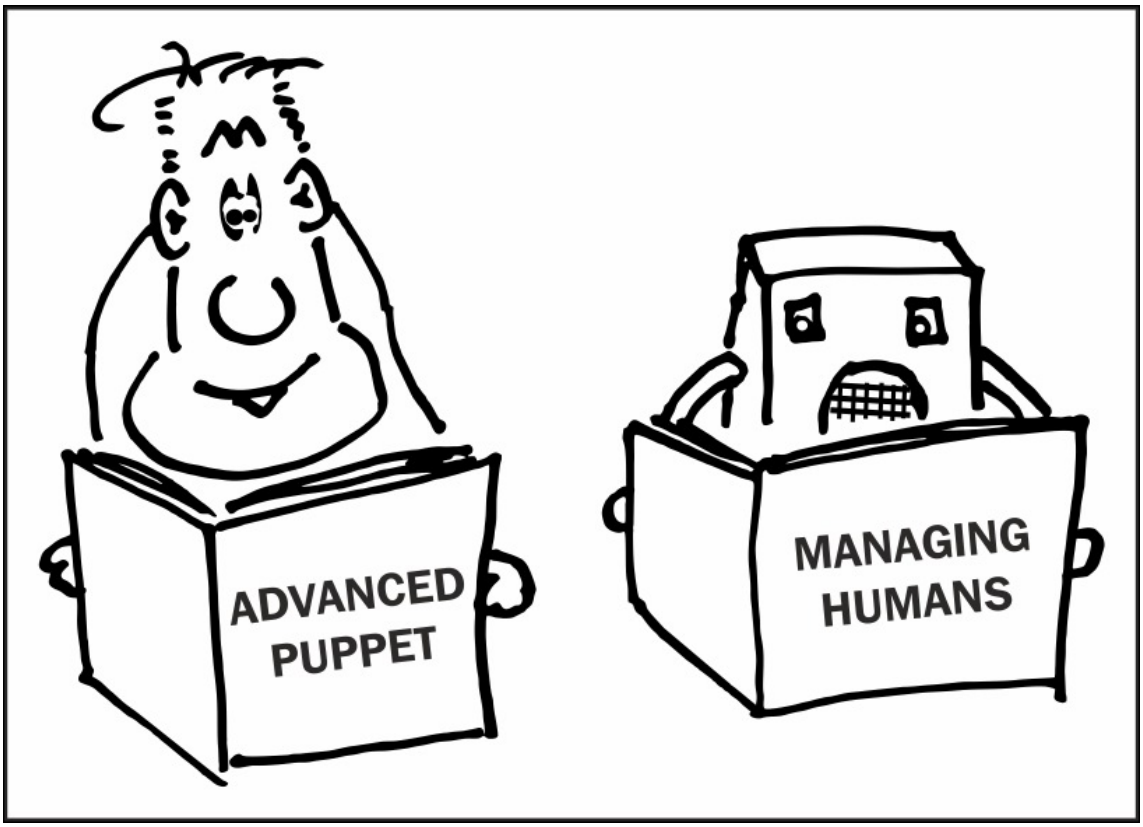
Lab 7. Mastering modules

In this lab you'll learn about Puppet Forge, the public repository for Puppet modules, and you'll see how to install and use third-party modules from Puppet Forge, using the `r10k` module management tool. Finally, working through a complete example, you'll learn how to develop your own Puppet module from scratch, how to add appropriate metadata for your module, and how to upload it to Puppet Forge.



What is the Puppet Forge?

The **Puppet Forge** is a public repository of Puppet modules, many of them officially supported and maintained by Puppet and all of which you can download and use. You can browse the Forge at the following URL:

https://forge.puppet.com/

Finding the module you need

The Puppet Forge home page has a search bar at the top. Type what you're looking for into this box, and the website will show you all the modules which match your search keywords. Often, there will be more than one result, so how do you decide which module to use?

The best choice is a **Puppet Supported** module, if one is available.

https://forge.puppet.com/modules?endorsements=supported

The next best option is a **Puppet Approved** module. While not officially supported, these modules are recommended by Puppet and have been checked to make sure they follow best practices and meet certain quality standards.

https://forge.puppet.com/modules?endorsements=approved

Note: Copy and open above url in Midori browser.

