Open Source Puppet

6.17 (latest)

+ Welcome to Puppet 6.17
+ Installing and configuring
+ The Puppet platform
- Developing Puppet code
  - The Puppet language
    » Puppet language overview
    » Puppet language syntax examples
    » The Puppet language style guide
    » Files and paths on Windows
    » Code comments
    » Variables

The Puppet language style guide

Open Source Puppet — 6.17 (latest)

Sections

+ Resource types
» Relationships and ordering
» Classes
» Defined resource types
» Bolt tasks
» Expressions and operators
» Conditional statements and expressions
» Function calls
» Built-in functions
» Node definitions
+ Facts and built-in variables
» Reserved words and acceptable names
+ Custom resources
+ Values, data types, and aliases
+ Templates
+ Advanced constructs
+ Details of complex behaviors

+ Modules
» Puppet Forge
» Puppet Development Kit (PDK)
» Puppet VSCode extension

» Automating ad-hoc tasks

This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.

This style guide applies to Puppet 4 and later. Puppet 3 is no longer supported, but we include some Puppet 3 guidelines in case you're maintaining older code.

> **Tip:** Use puppet-lint and metadata-json-lint to check your module for compliance with the style guide.

No style guide can cover every circumstance you might run into when developing Puppet code. When you need to make a judgement call, keep in mind a few general principles.

Readability matters

If you have to choose between two equal alternatives, pick the more readable one. This is subjective, but if you can read your own code three months from now, it's a great start. In particular, code that generates readable diffs is highly preferred.

Scoping and simplicity are key
: When in doubt, err on the side of simplicity. A module should contain related resources that enable it to accomplish a task. If you describe the function of your module and you find yourself using the word "and," consider splitting the module. Have one goal, with all your classes and parameters focused on achieving it.

Your module is a piece of software
: At least, we recommend that you treat it that way. When it comes to making decisions, choose the option that is easier to maintain in the long term.

These guidelines apply to Puppet code, for example, code in Puppet modules or classes. To reduce repetitive phrasing, we don't include the word 'Puppet' in every description, but you can assume it.

For information about the specific meaning of terms like 'must,' 'must not,' 'required,' 'should,' 'should not,' 'recommend,' 'may,' and 'optional,' see RFC 2119.

Module design practices

Consistent module design practices makes module contributions easier.

Spacing, indentation, and whitespace

Module manifests should follow best practices for spacing, indentation, and whitespace.

Manifests:

- Must use two-space soft tabs.
- Must not use literal tab characters.
- Must not contain trailing whitespace.
- Must include trailing commas after all resource attributes and parameter definitions.
- Must end the last line with a new line.
- Must use one space between the resource type and opening brace, one space between the opening brace and the title, and no spaces between the title and colon.
  Good:

```
file { '/tmp/sample':
```

Bad: Space between title and colon:

```
file { '/tmp/sample' :
```

Bad: No spaces:

```
file{'/tmp/sample':
```

Bad: Too many spaces:

```
file     { '/tmp/sample':
```

- Should not exceed a 140-character line width, except where such a limit would be impractical.
- Should leave one empty line between resources, except when using dependency chains.
- May align hash rockets (=>) within blocks of attributes, one space after the longest resource key, arranging hashes for maximum readability first.

Arrays and hashes

To increase readability of arrays and hashes, it is almost always beneficial to break up the elements on separate lines.

Use a single line only if that results in overall better readability of the construct where it appears, such as when it is very short. When breaking arrays and hashes, they should have:

- Each element on its own line.
- Each new element line indented one level.
- First and last lines used only for the syntax of that data type.

Good: Array with multiple elements on multiple lines:

```
service { 'sshd':
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

Good: Hash with multiple elements on multiple lines:

```
$myhash = {
  key       => 'some value',
  other_key => 'some other value',
}
```

Bad: Array with multiple elements on same line:

```
service { 'sshd':
  require => [ Package['openssh-server'], File['/etc/ssh/sshd_config'], ],
}
```

Bad: Hash with multiple elements on same line:

```
$myhash = { key => 'some value', other_key => 'some other value', }
```

Bad: Array with multiple elements on different lines, but syntax and element share a line:

```
service { 'sshd':
  require => [ Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

Bad: Hash with multiple elements on different lines, but syntax and element share a line:

```
$myhash = { key => 'some value',
  other_key     => 'some other value',
}
```

Bad: Array with an indention of elements past two spaces:

```
service { 'sshd':
  require => [
              Package['openssh-server'],
              File['/etc/ssh/sshd_config'],
  ],
}
```

Quoting

As long you are consistent, strings may be enclosed in single or double quotes, depending on your preference.

