# Lab 2. Writing Better Manifests


In this lab, we will cover the following recipes:

- Using arrays of resources
- Using resource defaults
- Using defined types
- Using tags
- Using run stages
- Using roles and profiles
- Using data types in Puppet
- Passing parameters to classes
- Passing parameters from Hiera
- Writing reusable, cross-platform manifests
- Getting information about the environment
- Importing dynamic information
- Passing arguments to shell commands

## Introduction

Your Puppet manifests are the living documentation for your entire infrastructure. Keeping them tidy and well-organized is a great way to make it easier to maintain and understand. Puppet gives you a number of tools to do this:

- Arrays
- Defaults
- Defined types
- Dependencies
- Class parameters

We'll see how to use all of these and more. As you read through the lab, try out the examples and look through your own manifests to see where these features might help you simplify and improve your Puppet code.

## Using arrays of resources

Anything that you can do to a resource, you can do to an array of resources. Use this idea to refactor your manifests to make them shorter and clearer.

### How to do it...

Here are the steps to refactor using arrays of resources:

1. Identify a class in your manifest where you have several instances of the same kind of resource, for example, packages:

```
package { 'sudo' : ensure     => installed }
package { 'unzip' : ensure    => installed }
package { 'locate' : ensure   => installed }
package { 'lsof' : ensure     => installed }
package { 'cron' : ensure     => installed }
package { 'rubygems' : ensure => installed }
```

2. Group them together and replace them with a single `package` resource using an array:

```
$pkgs = [ 'cron',
          'locate',
          'lsof',
          'rubygems',
          'sudo',
          'unzip' ]
package { $pkgs:
  ensure => installed,
}
```

### How it works...

Most of Puppet's resource types can accept an array instead of a single name, and will create one instance for each of the elements in the array. All the parameters you provide for the resource (for example, `ensure => installed`) will be assigned to each of the new resource instances. This shorthand will only work when all the resources have the same attributes.

### There's more...

Using an array, you can add or change parameters for all the resources at once. For example, you could add a `hasrestart` option with the following modification:

```
package { $pkgs:
  ensure     => installed,
  hasrestart => true,
}
```

## Using resource defaults

Resource defaults allow you to specify the default attribute values for a resource. Resource defaults are valid for a given resource type and within the current scope. If you define a resource default in a class, then all resources of that type within the class will be given those defaults. In this example, we'll show you how to specify a resource default for the `File` type.

### How to do it...

To show you how to use resource defaults, we'll create an `apache` module. Within this module, we will specify that all file resources require the `httpd` package and the default `owner` and `group` are the `apache` user:

1. Create an `apache` module and create a resource default for the `File` type:

```
class apache {
  File {
    owner   => 'apache',
    group   => 'apache',
    mode    => '0644',
    require => Package['httpd']
  }
  package {'httpd': ensure => 'installed'}
}
```

2. Create `html` files within the `/var/www/html` directory:

```
$index = @(INDEX)
  <html>
    <body>
      <h1><a href='fenago.html'>Puppet! </a></h1>
    </body>
  </html>
  | INDEX
file {'/var/www/html/index.html':
  content => $index,
}
$fenago = @(FENAGO)
<html>
    <body>
      <h2>Fenago</h2>
    </body>
  </html>
  | FENAGO
file {'/var/www/html/fenago.html':
  content => $fenago
}
```

3. Apply the module to a node:

```
t@mylaptop ~ $ sudo /opt/puppetlabs/bin/puppet apply -e 'include apache' --modulepath
/home/thomas/.puppetlabs/etc/code/modules
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.54
seconds
Notice: /Stage[main]/Apache/File[/var/www/html/index.html]/ensure: defined content as
'{md5}05504e959eee487f44e9c0ddfa741829'
Notice: /Stage[main]/Apache/File[/var/www/html/fenago.html]/ensure: defined content as
'{md5}e83a49b87d91cf41ee30c0b755f3712e'
Notice: Applied catalog in 0.55 seconds
t@mylaptop ~ $ ls -l /var/www/html
total 8
-rw-r--r--. 1 apache apache 56 Feb 4 21:42 fenago.html
-rw-r--r--. 1 apache apache 86 Feb 4 21:42 index.html
```

### How it works...

The resource default we defined specifies the owner, group, and mode for all file resources within this class (also known as within this scope). We also specify that the `httpd` package is required before creating these files. This is useful since the package creates the `the /var/www/html` directory, into which we are going to place these files. Unless you specifically override a resource default, the value for an attribute will be taken from the default.