Using r10k

`r10k` is the de facto standard module manager for Puppet deployments, and we'll be using it to manage modules throughout the rest of this course.

In this example, we'll use `r10k` to install the `puppetlabs/stdlib` module. The Puppetfile in the example repo contains a list of all the modules we'll use in this course. Here it is (we'll look more closely at the syntax in a moment):

```
forge 'http://forge.puppetlabs.com'

mod 'puppet/archive', '1.3.0'
mod 'puppetlabs/apache', '2.0.0'
mod 'puppetlabs/apt', '3.0.0'
mod 'puppetlabs/concat', '4.0.1'
mod 'puppetlabs/mysql', '3.11.0'
mod 'puppetlabs/stdlib', '4.17.1'
mod 'stahnma/epel', '1.2.2'

mod 'pbg_ntp',
  :git => 'https://github.com/fenago/pbg_ntp.git',
  :tag => '0.1.4'
```

Follow these steps:

1. Run the following commands to clear out your `modules/` directory, if there's anything in it (make sure you have backed up anything here you want to keep):

   ```
   cd /etc/puppetlabs/code/environments/pbg
   sudo rm -rf modules/
   ```

2. Run the following command to have `r10k` process the example Puppetfile here and install your requested modules:

   ```
   sudo r10k puppetfile install --verbose
   ```

`r10k` downloads all the modules listed in the Puppetfile into the `modules/` directory. All modules in this directory will be automatically loaded by Puppet and available for use in your manifests. To test that the `stdlib` module is correctly installed, run the following command:

```
puppet apply --environment pbg -e "notice(upcase('hello'))"
Notice: Scope(Class[main]): HELLO
```

The `upcase` function, which converts its string argument to uppercase, is part of the `stdlib` module. If this doesn't work, then `stdlib` has not been properly installed. As in previous examples, we're using the `--environment pbg` switch to tell Puppet to look for code, modules, and data in the `/etc/puppetlabs/code/environments/pbg` directory.

Understanding the Puppetfile

The example Puppetfile begins with the following:

```
forge 'http://forge.puppetlabs.com'
```

The `forge` statement specifies the repository where modules should be retrieved from.

There follows a group of lines beginning with `mod`:

```
mod 'puppet/archive', '1.3.0'
mod 'puppet/staging', '2.2.0'
...
```

The `mod` statement specifies the name of the module (`puppetlabs/stdlib`) and the specific version of the module to install (`4.17.0`).

Managing dependencies with generate-puppetfile

`r10k` does not automatically manage dependencies between modules. For example, the `puppetlabs/apache` module depends on having both `puppetlabs/stdlib` and `puppetlabs/concat` installed. `r10k` will not automatically install these for you unless you specify them, so you also need to include them in your Puppetfile.

However, you can use the `generate-puppetfile` tool to find out what dependencies you need so that you can add them to your Puppetfile.

1. Run the following command to install the `generate-puppetfile` gem:

   ```
   sudo gem install generate-puppetfile
   ```

2. Run the following command to generate the Puppetfile for a list of specified modules (list all the modules you need on the command line, separated by spaces):

   ```
   generate-puppetfile puppetlabs/docker_platform

   Installing modules. This may take a few minutes.
   Your Puppetfile has been generated. Copy and paste between the markers:
   ===========================================
   forge 'http://forge.puppetlabs.com'

   # Modules discovered by generate-puppetfile
   mod 'garethr/docker', '5.3.0'
   mod 'puppetlabs/apt', '3.0.0'
   mod 'puppetlabs/docker_platform', '2.2.1'
   mod 'puppetlabs/stdlib', '4.17.1'
   mod 'stahnma/epel', '1.2.2'
   ===========================================
   ```

3. Run the following command to generate a list of updated versions and dependencies for an existing Puppetfile:

   ```
   generate-puppetfile -p /etc/puppetlabs/code/environments/pbg/Puppetfile
   ```

This is an extremely useful tool both for finding dependencies you need to specify in your Puppetfile and for keeping your Puppetfile up to date with the latest versions of all the modules you use.