Regardless of your preferred quoting style, all variables MUST be enclosed in braces when interpolated in a string.

For example:
Good:

```
"/etc/${file}.conf"
```

```
"${facts['operatingsystem']} is not supported by ${module_name}"
```

Bad:

```
"/etc/$file.conf"
```

Option 1: Prefer single quotes

Modules that adopt this string quoting style MUST enclose all strings in single quotes, except as listed below.

For example:
Good:

```
owner => 'root'
```

Bad:

```
owner => "root"
```

A string MUST be enclosed in double quotes if it:

- Contains variable interpolations.
    - Good:

      ```
      "/etc/${file}.conf"
      ```

    - Bad:

      ```
      '/etc/${file}.conf'
      ```

- Contains escaped characters not supported by single-quoted strings.

  - Good:

    ```
    content => "nameserver 8.8.8.8\n"
    ```

  - Bad:

    ```
    content => 'nameserver 8.8.8.8\n'
    ```

A string SHOULD be enclosed in double quotes if it:

- Contains single quotes.
  - Good:

    ```
    warning("Class['apache'] parameter purge_vdir is deprecated in favor of purg
    ```

  - Bad:

    ```
    warning('Class[\'apache\'] parameter purge_vdir is deprecated in favor of pu
    ```

Option 2: Prefer double quotes

Modules that adopt this string quoting style MUST enclose all strings in double quotes, except as listed below.

For example:
Good:

```
owner => "root"
```

Bad:

```
owner => 'root'
```

A string SHOULD be enclosed in single quotes if it does not contain variable interpolations AND it:

- Contains double quotes.
  - Good:

    ```
    warning('Class["apache"] parameter purge_vdir is deprecated in favor of purg
    ```

  - Bad:

    ```
    warning("Class[\"apache\"] parameter purge_vdir is deprecated in favor of pu
    ```

- Contains literal backslash characters that are not intended to be part of an escape sequence.
  - Good:

    ```
    path => 'c:\windows\system32'
    ```

  - Bad:

    ```
    path => "c:\\windows\\system32"
    ```

If a string is a value from an enumerable set of options, such as `present` and `absent`, it SHOULD NOT be enclosed in quotes at all.

For example:

Good:

```
ensure => present
```

Bad:

```
ensure => "present"
```

Escape characters

Use backslash (`\`) as an escape character.

For both single- and double-quoted strings, escape the backslash to remove this special meaning: `\\` This means that for every backslash you want to include in the resulting string, use two backslashes. As an example, to include two literal backslashes in the string, you would use four backslashes in total.

Do not rely on unrecognized escaped characters as a method for including the backslash and the character following it.

Unicode character escapes using fewer than 4 hex digits, as in `\u040`, results in a backslash followed by the string `u040`. (This also causes a warning for the unrecognized escape.) To use a number of hex digits not equal to 4, use the longer `u{digits}` format.

Comments

Comments must be hash comments (`# This is a comment`). Comments should explain the why, not the how, of your code.

Do not use `/* */` comments in Puppet code.
Good:

```
# Configures NTP
file { '/etc/ntp.conf': ... }
```

Bad:

```
/* Creates file /etc/ntp.conf */
file { '/etc/ntp.conf': ... }
```

Note: Include documentation comments for Puppet Strings for each of your classes, defined types, functions, and resource types and providers. If used, documentation comments precede the name of the element. For documentation recommendations, see the Modules section of this guide.

Functions

Avoid the `inline_template()` and `inline_epp()` functions for templates of more than one line, because these functions don't permit template validation. Instead, use the `template()` and `epp()` functions to read a template from the module. This method allows for syntax validation.

You should avoid using calls to Hiera functions in modules meant for public consumption, because not all users have implemented Hiera. Instead, we recommend using parameters that can be overridden with Hiera.

Improving readability when chaining functions

In most cases, especially if blocks are short, we recommend keeping functions on the same line. If you have a particularly long chain of operations or block that you find difficult to read, you can break it up on multiples lines to improve readability. As long as your formatting is consistent throughout the chain, it is up to your own judgment.
For example, this:

```
$foodgroups.fruit.vegetables
```

Is better than this:

```
   $foodgroups
           .fruit
           .vegetables
```

But, this:

```
$foods = {
 "avocado"    => "fruit",
 "eggplant"   => "vegetable",
 "strawberry" => "fruit",
 "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| {
 # Choose only fruits
 $kind == "fruit"
}.map |$name, $kind| {
 # Return array of capitalized fruits
 String($name, "%c")
}.filter |$fruit| {
 # Only keep fruits named "berry"
 $fruit =~ /berry$/
}
```

Is better than this:

```
$foods = {
 "avocado"    => "fruit",
 "eggplant"   => "vegetable",
 "strawberry" => "fruit",
 "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| { $kind == "fruit" }.map |$name, $kind| { St
```

Related information

- Modules

Resources

Resources are the fundamental unit for modeling system configurations. Resource declarations have a lot of possible features, so your code's readability is crucial.

