Learning Puppet

(Generated on July 01, 2013, from git revision 46784ac1656bd7b57fcfb51d0865ec7ff65533d9)

Learning Puppet — Index

Welcome

This is Learning Puppet, a series of lessons about managing system configuration with Puppet Enterprise. Installation instructions and a complete reference to the Puppet language are available elsewhere on the site; this series is meant to be a guided tour to building things with Puppet.

If you've heard good things about Puppet but don't know where to start, this is the best place to begin.

Get the Free VM

Puppet can configure or misconfigure nearly any aspect of a system, so when learning how to use it, it's best to have some test systems around.

To help with this, we provide a free virtual machine with Puppet already installed. Experiment fearlessly!

Get the Learning Puppet VM

While it downloads, move on to the first chapter of Learning Puppet. If you have problems with getting the VM running, see "VM Tips" below.

The Learning Puppet VM is available in VMWare .vmx format and the cross-platform OVF format, and has been tested with VMWare Fusion and VirtualBox.

Login Info

- Log in as `root`, with the password `puppet`.
- The VM is configured to write its current IP address to the login screen about ten seconds after it boots. If you prefer to use SSH, wait for the IP address to print and ssh to `root@<ip address>`.
- To view the Puppet Enterprise web console, navigate to https://(your VM's IP address) in your web browser. Log in as `puppet@example.com`, with the password `learningpuppet`.
- Note: If you want to create new user accounts in the console, the confirmation emails will contain incorrect links. You can work around this by copy-pasting the links into a web browser and fixing the hostname before hitting enter, or you can make sure the console is available at a reliable hostname and follow the instructions for changing the authentication hostname.

If you'd rather cook up your own VM than download one from the web, you can imitate it fairly easily: this is a stripped-down CentOS system with a hostname of "learn.localdomain," Puppet Enterprise installed, and `iptables` disabled. (It also has Puppet language modes installed for Vim and Emacs, but that's not strictly necessary.)

To begin with, you won't need separate agent and master VMs; this VM can act in both roles. When you reach the agent/master exercises, we'll walk through duplicating the system into a new agent node.

Contents

- Part one: Serverless Puppet
  - Resources and the RAL — Learn about the fundamental building blocks of system configuration.
  - Manifests — Start controlling your system by writing actual Puppet code.
  - Ordering — Learn about dependencies and refresh events, manage the relationships between resources, and discover the fundamental Puppet design pattern.
  - Variables, Conditionals, and Facts — Make your manifests versatile by reading system information.
  - Modules and Classes (Part One) — Start building your manifests into self-contained modules.
  - Templates — Use ERB to make your config files as flexible as your Puppet manifests.
  - Parameterized Classes (Modules, Part Two) — Learn how to pass parameters to classes and make your modules more adaptable.
  - Defined Types — Model repeatable chunks of configuration by grouping basic resources into super-resources.
- Part two: Master/Agent Puppet
  - Preparing an Agent VM — Prepare your tools for the next few chapters with our step-by-step walkthrough.
  - Basic Agent/Master Puppet — Tour the agent/master workflow: sign an agent node's certificate, pick which classes a node will get, and pull and apply a catalog.

VM Tips

Importing the VM into VirtualBox

There are several quirks and extra considerations to manage when importing this VM into VirtualBox:

- If you are using VirtualBox with the OVF version of the VM, choose "Import Appliance" from the File menu and browse to the `.ovf` file included with your download; alternately, you can drag the OVF file and drop it onto VirtualBox's main window.

Do not use the “New Virtual Machine Wizard” and select the included `.vmdk` file as the disk; machines created this way will kernel panic during boot.

- If you find the system hanging during boot at a “registered protocol family 2” message, you may need to go to the VM’s “System” settings and check the “Enable IO APIC” option. (Many users are able to leave the IO APIC option disabled; we do not currently know what causes this problem.)
- The VM should work without modification on 4.x versions of VirtualBox. However, on 3.x versions, it may fail to import, with an error like “Failed to import appliance. Error reading ‘filename.ovf’: unknown resource type 1 in hardware item, line 95.” If you see this error, you can either upgrade your copy of VirtualBox, or work around it by editing the .ovf file and recalculating the sha1 hash, as described here. Thanks to Mattias for this workaround.

Importing the VM into Parallels Desktop

Parallels Desktop 7 on OS X can import the VMX version of this VM, but it requires extra configuration before it can run:

1. First, convert the VM. Do not start the VM yet.
2. Navigate to the Virtual Machine menu, then choose Configure -> Hardware -> Hard Disk 1 and change its location from SATA to IDE (e.g. IDE 0:1).
3. You can now start the VM.

If you attempt to start the VM without changing the location of the disk, it will probably kernel panic.