Using modules in your manifests

Now that we know how to find and install public Puppet modules, let's see how to use them. We'll work through a few examples, using the `puppetlabs/apache` module to set up an Apache website, and using `puppet/archive` to download and unpack a compressed archive. After you've tried out these examples, you should feel quite confident in your ability to find an appropriate Puppet module, add it to your `Puppetfile`, and deploy it with `r10k`.

Using puppetlabs/apache

Here's an example manifest which uses the `apache` module to create a simple virtual host serving an image file (`module_apache.pp`):

```
include apache

apache::vhost { 'localhost':
  port          => '81',
  docroot       => '/var/www/cat-pictures',
  docroot_owner => 'www-data',
  docroot_group => 'www-data',
}

file { '/var/www/cat-pictures/index.html':
  content => "<img  src='https://raw.githubusercontent.com/fenago/puppet-
course/master/quickstart/md/images/8880_07_02.jpg'>",
  owner   => 'www-data',
  group   => 'www-data',
}
```

Follow these steps to apply the manifest:

1. If you've previously followed the steps in the [*Using r10k*] section, the required module will already be installed. If not, run the following commands to install it:

   ```
   cd /etc/puppetlabs/code/environments/pbg
   sudo r10k puppetfile install
   ```

2. Run the following command to apply the manifest:

   ```
   puppet apply --environment=pbg /examples/module_apache.pp
   ```

Verify that virtualhost is created: `ls -ltr /etc/apache2/sites-enabled/`

Apache Port

Run following command to start apache server:

```
apachectl start
```

Note: You will get and error because port 80 is already in use for the lab environment. We will change the port in the next step.

Change Apache Port

Let's change the default Apache port. Open `/etc/apache2/ports.conf` in vscode and update `80` with port `81`. Save and close the file.

```
apachectl start
```

3. To test the new website, browse to port `81` `http://localhost:81/`

You should see a picture of a happy cat:

Let's go through the manifest and see how it works in detail.

1. It starts with the `include` declaration which actually installs Apache on the server (`module_apache.pp`):

   ```
   include apache
   ```

2. There are many parameters you could set for the `apache` class, but in this example, we only need to set one, and as with the other examples, we set it using Hiera data in the example Hiera file:

   ```
   apache::default_vhost: false
   ```

```
This disables the default **Apache 2 Test Page** virtual host.
```

3. Next comes a resource declaration for an `apache::vhost` resource, which creates an Apache virtual host or website.

```
apache::vhost { 'localhost':
  port          => '81',
  docroot       => '/var/www/cat-pictures',
  docroot_owner => 'www-data',
  docroot_group => 'www-data',
}
```

4. Finally, we create an `index.html` file to add some content to the website, in this case, an image of a happy cat.

```
file { '/var/www/cat-pictures/index.html':
  content => "<img
    src='https://raw.githubusercontent.com/fenago/puppet-
course/master/quickstart/md/images/8880_07_02.jpg'>",
  owner   => 'www-data',
  group   => 'www-data',
}
```

Using puppet/archive

While installing software from packages is a common task, you'll also occasionally need to install software from archive files, such as a tarball (a `.tar.gz` file) or ZIP file. The `puppet/archive` module is a great help for this, as it provides an easy way to download archive files from the Internet, and it can also unpack them for you.

In the following example, we'll use the `puppet/archive` module to download and unpack the latest version of the popular WordPress blogging software. Follow these steps to apply the manifest:

1. If you've previously followed the steps in the [*Using r10k*] section, the required module will already be installed. If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg
sudo r10k puppetfile install
```

2. Run the following command to apply the manifest:

```
puppet apply --environment=pbg /examples/module_archive.pp

Notice: Compiled catalog for ubuntu-xenial in environment production in 2.50
seconds
Notice: /Stage[main]/Main/Archive[/tmp/wordpress.tar.gz]/ensure: download
archive from https://wordpress.org/latest.tar.gz to /tmp/wordpress.tar.gz and
extracted in /var/www with cleanup
```

Unlike the previous modules in this lab, there's nothing to install with `archive`, so we don't need to include the class itself. All you need to do is declare an `archive` resource. Let's look at the example in detail to see how it works (`module_archive.pp`):

```
archive { '/tmp/wordpress.tar.gz':
  ensure       => present,
  extract      => true,
  extract_path => '/var/www',
```

