Lab 2. Puppet Language and Style

We will cover the following recipes in this lab:

- Using in-line templates
- Iterating over multiple terms
- Writing powerful conditional statements
- Using regular expressions in `if` statements
- Using selectors and case statements
- Using the `in` operator
- Using regular expression substitutions
- Puppet 5 changes
- Puppet 4/5 Changes

Using inline templates

Templates are a powerful way of using **Embedded Puppet** (**EPP**) or Embedded Ruby (ERB) to help build config files dynamically. You can also use `EPP` or `ERB` syntax directly without having to use a separate file by calling the `inline_epp` or `inline_template` function. `EPP` and `ERB` allow you to use conditional logic, iterate over arrays, and include variables. `EPP` is the replacement of `ERB`; `EPP` uses native Puppet language. `ERB` uses Ruby language. `ERB` allows for using native Ruby functions which may not be available in `EPP`, so unless you need something Ruby specific, it is better to go with the native `EPP` templates. In the following example, we'll use a Ruby construct, so we'll use an `ERB` inline template.

How to do it...

Here's an example of how to use inline_template.

Pass your Ruby code to `inline_template` within the Puppet manifest, as follows:

```
cron { 'chkrootkit':
  command => '/usr/sbin/chkrootkit > /var/log/chkrootkit.log 2>&amp;1',
  hour    => inline_template('<%= @hostname.sum % 24 %>'),
  minute  => '00',
}
```

How it works...

Anything inside the string passed to `inline_template` is executed as if it were an ERB template. That is, anything inside the `<%= and %>` delimiters will be executed as Ruby code, and the rest will be treated as a string.

In this example, we use `inline_template` to compute a different hour for this `cron` resource (a scheduled job) for each machine, so that the same job does not run at the same time on all machines. For more on this technique, see the [*Efficiently distributing cron jobs] recipe in [Lab 5], [Users and Virtual Resources*].

There's more...

In `ERB` code, whether inside a template file or an `inline_template` string, you can access your Puppet variables directly by name using an `@` prefix, if they are in the current scope or the top scope (facts):

```
<%= @fqdn %>
```

To reference variables in another scope, use `scope.lookupvar`, as follows:

```
<%= "The value of something from otherclass is " +
scope.lookupvar('otherclass::something') %>
```

You should use inline templates sparingly. If you really need to use some complicated logic in your manifest, consider using a custom function instead (see the [*Creating custom functions] recipe in [Lab 8], [External Tools and the Puppet Ecosystem*]). As we'll see later, EPP templates use global scope for their variables; you always refer to variables with their full scope.

See also

- The [*Using ERB templates] recipe in [Lab 4], [Working with Files and Packages*]
- The [*Using array iteration in templates] recipe in [Lab 4], [Working with Files and Packages*]

Iterating over multiple items

Arrays are a powerful feature in Puppet; wherever you want to perform the same operation on a list of things, an array may be able to help. You can create an array just by putting its content in square brackets:

```
$lunch = [ 'franks', 'beans', 'mustard' ]
```

How to do it...

Here's a common example of how arrays are used:

1. Add the following code to your manifest:

```
$packages = [
  'ruby1.8-dev', 'ruby1.8',
  'ri1.8', 'rdoc1.8',
  'irb1.8', 'libreadline-ruby1.8',
  'libruby1.8', 'libopenssl-ruby'
]
package { $packages: ensure => installed }
```

2. Run `Puppet` and note that each package should now be installed.

How it works...

Where Puppet encounters an array as the name of a resource, it creates a resource for each element in the array. In the example, a new package resource is created for each of the packages in the `$packages` array, with the same `ensure => installed` parameters. This is a very compact way to instantiate many similar resources.

There's more...

Although arrays will take you a long way with Puppet, it's also useful to know about an even more flexible data structure: the hash.