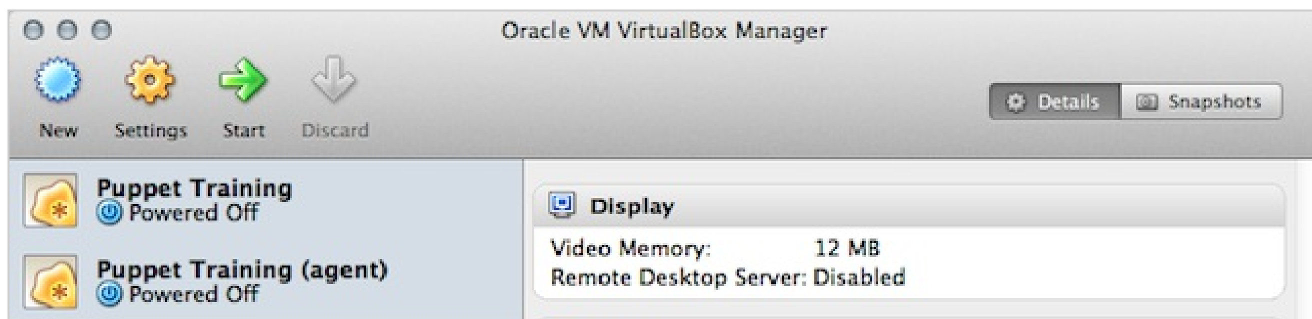
Configuring Virtual Networking

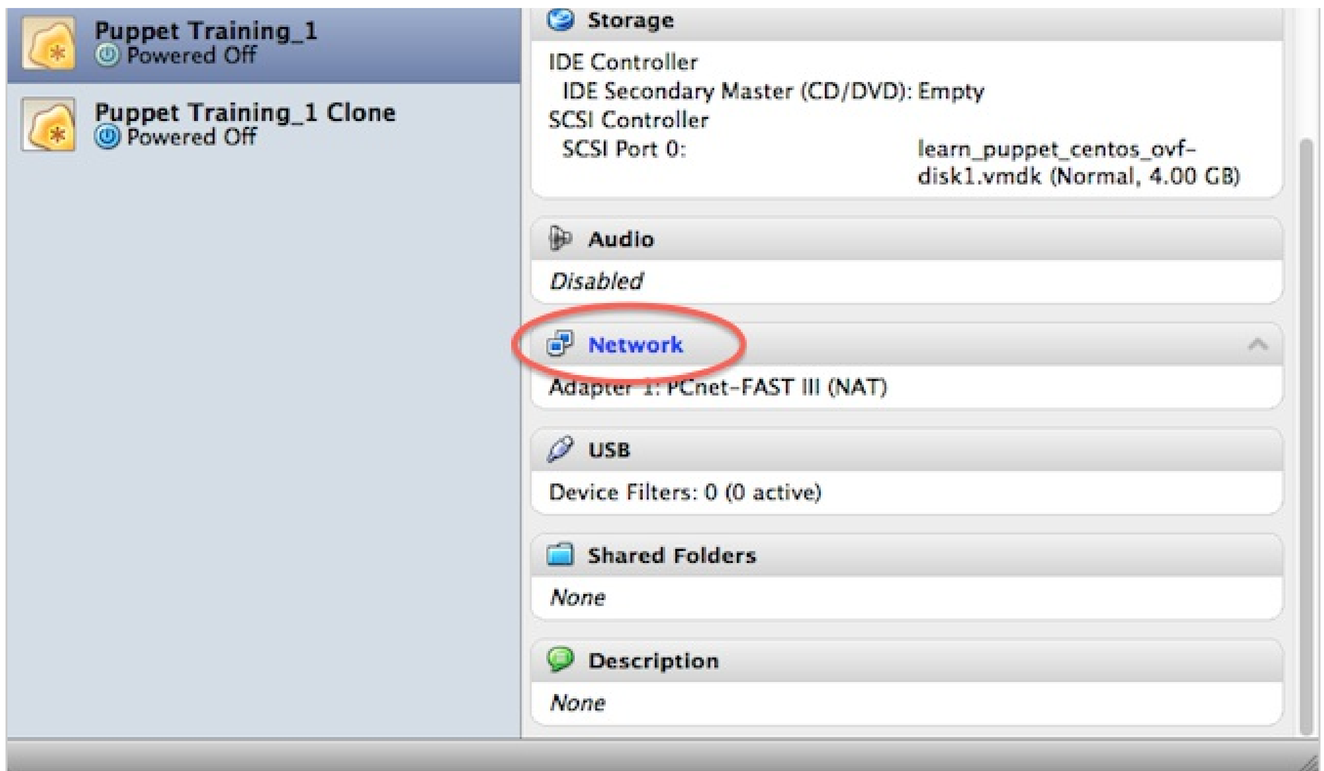
WITH VMWARE

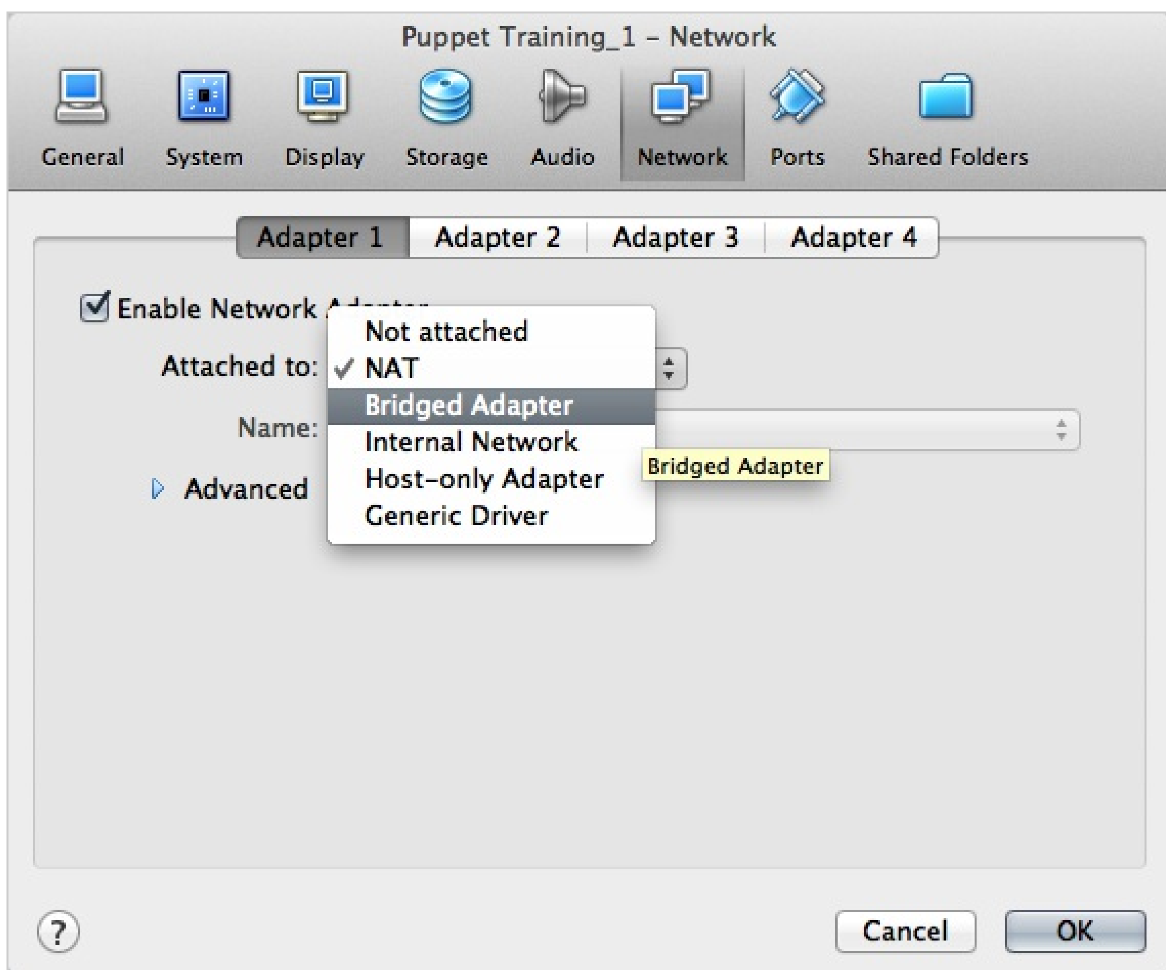
If you are using a VMware virtualization product, you can leave the VM’s networking in its default NAT mode. This will let it contact your host computer, any other VMs being run in NAT mode, the local network, and the outside internet; the only restriction is that computers outside your host computer can’t initiate connections with it. If you eventually need other computers to be able to contact your VM, you can change its networking mode to Bridged.

WITH VIRTUALBOX

VirtualBox’s NAT mode is severely limited, and will not work with the later agent/master lessons. You should change the VM’s network mode to Bridged Adapter before starting the VM for the first time.







If for some reason you cannot expose the VM as a peer on your local network, or you are not on a network with working DHCP, you must configure the VM to have two network adapters: one in NAT mode (for accessing the local network and the internet) and one in Host Only Adapter mode (for accessing the host computer and other VMs). You will also have to either

assign an IP address to the host-only adapter manually, or configure VirtualBox's DHCP server.

See here for more information about VirtualBox's networking modes, and see here for more about VirtualBox's DHCP server.

To manually assign an IP address to a host-only adapter:

- Find the host computer's IP address by looking in VirtualBox's preferences — go to the "Network" section, double-click on the host-only network you're using, go to the "Adapter" tab, and note the IP address in the "IPv4 Address" field.
- Once your VM is running, log in on its console and run `ifconfig eth1 <NEW IP ADDRESS>`, where `<NEW IP ADDRESS>` is an unclaimed IP address on the host-only network's subnet.

Learning Puppet — Resources and the RAL

Welcome to Learning Puppet! This series covers the basics of writing Puppet code and using Puppet Enterprise. You should already have a copy of the Learning Puppet VM; if you don't, you can download it for free.

Begin

Log into the Learning Puppet VM as root, and run the following command:

```
# puppet resource service

service { 'NetworkManager':
  ensure => 'stopped',
  enable => 'false',
}
service { 'acpid':
  ensure => 'running',
  enable => 'true',
}
service { 'anacron':
  ensure => 'stopped',
  enable => 'true',
}
service { 'apmd':
  ensure => 'running',
  enable => 'true',
}
...
... (etc.)
```

Okay! You've just met your first Puppet resources.

> **What Just Happened?**
>
> - `puppet`: Most of Puppet's functionality comes from a single `puppet` command, which has many subcommands.
> - `resource`: The `resource` subcommand can inspect and modify resources interactively.
> - `service`: The first argument to the `puppet resource` command must be a resource type, which you'll learn more about below. A full list of types can be found at [the Puppet type reference](#).
>
> Taken together, this command inspected every service on the system, whether running or stopped.

Resources

Imagine a system's configuration as a collection of many independent atomic units; call them "resources."