### There's more...

You can specify resource defaults for any resource type. You can also specify resource defaults in `site.pp`. I find it useful to specify the default action for the `Package` and `Service` resources, as follows:

```
Package { ensure => 'installed' }
Service {
  hasrestart => true,
```

```
  enable     => true,
  ensure     => true,
}
```

With these defaults, whenever you specify a package, the package will be installed. Whenever you specify a service, the service will be started and enabled to run at boot. These are the usual reasons you specify packages and services; most of the time these defaults will do what you prefer and your code will be cleaner. When you need to disable a service, simply override the defaults.

## Using defined types

In the previous example, we saw how to reduce redundant code by grouping identical resources into arrays. However, this technique is limited to resources where all the parameters are the same. When you have a set of resources that have some parameters in common, you need to use a defined type to group them together.

### How to do it...

The following steps will show you how to create a definition:

1. Create the following manifest:

```
define tmpfile() {
  file { "/tmp/${name}":
    content => "Hello, world\n",
  }
}
tmpfile { ['a', 'b', 'c']: }
```

2. Run `puppet apply`:

```
t@mylaptop ~ $ puppet apply tmpfile.pp
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.02
seconds
Notice: /Stage[main]/Main/Tmpfile[a]/File[/tmp/a]/ensure: defined content as
'{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: /Stage[main]/Main/Tmpfile[b]/File[/tmp/b]/ensure: defined content as
'{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: /Stage[main]/Main/Tmpfile[c]/File[/tmp/c]/ensure: defined content as
'{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: Applied catalog in 0.13 seconds
```

### How it works...

You can think of a defined type (introduced with the `define` keyword) as a cookie-cutter. It describes a pattern that Puppet can use to create lots of similar resources. Any time you declare a `tmpfile` instance in your manifest, Puppet will insert all the resources contained in the `tmpfile` definition.

In our example, the definition of `tmpfile` contains a single file resource whose content is `Hello world\n` and whose path is `/tmp/${name}`. If you declared an instance of `tmpfile` with the name `foo`, Puppet will create a file with the `/tmp/foo` path:

```
tmpfile { 'foo': }
```

In other words, `${name}` in the definition will be replaced by the name of any actual instance that Puppet is asked to create. It's almost as though we created a new kind of resource, `tmpfile`, which has one parameter: its name.

Just like with regular resources, we don't have to pass just one title; as in the preceding example, we can provide an array of titles and Puppet will create as many resources as required.

A note on `name`, the `namevar`: every resource you create must have a unique name, the `namevar`. This is different than the title, which is how Puppet refers to the resource internally (although they are often the same).

### There's more...

In the example, we created a definition where the only parameter that varies between instances is the name parameter. But we can add whatever parameters we want, so long as we declare them in the definition in parentheses after the name parameter, as follows:

```
define tmpfile (
  String $greeting = "Hello, World!\n"
  ) {
    file { "/tmp/${name}":
      content => $greeting,
  }
}

tmpfile { 'd': greeting => "Good Morning!\n" }
```

In this example, we've specified that the `$greeting` parameter is a String. Puppet will not allow us to try using `tmpfile` without a String:

```
Error: Evaluation Error: Error while evaluating a Resource Statement, Tmpfile[e]:
parameter 'greeting' expects a String value, got Integer at
/home/thomas/tmpfile2.pp:10 on node mylaptop.example.com
```

We also specified a default value for the greeting; if you fail to pass a greeting parameter, the default value will be used. You can declare multiple parameters as a comma-separated list:

```
define mywebapp (
 String $domain = $facts['domain'],
 String $path,
 String $platform,
) {
 notify {"${domain} ${path} ${platform}": }
}

mywebapp { 'mywizzoapp':
 domain   => 'Rails',
 path     => '/var/www/apps/mywizzoapp',
 platform => 'mywizzoapp.com',
}
```

This is a powerful technique for abstracting out everything that's common to certain resources, and keeping it in one place so that you don't repeat yourself. In the preceding example, there might be many individual resources contained within `mywebapp`: packages, config files, source code checkouts, virtual hosts, and so on. But all of them

are the same for every instance of `mywebapp` except in the parameters we provide. These might be referenced in a template, for example, to set the domain for a virtual host.

## Using tags

Sometimes one Puppet class needs to know about another, or at least know whether or not it's present. For example, a class that manages the firewall may need to know whether or not the node is a web server.

Puppet's tagged function will tell you whether a named class or resource is present in the catalog for this node. You can also apply arbitrary tags to a node or class and check for the presence of these tags. Tags are another metaparameter, similar to `require` and `notify`, which we introduced in [*[Lab 1], Puppet Language and Style*]. Metaparameters are used in the compilation of the Puppet catalog but are not attributes of the resource to which they are attached.

