Best Practices: Puppet Module

## Modules Layout

* Module Name
  + manifests
  + files
  + templates
  + lib
  + facts.d
  + tests
  + spec

|  |  |  |  |
| --- | --- | --- | --- |
| Directory | Sub-Directory | Example | Description |
| Module Name/ |  | standards | Module Directory, the name must match the module-name. |
|  | manifests/ |  | Contains all of the manifests in the module. |
|  | manifests/init.pp | class standards (...){...} | Contains all of the manifests in the module. |
|  | manifests/profile\_d.pp | class standards::profile\_d (...){...} | Contains a class definition. **This class’s name must match the module’s name.** |
|  | manifests/post\_receive.pp | **define**standards::post\_receive (...){...} | Contains a defined type named **my\_module::my\_defined\_type**. |
|  | manifests/**implementation/** |  | This directory’s name affects the class names beneath it. |
|  | manifests/**implementation/**foo.pp | class my\_module::implementation::foo(...){...} | Contains a class named **my\_module::implementation::foo**. |
|  | manifests/**implementation/**bar.pp | class my\_module::implementation::bar(...){...} | Contains a class named **my\_module::implementation::bar**. |
|  | files/ |  | Contains static files, which managed nodes can download. |
|  | files/service.conf | source => puppet:///modules/my\_module/service.conf | Access the static file from source Attribut. |
|  | files/service.conf | content => file('my\_module/service.conf') | Access the static file from content Attribut with the *file()-function*. |
|  | templates/ |  | Contains templates, which the module’s manifests can use. |
|  | templates/component.erb |  | A manifest can render this template with template('my\_module/component.erb'). |
|  | lib/ |  | Contains plugins, like custom facts and custom resource types. |
|  | facts.d/ |  | Contains external facts, which are an alternative to Ruby-based custom facts. |
|  | tests/ |  | Contains examples showing how to declare the module’s classes and defined types. |
|  | tests/init.pp |  |  |
|  | tests/profile\_d.pp |  | Each *class* or *type* should have an example in the tests directory. |
|  | spec/ |  | Contains spec tests for any plugins in the lib directory. |

## Writing Modules

**Checklist for an reuseable module:** - The classes, defined types, and plugins in a module should all be related. - The module should aim to be as self-contained as possible. - Manifests in one module should never reference files or templates stored in another module. - When possible, it’s best to isolate “super-classes” that declare many other classes in a local “site” module. - Usage of classes and defined types, in addition to the built-in managed types(file, service, package,...), is very helpful towards having a managable Puppet infrastructure. - One repository per puppet module. - All modules in a single repository - Remember – your module should just accept a value and use it somewhere. Don’t get TOO smart with your component module – leave the logic for other places. - no crossreferences in an module to an other module. - The best way I’ve found to determine if your module is ‘generic enough’ is if I asked you TODAY to give me your module, would you give it to me, or would you be worried that there was some company-specific data locked in there? - Do all Hiera lookups in the profile. - Do all Hiera lookups with having NO DEFAULT VALUES – this is BY DESIGN! - For most people, their Hiera data is PROBABLY located in a separate repository as their Puppet module data. - Use parameterized class declarations and explicitly pass values you care about.

**Style** For recommendations on syntax and formatting, follow the [The Puppet Language Style Guide](https://docs.puppetlabs.com/guides/style_guide.html)

**Naming Conventions** When naming classes, a class that disables ssh should be inherited from the ssh class and be named "ssh::disabled".

**Classes Vs Defined Types** *Classes:* - Classes are not to be thought of in the ‘object oriented’ meaning of a class. This means a machine belongs to a particular class of machine. - Example: A generic webserver would be a class. You would include that class as part of any node that needed to be built as a generic webserver.

*Defined Types:* - Defined types on the other hand can have many instances on a machine, and can encapsulate classes and other resources. - Example: a *defined type* which manages iptables rules, creates one-rule per defined type instance.

**Generating Module (-Layout Structure)**

Generating the Module Structure via commandline: puppet module generate <USERNAME>-<MODULE NAME>

**Example of an Generated Module:**

$ puppet module generate examplecorp-mymodule  
  
We need to create a metadata.json file for this module. Please answer the  
following questions; if the question is not applicable to this module, feel free  
to leave it blank.  
  