These pieces vary in size, complexity, and lifespan. Any of the following (and more) can be modeled as a single resource:

- A user account
- A specific file
- A directory of files
- A software package
- A running service
- A scheduled cron job
- An invocation of a shell command, when certain conditions are met

Any single resource is very similar to a group of related resources:

- Every file has a path and an owner
- Every user has a name, a UID, and a group

The implementation might differ — for example, you'd need a different command to start or stop a service on Windows than you would on Linux, and even across Linux distributions there's some variety. But conceptually, you're still starting or stopping a service, regardless of what you type into the console.

Abstraction

If you think about resources in this way, there are two notable insights you can derive:

- Similar resources can be grouped into types. Services will tend to look like services, and users will tend to look like users.
- The description of a resource type can be separated from its implementation. You can talk about whether a service is started without needing to know how to start it.

To these, Puppet adds a third insight:

- With a good enough description of a resource type, it's possible to declare a desired state for a resource — instead of saying "run this command that starts a service," say "ensure this service is running."

These three insights form Puppet's resource abstraction layer (RAL). The RAL consists of types (high-level models) and providers (platform-specific implementations) — by splitting the two, it lets you describe desired resource states in a way that isn't tied to a specific OS.

Anatomy of a Resource

In Puppet, every resource is an instance of a resource type and is identified by a title; it has a number of attributes (which are defined by the type), and each attribute has a value.

Puppet uses its own language to describe and manage resources:

```
user { 'dave':
  ensure     => present,
  uid        => '507',
  gid        => 'admin',
  shell      => '/bin/zsh',
  home       => '/home/dave',
  managehome => true,
}
```

This syntax is called a resource declaration. You saw it earlier when you ran `puppet resource service`, and it's the heart of the Puppet language. It describes a desired state for a resource, without mentioning any steps that must be taken to reach that state.

Try and identify all four parts of the resource declaration above:

- Type
- Title
- Attributes
- Values

Resource Types

As mentioned above, every resource has a type.

Puppet has many built-in resource types, and you can install even more as plugins. Each type can behave a bit differently, and has a different set of attributes available.

There are several ways to get information about resource types:

The Cheat Sheet

Not all resource types are equally common or useful, so we've made a printable cheat sheet that explains the eight most useful types. Download the core types cheat sheet here.

The Type Reference

Experienced Puppet users spend most of their time in the type reference.

This page list all of Puppet's built-in resource types, in extreme detail. It can be a bit overwhelming for a new user, but it has most of the info you'll need in a normal day of writing Puppet code.

We generate a new type reference for every new version of Puppet, to help ensure that the descriptions stay accurate.

Puppet Describe

The `puppet describe` subcommand can list info about the currently installed resource types on a given machine. This is different from the type reference because it also catches plugins installed by a user, in addition to the built-in types.

- `puppet describe -l` — List all of the resource types available on the system.
- `puppet describe -s <TYPE>` — Print short information about a type, without describing every attribute
- `puppet describe <TYPE>` — Print long information, similar to what appears in the type reference.

Browsing and Inspecting Resources

In the next few chapters, we'll talk about using the Puppet language to manage resources. For now, though, let's just look at them for a while.

Live Management in the Console

Puppet Enterprise includes a web console for controlling many of its features. One of the things it can do is browse and inspect resources on any PE systems the console can reach. This supports a limited number of resource types, but has some useful comparison features for correlating data across a large number of nodes.

LOGGING IN

When you first started your VM, it gave you the URL, username, and password for accessing the console. The user and password should always be `puppet@example.com` and `learningpuppet`. The URL will be `https://<IP ADDRESS>`; you can get your VM's IP address by running `facter ipaddress` at the command line.

Once logged in, navigate to "Live Management" in the top menu bar, then click the "Manage Resources" tab. You can then follow these instructions to find and inspect resources.

Since you're only using a single node, you won't see much in the way of comparisons, but you can see the current states of packages, user accounts, etc.

The Puppet Resource Command

Puppet includes a command called `puppet resource`, which can interactively inspect and modify resources on a single system.

Usage of puppet resource is as follows:

```
# puppet resource <TYPE> [<NAME>] [ATTRIBUTE=VALUE ...]
```

- The first argument must be a resource type. If no other arguments are given, it will inspect every resource of that type it can find.
- The second argument (optional) is the name of a resource. If no other arguments are given, it will inspect that resource.
- After the name, you can optionally specify any number of attributes and values. This will sync those attributes to the desired state, then inspect the final state of the resource.
- Alternately, if you specify a resource name and use the `--edit` flag, you can change that resource in your text editor; after the buffer is saved and closed, Puppet will modify the resource to match your changes.

EXERCISES

Inspecting a single resource:

```
# puppet resource user root

user { 'root':
  ensure           => 'present',
  comment          => 'root',
  gid              => '0',
  groups           => ['root', 'bin', 'daemon', 'sys', 'adm', 'disk',
'wheel'],
  home             => '/root',
  password         => '$1$jrm5tnjw$h8JJ9mCZLmJvIxvDLjw1M/',
  password_max_age => '99999',
  password_min_age => '0',
```

```
  shell            => '/bin/bash',
  uid              => '0',
}
```

Setting a new desired state for a resource:

```
# puppet resource user katie ensure=present shell="/bin/zsh"
home="/home/katie" managehome=true

notice: /User[katie]/ensure: created

user { 'katie':
  ensure => 'present',
  home   => '/home/katie',
  shell  => '/bin/zsh'
}
```

Next

Next Lesson:

The puppet resource command can be useful for one-off jobs, but Puppet was born for greater things. Time to write some manifests.

Off-Road:

The LP VM is a tiny sandbox system, and it doesn't have much going on. If you have some dev machines that look more like your actual servers, why not download Puppet Enterprise for free and inspect them? Follow the quick start guide to get a small environment installed, then try using the console to inspect resources for many systems at once.

Learning Puppet — Manifests

Begin

Did you do the `puppet resource` exercises from the last chapter? Let's remove the user account you created.

In a text editor — `vim`, `emacs`, or `nano` — create a file with the following contents and filename:

```
# /root/examples/user-absent.pp
user {'katie':
  ensure => absent,
}
```

Save and close the editor, then run:

```
# puppet apply /root/examples/user-absent.pp
notice: /Stage[main]//User[katie]/ensure: removed
notice: Finished catalog run in 0.07 seconds
```

Now run it again:

```
# puppet apply /root/examples/user-absent.pp
notice: Finished catalog run in 0.03 seconds
```

Cool: You've just written and applied your first Puppet manifest.

Manifests

Puppet programs are called "manifests," and they use the `.pp` file extension.

The core of the Puppet language is the resource declaration. A resource declaration describes a desired state for one resource.

(Manifests can also use various kinds of logic: conditional statements, collections of resources, functions to generate text, etc. We'll get to these later.)

Puppet Apply

Like `resource` in the last chapter, `apply` is a Puppet subcommand. It takes the name of a manifest file as its argument, and enforces the desired state described in the manifest.

We'll use it below to test small manifests, but it can be used for larger jobs too. In fact, it can do nearly everything an agent/master Puppet environment can do.

Resource Declarations

Let's start by looking at a single resource:

```
    # /root/examples/file-1.pp

    file {'testfile':
      path    => '/tmp/testfile',
      ensure  => present,
      mode    => 0640,
      content => "I'm a test file.",
    }
```

The complete syntax and behavior of resource declarations are documented in the Puppet

reference manual, but in short, they consist of:

- The type (`file`, in this case)
- An opening curly brace (`{`)
  - The title (`testfile`)
  - A colon (`:`)
  - A set of attribute `=>` value pairs, with a comma after each pair (`path => '/tmp/testfile',` etc.)
- A closing curly brace (`}`)

Try applying the short manifest above:

```
# puppet apply /root/examples/file-1.pp
notice: /Stage[main]//File[testfile]/ensure: created
notice: Finished catalog run in 0.05 seconds
```

This is just the reverse of what we saw above when we removed the user account: Puppet noticed that the file didn't exist, and created it. It set the desired content and mode at the same time.

```
# cat /tmp/testfile
I'm a test file.
# ls -lah /tmp/testfile
-rw-r----- 1 root root 16 Feb 23 13:15 /tmp/testfile
```

If we try changing the mode and applying the manifest again, Puppet will fix it:

```
# chmod 0666 /tmp/testfile
# puppet apply /root/examples/file-1.pp
notice: /Stage[main]//File[testfile]/mode: mode changed '0666' to '0640'
notice: Finished catalog run in 0.04 seconds
```

And if you run the manifest again, you'll see that Puppet doesn't do anything — if a resource is in the desired state already, Puppet will leave it alone.