Resource names

All resource names or titles must be quoted. If you are using an array of titles you must quote each title in the array, but cannot quote the array itself.

Good:

```
package { 'openssh': ensure => present }
```

Bad:

```
package { openssh: ensure => present }
```

These quoting requirements do not apply to expressions that evaluate to strings.

Arrow alignment

To align hash rockets (=>) in a resource's attribute/value list or in a nested block, place the hash rocket one space ahead of the longest attribute name. Indent the nested block by two spaces, and place each attribute on a separate line. Declare very short or single purpose resource declarations on a single line.

Good:

```
exec { 'hambone':
  path => '/usr/bin',
  cwd  => '/tmp',
}

exec { 'test':
  subscribe   => File['/etc/test'],
  refreshonly => true,
}

myresource { 'test':
  ensure => present,
  myhash => {
    'myhash_key1' => 'value1',
    'key2'        => 'value2',
  },
}

notify { 'warning': message => 'This is an example warning' }
```

Bad:

```
exec { 'hambone':
path  => '/usr/bin',
cwd => '/tmp',
}

file { "/path/to/my-filename.txt":
  ensure => file, mode => $mode, owner => $owner, group => $group,
  source => 'puppet:///modules/my-module/productions/my-filename.txt'
}
```

Attribute ordering

If a resource declaration includes an `ensure` attribute, it should be the first attribute specified so that a user can quickly see if the resource is being created or deleted.

Good:

```
file { '/tmp/readme.txt':
  ensure => file,
  owner  => '0',
  group  => '0',
  mode   => '0644',
}
```

When using the special attribute `*` (asterisk or splat character) in addition to other attributes, splat should be ordered last so that it is easy to see. You may not include multiple splats in the same body.

Good:

```
$file_ownership = {
  'owner' => 'root',
  'group' => 'wheel',
  'mode'  => '0644',
}

file { '/etc/passwd':
  ensure => file,
  *      => $file_ownership,
}
```

Resource arrangement

Within a manifest, resources should be grouped by logical relationship to each other, rather than by resource type.

Good:

```
file { '/tmp/dir':
  ensure => directory,
}

file { '/tmp/dir/a':
  content => 'a',
}

file { '/tmp/dir2':
  ensure => directory,
}

file { '/tmp/dir2/b':
  content => 'b',
}
```

Bad:

```
file { '/tmp/dir':
  ensure => directory,
}

file { '/tmp/dir2':
  ensure => directory,
}

file { '/tmp/dir/a':
  content => 'a',
}

file { '/tmp/dir2/b':
  content => 'b',
}
```

Use semicolon-separated multiple resource bodies only in conjunction with a local default body.

Good:

```
$defaults = { < hash of defaults > }

file {
  default:
    * => $defaults,;

  '/tmp/testfile':
    content => 'content of the test file',
}
```

Good: Repeated pattern with defaults:

```
$defaults = { < hash of defaults > }

file {
  default:
    * => $defaults,;

  '/tmp/motd':
    content => 'message of the day',;

  '/tmp/motd_tomorrow':
    content => 'tomorrows message of the day',;
}
```

Bad: Unrelated resources grouped:

```
file {
  '/tmp/testfile':
    owner    => 'admin',
    mode     => '0644',
    contents => 'this is the content',;

  '/opt/myapp':
    owner  => 'myapp-admin',
    mode   => '0644',
    source => 'puppet://<someurl>',;

  # etc
}
```

You cannot set any attribute more than one time for a given resource; if you try, Puppet raises a compilation error. This means:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if mode is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, because `*` itself acts like an attribute.
- To use some attributes from a hash and override others, either use a hash to set per-expression defaults, or use the `+` (merging) operator to combine attributes from two hashes (with the right-hand hash overriding the left-hand one).

Symbolic links

Declare symbolic links with an ensure value of `ensure => link`. To inform the user that you are creating a link, specify a value for the `target` attribute.