Puppet uses Semantic Versioning (semver.org) to version modules.  
What version is this module? [0.1.0]  
--> 0.1.0  
  
Who wrote this module? [examplecorp]  
--> Pat  
  
What license does this module code fall under? [Apache 2.0]  
--> Apache 2.0  
  
How would you describe this module in a single sentence?  
--> It examples with Puppet.  
  
Where is this module's source code repository?  
--> https://github.com/examplecorp/examplecorp-mymodule  
  
Where can others go to learn more about this module?  
--> https://forge.puppetlabs.com/examplecorp/mymodule  
  
Where can others go to file issues about this module?  
-->  
  
  
{  
 "name": "examplecorp-mymodule",  
 "version": "0.1.0",  
 "author": "Pat",  
 "summary": "It examples with Puppet.",  
 "license": "Apache 2.0",  
 "source": "https://github.com/examplecorp/examplecorp-mymodule",  
 "project\_page": "(https://forge.puppetlabs.com/examplecorp/mymodule)",  
 "issues\_url": null,  
 "dependencies": [  
 {  
 "name": "puppetlabs-stdlib",  
 "version\_range": ">= 1.0.0"  
 }  
 ]  
}  
  
  
About to generate this metadata; continue? [n/Y]  
--> Y  
  
Notice: Generating module at /Users/Pat/Development/examplecorp-mymodule...  
Notice: Populating ERB templates...  
Finished; module generated in examplecorp-mymodule.  
examplecorp-mymodule/manifests  
examplecorp-mymodule/manifests/init.pp  
examplecorp-mymodule/metadata.json  
examplecorp-mymodule/Rakefile  
examplecorp-mymodule/README.md  
examplecorp-mymodule/spec  
examplecorp-mymodule/spec/classes  
examplecorp-mymodule/spec/classes/init\_spec.rb  
examplecorp-mymodule/spec/spec\_helper.rb  
examplecorp-mymodule/tests  
examplecorp-mymodule/tests/init.pp

**Module Parameters**

* API to your module: top-level class apache, standards and **not** apache::install.
  + IDEALLY, they’re the ONLY THING that a user needs to modify when using your module.
* Whenever possible, it should be the case that a user need ONLY interact with the top-level class when using your module.
  + Of course, defined resource types like apache::vhost are used on an ad-hoc basis, and thus are the exception here.

**Inherit the ::params class**

* The idea is that the ::params class is the one-stop-shop to see where a variable is set.
* To get access to a variable that’s set in a Puppet class: you have to declare the class.

*Declare a class: useing inherit from that class*

class apache (  
 $port = $apache::params::port,  
 $user = $apache::params::user,  
) inherits apache::params {  
...  
}

*Declare a class: useing the include() function*

class apache {  
include (apache::params)  
}

* When you declare a class that has both variables AND resources, those resources get put into the catalog, which means that Puppet ENFORCES THE STATE of those resources.
* What if you only needed a variable’s value and didn’t want to enforce the rest of the resources in that class?
  + There’s no good way in Puppet to do that.
* Finally, when you inherit from a class in Puppet that has assigned variable values, you ALSO get access to those variables in the parameter definition section of your class.
* class apache ( $port = $apache::params::port, $user = $apache::params::user, ) inherits apache::params { ... }

See how I set the default value of $apache::port to $apache::params::port? I could only access the value of the variable $apache::params::port in that section by inheriting from the apache::params class. I couldn’t insert include apache::params below that section and be allowed access to the variable up in the parameter defaults section (due to the way that Puppet parses classes).

**FOR THIS REASON, THIS IS THE ONLY RECOMMENDED USAGE OF INHERITANCE IN PUPPET!**,

**NOTE: Data in Modules** – There’s a ‘Data in Modules’ pattern out there that attempts to eliminate the ::params class. [I wrote about it in a previous post](http://garylarizza.com/blog/2013/12/08/when-to-hiera/), and I recommend you read that post for more info (it’s near the bottom).