> Exercise: Declare another file resource in a manifest and apply it. Try setting a new desired state for an existing file — for example, changing the login message by setting the content of `/etc/motd`. You can see the attributes available for the file type here.

Syntax Hints

Watch out for these common errors:

- Don’t forget commas and colons! Forgetting them causes errors like `Could not parse for environment production: Syntax error at 'mode'; expected '}' at /root/manifests/1.file.pp:6 on node learn.localdomain`.
- Capitalization matters! The resource type and the attribute names should always be lowercase.
- The values used for titles and attribute values will usually be strings, which you should usually quote. Read more about Puppet’s data types here.
  - There are two kinds of quotes in Puppet: single (`'`) and double (`"`). The main difference is that double quotes let you interpolate $variables, which we cover in another lesson.
  - Attribute names (like `path`, `ensure`, etc.) are special keywords, not strings. They shouldn’t be quoted.

Also, note that Puppet lets you use whatever whitespace makes your manifests more readable. We suggest visually lining up the `=>` arrows, because it makes it easier to understand a manifest at a glance. (The Vim plugins on the Learning Puppet VM will do this automatically as you type.)

Once More, With Feeling!

Now that you know resource declarations, let’s play with the file type some more. We’ll:

- Put multiple resources of different types in the same manifest
- Use new values for the `ensure` attribute
- Find an attribute with a special relationship to the resource title
- See what happens when we leave off certain attributes
- See some automatic permission adjustments on directories

```
# /root/examples/file-2.pp

file {'/tmp/test1':
  ensure  => file,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 0644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}

user {'katie':
  ensure => absent,
}

notify {"I'm notifying you.":}
```

```
    notify {"So am I!":}
```

Apply:

```
# puppet apply /root/examples/file-2.pp
notice: /Stage[main]//File[/tmp/test1]/ensure: created
notice: /Stage[main]//File[/tmp/test3]/ensure: created
notice: /Stage[main]//File[/tmp/test2]/ensure: created
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as
'I'm notifying you.'
notice: Finished catalog run in 0.05 seconds
```

Cool. What just happened?

New Ensure Values, Different States

The `ensure` attribute is somewhat special. It's available on most (but not all) resource types, and it controls whether the resource exists, with the definition of "exists" being somewhat local.

With files, there are several ways to exist:

- As a normal file (`ensure => file`)
- As a directory (`ensure => directory`)
- As a symlink (`ensure => link`)
- As any of the above (`ensure => present`)
- As nothing (`ensure => absent`).

A quick check shows how our manifest played out:

```
# ls -lah /tmp/test*
-rw-r--r--  1 root root    3 Feb 23 15:54 test1
lrwxrwxrwx  1 root root   10 Feb 23 15:54 test3 -> /tmp/test1

/tmp/test2:
total 16K
drwxr-xr-x 2 root root 4.0K Feb 23 16:02 .
drwxrwxrwt 5 root root 4.0K Feb 23 16:02 ..

# cat /tmp/test3
Hi.
```

Titles and Namevars

Notice how our original file resource had a `path` attribute, but our next three left it out?

Almost every resource type has one attribute whose value defaults to the resource's title. For the `file` resource, that's `path`. Most of the time (`user`, `group`, `package`…), it's `name`.

These attributes are called "namevars." They are generally the attribute that corresponds to the resource's identity, the one thing that should always be unique.

If you leave out the namevar for a resource, Puppet will re-use the title as its value. If you do specify a value for the namevar, the title of the resource can be anything.

IDENTITY AND IDENTITY

So why even have a namevar, if Puppet can just re-use the title?

There are two kinds of identity that Puppet recognizes:

- Identity within Puppet itself
- Identity on the target system

Most of the time these are the same, but sometimes they aren't. For example, the NTP service has a different name on different platforms: on Red Hat-like systems, it's called `ntpd`, and on Debian-like systems, it's `ntp`. These are logically the same resource, but their identity on the target system isn't the same.

Also, there are cases (usually exec resources) where the system identity has no particular meaning, and putting a more descriptive identity in the title can help tell your colleagues (or yourself in two months) what a resource is supposed to be doing.

By allowing you to split the title and namevar, Puppet makes it easy to handle these cases. We'll cover this later when we get to conditional statements.

UNIQUENESS

Note that you can't declare the same resource twice: Puppet always disallows duplicate titles within a given type, and usually disallows duplicate namevar values within a type.

This is because resource declarations represent desired final states, and it's not at all clear what should happen if you declare two conflicting states. So Puppet will fail with an error instead of accidentally doing something wrong to the system.

Missing Attributes: "Desired State = Whatever"

On the `/tmp/test1` file, we left off the `mode` and `owner` attributes, among others. When we omit attributes, Puppet doesn't manage them, and any value is assumed to be the desired state.

If a file doesn't exist, Puppet will default to creating it with permissions mode 0644, but if you change that mode, Puppet won't change it back.

Note that you can even leave off the `ensure` attribute, as long as you don't specify `content` or `source` — this can let you manage the permissions of a file if it exists, but not create it if it doesn't.

Directory Permissions: 644 = 755

We said `/tmp/test2/` should have permissions mode 0644, but our `ls -lah` showed mode 0755. That's because Puppet groups the read bit and the traverse bit for directories.

This helps with recursively managing directories (with `recurse => true`), so you can allow traversal without making all of the contents of the directory executable.

Destinations, Not Journeys

You've noticed that we talk about "desired states" a lot, instead of talking about making changes to the system. This is the core of thinking like a Puppet user.

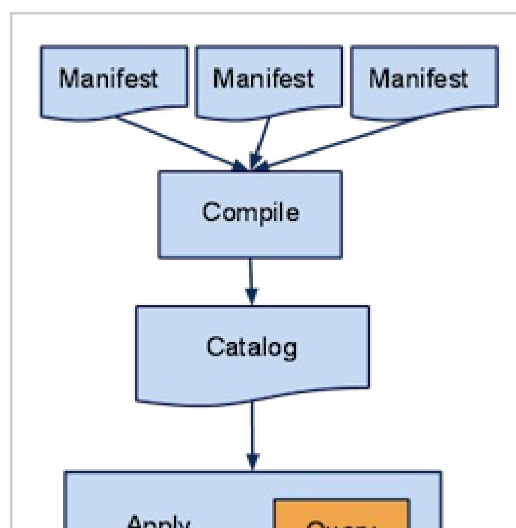
If you were writing an explanation to another human of how to put a system into a desired state, using the OS's default tools, it would read something like "Check whether the mode of the sudoers file is 0440, using `ls -l`. If it's already fine, move on to the next step; otherwise, run `chmod 0440 /etc/sudoers`."

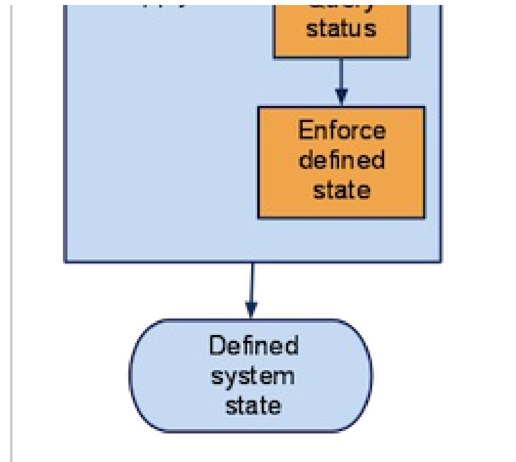
Under the hood, Puppet is actually doing the same thing, with some of the same OS tools. But it wraps the "check" step together with the "and fix if needed" step, and presents them as a single interface.

