Writing Nginx Module

Modules, Ordering and The Trifecta

Objectives

- Understand what a Module is

Objectives

- Understand what a Module is
- Learn about classes and naming conventions

Objectives

- Understand what a Module is
- Learn about classes and naming conventions
- Understand how to use package, service and file resources, the trifecta

Objectives

- Understand what a Module is
- Learn about classes and naming conventions
- Understand how to use package, service and file resources, the trifecta
- Learn what meta parameters are and how to use them to specify ordering

What is a Module ?

- a 'Package' for Puppet recipes.
  - It contains all manifests, the core logic and supporting entities such as files, templates, libraries etc.
- Typically maps 1:1 to a piece of software or functionality.

Scenario

Problem Statement: We need a web server configured to serve up our home page.

Validation Criteria: We can see the homepage in a web browser.

Which Web Server ?

Our choice for web servers this time is nginx, a lightweight and scalable alternative to apache.

Lets now get into puppet's shoe and start breaking it down into sub tasks and identify resources for each.

Required Steps

- Install Nginx
- Start Service
- Create a configuration file
- Write out the home page

Required Steps

• Install Nginx	→	• Package
• Start Service	→	• Service
• Create a configuration file	→	• File
• Write out the home page	→	• File

Modular Code

To make your Puppet manifests more readable and maintainable, it's a good idea to arrange them into modules.

we're going to make an nginx module that will contain all Puppet code relating to Nginx.

Lets create a nginx module

In your puppet directory, create the following subdirectories:

```
$ cd puppet
$ mkdir  -p  modules/nginx/manifests
```

We need to now start writing a resource to define the desired state of package

Manifests Best Practice

- One manifest per stage/feature

Phases

- Install
- Configure
- Start Service

Features

- Add SSL Support
- Configure PHP module
- Create Virtual Hosts

Packages

- install/remove/upgrade packages
- maps to package manager internally
  e.g. apt-get, yum, zypper, homebrew

Creating Manifest for Install Phase

Create install.pp and add the resource

```
package {'nginx':
  ensure => installed,
}
```

- Each manifest in a module may contain more than one resources
- These resources need to contained in a single named bundle, which can then be applied to nodes
- Most commonly this named bundle is

- Each manifest in a module may contain more than one resources
- These resources need to contained in a single named bundle, which can then be applied to nodes
- Most commonly this named bundle is

Class

Class

- Classes contain resources

- Classes can be declared in node declarations and even be called from other classes

Class Names

- Class Name Starts with a lowercase letter
- Can contain letters, numbers and underscores
- Can also use a double colon (::) as a namespace separator
- Namespace must map to module layout

Classes and manifests

- Classes and manifests have 1:1 mapping
- Manifests filename must map to class (except init.pp)
- Classes are named as

  module_name::manifestname

  module_name::subdir::manifestname

Adding class to install.pp

```
class nginx::install {
        package {'nginx':
          ensure => installed,
        }
}
```

- We have defined a class. Thats not enough. We need to apply it on the node
- Q: How do we apply this class to node demo ?

- We have defined a class. Thats not enough. We need to apply it on the node

- Q: How do we apply this class to node demo ?

➡ Using Node Declaration

Defining vs Declaring

Defining a class makes it available by name, but doesn’t automatically evaluate the code inside it.

```
class my_class {
  ... puppet code ...
}
```

Declaring a class evaluates the code in the class, and applies all of its resources.