```
  source       => 'https://wordpress.org/latest.tar.gz',
  creates      => '/var/www/wordpress',
  cleanup      => true,
}
```

Note

Once the file has been deleted by `cleanup`, Puppet won't redownload the archive file `/tmp/wordpress.tar.gz` the next time you apply the manifest, even though it has `ensure => present`. The `creates` clause tells Puppet that the archive has already been downloaded and extracted.

Exploring the standard library

One of the oldest-established Puppet Forge modules is `puppetlabs/stdlib`, the official Puppet standard library. We looked at this briefly earlier in the lab when we used it as an example of installing a module with `r10k`, but let's look more closely now and see what the standard library provides and where you might use it.

Before trying the examples in this section, make sure the `stdlib` module is installed by following these steps: If you've previously followed the steps in the [*Using r10k*] section, the required module will already be installed. If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg
sudo r10k puppetfile install
```

Safely installing packages with ensure_packages

As you know, you can install a package using the `package` resource, like this (`package.pp`):

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

But what happens if you also install the same package in another class in a different part of your manifest? Puppet will refuse to run, with an error like this:

```
Error: Evaluation Error: Error while evaluating a Resource Statement, Duplicate
declaration: Package[cowsay] is already declared in file /examples/package.pp:1;
cannot redeclare at /examples/package.pp:4 at /examples/package.pp:4:1 on node ubuntu-
xenial
```

If both of your classes really require the package, then you have a problem. You could create a class which simply declares the package, and then include that in both classes, but that is a lot of overhead for a single package. Worse, if the duplicate declaration is in a third-party module, it may not be possible, or advisable, to change that code.

What we need is a way to declare a package which will not cause a conflict if that package is also declared somewhere else. The standard library provides this facility in the `ensure_packages()` function. Call `ensure_packages()` with an array of package names, and they will be installed if they are not already declared elsewhere (`package_ensure.pp`):

```
ensure_packages(['cowsay'])
```

To apply this example, run the following command:

```
puppet apply --environment=pbg /examples/package_ensure.pp
```

You can try all the remaining examples in this lab in the same way. Make sure you supply the `--environment=pbg` switch to `puppet apply`, as the necessary modules are only installed in the `pbg` environment.

If you need to pass additional attributes to the `package` resource, you can supply them in a hash as the second argument to `ensure_packages()`, like this (`package_ensure_params.pp`):

```
ensure_packages(['cowsay'],
  {
    'ensure' => 'latest',
  }
)
```

Why is this better than using the `package` resource directly? When you declare the same `package` resource in more than one place, Puppet will give an error message and refuse to run. If the package is declared by `ensure_packages()`, however, Puppet will run successfully.

Since it provides a safe way to install packages without resource conflicts, you should always use `ensure_packages()` instead of the built-in `package` resource. It is certainly essential if you're writing modules for public release, but I recommend you use it in all your code. We'll use it to manage packages throughout the rest of this course.

Modifying files in place with file_line

Often, when managing configuration with Puppet, we would like to change or add a particular line to a file, without incurring the overhead of managing the whole file with Puppet. Sometimes it may not be possible to manage the whole file in any case, as another Puppet class or another application may be managing it. We could write an `exec` resource to modify the file for us, but the standard library provides a resource type for exactly this purpose: `file_line`.

Here's an example of using the `file_line` resource to add a single line to a system config file (`file_line.pp`):

```
file_line { 'set ulimits':
  path => '/etc/security/limits.conf',
  line => 'www-data         -     nofile          32768',
}
```

If there is a possibility that some other Puppet class or application may need to modify the target file, use `file_line` instead of managing the file directly. This ensures that your class won't conflict with any other attempts to control the file.

You can also use `file_line` to find and modify an existing line, using the `match` attribute (`file_line_match.pp`):

```
file_line { 'adjust ulimits':
  path  => '/etc/security/limits.conf',
  line  => 'www-data         -     nofile          9999',
  match => '^www-data .* nofile',
}
```

The value of `match` is a regular expression, and if Puppet finds a line in the file which matches this expression, it will replace it with the value of `line`. (If you need to potentially change multiple lines, set the `multiple` attribute to `true` or Puppet will complain when more than one line matches the expression.)