### How to do it...

To help you find out whether you're running on a particular node or class of nodes, all nodes are automatically tagged with the node name and the names of any classes they include. Here's an example that shows you how to use `tagged` to get this information:

1. Add the following code to your `site.pp` file (replacing `fenago` with your machine's hostname):

```
node 'fenago' {
  if tagged('fenago') {
    notify { 'tagged fenago': }
  }
}
```

2. Run Puppet:

```
[root@fenago ~]# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for test.example.com
Info: Applying configuration version '1524111400'
Notice: tagged fenago
Notice: /Stage[main]/Main/Node[fenago]/Notify[tagged fenago]/message: defined
'message' as 'tagged fenago'
Notice: Applied catalog in 0.03 seconds
```

Nodes are also automatically tagged with the names of all the classes they include in, addition to several other automatic tags. You can use tagged to find out what classes are included on the node. You're not just limited to checking the tags automatically applied by Puppet. You can also add your own. To set an arbitrary `tag` on a node, use the `tag` function, as in the following example:

1. Modify your `site.pp` file, as follows:

```
node 'fenago' {
  tag('tagging')
  class {'tag_test': }
}
```

2. Add a `tag_test` module with the following `init.pp`:

```
class tag_test {
  if tagged('tagging') {
    notify { 'containing node/class was tagged.': }
  }
}
```

3. Run Puppet:

```
root@fenago:~# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for fenago.example.com
Info: Applying configuration version '1517851735'
Notice: containing node/class was tagged.
Notice: /Stage[main]/Tag_test/Notify[containing node/class was tagged.]/message:
defined 'message' as 'containing node/class was tagged.'
Notice: Applied catalog in 0.30 seconds
```

4. You can also use tags to determine which parts of the manifest to apply. If you use the `--tags` option on the Puppet command line, Puppet will apply only those classes or resources tagged with the specific tags you include. For example, we can define our fenago class with two classes:

```
node fenago {
  class {'first_class': }
  class {'second_class': }
}
class first_class {
  notify { 'First Class': }
}
class second_class {
  notify {'Second Class': }
}
```

5. Now, when we run `puppet agent` on the fenago node, we see both `notify`:

```
root@fenago:~# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for fenago.example.com
Info: Applying configuration version '1517851837'
Notice: First Class
Notice: /Stage[main]/First_class/Notify[First Class]/message: defined 'message' as
'First Class'
Notice: Second Class
Notice: /Stage[main]/Second_class/Notify[Second Class]/message: defined 'message' as
'Second Class'
Notice: Applied catalog in 0.27 seconds
```

6. Apply only the `first_class` add `--tags` function to the command line:

```
root@fenago:~# puppet agent -t --tags first_class
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for fenago.example.com
Info: Applying configuration version '1517851867'
Notice: First Class
Notice: /Stage[main]/First_class/Notify[First Class]/message: defined 'message' as
'First Class'
Info: Stage[main]: Unscheduling all events on Stage[main]
Notice: Applied catalog in 0.30 seconds
```

### There's more...

You can use tags to create a collection of resources, and then make the collection a dependency for some other resource. For example, say some service depends on a config file that is built from a number of file snippets, as in the following example:

```
class firewall::service {
  service { 'firewall':
    ...
  }
  File <| tag == 'firewall-snippet' |> ~> Service['firewall']
}
class myapp {
  file { '/etc/firewall.d/myapp.conf':
    tag => 'firewall-snippet',
    ...
  }
}
```

Here, we've specified that the firewall service should be notified if any file resource tagged `firewall-snippet` is updated. All we need to do to add a firewall config snippet for any particular application or service is to tag it firewall-snippet, and Puppet will do the rest.

Although we could add a notify `=> Service["firewall"]` function to each snippet resource if our definition of the firewall service were ever to change, we would have to hunt down and update all the snippets accordingly. The tag lets us encapsulate the logic in one place, making future maintenance and refactoring much easier.

What's `<| tag == 'firewall-snippet' |> syntax`? This is called a resource collector, and it's a way of specifying a group of resources by searching for some piece of data about them; in this case, the value of a tag.

What does `~>` mean? This is a chaining arrow with notification. The resource(s) on the left must come before the resource(s) on the right. If any resources on the left update, then they notify the resources on the right.

#### Note

More information on resource relationships may be found on the puppet website: https://puppet.com/docs/puppet/5.0/lang_relationships.html.

## Using run stages

A common requirement is to apply a certain group of resources before other groups (for example, installing a package repository or a custom Ruby version), or after others (for example, deploying an application once its dependencies are installed). Puppet's run stages feature allows you to do this.

By default, all resources in your manifest are applied in a single stage named `main`. If you need a resource to be applied before all others, you can assign it to a new run stage that is specified to come before main. Similarly, you could define a run stage that comes after `main`. In fact, you can define as many run stages as you need and tell Puppet which order they should be applied in.

In this example, we'll use stages to ensure one class is applied first and another last.