The effect is that, instead of writing a bash script that looks like a step-by-step for a beginning user, you can write Puppet manifests that look like shorthand notes for an expert user.

Aside: Compilation

Manifests don't get used directly when Puppet syncs resources. Instead, the flow of a Puppet run goes a little like this:





As we mentioned above, manifests can contain conditional statements, variables, functions, and other forms of logic. But before being applied, manifests get compiled into a document called a "catalog," which only contains resources and hints about the order to sync them in.

With puppet apply, the distinction doesn't mean much. In a master/agent Puppet environment, though, it matters more, because agents only see the catalog:

- By using logic, manifests can be flexible and describe many systems at once. A catalog describes desired states for one system.
- By default, agent nodes can only retrieve their own catalog; they can't see information meant for any other node. This separation improves security.
- Since catalogs are so unambiguous, it's possible to simulate a catalog run without making any changes to the system. (This is usually done by running `puppet agent --test --noop`.) You can even use special diff tools to compare two catalogs and see the differences.

The Site Manifest and Puppet Agent

We've seen how to use puppet apply to directly apply manifests on one system. The puppet master/agent services work very similarly, but with a few key differences:

Puppet apply:

- A user executes a command, triggering a Puppet run.
- Puppet apply reads the manifest passed to it, compiles it into a catalog, and applies the catalog.

Puppet agent/master:

- Puppet agent runs as a service, and triggers a Puppet run about every half hour (configurable).
    - On your VM, which runs Puppet Enterprise, the agent service is named `pe-puppet`. (Puppet agent can also be configured to run from cron, instead of as a service.)
- Puppet agent does not have access to any manifests; instead, it requests a pre-compiled catalog

from a puppet master server.
  - On your VM, the puppet master appears as the `pe-httpd` service. A sandboxed copy of Apache with Passenger manages the puppet master application, spawning and killing new copies of it as needed.

- The puppet master always reads one special manifest, called the "site manifest" or site.pp. It uses this to compile a catalog, which it sends back to the agent.
  - On your VM, the site manifest is at `/etc/puppetlabs/puppet/manifests/site.pp`.

- After getting the catalog, the agent applies it.

This way, you can have many machines being configured by Puppet, while only maintaining your manifests on one (or a few) servers. This also gives some extra security, as described above under "Compilation."

Exercise: Use Puppet Agent/Master to Apply the Same Configuration

To see how the same manifest code works in puppet agent:

- Edit `/etc/puppetlabs/puppet/manifests/site.pp` and paste in the three file resources from the manifest above.
  - Watch out for some of the existing code in site.pp, and don't disturb any `node` statements yet. You can paste the resources at the bottom of the file and they'll work fine.

- Delete or mutilate the files and directories we created in `/tmp`.
- Run `puppet agent --test`, which will trigger a single puppet agent run in the foreground so you can see what it's doing in real time.
- Check `/tmp`, and notice that the files are back to their desired state.

Exercise: SSH Authorized Key

Write and apply a manifest that uses the `ssh_authorized_key` type to let you log into the learning VM as root without a password.

Bonus work: Try putting it directly into the site manifest, instead of using puppet apply. Use the console to trigger a puppet agent run, and check the reports in the console to see whether the manifest worked.

- You'll need to have an SSH key pair, a terminal application on your host system, and some basic understanding of how SSH works. You can get all of these with a little outside research.

- Watch out: you can't just paste the line from `id_rsa.pub` into the `key` attribute of the resource. You'll need to separate its components out into multiple attributes. Read the documentation for the `ssh_authorized_key` type to see how.

Next

Next Lesson:

You know how to use the fundamental building blocks of Puppet code, so now it's time to learn how those blocks fit together.

Off-Road:

You already know how to do a bit with Puppet, and managing file ownership and permissions is important.

Are there any files on your systems that you've had a hard time keeping in sync? You already know enough to lock them down. Download Puppet Enterprise for free, follow the quick start guide to get a small environment installed, then try putting some file resources at the bottom of the puppet master's `site.pp` file to manage those files on every machine.

Learning Puppet — Resource Ordering

Disorder

Let's look back on one of our manifests from the last page:

```
# /root/training-manifests/2.file.pp

file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}

notify {"I'm notifying you.":}
notify {"So am I!":}
```

When we ran this, the resources weren't synced in the order we wrote them: it went `/tmp/test1`, `/tmp/test3`, `/tmp/test2`, `So am I!`, and `I'm notifying you.`

Like we mentioned in the last chapter, Puppet combines "check the state" and "fix any problems" into a single declaration for each resource. Since each resource is represented by one atomic statement, ordering within a file matters a lot less than it would for an equivalent script.

Or rather, it matters less as long as the resources are independent and not related to each other. And most resources are! But some resources depend on other resources. Consider a service which is installed by a package — it's impossible to get the service into its desired state if the package isn't installed yet. The service has a dependency on the package.

So when dealing with related resources, Puppet has ways to express those relationships.

Summary of This Page

- You can embed relationship information in a resource with the `before`, `require`, `notify`, and `subscribe` metaparameters.
- You can also declare relationships outside a resource with the `->` and `~>` chaining arrows.
- Relationships can be either ordering (this before that) or ordering-with-notification (this before that, and tell that whether this was changed).
- Puppet's relationship behaviors and syntaxes are documented in the Puppet reference manual page on relationships.

Metaparameters, Resource References, and Ordering

Here's a notify resource that depends on a file resource:

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

notify {'/tmp/test1 has already been synced.':
  require => File['/tmp/test1'],
}
```

Each resource type has its own set of attributes, but there's another set of attributes, called metaparameters, which can be used on any resource. (They're "meta" because they don't describe any feature of the resource that you could observe on the system after Puppet finishes; they only describe how Puppet should act.)

There are four metaparameters that let you arrange resources in order:

- `before`
- `require`
- `notify`
- `subscribe`

All of them accept a resource reference (or an array of them) as their value. Resource references look like this:

```
Type['title']
```

(Note the square brackets and capitalized resource type!)

> **ASIDE: WHEN TO CAPITALIZE**
>
> The easy way to remember this is that you only use the lowercase type name when declaring a new resource. Any other situation will always call for a capitalized type name.

Before and Require

`before` and `require` make simple dependency relationships, where one resource must be synced before another. `before` is used in the earlier resource, and lists resources that depend on it; `require` is used in the later resource, and lists the resources that it depends on.

These two metaparameters are just different ways of writing the same relationship — our example above could just as easily be written like this:

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
  before  => Notify['/tmp/test1 has already been synced.'],
}

notify {'/tmp/test1 has already been synced.':}
```

Notify and Subscribe

A few resource types (`service`, `exec`, and `mount`) can be “refreshed” — that is, told to react to changes in their environment. For a service, this usually means restarting when a config file has been changed; for an `exec` resource, this could mean running its payload if any user accounts have been changed. (Note that refreshes are performed by Puppet, so they only occur during Puppet runs.)

The `notify` and `subscribe` metaparameters make dependency relationships the way `before` and `require` do, but they also make notification relationships. Not only will the earlier resource in the pair get synced first, but if Puppet makes any changes to that resource, it will send a refresh event to the later resource, which will react accordingly.

An example of a notification relationship:

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => 'puppet:///modules/ssh/sshd_config',
}
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

In this example, the `sshd` service will be restarted if Puppet has to edit its config file.

Chaining Arrows

There's one last way to declare relationships: chain resource references with the ordering (`->`) and notification (`~>`; note the tilde) arrows. Think of them as representing the flow of time: the resource behind the arrow will be synced before the resource the arrow points at.