Using hashes

A hash is like an array, but each of the elements can be stored and looked up by name (referred to as the key); for example, `hash.pp` :

```
$interface = {
 'name' => 'eth0',
 'ip'   => '192.168.0.1',
```

```
 'mac'  => '52:54:00:4a:60:07'
}
notify { "(${interface['ip']}) at ${interface['mac']} on ${interface['name']}": }
```

When we run `Puppet` on this, we see the following notice in the output:

```
t@fenago:~/.puppet/manifests$ puppet apply hash.pp
Notice: Compiled catalog for fenago.example.com in environment production in 0.04
seconds
Notice: (192.168.0.1) at 52:54:00:4a:60:07 on eth0
```

Hash values can be anything that you can assign to variables, strings, function calls, expressions, and even other hashes or arrays. Hashes are useful to store a bunch of information about a particular thing because by accessing each element of the hash using a key, we can quickly find the information we are looking for.

Creating arrays with the split function

You can declare literal arrays using square brackets, as follows:

```
define lunchprint() {
 notify { "Lunch included ${name}":}
}
$lunch = ['egg', 'beans', 'chips']
lunchprint { $lunch: }
```

Now, when we run Puppet on the preceding code, we see the following notice messages in the output:

```
t@fenago:~$ puppet apply lunchprint.pp
Notice: Compiled catalog for fenago.strangled.net in environment production in 0.02
seconds
Notice: Lunch included egg
Notice: Lunch included beans
Notice: Lunch included chips
Notice: Applied catalog in 0.04 seconds
```

However, Puppet can also create arrays for you from strings, using the `split` function, as follows:

```
$menu = 'egg beans chips'
$items = split($menu, ' ')
lunchprint { $items: }
```

Running `puppet apply` against this new manifest, we see the same messages in the output:

```
t@fenago:~$ puppet apply lunchprint2.pp
Notice: Compiled catalog for fenago.strangled.net in environment production in 0.02
seconds
Notice: Lunch included egg
Notice: Lunch included beans
Notice: Lunch included chips
Notice: Applied catalog in 0.21 seconds
```

Note

The split takes two arguments: the first argument is the string to be split. The second argument is the character to split on. In this example, it's a single space. As Puppet works its way through the string, when it encounters a space, it will interpret it as the end of one item and the beginning of the next. So, given the string egg, beans, and chips, this will be split into three items.

The character to split on can be any character or string:

```
$menu = 'egg and beans and chips' $items = split($menu, ' and ')
```

The character can also be a regular expression, for example, a set of alternatives separated by a `|` (pipe) character:

```
$lunch = 'egg:beans,chips'
$items = split($lunch, ':|,')
```

Writing powerful conditional statements

Puppet's `if` statement allows you to change the manifest behavior based on the value of a variable or an expression. With it, you can apply different resources or parameter values depending on certain facts about the node; for example, the operating system or the memory size.

You can also set variables within the manifest, which can change the behavior of included classes. For example, nodes in data center A might need to use different DNS servers than nodes in data center B, or you might need to include one set of classes for an Ubuntu system, and a different set for other systems.

How to do it...

Here's an example of a useful conditional statement. Add the following code to your manifest:

```
if $::timezone == 'UTC' {
  notify { 'Universal Time Coordinated':}
} else {
  notify { "$::timezone is not UTC": }
}
```

How it works...

Puppet treats whatever follows an if keyword as an expression and evaluates it. If the expression evaluates to true, Puppet will execute the code within the curly braces.

Optionally, you can add an else branch, which will be executed if the expression evaluates to false.

There's more...

Lets take a look at some more tips on using `if` statements.

elsif branches

You can add further tests using the `elsif` keyword, as follows:

```
if $::timezone == 'UTC' {
notify { 'Universal Time Coordinated': }
} elsif $::timezone == 'GMT' {
notify { 'Greenwich Mean Time': }
} else {
```

```
notify { "$::timezone is not UTC": }
}
```

Comparisons

You can check whether two values are equal using the `==` syntax, as in our example:

```
if $::timezone == 'UTC' {
  ...
}
```

Alternatively, you can check whether they are not equal using `!=`:

```
if $::timezone != 'UTC' {
  ...
}
```

You can also compare numeric values using `< and >`:

```
if $::uptime_days > 365 {
  notify { 'Time to upgrade your kernel!': }
}
```