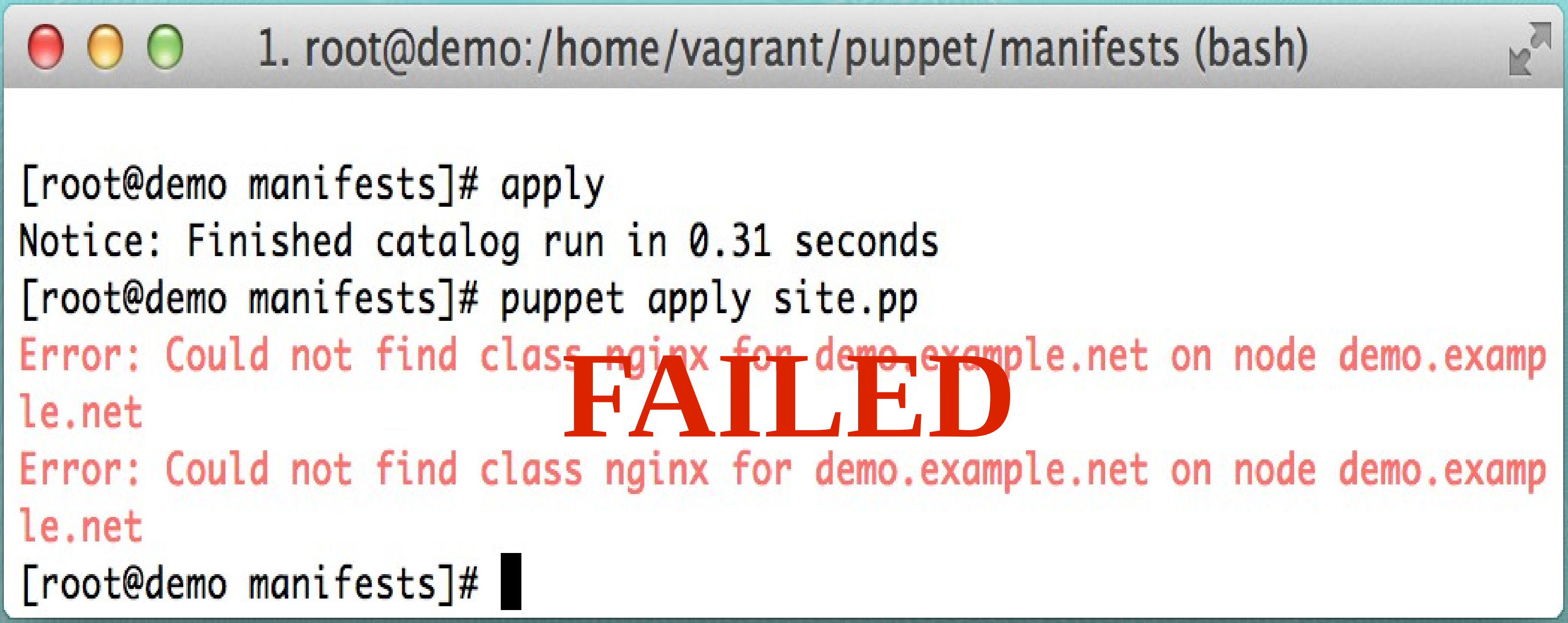
```
include  module::my_class
```

nodes.pp

```
node 'demo' {

   include nginx::install

 }
```

```
puppet
    |
    |__manifests
    |         |
    |         |__ site.pp.bak
    |         |__ nodes.pp
    |
    |__modules
          |
          |___nginx
               |
               |__manifests
                       |__install.pp
```

```
$ sudo   puppet   apply    /vagrant/puppet/manifests/
$ sudo   puppet   apply   /vagrant/puppet/manifests/
```



1. root@demo:/home/vagrant/puppet/manifests (bash)

```
[root@demo manifests]# apply
Notice: Finished catalog run in 0.31 seconds
[root@demo manifests]# puppet apply site.pp
Error: Could not find class nginx for demo.example.net on node demo.examp
le.net
Error: Could not find class nginx for demo.example.net on node demo.examp
le.net
[root@demo manifests]#
```

FAILED

Discover current module path

```
$puppet apply --configprint
modulepath
```

Output

```
/etc/puppet/modules:/usr/share/puppet/modules
```

Providing Modulepath

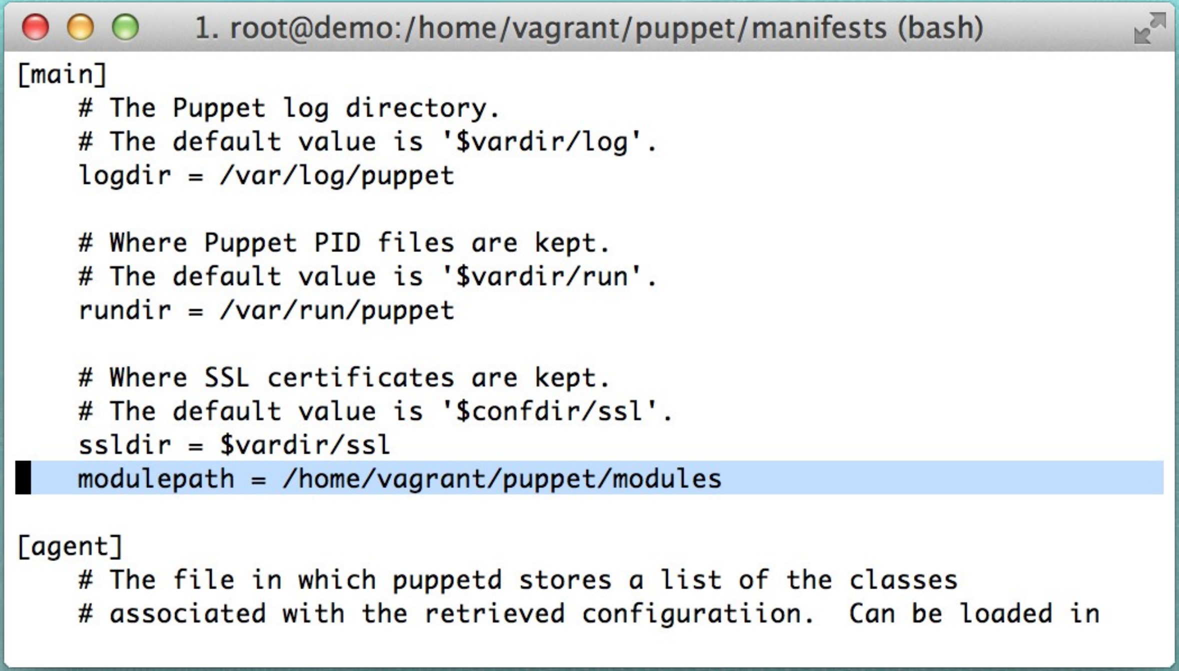
```
$ sudo   puppet   apply
/vagrant/puppet/manifests/
--modulepath /vagrant/puppet/modules/
```