**Do NOT do Hiera lookups in your component modules!**

The pattern that I’ll be pushing DOES INDEED use Hiera, BUT it confines all Hiera calls to a higher-level wrapper class we call a **profile**.

The reasons for NOT using Hiera in your module are:

* By doing Hiera calls at a higher level, you have a greater visibility on exactly what parameters were set by Hiera and which were set explicitly or by default values.
* By doing Hiera calls elsewhere, your module is backwards-compatible for those folks who are NOT using Hiera

Remember – your module should just accept a value and use it somewhere. Don’t get TOO smart with your component module – leave the logic for other places.

**Hiera Usage Example:**

class profiles::wordpress {  
  
 ## Hiera lookups  
 $site\_name = hiera('profiles::wordpress::site\_name')  
 $wordpress\_user\_password = hiera('profiles::wordpress::wordpress\_user\_password')  
 $mysql\_root\_password = hiera('profiles::wordpress::mysql\_root\_password')  
 $wordpress\_db\_host = hiera('profiles::wordpress::wordpress\_db\_host')  
 $wordpress\_db\_name = hiera('profiles::wordpress::wordpress\_db\_name')  
 $wordpress\_db\_password = hiera('profiles::wordpress::wordpress\_db\_password')  
 $wordpress\_user = hiera('profiles::wordpress::wordpress\_user')  
 $wordpress\_group = hiera('profiles::wordpress::wordpress\_group')  
 $wordpress\_docroot = hiera('profiles::wordpress::wordpress\_docroot')  
 $wordpress\_port = hiera('profiles::wordpress::wordpress\_port')  
  
 ## Create user  
 group { 'wordpress':  
 ensure => present,  
 name => $wordpress\_group,  
 }  
 user { 'wordpress':  
 ensure => present,  
 gid => $wordpress\_group,  
 password => $wordpress\_user\_password,  
 name => $wordpress\_group,  
 home => $wordpress\_docroot,  
 }  
  
 ## Configure mysql  
 class { 'mysql::server':  
 root\_password => $wordpress\_root\_password,  
 }  
  
 class { 'mysql::bindings':  
 php\_enable => true,  
 }  
  
 ## Configure apache  
 include apache  
 include apache::mod::php  
 apache::vhost { $::fqdn:  
 port => $wordpress\_port,  
 docroot => $wordpress\_docroot,  
 }  
  
 ## Configure wordpress  
 class { '::wordpress':  
 install\_dir => $wordpress\_docroot,  
 db\_name => $wordpress\_db\_name,  
 db\_host => $wordpress\_db\_host,  
 db\_password => $wordpress\_db\_password,  
 }  
}

**Use parameterized class declarations and explicitly pass values you care about**

**An annoying Puppet bug – top-level class declarations and profiles**

*File: modules/profiles/wordpress.pp*

class wordpress {  
 include wordpress # means-> profiles::wordpress  
}

-> We **expected** to include the module: wordpress -> modules/wordpress from an class wordpress of the module profiles. -> Puppet will include the class profiles::wordpress from the current scope profiles (module scope). -> The Solution is to use class declaration with top-level-scope prefix :: to include the module-class wordpress and not the class profiles::wordpress. All Module-Classes are stored at the top-level-scope ::.

*File: modules/profiles/wordpress.pp*

class wordpress {  
 class { '::wordpress':  
 install\_dir => $wordpress\_docroot,  
 db\_name => $wordpress\_db\_name,  
 db\_host => $wordpress\_db\_host,  
 db\_password => $wordpress\_db\_password,  
 }  
}

Our *Profile Class Naming Convention* says that we name profile-classes as the technology which they bring in. For an Wordpress Installation, we declare an Profile Class with the name "Wordpress" form which we include the module "Wordpress" to install an wordpress on the node.

When you use profiles and roles, you’ll need to do this namespacing trick when declaring classes because you’re frequently going to have a profile::<sometech> that will declare the <sometech> top-level class.

**What does and DOESN’T go in Hiera?**

***Data in site-specific Hiera datastore*** - Business-specific data (i.e. internal NTP server, VIP address, per-environment java application versions, etc…) - Sensitive data - Data that you don’t want to share with anyone else

***Data that does NOT go in the site-specific Hiera datastore*** - OS-specific data - Data that EVERYONE ELSE who uses this module will need to know (paths to config files, package names, etc…)

Basically, if I ask you if I can publish your module to the Puppet Forge, and you object because it has business-specific or sensitive data in it, then you probably need to pull that data out of the module and put it in Hiera.