### How to do it...

Here are the steps to create an example using run stages:

1. Create the `modules/admin/manifests/stages.pp` file with the following contents:

```
class admin::stages {
  stage { 'first': before => Stage['main'] }
  stage { 'last': require => Stage['main'] }

  class { 'admin::me_last': stage => 'last', }
  class { 'admin::me_first': stage => 'first', }
}
```

2. Create the `admin::me_first and admin::me_last` classes, as follows:

```
class admin::me_first {
  notify { 'This will be done first': }
}
class admin::me_last {
  notify { 'This will be done last': }
}
```

3. Modify your `site.pp` file, as follows:

```
node 'fenago' {
  class {'first_class': }
  class {'second_class': }
  include admin::stages
}
```

4. Run Puppet:

```
root@fenago:~# puppet agent -t
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for fenago.example.com
Info: Applying configuration version '1517854357'
Notice: This will be done first
Notice: /Stage[first]/Admin::Me_first/Notify[This will be done first]/message: defined
'message' as 'This will be done first'
```

```
Notice: First Class
Notice: /Stage[main]/First_class/Notify[First Class]/message: defined 'message' as
'First Class'
Notice: Second Class
Notice: /Stage[main]/Second_class/Notify[Second Class]/message: defined 'message' as
'Second Class'
Notice: This will be done last
Notice: /Stage[last]/Admin::Me_last/Notify[This will be done last]/message: defined
'message' as 'This will be done last'
Notice: Applied catalog in 0.25 seconds
```

### How it works...

Let's examine this code in detail to see what's happening. First, we declare the first and last run stages, as follows:

```
stage { 'first':
  before => Stage['main']
}
stage { 'last':
  require => Stage['main']
}
```

For the first stage, we've specified that it should come before `main`. That is, every resource marked as being in the first stage will be applied before any resource in the main stage (the default stage).

The last stage requires the main stage, so no resource in the last stage can be applied until after every resource in the main stage.

We then declare some classes that we'll later assign to these run stages:

```
class admin::me_first {
  notify { 'This will be done first': }
}
class admin::me_last {
  notify { 'This will be done last': }
}
```

We can now put it all together and include these classes on the node, specifying the run stages for each as we do so:

```
class { 'me_first':
  stage => 'first',
}
class { 'me_last':
  stage => 'last',
}
```

Note that, in the class declarations for `me_first` and `me_last`, we didn't have to specify that they take a stage parameter. The stage parameter is another metaparameter, which means it can be applied to any class or resource without having to be explicitly declared. When we ran `puppet agent` on our `Puppet` node, the notify from the `me_first` class was applied before the notifies from `first_class` and `second_class`. The notify from `me_last` was applied after the main stage, so it comes after the two notifies from `first_class` and `second_class`. If you run puppet agent multiple times, you will see that the notifies from `first_class` and

`second_class` may not always appear in the same order, but the `me_first` class will always come first and the `me_last` class will always come last.

### There's more...

A caveat: many people don't like to use run stages, feeling that Puppet already provides sufficient resource-ordering control, and that using run stages indiscriminately can make your code very hard to follow. The use of run stages should be kept to a minimum wherever possible. There are a few key examples where the use of stages creates less complexity. The most notable is when a resource modifies the system used to install packages on the system. It helps to have a package management stage that comes before the main stage. When packages are defined in the main (default) stage, your manifests can count on the updated package management configuration information being present. For instance, for a Yum-based system, you would create a `yumrepos` stage that comes before `main`. You can specify this dependency using chaining arrows, as shown in the following code snippet:

```
stage {'yumrepos': }
Stage['yumrepos'] -> Stage['main']
```

We can then create a class that creates a Yum repository ( `yumrepo` ) resource and assign it to the yumrepos stage, as follows:

```
class yums {
  notify {'always before the rest': }
  yumrepo {'testrepo':
    baseurl => 'file:///var/yum',
    ensure => 'present',
  }
}
class {'yums':
  stage => 'yumrepos',
}
```

For Apt-based systems, the same example would be a stage where Apt sources are defined. The key with stages is to keep their definitions in your `site.pp` file, where they are highly visible and to use them sparingly where you can guarantee that you will not introduce dependency cycles.

## Using roles and profiles

Well-organized Puppet manifests are easy to read; the purpose of a module should be evident in its name. The purpose of a node should be defined in a single class. This single class should include all classes that are required to perform that purpose.

In this model, roles are the single purpose of a node; a node may only have one role, a role may contain more than one profile, and a profile contains all the resources related to a single service. In this example, we will create a web server role that uses several profiles.