Good:

```
file { '/var/log/syslog':
  ensure => link,
  target => '/var/log/messages',
}
```

Bad:

```
file { '/var/log/syslog':
  ensure => '/var/log/messages',
}
```

File modes

- POSIX numeric notation must be represented as 4 digits.
- POSIX symbolic notation must be a string.
- You should not use file mode with Windows; instead use the **acl module**.
- You should use numeric notation whenever possible.
- The file mode attribute should always be a quoted string or (unquoted) variable, never an integer.

Good:

```
file { '/var/log/syslog':
  ensure => file,
  mode   => '0644',
}
```

Bad:

```
file { '/var/log/syslog':
  ensure => present,
  mode   => 644,
}
```

Multiple resources

Multiple resources declared in a single block should be used only when there is also a default set of options for the resource type.

Good:

```
file {
  default:
    ensure => 'file',
    mode   => '0666',;

  '/owner':
    user => 'owner',;

  '/staff':
    user => 'staff',;
}
```

Good: Give the defaults a name if used several times:

```
$our_default_file_attributes = {
  'ensure' => 'file',
  'mode'   => '0666',
}

file {
  default:
    * => $our_default_file_attributes,;

  '/owner':
    user => 'owner',;

  '/staff':
    user => 'staff',;
}
```

Good: Spell out 'magic' iteration:

```
['/owner', '/staff'].each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Good: Spell out 'magic' iteration:

```
$array_of_paths.each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Bad:

```
file {
  '/owner':
    ensure => 'file',
    user   => owner,
    mode   => '0666',;

  '/staff':
    ensure => 'file',
    user   => staff,
    mode   => '0774',;
}

file { ['/owner', '/staff']:
  ensure => 'file',
}

file { $array_of_paths:
  ensure => 'file',
}
```

Legacy style defaults

Avoid legacy style defaults. If you do use them, they should occur only at top scope in your site manifest. This is because resource defaults propagate through dynamic scope, which can have unpredictable effects far away from where the default was declared.

Acceptable: `site.pp`:

```
Package {
  provider => 'zypper',
}
```

Bad: `/etc/puppetlabs/puppet/modules/apache/manifests/init.pp`:

```
File {
  owner => 'nobody',
  group => 'nogroup',
  mode  => '0600',
}

concat { $config_file_path:
  notify  => Class['Apache::Service'],
  require => Package['httpd'],
}
```

Attribute alignment

Resource attributes must be uniformly indented in two spaces from the title.

Good:

```
file { '/owner':
  ensure => 'file',
  owner  => 'root',
}
```

Bad: Too many levels of indentation:

```
file { '/owner':
    ensure => 'file',
    owner  => 'root',
}
```

Bad: No indentation:

```
file { '/owner':
ensure => 'file',
owner  => 'root',
}
```

Bad: Improper and non-uniform indentation:

```
file { '/owner':
  ensure => 'file',
   owner => 'root',
}
```

Bad: Indented the wrong direction:

```
  file { '/owner':
ensure => 'file',
owner  => 'root',
  }
```

For multiple bodies, each title should be on its own line, and be indented. You may align all arrows across the bodies, but arrow alignment is not required if alignment per body is more readable.

```
file {
  default:
    * => $local_defaults,;

  '/owner':
    ensure => 'file',
    owner  => 'root',
}
```

Defined resource types

Because defined resource types can have multiple instances, resource names must have a unique variable to avoid duplicate declarations.

Good: Template uses `$listen_addr_port`:

```
define apache::listen {
  $listen_addr_port = $name

  concat::fragment { "Listen ${listen_addr_port}":
    ensure  => present,
    target  => $::apache::ports_file,
    content => template('apache/listen.erb'),
  }
}
```

Bad: Template uses `$name`:

```
define apache::listen {

  concat::fragment { 'Listen port':
    ensure  => present,
    target  => $::apache::ports_file,
    content => template('apache/listen.erb'),
  }
}
```

Classes and defined types

Classes and defined types should follow scope and organization guidelines.

Separate files

Put all classes and resource type definitions (defined types) as separate files in the `manifests` directory of the module. Each file in the manifest directory should contain nothing other than the class or resource type definition.

Good: `etc/puppetlabs/puppet/modules/apache/manifests/init.pp`:

```
class apache { }
```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/ssl.pp`:

```
class apache::ssl { }
```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/virtual_host.pp`:

```
define apache::virtual_host () { }
```

Separating classes and defined types into separate files is functionally identical to declaring them in `init.pp`, but has the benefit of highlighting the structure of the module and making the function and structure more legible.