This example causes the same dependency as the similar examples above:

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

notify {'after':
  message => '/tmp/test1 has already been synced.',
}

File['/tmp/test1'] -> Notify['after']
```

Chaining arrows can take several things as their operands: this example uses resource references, but they can also take resource declarations and resource collectors.

Since whitespace is freely adjustable in Puppet, and since chaining arrows can go between resource declarations, it's easy to make a short run of resources be synced in the order they're written — just put chaining arrows between them:

```
file {'/tmp/test1':
```

```
    ensure  => present,
    content => "Hi.",
  }
  ->
  notify {'after':
    message => '/tmp/test1 has already been synced.',
  }
```

Again, this creates the same relationship we've seen previously.

Autorequire

Some of Puppet's resource types will notice when an instance is related to other resources, and they'll set up automatic dependencies. The one you'll use most often is between files and their parent directories: if a given file and its parent directory are both being managed as resources, Puppet will make sure to sync the parent directory before the file. This never creates new resources; it only adds dependencies to resources that are already being managed.

Don't sweat much about the details of autorequiring; it's fairly conservative and should generally do the right thing without getting in your way. If you forget it's there and make explicit dependencies, your code will still work. Explicit dependencies will also override autorequires, if they conflict.

Example: sshd

Hopefully that's all pretty clear! But even if it is, it's rather abstract — making sure a notify fires after a file is something of a "hello world" use case, and not very illustrative. Let's break something!

You've probably been using SSH and your favorite terminal app to interact with the Learning Puppet VM, so let's go straight for the most-annoying-case scenario: we'll pretend someone accidentally gave the wrong person (i.e., us) sudo privileges, and they ruined root's ability to SSH to this box.

Prepare

Let's get a copy of the current sshd config file; going forward, we'll use our new copy as the canonical source for that file.

```
# cp /etc/ssh/sshd_config ~/examples/
```

Now we'll write some Puppet code to manage the file:

```
  # /root/examples/break_ssh.pp (incomplete)
  file { '/etc/ssh/sshd_config':
    ensure => file,
    mode   => 600,
    source => '/root/examples/sshd_config',
    # And yes, that's the first time we've used the "source" attribute. It
accepts
```

```
        # absolute local paths and puppet:/// URLs, which we'll say more about
later.
    }
```

This is only half of what we need, though. It will change the config file, but those changes will only take effect when the service restarts, which could be years from now.

To make the service restart whenever we make changes to the config, we should tell Puppet to manage the `sshd` service and have it subscribe to the config file:

```
    # /root/examples/break_ssh.pp
    file { '/etc/ssh/sshd_config':
      ensure => file,
      mode   => 600,
      source => '/root/examples/sshd_config',
    }
    service { 'sshd':
      ensure     => running,
      enable     => true,
      subscribe  => File['/etc/ssh/sshd_config'],
    }
```

Manage

Great, we have our Puppet code; now paste it into `/etc/puppetlabs/puppet/manifests/site.pp`, so that puppet agent will manage those resources.

Break

Next, edit the original `/etc/ssh/sshd_config` file. There's a commented-out line in there that says `#PermitRootLogin yes`; find it, remove the comment, and change the yes to a no:

```
PermitRootLogin no
```

Now manually restart the sshd service:

```
# service sshd restart
```

… and log out. You should no longer be able to log in as root over SSH; test it to make sure. (Although you can still log in via your virtualization software's console.)

Fix

Actually, now that you've added those resources to site.pp, Puppet will fix this automatically within about half an hour. But if you're impatient, you can log in to the Puppet Enterprise console, then trigger a puppet agent run in the live management page.

And that'll do it! After the Puppet run has completed and you can see the report appear in the console (it will have a blue icon, to show that changes were made), you should be able to log in as root via SSH again. Victory.

> NO CHANGES? NO REFRESH
>
> There's an odd situation you can get into if you apply a manifest that makes config file changes before you finish writing it.
>
> Puppet only sends refresh events if it makes changes to the notifying resource in this run. So if you wrote a file resource with new desired content for a config file, applied the manifest, then edited the manifest again to create a refresh relationship with a service, the service would miss its refresh, since the file resource would already be in its desired state.
>
> This will generally only happen to you on the machines you're testing early versions of manifests on, rather than your production boxes. If it does bite you, you can restart the service manually with the "Advanced Tasks" section of the PE console's live management page — use the "restart" action in the "service tasks" section.

Package/File/Service

The example we just saw was very close to a pattern you'll see constantly in production Puppet code, but it was missing a piece. Let's complete it:

```
# /root/examples/break_ssh.pp
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => '/root/examples/sshd_config',
}
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe  => File['/etc/ssh/sshd_config'],
}
```

This is the package/file/service pattern, one of the most useful idioms in Puppet: the package resource makes sure the software and its config file are installed, the config file depends on the package resource, and the service subscribes to changes in the config file.

It's hard to overstate the importance of this pattern! If you stopped here and only learned this, you could still get a lot of work done.

> **Exercise: Apache**
>
> Write and apply a manifest that will install the Apache package, then make sure the Apache service is running. Prove that it worked by using a web browser on your host OS to view the Apache welcome page.
>
> Bonus work: Manage the httpd.conf file, and have it notify the service. Force Apache to be kept at a certain version (note that you'll have to research the format of the version strings for your operating system, as well as which versions are available).
>
> Hints:
>
> - On modern Red Hat-like (your VM) and Debian-like Linux systems, packages are installed from the operating system's Apt or Yum repositories. Since the system tools know how to find and install a package (or even a specific version of a package), all Puppet needs to know is `ensure => installed`; it doesn't need to know where the package lives.
> - The names of package and service resources depend on the OS's own naming conventions. This means you often need to do a bit of research before writing a manifest, to learn what the local name for, e.g., the Apache package and service are. On CentOS, which your VM runs, both the package and service are named `httpd`.
> - Make sure you're using the right `ensure` values for each resource type; they aren't the same for package, file, and service.
> - The core types cheat sheet and the type reference are your friends.

Next

Next Lesson:

Now that you can express dependencies between resources, it's time to make your manifests more aware of the outside world with variables, facts, and conditionals.

Off-Road:

Now that you can manage a complete service from top to bottom, try managing an important service on your own test systems. Download Puppet Enterprise for free, follow the quick start guide to get a small environment installed, then try building a package/file/service pattern at the bottom of the puppet master's `/etc/puppetlabs/puppet/manifests/site.pp` file. MySQL? Memcached? You decide.

Learning Puppet — Variables, Conditionals, and Facts

Begin

```
$my_variable = "A bunch of text"
notify {$my_variable:}
```

Yup, that's a variable, all right.

Variables

Variables! You've almost definitely used variables before in some other programming or scripting language, so we'll cover the basics very quickly. A more complete explanation of the syntax and behavior of variables is available in the variables chapter of the Puppet reference manual.

- `$variables` always start with a dollar sign. You assign to variables with the `=` operator.
- Variables can hold strings, numbers, booleans, arrays, hashes, and the special `undef` value. See the data types chapter of the Puppet reference manual for more information.
- If you've never assigned a variable, you can actually still use it — its value will be `undef`.
- You can use variables as the value for any resource attribute, or as the title of a resource.
- You can also interpolate variables inside double-quoted strings. To distinguish a variable from the surrounding text, you can wrap its name in curly braces. (`"This is the ${variable} name."`) This isn't mandatory, but it is recommended.
- Every variable has two names:
  - A short local name
  - A long fully-qualified name

  Fully qualified variables look like `$scope::variable`. Top scope variables are the same, but their scope is nameless. (For example: `$::top_scope_variable`.)
- If you reference a variable with its short name and it isn't present in the local scope, Puppet will also check the global top scope; this means you can almost always refer to global variables with just their short names. You can see more about this in the scope chapter of the Puppet reference manual: scope in Puppet Enterprise 2.x and Puppet 2.7, scope in Puppet 3
- You can only assign the same variable once in a given scope. In this way, they're more like constants from other programming languages.