### How to do it...

We'll create two modules to store our roles and profiles. Roles will contain one or more profiles. Each role or profile will be defined as a subclass, such as `profile::base`:

1. Decide on a naming strategy for your roles and profiles. In our example, we will create two modules, `roles` and `profiles`, that will contain our roles and profiles, respectively:

```
t@t510 $ puppet module generate thomas-profiles --skip-interview
Notice: Generating module at /home/thomas/puppet/profiles...
Notice: Populating templates...
...
t@mylaptop $ puppet module generate thomas-roles --skip-interview
Notice: Generating module at /home/thomas/puppet/roles...
Notice: Populating templates...
...
```

2. Begin defining the constituent parts of our webserver role as profiles. To keep this example simple, we will create two profiles. First, a `base` profile to include our basic server configuration classes. Second, an `apache` class to install and configure the Apache web server ( `httpd` ), as follows: `profiles/manifests/ base.pp`

```
class profiles::base {
  include base
}
```

The other file is `profiles/manifests/apache.pp` :

```
class profiles::apache {
  $apache = $::osfamily ? {
    'RedHat' => 'httpd',
    'Debian' => 'apache2',
  }
  service { $apache:
    enable => true,
    ensure => true,
  }
  package { $apache:
    ensure => 'installed',
  }
}
```

3. Define a `roles::webserver` class for our webserver role as follows: `roles/manifests/webserver.pp`

```
class roles::webserver {
  include profiles::apache
  include profiles::base
}
```

4. Apply the `roles::webserver` class to a node. In a centralized installation, you would use either an **External Node Classifier** (**ENC**) to apply the class to the node, or you would use Hiera to define the role:

```
node 'webtest' {
  include roles::webserver
}
```

### How it works...

Breaking down the parts of the web server configuration into different profiles allows us to apply those parts independently. We created a base profile that we can expand to include all the resources we would like applied to all nodes. Our `roles::webserver` class simply includes the `base` and `apache` classes.

### There's more...

As we'll see in the next section, we can pass parameters to classes to alter how they work. In our `roles::webserver` class, we can use the class instantiation syntax instead of `include`, and override it with parameters in the classes. For instance, to pass a parameter to the `base` class, we would use the following:

```
class {'profiles::base':
  parameter => 'newvalue'
}
```

This replaces our previous use:

```
include profiles::base
```

In previous versions of this course, node and class inheritance were used to achieve a similar goal, code reuse. Node inheritance is deprecated in Puppet Version 3.7 and higher. Node and class inheritance should be avoided. Using roles and profiles achieves the same level of readability and is much easier to follow.

## Using data types in Puppet

In previous releases of Puppet, variables were not typed. A variable could hold any sort of value. Although this makes writing code somewhat easier, it leads to many problems. Variables that expect an array could be passed a string; variables that expect an integer may be passed a String. Type mismatch can have very bad affects so, to combat this problem, several helper functions were created in `stdlib` to validate the type of a variable. The validation functions were named for the data type they validated and included `validate_array`, `validate_hash`, `validate_numeric`, and `validate_string`. All these functions have been deprecated in Puppet 5 and replaced with the `assert_type` function. The `assert_type` function can be used to ensure that a variable is of any given type. Puppet5 also enforces types when they are assigned to class parameters, as we'll see in the next section.

### How to do it...

In this example, we'll create a password variable and ensure that it is at least eight characters long:

1. Create a password manifest, as follows:

```
$password = "pass"
$valid_password = assert_type(String[8],$password)

notify {"v=${valid_password}": }
```

2. Next, use `puppet apply` on the manifest:

```
t@mylaptop $ puppet apply password.pp
Error: Evaluation Error: Error while evaluating a Function Call, assert_type():
expects a String[8, default] value, got String at /home/thomas/password.pp:2:19 on
node mylaptop.example.com
```

3. Our password was not long enough, so change the password to an eight-character password and rerun `puppet apply`:

```
t@mylaptop $ puppet apply password.pp
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.01
seconds
Notice: v=password
```

```
Notice: /Stage[main]/Main/Notify[v=password]/message: defined 'message' as
'v=password'
Notice: Applied catalog in 0.02 seconds
```

### There's more...

The following types are available in Puppet5:

- String
- Integer, float, and numeric
- Boolean
- Array
- Hash
- Regexp (a regular expression matcher)
- Undef
- Default (the default of a case statement, for example)

In addition to these data types, there are several abstract data types built upon the code data types. The abstract data types can be used to ensure that variables have very specific values. For example, when specifying a port to use for a package, you can use the `enum` type to enforce that the port be one of a selection of String values, as shown here:

```
$state = 'install'
$pkg = assert_type(Enum['installed','absent'], $state)
```

When we run `puppet apply` on this manifest, we return an error:

```
t@mylaptop $ puppet apply enum.pp
Error: Evaluation Error: Error while evaluating a Function Call, assert_type():
expects a match for Enum['absent', 'installed'], got 'install' at
/home/thomas/enum.pp:2:8 on node mylaptop.example.com
```

There are a few abstract data types. Variant is another useful one, if you want to accept a String or an array of String, you can define your variable as follows:

```
$pkgs = ['ssh','gcc']
assert_type(Variant[Array[String],String],$pkgs)
```

## Passing parameters to classes

Sometimes it's very useful to parameterize some aspect of a class. For example, you might need to manage different versions of a gem package and, rather than making separate classes for each that differ only in the version number, you can pass in the version number as a parameter.