To test whether a value is greater (or less) than or equal to another value, use `<=` or `>=`:

```
if $::mtu_eth0 <= 1500 {
  notify {"Not Jumbo Frames": }
}
```

Combining expressions

You can put together the kind of simple expressions described previously into more complex logical expressions, using `and`, `or`, and `not`:

```
if ($::uptime_days > 365) and ($::kernel == 'Linux') {
  …
}
if ($role == 'webserver') and ( ($datacenter == 'A') or ($datacenter == 'B') ) {
  …
}
```

See also

- [*The][Using the in operator*] recipe in this lab
- [*The][Using selectors and case statements*] recipe in this lab

Using regular expressions in if statements

Another kind of expression you can test in `if` statements and other conditionals is the regular expression. A regular expression is a powerful way to compare strings using pattern matching.

How to do it...

This is one example of using a regular expression in a conditional statement. Add the following to your manifest:

```
if $::architecture =~ /64/ {
  notify { '64Bit OS Installed': }
} else {
  notify { 'Upgrade to 64Bit': }
  fail('Not 64 Bit')
}
```

How it works...

Puppet treats the text supplied between the forward slashes as a regular expression, specifying the text to be matched. If the match succeeds, the if expression will be true and so the code between the first set of curly braces will be executed. In this example, we used a regular expression because different distributions have different ideas on what to call 64 bit; some use `amd64`, while others use `x86_64`. The only thing we can count on is the presence of the number 64 within the fact. Some facts that have version numbers in them are treated as strings to Puppet. For instance, `$::facterversion`. On my test system, this is 3.9.3, but when I try to compare that with 3, Puppet fails to make the following comparison:

```
if $::facterversion > 3 {
 notify {"Facter version 3": }
}
```

Which produces the following output when run with `puppet apply`:

```
t@fenago:~$ puppet apply version.pp
Error: Evaluation Error: Comparison of: String > Integer, is not possible. Caused by
'A String is not comparable to a non String'. at /home/vagrant/version.pp:1:21 on node
fenago.example.com
```

We could make the comparison with `=~` but that would match a `3` in any position in the `version` string. Puppet provides a function to compare versions, `versioncmp`, as shown in this example:

```
if versioncmp($::facterversion,'3') > 0 {
  notify {"Facter version 3": }
}
```

Which now produces the desired result:

```
t@fenago:~$ puppet apply version2.pp
Notice: Compiled catalog for fenago.strangled.net in environment production in 0.01
seconds
Notice: Facter version 3
```

The `versioncmp` function returns `-1` if the first parameter is a lower version than the second, `0` if the two parameters are equal, or `1` if the second parameter is lower than the first.

If you wanted instead to do something if the text does not match, use `!~` rather than `=~`:

```
if $::kernel !~ /Linux/ {
  notify { 'Not Linux, could be Windows, MacOS X, AIX, or ?': }
}
```

There's more...

Regular expressions are very powerful, but can be difficult to understand and debug. If you find yourself using a regular expression so complex that you can't see at a glance what it does, think about simplifying your design to make it easier. However, one particularly useful feature of regular expressions is their ability to capture patterns.

Capturing patterns

You can not only match text using a regular expression, but also capture the matched text and store it in a variable:

```
$input = 'Puppet is better than manual configuration'
if $input =~ /(.*) is better than (.*)/ {
  notify { "You said '${0}'. Looks like you're comparing ${1} to ${2}!": }
}
```

The preceding code produces this output:

```
Notice: You said 'Puppet is better than manual configuration'. Looks like you're
comparing Puppet to manual configuration!
```

The `$0` variable stores the whole matched text (assuming the overall match succeeded). If you put brackets around any part of the regular expression, it creates a group, and any matched groups will also be stored in variables. The first matched group will be `$1`, the second `$2`, and so on, as shown in the preceding example.

Regular expression syntax

Puppet's regular expression syntax is the same as Ruby's, so resources that explain Ruby's regular expression syntax will also help you with Puppet. You can find a good introduction to Ruby's regular expression syntax at this website: http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm.