You can also use `file_line` to delete a line in a file if it is present (`file_line_absent.pp`):

```
file_line { 'remove dash from valid shells':
  ensure            => absent,
  path              => '/etc/shells',
  match             => '^/bin/dash',
  match_for_absence => true,
}
```

Note that when using `ensure => absent`, you also need to set the `match_for_absence` attribute to `true` if you want Puppet to actually delete matching lines.

Introducing some other useful functions

The `grep()` function will search an array for a regular expression and return all matching elements (`grep.pp`):

```
$values = ['foo', 'bar', 'baz']
notice(grep($values, 'ba.*'))

# Result: ['bar', 'baz']
```

The `member()` and `has_key()` functions return `true` if a given value is in the specified array or hash, respectively (`member_has_key.pp`):

```
$values = [
  'foo',
  'bar',
  'baz',
]
notice(member($values, 'foo'))

# Result: true

$valuehash = {
  'a' => 1,
  'b' => 2,
  'c' => 3,
}
notice(has_key($valuehash, 'b'))

# Result: true
```

The `empty()` function returns `true` if its argument is an empty string, array, or hash (`empty.pp`):

```
notice(empty(''))

# Result: true

notice(empty([]))
```

```
# Result: true

notice(empty({}))

# Result: true
```

The `join()` function joins together the elements of a supplied array into a string, using a given separator character or string (`join.pp`):

```
$values = ['1', '2', '3']
notice(join($values, '... '))

# Result: '1... 2... 3'
```

The `pick()` function is a neat way to provide a default value when a variable happens to be empty. It takes any number of arguments and returns the first argument which is not undefined or empty (`pick.pp`):

```
$remote_host = ''
notice(pick($remote_host, 'localhost'))

# Result: 'localhost'
```

Sometimes you need to parse structured data in your Puppet code which comes from an outside source. If that data is in YAML format, you can use the `loadyaml()` function to read and parse it into a native Puppet data structure (`loadyaml.pp`):

```
$db_config = loadyaml('/examples/files/database.yml')
notice($db_config['development']['database'])

# Result: 'dev_db'
```

The `dirname()` function is very useful if you have a string path to a file or directory and you want to reference its parent directory, for example to declare it as a Puppet resource (`dirname.pp`):

```
$file = '/var/www/vhosts/mysite'
notice(dirname($file))

# Result: '/var/www/vhosts'
```

The pry debugger

When a Puppet manifest doesn't do quite what you expect, troubleshooting the problem can be difficult. Printing out the values of variables and data structures with `notice()` can help as can running `puppet apply -d` to see detailed debug output, but if all else fails, you can use the standard library's `pry()` method to enter an interactive debugger session (`pry.pp`):

```
pry()
```

With the `pry` gem installed in Puppet's context, you can call `pry()` at any point in your code. When you apply the manifest, Puppet will start an interactive Pry shell at the point where the `pry()` function is called. You can then

run the `catalog` command to inspect Puppet's catalog, which contains all the resources currently declared in your manifest:

```
puppet apply --environment=pbg /examples/pry_install.pp
puppet apply --environment=pbg /examples/pry.pp
...
[1] pry(#<Puppet::Parser::Scope>)> catalog
=> #<Puppet::Resource::Catalog:0x00000001bbcf78
...
 @resource_table={["Stage", "main"]=>Stage[main]{}, ["Class",
"Settings"]=>Class[Settings]{}, ["Class", "main"]=>Class[main]{}},
 @resources=[["Stage", "main"], ["Class", "Settings"], ["Class", "main"]],
...
```

Once you've finished inspecting the catalog, type `exit` to quit the debugger and continue applying your Puppet manifest.

Writing your own modules

As we've seen, a Puppet module is a way of grouping together a set of related code and resources that performs some particular task, like managing the Apache web server or dealing with archive files. But how do you actually create a module? In this section, we'll develop a module of our own to manage the NTP service, familiar to most system administrators as the easiest way to keep server clocks synchronized with the Internet time standard. (Of course, it's not necessary to write your own module for this because a perfectly good one exists on Puppet Forge. But we'll do so anyway, for learning purposes.)

Creating a repo for your module

If we're going to use our new module alongside others that we've installed from Puppet Forge, then we should create a new Git repo just for our module. Then we can add its details to our Puppetfile and have `r10k` install it for us.