When a resource or include statement is placed outside of a class, node definition, or defined type, it is included in all catalogs. This can have undesired effects and is not always easy to detect.

Good: `manifests/init.pp`:

```
# class ntp
class ntp {
  ntp::install
}
# end of file
```

Bad: `manifests/init.pp`:

```
class ntp {
  #...
}
ntp::install
```

Internal organization of classes and defined types

Structure classes and defined types to accomplish one task.

Documentation comments for Puppet Strings should be included for each class or defined type. If used, documentation comments must precede the name of the element. For complete documentation recommendations, see the Modules section.

Put the lines of code in the following order:

1. First line: Name of class or type.
2. Following lines, if applicable: Define parameters. Parameters should be **typed**.
3. Next lines: Includes and validation come after parameters are defined. Includes may come before or after validation, but should be grouped separately, with all includes and requires in one group and all validations in another. Validations should validate any parameters and fail catalog compilation if any parameters are invalid. See **puppetlabs-ntp** for an example.
4. Next lines, if applicable: Should declare local variables and perform variable munging.
5. Next lines: Should declare resource defaults.
6. Next lines: Should override resources if necessary.

The following example follows the recommended style.
In `init.pp`:

- The `myservice` class installs packages, ensures the state of `myservice`, and creates a tempfile with given content. If the tempfile contains digits, they are filtered out.
- `@param service_ensure` the wanted state of services.
- `@param package_list` the list of packages to install, at least one must be given, or an error of unsupported OS is raised.
- `@param tempfile_contents` the text to be included in the tempfile, all digits are filtered out if present.

```
class myservice (
  Enum['running', 'stopped'] $service_ensure,
  String                     $tempfile_contents,
  Optional[Array[String[1]]] $package_list = undef,
) {
```

- Rather than just saying that there was a type mismatch for `$package_list`, this example includes an additional assertion with an improved error message. The list can be "not given", or have an empty list of packages to install. An assertion is made that the list is an array of at least one String, and that the String is at least one character long.

```
    assert_type(Array[String[1], 1], $package_list) |$expected, $actual| {
      fail("Module ${module_name} does not support ${facts['os']['name']} as the l
    }

    package { $package_list:
      ensure => present,
    }

    file { "/tmp/${variable}":
      ensure   => present,
      contents => regsubst($tempfile_contents, '\d', '', 'G'),
      owner    => '0',
      group    => '0',
      mode     => '0644',
    }

    service { 'myservice':
      ensure    => $service_ensure,
      hasstatus => true,
    }

    Package[$package_list] -> Service['myservice']
  }
```

In `hiera.yaml`: The default values can be merged if you want to extend with additional packages. If not, use `default_hierarchy` instead of `hierarchy`.

```
---
version: 5
defaults:
  data_hash: yaml_data

hierarchy:
- name: 'Per Operating System'
  path: "os/%{os.name}.yaml"
- name: 'Common'
  path: 'common.yaml'
```

In `data/common.yaml`:

```
myservice::service_ensure: running
```

In `data/os/centos.yaml`:

```
myservice::package_list:
- 'myservice-centos-package'
```

In `data/os/solaris.yaml`:

```
myservice::package_list:
- 'myservice-solaris-package1'
- 'myservice-solaris-package2'
```

Public and private

Split your module into public and private classes and defined types where possible. Public classes or defined types should contain the parts of the module meant to be configured or customized by the user, while private classes should contain things you do not expect the user to change via parameters. Separating into public and private classes or defined types helps build reusable and readable code.

Help indicate to the user which classes are which by making sure all public classes have complete comments and denoting public and private classes in your documentation. Use the documentation tags “@api private” and “@api public” to make this clear. For complete documentation recommendations, see the Modules section.

Chaining arrow syntax

Most of the time, use **relationship metaparameters** rather than **chaining arrows**. When you have many **interdependent or order-specific items**, chaining syntax may be used. A chain operator should appear on the same line as its right-hand operand. Chaining arrows must be used left to right.

Good: Points left to right:

```
Package['httpd'] -> Service['httpd']
```

Good: On the line of the right-hand operand:

```
Package['httpd']
-> Service['httpd']
```

Bad: Arrows are not all pointing to the right:

```
Service['httpd'] <- Package['httpd']
```

Bad: Must be on the right-hand operand's line:

```
Package['httpd'] ->
Service['httpd']
```

Nested classes or defined types

Don't define classes and defined resource types within other classes or defined types. Declare them as close to node scope as possible. If you have a class or defined type which requires another class or defined type, put graceful failures in place if those required classes or defined types are not declared elsewhere.