See also

- Refer to the [*Using regular expression substitutions*] recipe in this lab

Using selectors and case statements

Although you could write any conditional statement using `if`, Puppet provides a couple of extra forms to help you express conditionals more easily: the selector and the case statement.

How to do it...

Here are some examples of selector and case statements:

1. Add the following code to your manifest:

```
$systemtype = $::operatingsystem ? {
  'Ubuntu' => 'debianlike',
  'Debian' => 'debianlike',
  'RedHat' => 'redhatlike',
  'Fedora' => 'redhatlike',
  'CentOS' => 'redhatlike',
  default  => 'unknown',
}
notify { "You have a ${systemtype} system": }
```

2. Add the following code to your manifest:

```
class debianlike {
  notify { 'Special manifest for Debian-like systems': }
}
class redhatlike {
  notify { 'Special manifest for RedHat-like systems': }
}
case $::operatingsystem {
  'Ubuntu', 'Debian': { include debianlike },
  'RedHat', 'Fedora', 'CentOS', 'Springdale': { include redhatlike }
  default: { notify { "I don't know what kind of system you have!": } }
}
```

How it works...

Our example demonstrates both the selector and the case statement, so let's see in detail how each of them works.

Selector

In the first example, we used a selector (the `?` operator) to choose a value for the `$systemtype` variable depending on the value of `$::operatingsystem`. This is similar to the ternary operator in C or Ruby, but instead of choosing between two possible values, you can have as many values as you like.

Puppet will compare the value of `$::operatingsystem` to each of the possible values we have supplied in Ubuntu, Debian, and so on. These values could be regular expressions (for example, for a partial string match or to use wildcards), but in our case, we have just used literal strings.

As soon as it finds a match, the selector expression returns whatever value is associated with the matching string. If the value of `$::operatingsystem` is fedora, for example, the selector expression will return the `redhatlike` string and this will be assigned to the `$systemtype` variable.

Case statement

Unlike selectors, the case statement does not return a value. Case statements come in handy when you want to execute different code depending on the value of an expression. In our second example, we used the case statement to include either the `debianlike` or `redhatlike` class, depending on the value of `$::operatingsystem`.

Again, Puppet compares the value of `$::operatingsystem` to a list of potential matches. These could be regular expressions or strings, or as in our example, comma-separated lists of strings. When it finds a match, the associated code between curly braces is executed. So, if the value of `$::operatingsystem` is Ubuntu, then the code including `debianlike` will be executed.

There's more...

Once you've got a grip on the basic use of selectors and case statements, you may find the following tips useful.

Regular expressions

As with `if` statements, you can use regular expressions with selectors and case statements, and you can also capture the values of the matched groups and refer to them using `$1`, `$2`, and so on:

```
case $::lsbdistdescription {
  /Ubuntu (.+)/: {
    notify { "You have Ubuntu version ${1}": }
  }
  /CentOS (.+)/: {
```

```
    notify { "You have CentOS version ${1}": }
  }
  default: {}
}
```

Defaults

Both selectors and case statements let you specify a default value, which is chosen if none of the other options match (the style guide suggests you always have a default clause defined):

```
$lunch = 'Filet mignon.' $lunchtype = $lunch ? {
  /fries/ => 'unhealthy',
  /salad/ => 'healthy',
  default => 'unknown',
}
notify { "Your lunch was ${lunchtype}": }
```

The output is as follows:

```
t@fenago:~$ puppet apply lunchtype.pp
Notice: Compiled catalog for fenago.strangled.net in environment production in 0.01
seconds
Notice: Your lunch was unknown
```

When the default action dosen't occur, use the `fail()` function to halt the Puppet run.

Using the in operator

The `in` operator tests whether one string contains another string. Here's an example:

```
if 'spring' in 'springfield'
```

The preceding expression is true if the spring string is a substring of `springfield`, which it is. The `in` operator can also test for membership of arrays as follows:

```
if $crewmember in ['Frank', 'Dave', 'HAL' ]
```

When `in` is used with a hash, it tests whether the string is a key of the hash:

```
$ifaces = {
  'lo'   => '127.0.0.1',
  'eth0' => '192.168.0.1'
}
if 'eth0' in $ifaces {
  notify { "eth0 has address ${ifaces['eth0']}": }
}
```

How to do it...