If you've already worked through [Lab 3], [*Managing your Puppet code with Git], you'll have created a GitHub account. If not, go to that lab and follow the instructions in the [Creating a GitHub account and project*] section before continuing:

1. Log in to your GitHub account and click on the **Start a project** button.
2. On the **Create a new repository** screen, enter a suitable name for your repo (I'm using `pbg_ntp` for the Puppet Beginner's Guide's NTP module).
3. Check the **Initialize this repository with a README** box.
4. Click on **Create repository**.
5. GitHub will take you to the project page for the new repository. Click on the **Clone or download** button.
6. On your own computer, or wherever you develop Puppet code, run the following command to clone the new repo (use the GitHub URL you copied in the previous step instead of this one):

   ```
   git clone https://github.com/fenago/pbg_ntp.git
   ```

When the clone operation completes successfully, you're ready to get started with creating your new module.

Writing the module code

As you'll see if you look inside the Puppet Forge modules you've already installed, modules have a standard directory structure. This is so that Puppet can automatically find the manifest files, templates, and other components within the module. Although complex modules have many subdirectories, the only ones we will be concerned with in this example are manifests and files. In this section, we'll create the necessary subdirectories, write the code to manage NTP, and add a config file which the code will install.

Note

All the code and files for this module are available in the GitHub repo at the following URL:

https://github.com/fenago/pbg_ntp

1. Run the following commands to create the `manifests` and `files` subdirectories:

```
cd pbg_ntp
mkdir manifests
mkdir files
```

2. Create the file `manifests/init.pp` with the following contents:

```
# Manage NTP
class pbg_ntp {
  ensure_packages(['ntp'])

  file { '/etc/ntp.conf':
    source  => 'puppet:///modules/pbg_ntp/ntp.conf',
    notify  => Service['ntp'],
    require => Package['ntp'],
  }

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

3. Create the file `files/ntp.conf` with the following contents:

```
driftfile /var/lib/ntp/ntp.drift

pool 0.ubuntu.pool.ntp.org iburst
pool 1.ubuntu.pool.ntp.org iburst
pool 2.ubuntu.pool.ntp.org iburst
pool 3.ubuntu.pool.ntp.org iburst
pool ntp.ubuntu.com

restrict -4 default kod notrap nomodify nopeer noquery limited
restrict -6 default kod notrap nomodify nopeer noquery limited
restrict 127.0.0.1
restrict ::1
```

4. Run the following commands to add, commit, and push your changes to GitHub (you'll need to enter your GitHub username and password if you're not using an SSH key):

```
git add manifests/ files/
git commit -m 'Add module manifest and config file'
[master f45dc50] Add module manifest and config file
 2 files changed, 29 insertions(+)
 create mode 100644 files/ntp.conf
 create mode 100644 manifests/init.pp
git push origin master
```

Notice that the `source` attribute for the `ntp.conf` file looks like the following:

```
puppet:///modules/pbg_ntp/ntp.conf
```

We haven't seen this kind of file source before, and it's generally only used within module code. The `puppet://` prefix indicates that the file comes from within the Puppet repo, and the path `/modules/pbg_ntp/` tells Puppet to look within the `pbg_ntp` module for it. Although the `ntp.conf` file is actually in the directory `modules/pbg_ntp/files/`, we don't need to specify the `files` part: that's assumed, because this is a `file` resource. (It's not just you: this confuses everybody).

Rather than installing the `ntp` package via a `package` resource, we use `ensure_packages()` from the standard library, as described earlier in this lab.

Creating and validating the module metadata

Every Puppet module should have a file in its top-level directory named `metadata.json`, which contains helpful information about the module that can be used by module management tools, including Puppet Forge.

Create the file `metadata.json` with the following contents (use your own name and GitHub URLs):