***The recommendations that I give when I go on-site with Puppet users is the following:***

* Use Roles/Profiles to create wrapper-classes for class declaration
* Do ALL Hiera lookups for site-specific data inside your ‘Profile’ wrapper classes
* All module-specific data (like paths to config files, names of packages to install, etc…) should be kept in the module in the params class
* All ‘Role’ wrapper classes should just include ‘Profile’ wrapper classes – nothing else.

https://www.devco.net/archives/2013/12/08/better-puppet-modules-using-hiera-data.php https://github.com/ripienaar/puppet-module-data https://forge.puppetlabs.com/ripienaar/module\_data https://www.devco.net/archives/2013/12/09/the-problem-with-params-pp.php https://www.devco.net/archives/2012/12/13/simple-puppet-module-structure-redux.php

https://docs.puppetlabs.com/guides/module\_guides/bgtm.html http://confreaks.tv/videos/puppetconf2012-puppet-modules-for-fun-and-profit

**What the hell does and does NOT belong in Hiera?**

**Puppet data models**

**The params class pattern**

*File: puppetlabs-mysql/manifests/server.pp*

class mysql::server (  
 $config\_file = $mysql::params::config\_file,  
 $manage\_config\_file = $mysql::params::manage\_config\_file,  
 $old\_root\_password = $mysql::params::old\_root\_password,  
 $override\_options = {},  
 $package\_ensure = $mysql::params::server\_package\_ensure,  
 $package\_name = $mysql::params::server\_package\_name,  
 $purge\_conf\_dir = $mysql::params::purge\_conf\_dir,  
 $remove\_default\_accounts = false,  
 $restart = $mysql::params::restart,  
 $root\_group = $mysql::params::root\_group,  
 $root\_password = $mysql::params::root\_password,  
 $service\_enabled = $mysql::params::server\_service\_enabled,  
 $service\_manage = $mysql::params::server\_service\_manage,  
 $service\_name = $mysql::params::server\_service\_name,  
 $service\_provider = $mysql::params::server\_service\_provider,  
 # Deprecated parameters  
 $enabled = undef,  
 $manage\_service = undef  
) inherits mysql::params {  
  
 ## Puppet goodness goes here  
}

*File: puppetlabs-mysql/manifests/params.pp*

class mysql::params {  
 case $::osfamily {  
 'RedHat': {  
 if $::operatingsystem == 'Fedora' and (is\_integer($::operatingsystemrelease) and $::operatingsystemrelease >= 19 or $::operatingsystemrelease == "Rawhide") {  
 $client\_package\_name = 'mariadb'  
 $server\_package\_name = 'mariadb-server'  
 } else {  
 $client\_package\_name = 'mysql'  
 $server\_package\_name = 'mysql-server'  
 }  
 $basedir = '/usr'  
 $config\_file = '/etc/my.cnf'  
 $datadir = '/var/lib/mysql'  
 $log\_error = '/var/log/mysqld.log'  
 $pidfile = '/var/run/mysqld/mysqld.pid'  
 $root\_group = 'root'  
 }  
  
 'Debian': {  
 ## More parameters defined here  
 }  
 }  
}

**Pros:**

* All conditional logic is in a single class
* You always know which class to seek out if you need to change any of the logic used to determine a variable’s value
* You can use the include function because parameters for each class will be defaulted to the values that came out of the params class
* If you need to override the value of a particular parameter, you can still use the parameterized -class declaration syntax to do so
* Anyone using Puppet version 2.6 or higher can use it (i.e. anyone who’s been using Puppet since about 2010).