```
$longthing = "Imagine I have something really long in here. Like an SSH
key, let's say."

file {'authorized_keys':
  path    => '/root/.ssh/authorized_keys',
  content => $longthing,
}
```

Pretty easy.

Aside: Why Do Everyone's Manifests Seem to Use $::ipaddress?

People who write manifests to share with the public often adopt the habit of always using the `$::variable` notation when referring to facts.

As mentioned above, the double-colon prefix specifies that a given variable should be found at top scope. This isn't actually necessary, since variable lookup will always reach top scope anyway. (See the scope chapter of the Puppet reference manual.)

However, explicitly asking for top scope helps work around two issues that can make public code behave unpredictably. One issue affects all versions of Puppet 2.x, and the other affected earlier versions of Puppet 2.7.x:

- In Puppet 2.x: if a user declares a class from a public module inside one of their own classes, and their personal class sets a variable whose name matches the name of a fact that the public class is trying to access, the public class will get the local variable instead of the fact. This will generally cause the public class to fail or do something really strange.
- In earlier versions of Puppet 2.7.x: the dynamic scope deprecation warnings would sometimes be improperly triggered when manifests accessed top scopes variables without the double-colon prefix. This was fixed in later versions, but was very annoying for a while.

Neither of these issues are relevant as of Puppet 3, but not everyone is using Puppet 3 yet, and a Puppet 3-based version of Puppet Enterprise is still forthcoming later this year. Since a lot of people are still writing public code meant to be used with Puppet 2.7, you still see this idiom a lot.

Facts

Puppet has a bunch of built-in, pre-assigned variables that you can use. Check it out:

```
    # /root/examples/motd.pp

    file {'motd':
      ensure  => file,
      path    => '/etc/motd',
      mode    => 0644,
      content => "This Learning Puppet VM's IP address is ${ipaddress}. It
thinks its
    hostname is ${fqdn}, but you might not be able to reach it there
    from your host machine. It is running ${operatingsystem}
${operatingsystemrelease} and
    Puppet ${puppetversion}.
    Web console login:
      URL: https://${ipaddress_eth0}
```

```
      User: puppet@example.com
      Password: learningpuppet
    ",
    }
```

```
# puppet apply /root/examples/motd.pp

notice: /Stage[main]//Host[puppet]/ensure: created
notice: /Stage[main]//File[motd]/ensure: defined content as
'{md5}bb1a70a2a2ac5ed3cb83e1a8caa0e331'

# cat /etc/motd
This Learning Puppet VM's IP address is 172.16.52.135. It thinks its
hostname is learn.localdomain, but you might not be able to reach it there
from your host machine. It is running CentOS 5.7 and
Puppet 2.7.21 (Puppet Enterprise 2.8.1).
Web console login:
  URL: https://172.16.52.135
  User: puppet@example.com
  Password: learningpuppet
```

Our manifests are becoming more flexible, with pretty much no real work on our part.

What Are These Hostname and IPaddress Variables?

And where did they come from?

They're "facts." Puppet uses a tool called Facter, which discovers some system information, normalizes it into a set of variables, and passes them off to Puppet. Puppet's compiler then has access to those facts when it's reading a manifest.

- See here for a list of all of the "core" facts built into Facter. Most of them are always available to Puppet, although some of them are only present on certain system types.
- You can see what Facter knows about a given system by running `facter` at the command line.
- You can also see all of the facts for any node in your Puppet Enterprise deployment by browsing to that node's page in the console and scrolling down to the inventory information.
- You can also add new custom facts to Puppet; see the custom facts guide for more information.

Other Built-In Variables

In addition to the facts from Facter, Puppet has a few extra built-in variables. You can see a list of them in the variables chapter of the Puppet reference manual.

Conditional Statements

Puppet has several kinds of conditional statements. You can see more complete info about them in the conditional statements chapter of the Puppet reference manual..

By using facts as conditions, you can easily make Puppet do different things on different kinds of

systems.

If

We'll start with your basic `if` statement. Same as it ever was:

```
if condition {
  block of code
}
elsif condition {
  block of code
}
else {
  block of code
}
```

- The `else` and any number of `elsif` statements are optional.
- The blocks of code for each condition can contain any Puppet code.
- The conditions can be any fragment of Puppet code that resolves to a boolean true/false value, including expressions, functions that return values, and variables. Follow the links for more detailed descriptions of expressions and functions.

An example `if` statement:

```
    if str2bool("$is_virtual") {
      service {'ntpd':
        ensure => stopped,
        enable => false,
      }
    }
    else {
      service { 'ntpd':
        name       => 'ntpd',
        ensure     => running,
        enable     => true,
        hasrestart => true,
        require => Package['ntp'],
      }
    }
```

Aside: Beware of the Fake False!

In the example above, we see something new: `str2bool("$is_virtual")`.

The condition for an `if` statement has to resolve to a boolean true/false value. However, all facts are strings, and all non-empty strings — including the string `"false"` — are true. This means that facts that are "false" need to be transformed before Puppet will treat them as

false.

In this case, we're:

- Surrounding the variable with double quotes — if it contained an actual boolean for some reason (and it usually wouldn't), this would convert it to a string.
- Passing the string to the `str2bool` function, which converts a string that looks like a boolean into a real true or false value.

The `str2bool` function is part of the puppetlabs/stdlib module, which is included with Puppet Enterprise. If you are running open source Puppet, you can install it by running `sudo puppet module install puppetlabs/stdlib`.

We could also use an expression instead: the expression `$is_virtual == 'true'` would resolve to true if the `is_virtual` fact has a value of true, and false otherwise.

Case

Another kind of conditional is the case statement. (Or switch, or whatever your language of choice calls it.)

```
case $operatingsystem {
  centos: { $apache = "httpd" }
  # Note that these matches are case-insensitive.
  redhat: { $apache = "httpd" }
  debian: { $apache = "apache2" }
  ubuntu: { $apache = "apache2" }
  default: { fail("Unrecognized operating system for webserver") }
}
package {'apache':
  name   => $apache,
  ensure => latest,
}
```

Instead of testing a condition up front, `case` matches a variable against a bunch of possible values. `default` is a special value, which does exactly what it sounds like.

In this example, we also see the `fail` function. Unlike the `str2bool` function above, `fail` doesn't resolve to a value; instead, it fails compilation immediately with an error message.

CASE MATCHING

Case matches can be simple strings (like above), regular expressions, or comma-separated lists of either.

Here's the example from above, rewritten to use comma-separated lists of strings:

```
case $operatingsystem {
```

```
    centos, redhat: { $apache = "httpd" }
    debian, ubuntu: { $apache = "apache2" }
    default: { fail("Unrecognized operating system for webserver") }
  }
```

And here's a regex example:

```
  case $ipaddress_eth0 {
    /^127[\d.]+$/: {
      notify {'misconfig':
        message => "Possible network misconfiguration: IP address of $0",
      }
    }
  }
```

String matching is case-insensitive, like the `==` comparison operator. Regular expressions are denoted with the slash-quoting used by Perl and Ruby; they're case-sensitive by default, but you can use the `(?i)` and `(?-i)` switches to turn case-insensitivity on and off inside the pattern. Regex matches also assign captured subpatterns to `$1`, `$2`, etc. inside the associated code block, with `$0` containing the whole matching string. See the regular expressions section of the Puppet reference manual's data types page for more details.

Selectors

Selectors might be less familiar; they're kind of like the common ternary operator, and kind of like the case statement.

Instead of choosing between a set of code blocks, selectors choose between a group of possible values. You can't use them on their own; instead, they're usually used to assign a variable.