Quick Fix

```
$ rm -rf /etc/puppet/modules

$ ln -s \
/vagrant/puppet/modules \
/etc/puppet/modules
```

Alternative (dont apply this change)

Alternately, you could edit /etc/puppet/puppet.conf, and add modulepath parameter

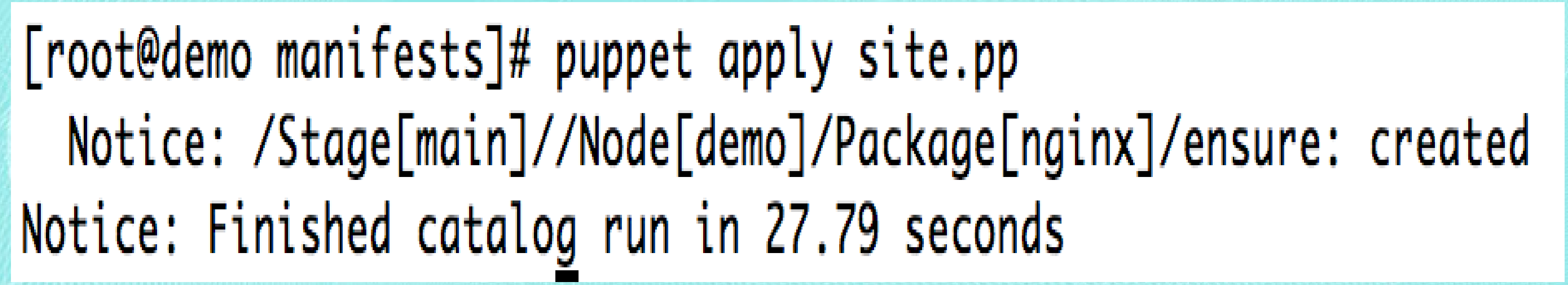


1. root@demo:/home/vagrant/puppet/manifests (bash)

```
[main]
    # The Puppet log directory.
    # The default value is '$vardir/log'.
    logdir = /var/log/puppet

    # Where Puppet PID files are kept.
    # The default value is '$vardir/run'.
    rundir = /var/run/puppet

    # Where SSL certificates are kept.
    # The default value is '$confdir/ssl'.
    ssldir = $vardir/ssl
    modulepath = /home/vagrant/puppet/modules

[agent]
    # The file in which puppetd stores a list of the classes
    # associated with the retrieved configuratiion.  Can be loaded in
```

apply

```
$ sudo puppet apply /vagrant/puppet/manifests/
```



```
[root@demo manifests]# puppet apply site.pp
  Notice: /Stage[main]//Node[demo]/Package[nginx]/ensure: created
Notice: Finished catalog run in 27.79 seconds
```