**Cons:**

* Conditional logic is repeated in every module
* You will need to use inheritance to inherit parameter values in each subclass
* It’s another place to look if you ALSO use Hiera inside the module
* Data is inside the manifest, so business logic is also inside params.pp

**Hiera defaults pattern**

*File: puppetlabs-mysql/manifests/server.pp*

class mysql::server (  
 $config\_file = hiera('mysql::params::config\_file', 'default value'),  
 $manage\_config\_file = hiera('mysql::params::manage\_config\_file', 'default value'),  
 $old\_root\_password = hiera('mysql::params::old\_root\_password', 'default value'),  
 ## Repeat the above pattern  
) {  
  
 ## Puppet goodness goes here  
}

**Pros:**

* All data is locked up in Hiera (and its multiple backends)
* Default values can be provided if a Hiera lookup fails

**Cons:**

* You need to have Hiera installed, enabled, and configured to use this pattern
* All data, including non-business logic, is in Hiera
* If you use the default value, data could either come from Hiera OR the default (multiple places to look when debugging)

**Hybrid data model**

*File: puppetlabs-mysql/manifests/server.pp*

class mysql::server (  
 $config\_file = hiera('mysql::params::config\_file', $mysql::params::config\_file),  
 $manage\_config\_file = hiera('mysql::params::manage\_config\_file', $mysql::params::manage\_config\_file),  
 $old\_root\_password = hiera('mysql::params::old\_root\_password', $mysql::params::old\_root\_password),  
 ## Repeat the above pattern  
) inherits mysql::params {  
  
}

**Pros:**

* Data is sought from Hiera first and then defaulted back to the params class parameter
* Keep non-business logic (i.e. OS specific data) in the params class and business logic in Hiera
* Added benefits of both models

**Cons:**

* Where did the variable get set – Hiera or the params class? Debugging can be hard
* Requires Hiera to be setup to use the module
* If you fudge a variable name in Hiera, you get the params class default – see Con #1

**Hiera data bindings in Puppet 3.x.x**

*File: parameterized class declaration*

class { 'apache':  
 package\_name => 'httpd',  
}

If a parameter was not passed with the parameterized class syntax (like the ‘package\_name’ parameter above’), Puppet would then look for a default value inside the class definition (i.e. the below:).

*File: parameter default in a class definition*

class ntp (  
 $ntpserver = 'default.ntpserver.org'  
) {  
 # Use $ntpserver in a file declaration...  
}

If the value of ‘ntpserver’ wasn’t passed with a parameterized class declaration, then the value would be set to ‘default.ntpserver.org’, since that’s the default set in the above class definition.

Failing both of these conditions, Puppet would throw a parse error and say that it couldn’t determine a value for a class parameter.

As of Puppet 3.0.0, Puppet will now do a Hiera lookup for the fully namespaced value of a class parameter.

**Roles and Profiles**

*File: profiles/manifests/wordpress.pp*

class profiles::wordpress {  
 # Data Lookups  
 $site\_name = hiera('profiles::wordpress::site\_name')  
 $wordpress\_user\_password = hiera('profiles::wordpress::wordpress\_user\_password')  
 $mysql\_root\_password = hiera('profiles::wordpress::mysql\_root\_password')  
 $wordpress\_db\_host = hiera('profiles::wordpress::wordpress\_db\_host')  
 $wordpress\_db\_name = hiera('profiles::wordpress::wordpress\_db\_name')  
 $wordpress\_db\_password = hiera('profiles::wordpress::wordpress\_db\_password')  
  
 ## Create user  
 group { 'wordpress':  
 ensure => present,  
 }  
 user { 'wordpress':  
 ensure => present,  
 gid => 'wordpress',  
 password => $wordpress\_user\_password,  
 home => '/var/www/wordpress',  
 }  
  
 ## Configure mysql  
 class { 'mysql::server':  
 root\_password => $wordpress\_root\_password,  
 }  
  
 class { 'mysql::bindings':  
 php\_enable => true,  
 }  
  
 ## Configure apache  
 include apache  
 include apache::mod::php  
}  
  
## Continue with declarations...

*File: roles/manifests/frontend.pp*

class roles::frontend {  
 include profiles::mysql  
 include profiles::apache  
 include profiles::java  
 include profiles::jboss  
 # include more profiles...  
}