The following steps will show you how to use the `in` operator:

1. Add the following code to your manifest:

```
if $::operatingsystem in [ 'Ubuntu', 'Debian' ] {
  notify { 'Debian-type operating system detected': }
} elsif $::operatingsystem in [ 'RedHat', 'Fedora', 'SuSE', 'CentOS' ] {
  notify { 'RedHat-type operating system detected': }
} else {
  notify { 'Some other operating system detected': }
}
```

2. Run Puppet:

```
t@fenago:~$ puppet apply in.pp
Notice: Compiled catalog for fenago.example.com in environment production in 0.01
seconds
Notice: RedHat-type operating system detected
```

There's more...

The value of an in expression is Boolean (true or false) so you can assign it to a variable:

```
$debianlike = $::operatingsystem in [ 'Debian', 'Ubuntu' ]
if $debianlike {
  notify { 'You are in a maze of twisty little packages, all alike': }
}
```

Using regular expression substitutions

Puppet's `regsubst` function provides an easy way to manipulate text, search and replace expressions within strings, or extract patterns from strings. We often need to do this with data obtained from a fact, for example, or from external programs.

In this example, we'll see how to use `regsubst` to extract the first three octets of an IPv4 address (the network part, assuming it's a /24 class C address).

How to do it...

Follow these steps to build the example:

1. Add the following code to your manifest:

```
$class_c = regsubst($::ipaddress, '(.*)\..*', '\1.0')
notify { "The network part of ${::ipaddress} is ${class_c}": }
```

2. Run Puppet:

```
t@fenago:~$ puppet apply regsubst.pp
Notice: Compiled catalog for fenago.strangled.net in environment production in 0.02
seconds
Notice: The network part of 10.0.2.15 is 10.0.2.0
```

How it works...

The `regsubst` function takes at least three parameters: source, pattern, and replacement. In our example, we specified the source string as `$::ipaddress`, which, on this machine, is as follows:

```
10.0.2.15
```

We specify the pattern function as follows:

```
(.*)\..*
```

We specify the replacement function as follows:

```
\1.0
```

The pattern captures all of the string up to the last period (`\.`) in the `\1` variable. We then match on `.*`, which matches everything to the end of the string, so when we replace the string at the end with `\1.0`, we end up with only the network portion of the IP address, which evaluates to the following:

```
10.0.2.0
```

We could have got the same result in other ways, of course, including the following:

```
$class_c = regsubst($::ipaddress, '\.\d+$', '.0')
```

Here, we only match the last octet and replace it with `.0`, which achieves the same result without capturing.

There's more...

The pattern function can be any regular expression, using the same (Ruby) syntax as regular expressions in `if` statements.

See also

- [*The][Importing dynamic information] recipe in [Lab 3], [Writing Better Manifests*]
- [*The Getting information about the environment] recipe in [Lab 3], [Writing Better Manifests*]
- [*The Using regular expressions in if statements*] recipe in this lab

Puppet 5 changes

Prior to Puppet 4, Puppet 3 had a preview of the Puppet 4 language named the future parser. The future parser feature allowed you to preview the language changes that would be coming to Puppet 4 before upgrading. Most of these features were related to iterating on objects and have been carried forward to Puppet 5. In this section, we will cover the major changes in Puppet 5.

Using the call function

Puppet 5 adds a new function, `call`. This function is useful for calling a function by name using a variable. In the following example, we change the function we use depending on a variable:

```
if versioncmp($::puppetversion,'4.0') {
  $func = 'lookup'
 } else {
  $func = 'hiera'
 }
 $val = call($func,'important_setting')
 notify {"\$val = $val, \$func = $func": }
```

If the version of Puppet is lower than `4.0`, the `hiera` function will be called; if not, `lookup` will be used.