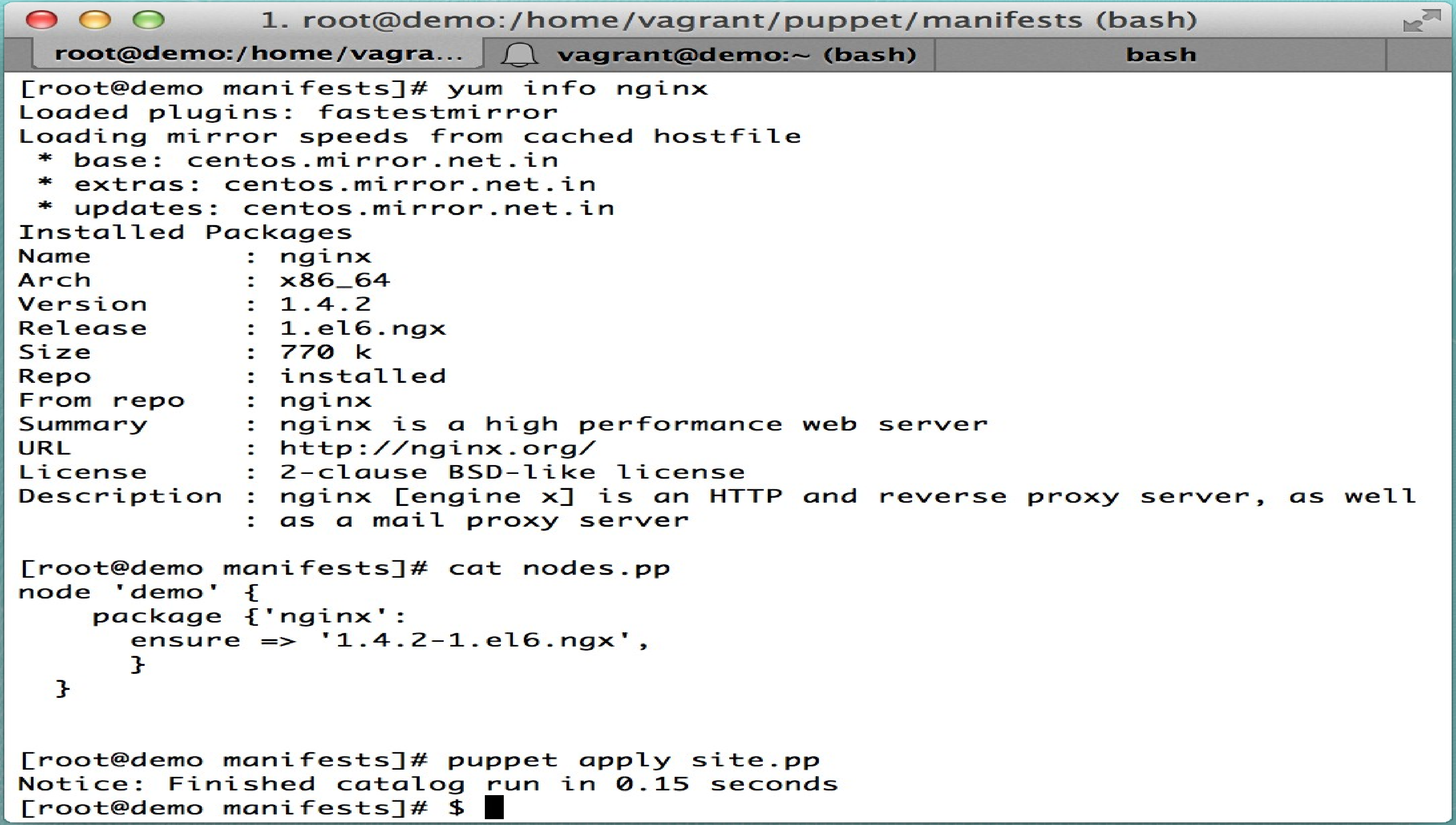
More About Packages...

Installing specific versions:

```
package { 'nginx':
        ensure => '1.6.2-1.el6.ngx',
}
```

The exact version string will depend on the Linux distribution and package repository you're using.

$yum info nginx


1. root@demo:/home/vagrant/puppet/manifests (bash)

root@demo:/home/vagra... | vagrant@demo:~ (bash) | bash

```
[root@demo manifests]# yum info nginx
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: centos.mirror.net.in
 * extras: centos.mirror.net.in
 * updates: centos.mirror.net.in
Installed Packages
Name        : nginx
Arch        : x86_64
Version     : 1.4.2
Release     : 1.el6.ngx
Size        : 770 k
Repo        : installed
From repo   : nginx
Summary     : nginx is a high performance web server
URL         : http://nginx.org/
License     : 2-clause BSD-like license
Description : nginx [engine x] is an HTTP and reverse proxy server, as well
            : as a mail proxy server

[root@demo manifests]# cat nodes.pp
node 'demo' {
    package {'nginx':
      ensure => '1.4.2-1.el6.ngx',
      }
  }


[root@demo manifests]# puppet apply site.pp
Notice: Finished catalog run in 0.15 seconds
[root@demo manifests]# $
```

Package names may be different on different operating system. If you have to write Puppet code that takes account of platform differences like this, you can use a Puppet construct called a **selector** to choose the appropriate package name.