Bad:

```
class apache {
  class ssl { ... }
}
```

Bad:

```
class apache {
  define config() { ... }
}
```

Display order of parameters

In parameterized class and defined resource type definitions, you can list required parameters before optional parameters (that is, parameters with defaults). Required parameters are parameters that are not set to anything, including undef. For example, parameters such as passwords or IP addresses might not have reasonable default values.

You can also group related parameters, order them alphabetically, or in the order you encounter them in the code. How you order parameters is personal preference.

Note that treating a parameter like a namevar and defaulting it to `$title` or `$name` does not make it a required parameter. It should still be listed following the order recommended here.

Good:

```
class dhcp (
  $dnsdomain,
  $nameservers,
  $default_lease_time = 3600,
  $max_lease_time     = 86400,
) {}
```

Bad:

```
class ntp (
  $options   = "iburst",
  $servers,
  $multicast = false,
) {}
```

Parameter defaults

Adding default values to the parameters in classes and defined types makes your module easier to use. Use Hiera data in your module to set parameter defaults. See **Defining classes** for details about setting parameter defaults with Hiera data. In simple cases, you can also specify the default values directly in the class or defined type.

Be sure to declare the data type of parameters, as this provides automatic type assertions.

Good: Parameter defaults set in the class with references to Hiera data:

```
class my_module (
  String $source,
  String $config,
) {
  # body of class
}
```

A `hiera.yaml` in the root of the module sets the hierarchy for assigning defaults:

```
---
version: 5
default_hierarchy:
- name: 'defaults'
  path: 'defaults.yaml'
  data_hash: yaml_data
```

And the file `data/defaults.yaml` specifies the actual default values:

```
my_module::source: 'default source value'
my_module::config: 'default config value'
```

This example places the values in the defaults hierarchy, which means that the defaults are not merged into overriding values. To merge the defaults into those values, change the `default_hierarchy` to `hierarchy`.

If you are maintaining old code created prior to Puppet 4.9, you might encounter the use of a `params.pp` pattern. This pattern makes maintenance and troubleshooting difficult — refactor such code to use the Hiera data-in-modules pattern instead. See **Adding Hiera data to a module** for a detailed example showing how to replace `params.pp` with data.

Bad: `params.pp`

```
class my_module (
  String $source = $my_module::params::source,
  String $config = $my_module::params::config,
) inherits my_module::params {
  # body of class
}
```

Exported resources

Exported resources should be opt-in rather than opt-out. Your module should not be written to use exported resources to function by default unless it is expressly required.

When using exported resources, name the property `collect_exported`.

Exported resources should be exported and collected selectively using a **search expression**, ideally allowing user-defined tags as parameters so tags can be used to selectively collect by environment or custom fact.

Good:

```
define haproxy::frontend (
  $ports             = undef,
  $ipaddress         = [$::ipaddress],
  $bind              = undef,
  $mode              = undef,
  $collect_exported = false,
  $options           = {
    'option' => [
      'tcplog',
    ],
  },
) {
  # body of define
}
```

Parameter indentation and alignment

Parameters to classes or defined types must be uniformly indented in two spaces from the title. The equals sign should be aligned.

Good:

```
class profile::myclass (
  $var1    = 'default',
  $var2    = 'something else',
  $another = 'another default value',
) {

}
```

Good:

```
class ntp (
  Boolean                          $broadcastclient = false,
  Optional[Stdlib::Absolutepath] $config_dir      = undef,
  Enum['running', 'stopped']     $service_ensure  = 'running',
  String                           $package_ensure  = 'present',
  # ...
) {
# ...
}
```

Bad: Too many level of indentation:

```
class profile::myclass (
      $var1    = 'default',
      $var2    = 'something else',
      $another = 'another default value',
) {

}
```

Bad: No indentation:

```
class profile::myclass (
$var1    = 'default',
$var2    = 'something else',
  $another = 'another default value',
) {

}
```

Bad: Misaligned equals sign:

```
class profile::myclass (
  $var1 = 'default',
  $var2  = 'something else',
  $another = 'another default value',
) {

}
```

Class inheritance

In addition to scope and organization, there are some additional guidelines for handling classes in your module.

Don't use class inheritance; use data binding instead of `params.pp` pattern. Inheritance is used only for `params.pp`, which is not recommended in Puppet 4.

If you use inheritance for maintaining older modules, do not use it across module namespaces. To satisfy cross-module dependencies in a more portable way, include statements or relationship declarations. Only use class inheritance for `myclass::params` parameter defaults. Accomplish other use cases by adding parameters or conditional logic.