**Pros:**

* Hiera data lookups are confined to a wrapper class OUTSIDE of the component modules (like mysql, apache, java, etc…)
* Data lookups for parameters containing business logic are done with Hiera
* Non-business specific data is pulled from the module (i.e. the params class)
* Wrapper modules can be ‘included’ with the include function, helping to eliminate multiple class declarations using the parameterized class declaration syntax
* Component modules are backward-compatible to Puppet 2.6 while wrapper modules still get to use a modern data lookup mechanism (Hiera)
* Component modules do NOT contain any business specific logic, which means they’re portable

**Cons:**

* Hiera must be setup to use the wrapper modules
* Wrapper modules add another debug path for variable data
* Wrapper modules add another layer of abstraction

**Data in Puppet Modules**

***Sure, but this is outside of Puppet and you’re losing visibility inside Puppet with your data***

You’re already doing that if you’re using the params class. In this case, visibility is moved to YAML files instead of separate Puppet classes.

**Setting it up**

[root@linux modules]# puppet module install ripienaar/module\_data  
Notice: Preparing to install into /etc/puppetlabs/puppet/modules ...  
Notice: Downloading from https://forge.puppetlabs.com ...  
Notice: Installing -- do not interrupt ...  
/etc/puppetlabs/puppet/modules  
└── ripienaar-module\_data (v0.0.1)  
  
  
[root@linux modules]# tree mysql/  
mysql/  
├── data  
│ ├── hiera.yaml  
│ └── RedHat.yaml  
└── manifests  
 └── init.pp  
  
[root@linux modules]# puppet config print modulepath  
/etc/puppetlabs/puppet/modules:/opt/puppet/share/puppet/modules

*File: mysql/data/hiera.yaml*

:hierarchy:  
 - "%{::osfamily}"

*File: mysql/data/RedHat.yaml*

---  
mysql::config\_file: '/path/from/data\_in\_modules'  
mysql::manage\_config\_file: true  
mysql::old\_root\_password: 'password\_from\_data\_in\_modules'

*File: mysql/manifests/init.pp*

class mysql (  
 $config\_file = 'module\_default',  
 $manage\_config\_file = 'module\_default',  
 $old\_root\_password = 'module\_default'  
) {  
 notify { "The value of config\_file: ${config\_file}": }  
 notify { "The value of manage\_config\_file: ${manage\_config\_file}": }  
 notify { "The value of old\_root\_password: ${old\_root\_password}": }  
}

**Testing data-in-modules**

[root@linux modules]# puppet apply -e 'include mysql'  
Notice: The value of config\_file: /path/from/data\_in\_modules  
Notice: /Stage[main]/Mysql/Notify[The value of config\_file: /path/from/data\_in\_modules]/message: defined 'message' as 'The value of config\_file: /path/from/data\_in\_modules'  
Notice: The value of manage\_config\_file: true  
Notice: /Stage[main]/Mysql/Notify[The value of manage\_config\_file: true]/message: defined 'message' as 'The value of manage\_config\_file: true'  
Notice: The value of old\_root\_password: password\_from\_data\_in\_modules  
Notice: /Stage[main]/Mysql/Notify[The value of old\_root\_password: password\_from\_data\_in\_modules]/message: defined 'message' as 'The value of old\_root\_password: password\_from\_data\_in\_modules'  
Notice: Finished catalog run in 0.62 seconds

**Testing outside of Puppet**

Currently not possible because hiera cmd-util uses one central configuration file /etc/puppet/hiera.yaml.

**Pros:**

* Parameters are defined in YAML and not Puppet DSL (i.e. you only need to know YAML and not the Puppet DSL)
* Adding parameters is as simple as adding another YAML file to the module
* Module authors provide module data that can be read by Puppet 3.x.x Hiera data bindings

**Cons:**

* Must be using Puppet 3.0.0 or higher
* Additional hierarchy and additional Hiera data file to consult when debugging
* Not (currently) an easy/straightforward way to use the hiera binary to test values
* Currently depends on a Puppet Forge module being installed on your system

**Data in site-specific Hiera datastore**

* Business-specific data (i.e. internal NTP server, VIP address, per-environment java application versions, etc…)
* Sensitive data
* Data that you don’t want to share with anyone else
* Data that does NOT go in the site-specific Hiera datastore