Adding Selector

```
$nginx_version = $osfamily ? {
      'Debian'            => '1.6.2-1ubuntu0.1',
      'RedHat'            => '1.6.2-1.el6.ngx',
       default            => '1.6.2',
}

package {'nginx':
   ensure => $nginx_version,
 }
```

More About Packages

Removing Packages

```
package { 'httpd':
    ensure => absent,
}
```

Lets add ensure=> absent

```
package { 'httpd':
  ensure => absent,
}

package {'nginx':
  ensure => $nginx_version,
  require => Package['httpd'],
}
```

More About Packages

Updating Packages

```
package { 'puppet':
    ensure => latest,
}
```

Warning

Upgrading a package version automatically can cause unexpected failures or problems.

Multiple includes

Its safe to declare a class multiple times.

The include function will declare the class only if its not been declared before, else will skip it.

Why group resources into class?

Well, for one thing, it means we could include the nginx class on many nodes without repeating the same resource declarations over and over:

```
node 'demo' {
    include nginx::install
}
node 'demo2' {
    include nginx::install
}
node 'demo3' {
    include nginx::install
}
```

More about Modules

- **Modules** are just directories with files, arranged in a specific, predictable structure. The manifest files within a module have to obey certain naming restrictions.
- **Auto Loading** If a class is defined in a module, you can declare that class by name in any manifest. Puppet will automatically find and load the manifest that contains the class definition.

Module Structure

```
module_name
    |
    |________manifests
    |
    |_____________files
    |
    |_________________templates
    |
    |_______________________lib
    |
    |__________________________tests or examples
    |
    |_________________________________spec
```

Modules

Splitting the logic into multiple classes inside a module, your main manifest becomes much smaller, more readable and policy focused.

Module Structure

- A module is a directory
- Module name = name of the directory (e.g. nginx)
- It contains a **manifests** directory, which contains any number of .pp files