```
{
  "name": "pbg_ntp",
  "version": "0.1.1",
  "author": "John Arundel",
  "summary": "Example module to manage NTP",
  "license": "Apache-2.0",
  "source": "https://github.com/fenago/pbg_ntp.git",
  "project_page": "https://github.com/fenago/pbg_ntp",
  "tags": ["ntp"],
  "dependencies": [
    {"name":"puppetlabs/stdlib",
      "version_requirement":">= 4.17.0 < 5.0.0"}
  ],
  "operatingsystem_support": [
    {
      "operatingsystem": "Ubuntu",
      "operatingsystemrelease": [ "16.04" ]
    }
  ]
}
```

Most of these are fairly self-explanatory. `tags` is an array of strings which will help people find your module if it is listed on Puppet Forge, and it's usual to tag your module with the name of the software or service it manages (in this case, `ntp`).

If your module relies on other Puppet modules, which is very likely (for example, this module relies on `puppetlabs/stdlib` for the `ensure_packages()` function) you use the `dependencies` metadata to record this. You should list each module used by your module along with the earliest and latest versions of that module which will work with your module. (If the currently-released version works, specify the next major release as the latest version. For example, if your module works with `stdlib` version 4.17.0 and that's the latest version available, specify 5.0.0 as the highest compatible version.)

Finally, the `operatingsystem_support` metadata lets you specify which operating systems and versions your module works with. This is very helpful for people searching for a Puppet module which will work with their operating system. If you know your module works with Ubuntu 16.04, as the example module does, you can list that in the `operatingsystem_support` section. The more operating systems your module can support, the better, so if possible, test your module on other operating systems and list them in the metadata once you know they work.

Note

For full details on module metadata and how to use it, see the Puppet documentation:

https://docs.puppet.com/puppet/latest/reference/modules_metadata.html

It's important to get the metadata for your module right, and there's a little tool that can help you with this, called `metadata-json-lint`.

1. Run the following commands to install `metadata-json-lint` and check your metadata:

   ```
   sudo gem install metadata-json-lint
   metadata-json-lint metadata.json
   ```

2. If `metadata-json-lint` produces no output, your metadata is valid and you can go on to the next steps. If you see error messages, fix the problem before continuing.

3. Run the following commands to add, commit, and push your metadata file to GitHub:

   ```
   git add metadata.json
   git commit -m 'Add metadata.json'
   git push origin master
   ```

Tagging your module

Just like when you use third-party Puppet Forge modules, it's important to be able to specify in your Puppetfile the exact version of your module to be installed. You can do this by using Git tags to attach a version tag to a specific commit in your module repo. As you develop the module further and make new releases, you can add a new tag for each release.

For the first release of your module, which according to the metadata is version 0.1.1, run the following commands to create and push the release tag:

```
git tag -a 0.1.1 -m 'Release 0.1.1'
git push origin 0.1.1
```

Installing your module

We can use `r10k` to install our new module, just as we did with the Puppet Forge modules, with one small difference. Since our module isn't on the Puppet Forge (yet), just specifying the name of the module in our Puppetfile isn't enough; we need to supply the Git URL so that `r10k` can clone the module from GitHub.

1. Add the following `mod` statement to your Puppetfile (using your GitHub URL instead of mine):

```
mod 'pbg_ntp',
  :git => 'https://github.com/fenago/pbg_ntp.git',
  :tag => '0.1.1'
```

2. Because the module also requires `puppetlabs/stdlib`, add this `mod` statement too:

```
mod 'puppetlabs/stdlib', '4.17.0'
```

3. Now install the module in the normal way with `r10k`:

```
sudo r10k puppetfile install --verbose
```

`r10k` can install a module from any Git repo you have access to; all you have to do is add the `:git` and `:tag` parameters to the `mod` statement in your Puppetfile.

Applying your module

Now that you've created, uploaded, and installed your module, we can use it in a manifest:

```
puppet apply --environment=pbg -e 'include pbg_ntp'
```

If you're using the lab environment or a recent version of Ubuntu, your server will most likely be running NTP already, so the only change you'll see Puppet apply will be the `ntp.conf` file. Nonetheless, it confirms that your module works.

Summary

In this lab, we've gained an understanding of Puppet modules, including an introduction to the Puppet Forge module repository. We've seen how to search for the modules we need and how to evaluate the results, including **Puppet Approved** and **Puppet Supported** modules, operating system support, and download count.

We've looked at using the `r10k` tool , covered the use of `ensure_packages()` to avoid package conflicts between modules, the `file_line` resource. Finally, we've looked at some of the features of more sophisticated modules and discussed uploading modules to the Puppet Forge.