**OS-specific data**

* Data that EVERYONE ELSE who uses this module will need to know (paths to config files, package names, etc…)

**The recommendations that I give when I go on-site with Puppet users is the following:**

* Use Roles/Profiles to create wrapper-classes for class declaration
* Do ALL Hiera lookups for site-specific data inside your ‘Profile’ wrapper classes
* All module-specific data (like paths to config files, names of packages to install, etc…) should be kept in the module in the params class
* All ‘Role’ wrapper classes should just include ‘Profile’ wrapper classes – nothing else

**Better Puppet Modules Using Hiera Data**

**Defacto standard solution for implenting class Parameter and default-values with the *params class pattern***

*File: ntp/manifests/init.pp*

class ntp (  
 $config = $ntp::params::config,  
 $keys\_file = $ntp::params::keys\_file  
 ) inherits ntp::params {  
   
 file{$config:  
 ....  
 }  
}

*File: ntp/manifests/params.pp*

class ntp::params {  
 case $::osfamily {  
 'AIX': {  
 $config = "/etc/ntp.conf"  
 $keys\_file = '/etc/ntp.keys'  
 }  
   
 'Debian': {  
 $config = "/etc/ntp.conf"  
 $keys\_file = '/etc/ntp/keys'  
 }  
   
 'RedHat': {  
 $config = "/etc/ntp.conf"  
 $keys\_file = '/etc/ntp/keys'  
 }  
   
 default: {  
 fail("The ${module\_name} module is not supported on an ${::osfamily} based system.")  
 }  
 }  
}

**Rework this module around this proposed solution**

We have to create an new directory with the name data inside the module.

*File: ntp/manifests/init.pp*

class ntp ($config, $keysfile) {  
 validate\_absolute\_path($config)  
 validate\_absolute\_path($keysfile)  
   
 file{$config:  
 ....  
 }  
}

*File: ntp/data/hiera.yaml*

---  
:hierarchy:  
 - "%{::osfamily}"

*File: ntp/data/RedHat.yaml*

---  
ntp::config: /etc/ntp.conf  
ntp::keys\_file: /etc/ntp/keys

**Pros:**

* Users of the module could add a new OS without contributing back to the module or forking the module by simply providing similar data to the site specific hierarchy leaving the downloaded module 100% untouched!

**This helps the contributor to a public module:**

* Adding a new OS is easy, just drop in a new YAML file. This can be done with confidence as it will not break existing code as it will only be read on machines of the new OS. No complex case statements or 100s of braces to get right
* On a busy module when adding a new OS they do not have to worry about complex merge problems, working hard at rebasing or any git escoteria – they’re just adding a file.
* Syntactically it’s very easy, it’s just a YAML file. No complex case statements etc.
* The contributor does not have to worry about breaking other Operating Systems he could not test on like AIX here. The change is contained to machines for the new OS
* In large environments this help with change control as it’s just data – no logic changes

**This helps the maintainer of a module:**

* Module maintenance is easier when it comes to adding new Operating Systems as it’s simple single files
* Easier contribution reviews
* Fewer merge commits, less git magic needed, cleaner commit history
* The code is a lot easier to read and maintain. Fewer tests and validations are needed.

**This helps the user of a module:**

* Well written modules now properly support supplying all data from Hiera
* He has a single place to look for the overridable data
* When using a module that does not support his OS he can deploy it into his site and just provide data instead of forking it

-> http://forge.puppetlabs.com/ripienaar/module\_data -> https://www.devco.net/archives/2013/12/08/better-puppet-modules-using-hiera-data.php -> https://github.com/ripienaar/

-> +dependency: ripienaar/module\_data -> puppet module install ripienaar/module\_data

$package = $operatingsystem ? {  
 "redhat" => $operatingsystemrelease ? {  
 '5.6' => "ntp-foo",  
 default => "ntp",  
 },  
 #debian => "ntp", #redundant here...  
 default => "ntp",  
}

Would it work to put just ::operatingsystem below your ::operatingsystemrelease tier? Then you can have Debian defaults with per release override possibilities.

:hierarchy:  
 - "%{::operatingsystem}/%{::operatingsystemrelease}"  
 - "%{::operatingsystem}/common"