Module Cheat Sheet

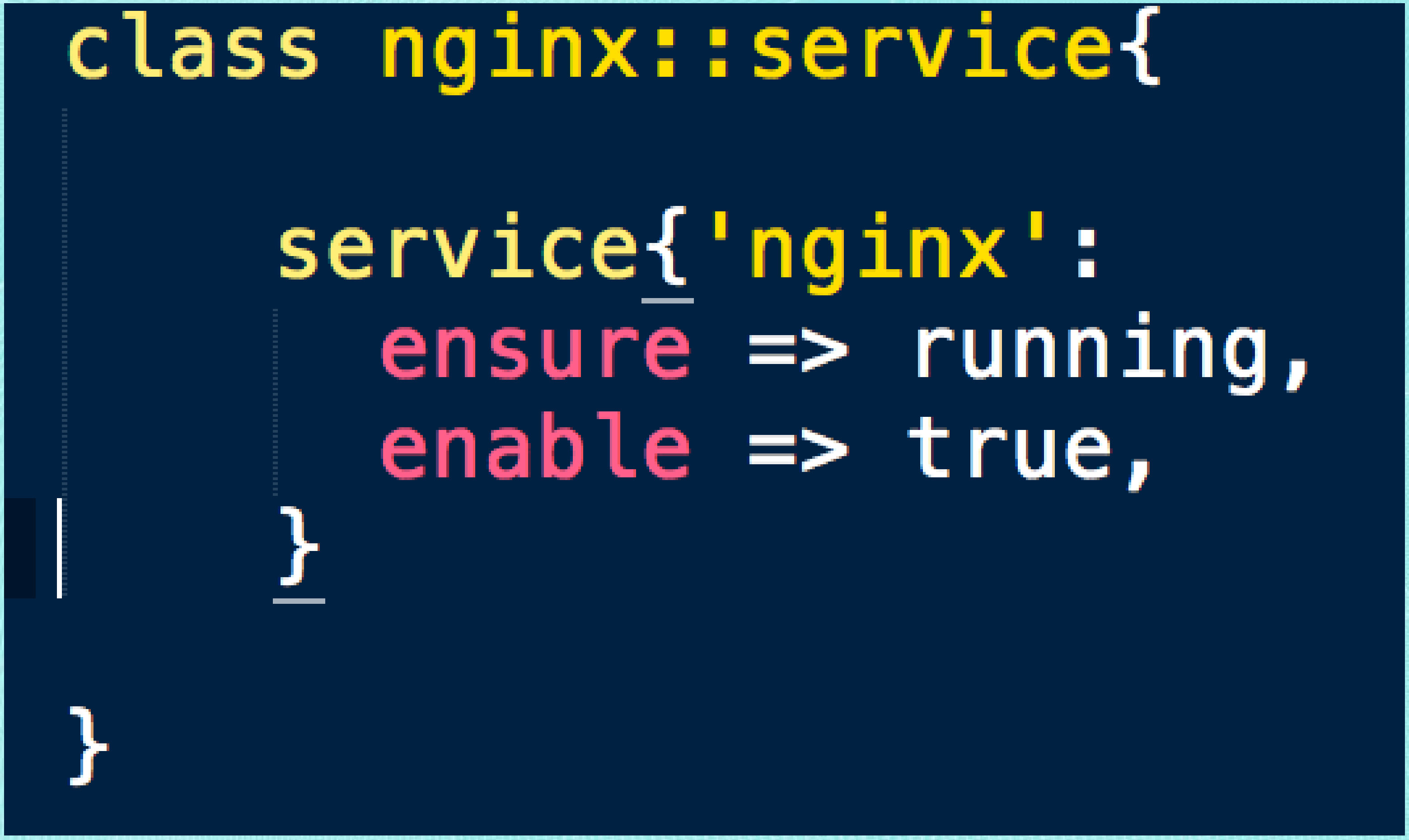
Services

Lets create a resource definition for starting nginx service.

Exercise

- Create service.pp manifest
- Write nginx::service class
- Create a resource definition for service with following specs
  - Type: service
  - Name: nginx
  - Attributes
    - State: running
    - enable: true

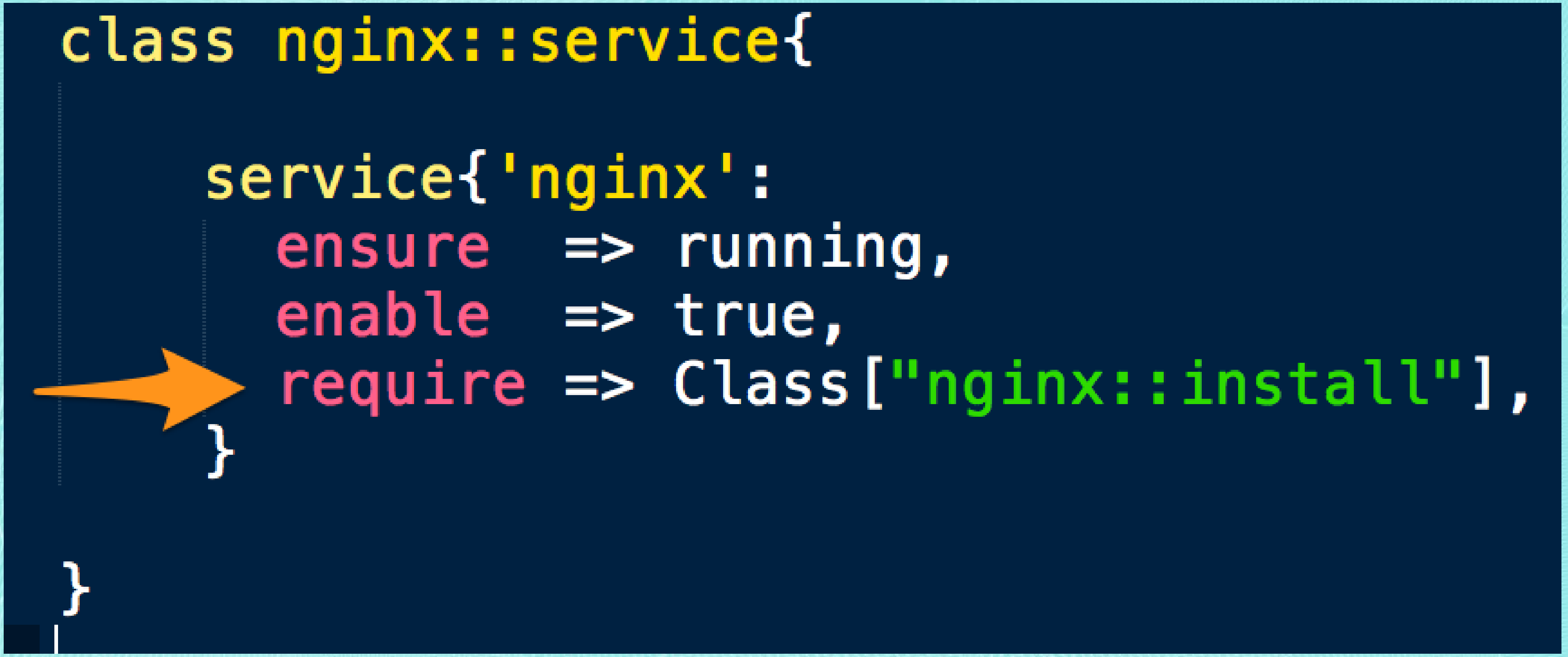
service.pp



```
class nginx::service{

    service{'nginx':
      ensure => running,
      enable => true,
    }

}
```