### How to do it...

In this example, we'll create a definition that accepts parameters:

1. Declare the parameter as a part of the class definition:

```
class eventmachine(
  String $version
  ) {
  package { 'eventmachine':
    provider => gem,
    ensure   => $version,
  }
}
```

2. Use the following syntax to include the class on a node:

```
class { 'eventmachine':
  version => '1.0.3',
}
```

### How it works...

The class definition class `eventmachine` ( `$version` ) { is just like a normal class definition except it specifies that the class takes one parameter: `$version`. Inside the class, we've defined a package resource:

```
package { 'eventmachine':
  provider => gem,
  ensure   => $version,
}
```

This is a gem package, and we're requesting to install the `$version` version.

Include the class on a node, but instead of the usual `include` syntax, `include eventmachine`, we use the class declaration syntax as follows:

```
class { 'eventmachine': version => '1.0.3', }
```

This has the same effect but also sets a value for the parameter version as `1.0.3`.

### There's more...

You can specify multiple parameters for a `class` as follows:

```
class mysql(
  Variant[String, Array[String]] $package,
  String $socket,
  Integer $port
  ) {
  ...
}
```

Then, supply them in the same way:

```
class { 'mysql':
  package => 'percona-server-server-5.5',
  socket  => '/var/run/mysqld/mysqld.sock',
  port    => 3306,
}
```

**Specifying default values**

You can also give default values for some of your parameters. When you include the class without setting a parameter, the default value will be used. For instance, if we created a MySQL class with three parameters, we could provide default values for any of the parameters, as shown in the code snippet:

```
class mysql(
  Variant[String, Array[String]] $package,
  String $socket,
  Integer $port=3306
) {
```

Or we can provide them for all:

```
class mysql(
  Variant[String, Array[String]] package = 'percona-server-server-5.5',
  String socket                          = '/var/run/mysqld/mysqld.sock',
  Integer port                           = 3306
  ) {
```

Defaults allow you to use a default value and override that default when you need to. Unlike a definition, only one instance of a parameterized class can exist on a node. When you need to have several different instances of the resource, use `define` instead.

## Passing parameters from Hiera

Like the parameter defaults we introduced in the previous lab, Hiera may be used to provide default values to classes. Automatic parameter lookup via Hiera has been on by default since version 3 of Puppet.

### Getting ready

Configure hiera as we did in [*[Lab 2], Puppet Infrastructure*]. Create a global or common YAML file; this will serve as the default for all values.

### How to do it...

1. We'll create a class with parameters and no default values. Create the directory `modules/mysql/manifests` and then create `modules/mysql/manifests/init.pp` with the following content:

```
class mysql (
  Integer $port,
  String $socket,
  Variant[String,Array[String]] $package
  ) {
    notify {"Port: ${port} Socket: ${socket} Package: ${package}": }
}
```

2. Update your `common.yaml` file in Hiera with the default values for the `mysql` class:

```
---
mysql::port: 3306
mysql::package: 'mysql-server'
mysql::socket: '/var/lib/mysql/mysql.sock'
```

3. Apply the class to a node; you can add the `mysql` class to your default node for now:

```
node default {
  class {'mysql': }
}
```

4. Run `puppet agent` and verify the output:

```
[root@testnode ~]# puppet agent -t
...
Notice: Port: 3306 Socket: /var/lib/mysql/mysql.sock Package: mysql-server
Notice: /Stage[main]/Mysql/Notify[Port: 3306 Socket: /var/lib/mysql/mysql.sock
Package: mysql-server]/message: defined 'message' as 'Port: 3306 Socket:
/var/lib/mysql/mysql.sock Package: mysql-server'
Notice: Applied catalog in 0.05 seconds
```

### How it works...

When we instantiate the MySQL class in our manifest, we provided no values for any of the attributes. Puppet knows to look for a value in Hiera that matches `class_ name::parameter_name:` or `::class_name::parameter_name:`.

When Puppet finds a value, it uses it as the parameter for the class. If Puppet fails to find a value in Hiera and no default is defined, a catalog failure will result in the following command line:

```
Error: Could not retrieve catalog from remote server: Error 500 on SERVER: Server
Error: Evaluation Error: Error while evaluating a Function Call, Class[Mysql]: expects
a value for parameter 'port' at
/etc/puppetlabs/code/environments/production/manifests/site.pp:21:3 on node
testnode.example.com
```

### There's more...

You can define a Hiera hierarchy and supply different values for parameters based on facts. You could, for instance, have `%{::facts.os.family}` in your hierarchy and have different YAML files based on the `facts.os.family` fact (that is, RedHat, Suse, and Debian).

