Puppet Best Practices

First, read through http://docs.puppetlabs.com/guides/best_practices.html and http://docs.puppetlabs.com/guides/style_guide.html .

Contrary to the style guide, I (jabba) usually lump similar resources into the same block, so I end up with an extra indendation, but overall it looks better to me.

```
class foo {
    file {
        "/etc/bar":
            ensure => directory,
            mode => "755";
        "/etc/bar/foobar":
            ensure => file,
            mode => 640,
            source => "puppet:///modules/foo/foobar";
    }
}
```

In the example above, I've lumped both file resources into one file {} code block. This is easier for me to follow and most of our stuff is set up that way. Puppetlabs suggests splitting every file out into individual file {} blocks. When doing this, make sure to add a comma after every parameter, and a semicolon after the last parameter of a particular resource.

Always quote resource names. Also feel free to quote any parameter values and if in doubt, go ahead and quote. Use single quotes if you have any special characters or double quotes and backslashes.

Modules

Most modules will only require three directories: files, templates, manifests.

Manifests is the only one required, as it contains init.pp. init.pp should contain exactly one class by the same name as the module.

For example: modules/foo/manifests/init.pp should contain something like this:

```
class foo {

}
```

The foo class can contain resources on its own or just other classes. It is best to split things out into sub classes for easier readability.

```
class foo {
    include foo::packages
}
```

and modules/foo/manifests/packages.pp would thusly look like this:

```
class foo::packages {
    package {
        "bar":
            ensure => latest;
        "foobar":
            ensure => present;
    }
}
```

Note the file name for any other class besides the main module named class foo, would just be the second part of the name, but referenced with foo::. When using the module structure like this, you don't need any "import" statements. If you declare a class foo::bar, this tells puppet to look in $::modulepath/foo/manifests/bar.pp . bar.pp needs to contain "class foo::bar {}". You can add another level here as well. class foo::bar::users would be found in $::modulepath/foo/manifests/bar/users.pp.

Once your class is all done, you can include it on a node like this:

```
node "example1.example.com" {
    include foo
}
node "example2.example.com" {
    class {
        "bar":
    }
}
```

In the example above, using the second method of calling a class as if it were a resource opens up the possibility to require or notify a class as well as use a parameterized class. Note that both types of classes can be included and called in other classes, it doesn't have to be at the node level.

Parameterized classes

A parameterized class can take optional parameters to provide more flexibility. These are becoming more and more important as we are trying to abstract our environment specific parameters from being hardcoded in a module and only called at a node level:

```
class bar::mount ( $mountpoint, $volume ) {
    mount {
        "${mountpoint}":
            device => $volume;
    }
}
```

This class can now be re-used over and over again on different nodes. It would be called thusly:

```
node "example1.example.com" {
    class {
        "bar::mount":
            mountpoint => "/mnt",
            volume => "/dev/sda1";
    }
}
```

The more things that can be genericized through the use of parameters, the better. It also makes sense to declare parameters with a default value though, so that you **can** override a default value but don't have to.

```
class foo::mount ( $mountpoint = "/mnt", $volume = "/dev/sda1" ) {
    mount {
        "${mountpoint}":
            device => $volume;
    }
}
```

Now you can call the class without any parameters, or optionally override either the mountpoint, the volume, or both.