Adding Dependency

- We also need define ordering between package and service
- Since we have different classes, we could define it at the class level ..... lets look at how (next slide)



```
class nginx::service{

    service{'nginx':
      ensure  => running,
      enable  => true,
      require => Class["nginx::install"],
    }

}
```

Exercise :

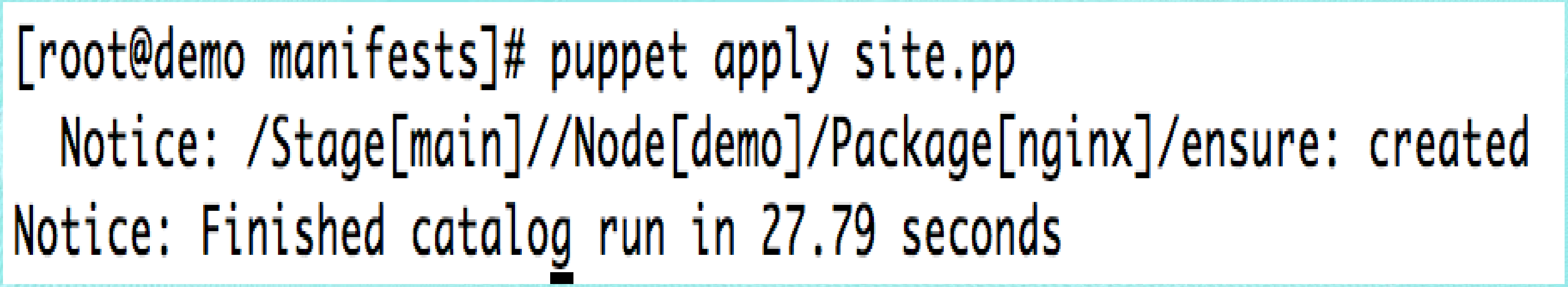
Add the new class (nginx::service) to node declaration for host “demo”

nodes.pp

```
node 'demo' {

  include nginx::install
  include nginx::service

}
```

apply

```
$ sudo   puppet   apply   /vagrant/puppet/manifests/
```



```
[root@demo manifests]# puppet apply site.pp
  Notice: /Stage[main]//Node[demo]/Package[nginx]/ensure: created
Notice: Finished catalog run in 27.79 seconds
```

Enabling Service

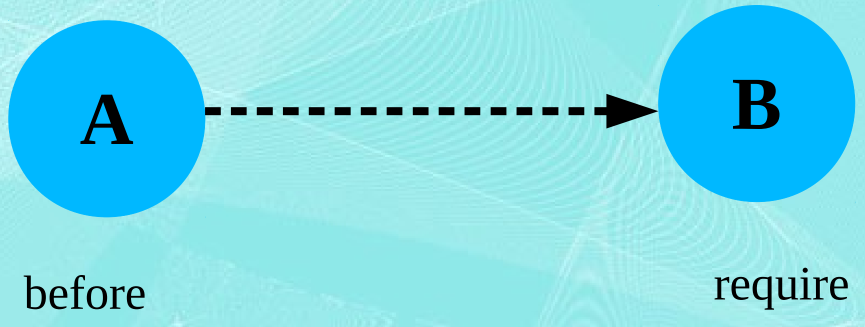
To enable service to be automatically started after rebooting the machine, use the following attribute

ensure => running,

Ordering

- Puppet does not follow serial order of execution
- This helps puppet to run resources in parallel
- On the flip side, ordering has to be explicit

Ordering



```
require => Package['httpd'],
require => Class['nginx'],
```

Resource Reference

Type['title']

Meta Parameters

You can embed relationship information in a resource with the following metaparameters.