Good:

```
class ssh { ... }

class ssh::client inherits ssh { ... }

class ssh::server inherits ssh { ... }
```

Bad:

```
class ssh inherits server { ... }

class ssh::client inherits workstation { ... }

class wordpress inherits apache { ... }
```

Public modules

When declaring classes in publicly available modules, use `include`, `contain`, or `require` rather than class resource declaration. This avoids duplicate class declarations and vendor lock-in.

Type signatures

We recommend always using type signatures for class and defined type parameters. Keep the parameters and = signs aligned.

When dealing with very long type signatures, you can define type aliases and use short definitions. Good naming of aliases can also serve as documentation, making your code easier to read and understand. Or, if necessary, you can turn the 140 line character limit off. For more information on type signatures, see **the `Type` data type**.

Related information

- **Modules**

Variables

Reference variables in a clear, unambiguous way that is consistent with the Puppet style.

Referencing facts

When referencing facts, prefer the `$facts` hash to plain top-scope variables (such as `$::operatingsystem`).

Although plain top-scope variables are easier to write, the `$facts` hash is clearer, easier to read, and distinguishes facts from other top-scope variables.

Namespacing variables

When referencing top-scope variables other than facts, explicitly specify absolute namespaces for clarity and improved readability. This includes top-scope variables set by the node classifier and in the main manifest.

This is not necessary for:

- the `$facts` hash.
- the `$trusted` hash.
- the `$server_facts` hash.

These special variable names are protected; because you cannot create local variables with these names, they always refer to top-scope variables.

Good:

```
$facts['operatingsystem']
```

Bad:

```
$::operatingsystem
```

Very bad:

```
$operatingsystem
```

Variable format

When defining variables you must only use numbers, lowercase letters, and underscores. Do not use upper-case letters within a word, such as “CamelCase”, as it introduces inconsistency in style. You must not use dashes, as they are not syntactically valid.

Good:

```
$server_facts
$total_number_of_entries
$error_count123
```

Bad:

```
$serverFacts
$totalNumberOfEntries
$error-count123
```

Conditionals

Conditional statements should follow Puppet code guidelines.

Simple resource declarations

Avoid mixing conditionals with resource declarations. When you use conditionals for data assignment, separate conditional code from the resource declarations.

Good:

```
$file_mode = $facts['operatingsystem'] ? {
  'debian' => '0007',
  'redhat' => '0776',
   default => '0700',
}

file { '/tmp/readme.txt':
  ensure  => file,
  content => "Hello World\n",
  mode    => $file_mode,
}
```

Bad:

```
file { '/tmp/readme.txt':
  ensure  => file,
  content => "Hello World\n",
  mode    => $facts['operatingsystem'] ? {
    'debian' => '0777',
    'redhat' => '0776',
    default  => '0700',
  }
}
```

Defaults for case statements and selectors

Case statements must have default cases. If you want the default case to be "do nothing," you must include it as an explicit `default: {}` for clarity's sake.

Case and selector values must be enclosed in quotation marks.

Selectors should omit default selections only if you explicitly want catalog compilation to fail when no value matches.

Good:

```
case $facts['operatingsystem'] {
  'centos': {
    $version = '1.2.3'
  }
  'solaris': {
    $version = '3.2.1'
  }
  default: {
    fail("Module ${module_name} is not supported on ${::operatingsystem}")
  }
}
```

When setting the default case, keep in mind that the default case should cause the catalog compilation to fail if the resulting behavior cannot be predicted on the platforms the module was built to be used on.

Conditional statement alignment

When using if/else statements, align in the following way:

```
if $something {
  $var = 'hour'
} elsif $something_else {
  $var = 'minute'
} else {
  $var = 'second'
}
```

For more information on if/else statements, see **Conditional statements and expressions**.

Modules

Develop your module using consistent code and module structures to make it easier to update and maintain.

Versioning

Your module must be versioned, and have metadata defined in the `metadata.json` file.

We recommend semantic versioning.

Semantic versioning, or **SemVer**, means that in a version number given as x.y.z:

- An increase in 'x' indicates major changes: backwards incompatible changes or a complete rewrite.
- An increase in 'y' indicates minor changes: the non-breaking addition of new features.
- An increase in 'z' indicates a patch: non-breaking bug fixes.

Module metadata

Every module must have metadata defined in the `metadata.json` file.