## Writing reusable, cross-platform manifests

Every system administrator dreams of a unified, homogeneous infrastructure of identical machines all running the same version of the same OS. As in other areas of life, however, the reality is often messy and doesn't conform to the plan.

You are probably responsible for a bunch of assorted servers of varying age and architecture running different kernels from different OS distributions, often scattered across different data centers and ISPs.

This situation should strike terror into the hearts of sysadmins of the `ssh` in a `for loop` persuasion because executing the same commands on every server can have different, unpredictable, and even dangerous results.

We should certainly strive to bring older servers up-to-date and get working as far as possible on a single reference platform to make administration simpler, cheaper, and more reliable. But until you get there, Puppet makes coping with heterogeneous environments slightly easier.

### How to do it...

Here are some examples of how to make your manifests more portable:

1. Where you need to apply the same manifest to servers with different OS distributions, the main differences will probably be the names of packages and services and the location of config files. Try to capture all these differences into a single class by using selectors to set global variables:

```
$ssh_service = $::operatingsystem? {
  /Ubuntu|Debian/ => 'ssh',
  default         => 'sshd',
}
```

You needn't worry about the differences in any other part of the manifest; when you refer to something, use the variable with confidence that it will point to the right thing in each environment:

```
service { $ssh_service:
  ensure => running,
}
```

2. Often we need to cope with mixed architectures; this can affect the paths to shared libraries, and also may require different versions of packages. Again, try to encapsulate all the required settings in a single architecture class that sets global variables:

```
$libdir = $::architecture ? {
  amd64   => '/usr/lib64',
  default => '/usr/lib',
}
```

Then you can use these wherever an architecture-dependent value is required in your manifests or even in templates:

```
; php.ini [PHP]
; Directory in which the loadable extensions (modules) reside.
extension_dir = <%= @libdir %>/php/modules
```

### How it works...

The advantage of this approach (which could be called top-down) is that you only need to make your choices once. The alternative, bottom-up approach would be to have a selector or case statement everywhere a setting is used:

```
service { $::operatingsystem? {
  /Ubuntu|Debian/ => 'ssh',
  default         => 'sshd' }:
  ensure          => running, }
```

This not only results in lots of duplication, but makes the code harder to read. And when a new operating system is added to the mix, you'll need to make changes throughout the whole manifest, instead of just in one place.

## Getting information about the environment

Often in a Puppet manifest, you need to know some local information about the machine you're on. Facter is the tool that accompanies Puppet to provide a standard way of getting information (facts) from the environment about things such as the following:

- Operating system
- Memory size
- Architecture
- Processor count

To see a complete list of the facts available on your system, run the following code:

```
$ sudo facter
aio_agent_version => 5.5.3
augeas => {
```

```
  version => "1.8.1"
}
...
```

While it can be handy to get this information from the command line, the real power of Facter lies in being able to access these facts in your Puppet manifests.

Some modules define their own facts; to see any facts that have been defined locally, add the `-p` (pluginsync) option to `facter`, as follows:

```
$ sudo facter -p
```

This is deprecated in Puppet5. To access facts defined in puppet, use the `puppet facts` face instead:

```
$ puppet facts
```

**Note**

puppet facts will return all the facts for a node and redirect the output to a pager or a file to inspect the results.

### How to do it...

Here's an example of using Facter facts in a manifest:

1. Reference a Facter fact in your manifest like any other variable. Facts are global variables in Puppet, so they should be prefixed with a double colon ( `::` ), as in the following code snippet:

```
$funfacts = @("FACTS")
  This is ${::facts['os']['name']}
  version ${::facts['os']['release']['full']},
  on ${::facts['os']['architecture']} architecture,
  kernel version ${::kernelversion}
  | FACTS

notify {'funfacts':
  message => $funfacts
}
```

2. When Puppet runs, it will fill in the appropriate values for the current node:

```
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.01
seconds
Notice: This is Fedora
version 27,
on x86_64 architecture,
kernel version 4.14.11
```

### How it works...

Facter provides a standard way for manifests to get information about the nodes to which they are applied. When you refer to a fact in a manifest, Puppet will query Facter to get the current value and insert it into the manifest. Facter facts are top-scope variables.

Always refer to them with leading double colons to ensure that you are using the facter value and not a local variable: `$::hostname` NOT `$hostname`. In Puppet5, I prefer to use the facts hash instead of referring to facts directly. `$::facts['hostname']` cannot be overridden by a variable, it must come from a fact.