- before
- require
- notify
- subscribe

More About Ordering

Relationships can be,

- ordering
- ordering-with-notification

Ordering Types

You can also declare relationships outside a resource with the -> and ~> chaining arrows.

```
Package['nginx'] -> Service['nginx']
Package['nginx.conf'] ~> Service['nginx']
```

More About Services

- Enabling Service
- Services that don't support "status"
- Specifying how to start, stop, or restart a service

Starting a service at boot time

```
service { 'nginx':
        ensure => running,
        enable => true,
}
```

Services that don't support "status"

```
service { 'my-service':
   ensure  => running,
   **hasstatus => false,**
}
```

Services that don't support "status"

```
service { 'my-service':
            Ensure      => running,
             hasstatus  => false,
             pattern     => 'nginx',
 }
```

Specifying how to start, stop, or restart a service

```
service { 'ssh':
        ensure => running,
        start => '/usr/sbin/start.sh',
        stop => '/usr/sbin/stop.sh',
        restart => '/usr/sbin/restart.sh',
}
```

Files

So Nginx is installed and running, but it's not yet serving a website. To do that, we have to have Puppet install a config file on the server to define an Nginx virtual host. This will tell Nginx how to respond to requests for the website.

Lets deploy a vhost

Create the directory modules/nginx/files:

```
vagrant@demo:~/puppet$ mkdir modules/nginx/files
```

vhost config

file: modules/nginx/files/cinema.conf

```
server {
            listen 80;
            root   /var/www/cinema;
            server_name   cinema.com;
        }
```

Index file

file: modules/nginx/files/index.html

```
<html>
  <h1>Hello World ! </h1>
</html>
```

configure.pp

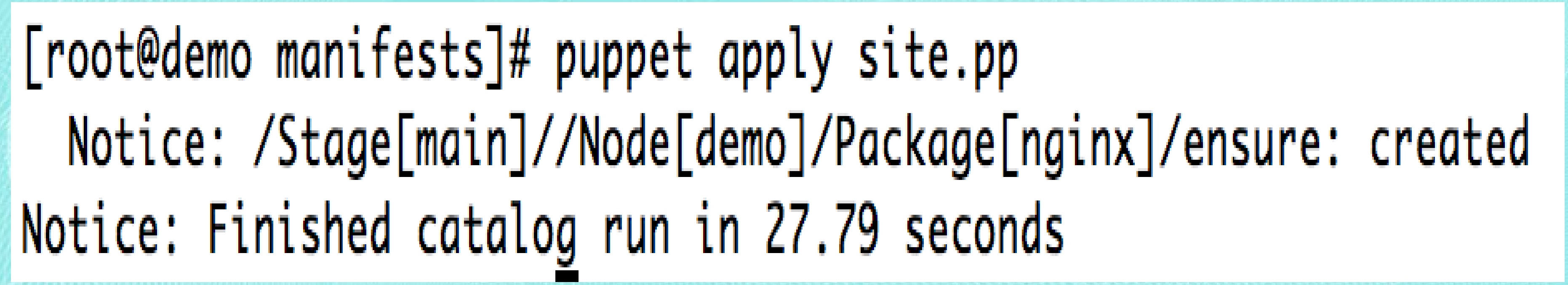
```
class nginx::configure{
    file { '/var/www':
        ensure => directory,
     }

     file { '/var/www/cinema':
        ensure => directory,
     }

      file { '/etc/nginx/conf.d/default.conf':
         source => 'puppet:///modules/nginx/cinema.conf',
         notify => Service['nginx'],
     }

      file { '/var/www/cinema/index.html':
         source => 'puppet:///modules/nginx/index.html',
     }
}
```

nodes.pp

```
node 'demo' {


   include nginx::install
   include nginx::configure
   include nginx::service


 }
```

apply

```
$ sudo puppet apply /vagrant/puppet/manifests/
```



```
[root@demo manifests]# puppet apply site.pp
  Notice: /Stage[main]//Node[demo]/Package[nginx]/ensure: created
Notice: Finished catalog run in 27.79 seconds
```

testing time

```
$ curl localhost
```

Congratulations ! You just got a web server installed with webpage serving!!

And you just learnt about THE TRIFECTA

Package => File => Service

If you can only do this, you can still do a lot !

Init.pp

- Special manifest
- Contain class named after module name
- Can be declared as

```
include module_name
```

Best Practice

- Use init.pp to do the base configurations for the module
- Could include other manifests which form the base configurations
- e.g. install, configure, service

For nginx module, by default we would like to run install, configure and service classes. Lets add them to init.pp

init.pp

```
class nginx {

include nginx::install
include nginx::configure
include nginx::service

}
```

nodes.pp

```
node 'demo' {

      include nginx::install
      include nginx::configure
      include nginx::service

 }
```

```
node 'demo' {

   include nginx

}
```

validate

```
$ sudo   puppet   apply    /vagrant/puppet/manifests/
```

Summary

- Modules
- Class
- Modulepath
- Ordering
- Resource Trifecta - Package, File, Service