Your metadata should follow the following format:

```
{
  "name": "examplecorp-mymodule",
  "version": "0.1.0",
  "author": "Pat",
  "license": "Apache-2.0",
  "summary": "A module for a thing",
  "source": "https://github.com/examplecorp/examplecorp-mymodule",
  "project_page": "https://github.com/examplecorp/examplecorp-mymodule",
  "issues_url": "https://github.com/examplecorp/examplecorp-mymodules/issues",
  "tags": ["things", "stuff"],
  "operatingsystem_support": [
    {
      "operatingsystem":"RedHat",
      "operatingsystemrelease": [
        "5.0",
        "6.0"
      ]
    },
    {
      "operatingsystem": "Ubuntu",
      "operatingsystemrelease": [
        "12.04",
        "10.04"
     ]
    }
  ],
  "dependencies": [
    { "name": "puppetlabs/stdlib", "version_requirement": ">= 3.2.0 <5.0.0" },
    { "name": "puppetlabs/firewall", "version_requirement": ">= 0.4.0 <5.0.0" },
  ]
}
```

For additional information regarding the `metadata.json` format, see **Adding module metadata in metadata.json**.

Dependencies

Hard dependencies must be declared explicitly in your module's metadata.json file.

Soft dependencies should be called out in the README.md, and must not be enforced as a hard requirement in your metadata.json. A soft dependency is a dependency that is only required in a specific set of use cases. For an example, see the **rabbitmq module**.

Your hard dependency declarations should not be unbounded.

README

Your module should have a README in `.md` (or `.markdown`) format. READMEs help users of your module get the full benefit of your work.

The **Puppet README template** offers a basic format you can use. If you create modules with Puppet Development Kit or the `puppet module generate` command, the generated README includes the template. Using the .md/.markdown format allows your README to be parsed and displayed by Puppet Strings, GitHub, and the Puppet Forge.

You can find thorough, detailed information on writing a great README in **Documenting modules**, but in general your README should:

- Summarize what your module does.

- Note any setup requirements or limitations, such as "This module requires the puppetlabs-apache module and only works on Ubuntu."
- Note any part of a user's system the module might impact (for example, "This module overwrites everything in animportantfile.conf.").
- Describe how to customize and configure the module.
- Include usage examples and code samples for the common use cases for your module.

Documenting Puppet code

Use **Puppet Strings** code comments to document your Puppet classes, defined types, functions, and resource types and providers. Strings processes the README and comments from your code into HTML or JSON format documentation. This allows you and your users to generate detailed documentation for your module.

Include comments for each element (classes, functions, defined types, parameters, and so on) in your module. If used, comments must precede the code for that element. Comments should contain the following information, arranged in this order:

- A description giving an overview of what the element does.
- Any additional information about valid values that is not clear from the data type. For example, if the data type is [String], but the value must specifically be a path.
- The default value, if any, for that element,

Multiline descriptions must be uniformly indented by at least one space:

```
# @param config_epp Specifies a file to act as a EPP template for the config file.
#  Valid options: a path (absolute, or relative to the module path). Example value:
#  'ntp/ntp.conf.epp'. A validation error is thrown if you supply both this param **
#  the `config_template` param.
```

If you use Strings to document your module, include information about Strings in the Reference section of your README so that your users know how to generate the documentation. See **Puppet Strings** documentation for details on usage, installation, and correctly writing documentation comments.

If you do not include Strings code comments, you should include a Reference section in your README with a complete list of all classes, types, providers, defined types, and parameters that the user can configure. Include a brief description, the valid options, and the default values (if any).

For example, this is a parameter for the `ntp` module's `ntp` class: `package_ensure`:

```
Data type: String.

Whether to install the NTP package, and what version to install. Values: 'present',

Default value: 'present'.
```

For more details and examples, see the **module documentation guide**.

CHANGELOG

Your module should include a change log file called `CHANGELOG.md` or `.markdown`. Your change log should:

- Have entries for each release.
- List bugfixes and features included in the release.
- Specifically call out backwards-incompatible changes.

Examples

In the `/examples` directory, include example manifests that demonstrate major use cases for your module.

```
modulepath/apache/examples/{usecase}.pp
```

The example manifest should provide a clear example of how to declare the class or defined resource type. It should also declare any classes required by the corresponding class to ensure `puppet apply` works in a limited, standalone manner.

Testing

Use one or more of the following community tools for testing your code and style:

- **puppet-lint** tests your code for adherence to the style guidelines.
- **metadata-json-lint** tests your `metadata.json` for adherence to the style guidelines.
- For testing your module, we recommend rspec. **rspec-puppet** can help you write rspec tests for Puppet.

puppet Puppet automates the delivery and operation of the software that powers some of the biggest brands in the world.