### There's more...

You can also use facts in **EPP** templates. For example, you might want to insert the node's hostname into a file or change a configuration setting for an application based on the memory size of the node. When you use fact names in templates, remember that they don't need a dollar sign because this is Ruby, not Puppet:

```
$KLogPath <%=
  case $::kernelversion
    {/^2/: { '/var/run/rsyslog/kmsg' }
    default: { '/proc/kmsg' }
  } %>
```

Variable references in epp templates are always fully scoped. To access variables defined in the class where you called the `epp` function, use the full path to that class. For example, to reference the `mysql::port` variable we defined earlier in the `mysql` modules, use the following:

```
MySQL Port = <%= $::mysql::port %>
```

## Importing dynamic information

Even though some system administrators like to wall themselves off from the rest of the office using piles of old printers, we all need to exchange information with other departments from time to time. For example, you may want to insert data into your Puppet manifests that is derived from some outside source. The `generate` function is ideal for this. Functions are executed on the machine compiling the catalog (the master for centralized deployments); an example such as the one shown here will only work in a masterless configuration.

### Getting ready

Follow these steps to prepare to run the example:

1. Create the `/usr/local/bin/message.rb` script with the following contents:

```
#!/opt/puppetlabs/puppet/bin/ruby
puts "This runs on the master if you are centralized"
```

2. Make the script executable:

```
$ sudo chmod a+x /usr/local/bin/message.rb
```

### How to do it...

This example calls the external script we created previously and gets its output:

1. Create a `message.pp` manifest containing the following:

```
$message = generate('/usr/local/bin/message.rb')
notify { $message: }
```

2. Run Puppet:

```
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.04
seconds
Notice: This runs on the master if you are centralized

Notice: /Stage[main]/Main/Notify[This runs on the master if you are centralized
]/message: defined 'message' as "This runs on the master if you are centralized\n"
Notice: Applied catalog in 0.02 seconds
```

### How it works...

The generate function runs the specified script or program and returns the result, in this case, a cheerful message from Ruby.

This isn't terribly useful as it stands, but you get the idea. Anything a script can do print, fetch, or calculate, for example, the results of a database query can be brought into your manifest using generate. You can also, of course, run standard UNIX utilities, such as `cat` and `grep`.

### There's more...

If you need to pass arguments to the executable called by generate, add them as extra arguments to the function call:

```
$message = generate('/bin/cat', '/etc/motd')
```

Puppet will try to protect you from malicious shell calls by restricting the characters you can use in a call to generate, so shell pipes and redirection aren't allowed, for example. The simplest and safest thing to do is to put all your logic into a script and then call that script.

## Passing arguments to shell commands

If you want to insert values into a command line (to be run by an `exec` resource, for example), they often need to be quoted, especially if they contain spaces. The `shellquote` function will take any number of arguments, including arrays, and quote each of the arguments and return them all as a space-separated string that you can pass to commands.

In this example, we would like to set up an exec resource that will rename a file; but both the source and the target name contain spaces, so they need to be correctly quoted in the command line.

### How to do it...

Here's an example of using the `shellquote` function:

1. Create a `shellquote.pp` manifest with the following command:

```
$source = 'Hello Jerry'
$target = 'Hello... Newman'
$argstring = shellquote($source, $target)
$command = "/bin/mv ${argstring}"
notify { $command: }
```

2. Run Puppet:

```
t@mylaptop ~ $ puppet apply shellquote.pp
Notice: Compiled catalog for mylaptop.example.com in environment production in 0.02
seconds
Notice: /bin/mv "Hello Jerry" "Hello... Newman"
Notice: /Stage[main]/Main/Notify[/bin/mv "Hello Jerry" "Hello... Newman"]/message:
defined 'message' as '/bin/mv "Hello Jerry" "Hello... Newman"'
Notice: Applied catalog in 0.02 seconds
```

### How it works...

First, we define the `$source` and `$target` variables, which are the two filenames we want to use in the command line:

```
$source = 'Hello Jerry'
$target = 'Hello... Newman'
```

Then we call `shellquote` to concatenate these variables into a quoted, space-separated string, as follows:

```
$argstring = shellquote($source, $target)
```

Then we put together the final command line:

```
$command = "/bin/mv ${argstring}"
```

The result will be as follows:

```
/bin/mv "Hello Jerry" "Hello... Newman"
```

This command line can now be run with an `exec` resource. What would happen if we didn't use shellquote? See the following:

```
$source = 'Hello Jerry'
$target = 'Hello... Newman'
$command = "/bin/mv ${source} ${target}"
notify { $command: }
Notice: /bin/mv Hello Jerry Hello... Newman
```

This won't work because `mv` expects space-separated arguments, so it will interpret this as a request to move the `Hello`, `Jerry`, and `Hello...` files into a directory named `Newman`, which probably isn't what we want.