```
  $apache = $operatingsystem ? {
    centos                => 'httpd',
    redhat                => 'httpd',
    /(?i)(ubuntu|debian)/ => 'apache2',
    default               => undef,
  }
```

Careful of the syntax, there: it looks kind of like we're saying `$apache = $operatingsystem`, but we're not. The question mark flags `$operatingsystem` as the control variable of a selector, and the actual value that gets assigned is determined by which option `$operatingsystem` matches. Also note how the syntax differs from the case syntax: it uses hash rockets and line-end commas instead of colons and blocks, and you can't use lists of values in a match. (If you want to match against a list, you have to fake it with a regular expression.)

It can look a little awkward, but there are plenty of situations where it's the most concise way to get a value sorted out; if you're ever not comfortable with it, you can just use a case statement to assign

the variable instead.

Selectors can also be used directly as values for a resource attribute, but try not to do that, because it gets ugly fast.

Exercises

> **Exercise: Build Environment**
>
> Use the $operatingsystem fact to write a manifest that installs a C build environment on Debian-based (“debian,” “ubuntu”) and Enterprise Linux-based (“centos,” “redhat”) machines. (Both types of system require the `gcc` package, but Debian-type systems also require `build-essential`.)

> **Exercise: Simple NTP**
>
> Write a manifest that installs and configures NTP for Debian-based and Enterprise Linux-based Linux systems. This will be a package/file/service pattern where both kinds of systems use the same package name (`ntp`), but you’ll be shipping different config files (Debian version, Red Hat version – remember the `file` type’s “source” attribute) and using different service names (`ntp` and `ntpd`, respectively).

Next

Next Lesson:

Now that your manifests can adapt to different kinds of systems, it’s time to start grouping resources and conditionals into meaningful units. Onward to classes, defined resource types, and modules!

Off-Road:

Since facts from every node show up in the console, Puppet Enterprise can be a powerful inventory tool. Download Puppet Enterprise for free, follow the quick start guide to get a small environment installed, then try browsing the console’s inventory for a central view of your operating system versions, hardware profiles, and more.

Learning Puppet — Modules and Classes

Begin

```
class my_class {
  notify {"This actually did something":}
}
```

This manifest does nothing.

```
class my_class {
  notify {"This actually did something":}
}

include my_class
```

This one actually does something.

Spot the difference?

The End of the One Huge Manifest

You can write some pretty sophisticated manifests at this point, but so far you've just been putting them in one file (either `/etc/puppetlabs/puppet/manifests/site.pp` or a one-off to use with puppet apply).

Past a handful of resources, this gets unwieldy. You can probably already see the road to the three thousand line manifest of doom, and you don't want to go there. It's much better to split chunks of logically related code out into their own files, and then refer to those chunks by name when you need them.

Classes are Puppet's way of separating out chunks of code, and modules are Puppet's way of organizing classes so that you can refer to them by name.

Classes

Classes are named blocks of Puppet code, which can be created in one place and invoked elsewhere.

- Defining a class makes it available by name, but doesn't automatically evaluate the code inside it.
- Declaring a class evaluates the code in the class, and applies all of its resources.

For the next five minutes, we'll keep working in a single manifest file; either a one-off, or site.pp. In a few short paragraphs, we'll start separating code out into additional files.

Defining a Class

Before you can use a class, you must define it, which is done with the `class` keyword, a name, curly braces, and a block of code:

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

What goes in that block of code? How about your answer from last chapter's NTP exercise? It should look a little like this:

```
# /root/examples/modules1-ntp1.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

  package { 'ntp':
    ensure => installed,
  }
  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/examples/answers/${conf_file}"
  }
  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }
}
```

That's a working class definition!

Note: You can download some basic NTP config files here: Debian version, Red Hat version.

ASIDE: CLASS NAMES

Class names must start with a lowercase letter, and can contain lowercase letters, numbers, and underscores.

Class names can also use a double colon (`::`) as a namespace separator. (This should look familiar.) Namespaces must map to module layout, which we'll cover below.

ASIDE: VARIABLE SCOPE

Each class definition introduces a new variable scope. This means:

- Any variables you assign inside the class won't be accessible by their short names outside the class; to get at them from elsewhere, you would have to use the fully-qualified name (e.g. `$ntp::service_name`, from our example above).
- You can assign new, local values to variable names that were already used at top scope. For example, you could specify a new local value for $fqdn.

Declaring

Okay, remember how we said that defining makes a class available, and declaring evaluates it? We can see that in action by trying to apply our manifest above:

```
# puppet apply /root/examples/modules1-ntp1.pp
notice: Finished catalog run in 0.04 seconds
```

...which does nothing, because we only defined the class.

To declare a class, use the `include` function with the class's name:

```
# /root/examples/modules1-ntp2.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

  package { 'ntp':
    ensure => installed,
  }
  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/examples/answers/${conf_file}"
  }
  service { 'ntp':
    name      => $service_name,
    ensure    => running,
```

```
    enable    => true,
    subscribe => File['ntp.conf'],
  }
}

include ntp
```

This time, Puppet will actually apply all those resources:

```
# puppet apply /root/examples/ntp-class1.pp

notice: /Stage[main]/Ntp/File[ntp.conf]/content: content changed
'{md5}5baec8bdbf90f877a05f88ba99e63685' to
'{md5}dc20e83b436a358997041a4d8282c1b8'
notice: /Stage[main]/Ntp/Service[ntp]/ensure: ensure changed 'stopped' to
'running'
notice: /Stage[main]/Ntp/Service[ntp]: Triggered 'refresh' from 1 events
notice: Finished catalog run in 0.76 seconds
```

Classes: Define, then declare.

Modules

You know how to define and declare classes, but we're still doing everything in a single manifest, where they're not very useful.

To help you split up your manifests into an easier to understand structure, Puppet uses modules and the module autoloader.

It works like this:

- Modules are just directories with files, arranged in a specific, predictable structure. The manifest files within a module have to obey certain naming restrictions.
- Puppet looks for modules in a specific place (or list of places). This set of directories is known as the `modulepath`, which is a configurable setting.
- 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.

This means you can have a pile of modules with sophisticated Puppet code, and your site.pp manifest can look like this:

```
# /etc/puppetlabs/puppet/manifests/site.pp
include ntp
include apache
include mysql
include mongodb
include build_essential
```

By stowing the implementation of a feature in a module, your main manifest can become much smaller, more readable, and policy-focused — you can tell at a glance what will be configured on your nodes, and if you need implementation details on something, you can delve into the module.

The Modulepath

Before we make a module, we need to know where to put it. So we'll find our modulepath, the set of directories that Puppet searches for modules.

The Puppet config file is called puppet.conf, and in Puppet Enterprise it is located at `/etc/puppetlabs/puppet/puppet.conf`:

```
# less /etc/puppetlabs/puppet/puppet.conf

[main]
    vardir = /var/opt/lib/pe-puppet
    logdir = /var/log/pe-puppet
    rundir = /var/run/pe-puppet
    modulepath =
/etc/puppetlabs/puppet/modules:/opt/puppet/share/puppet/modules
    user = pe-puppet
    group = pe-puppet
    archive_files = true
    archive_file_server = learn.localdomain

[master]
    ... etc.
```

The format of puppet.conf is explained in the configuration guide, but in short, the `[main]` section has settings that apply to everything (puppet master, puppet apply, puppet agent, etc.), and it sets the value of `modulepath` to a colon-separated list of two directories:

- `/etc/puppetlabs/puppet/modules`
- `/opt/puppet/share/puppet/modules`

The first, `/etc/puppetlabs/puppet/modules`, is the main module directory we'll be using. (The other one contains special modules that Puppet Enterprise uses to configure its own features; you can look in these, but shouldn't change them or add to them.)

> **ASIDE: CONFIGPRINT**
>
> You can also get the value of the modulepath by running `puppet master --configprint modulepath`. The `--configprint` option lets you get the value of any Puppet setting; by using the `master` subcommand, we're making sure we get the value the puppet master will use.

Module Structure

- A module is a directory.
- The module's name must be the name of the directory.
- It contains a `manifests` directory, which can contain any number of .pp files.
- The `manifests` directory should always contain an `init.pp` file.
    - This file must contain a single class definition. The class's name must be the same as the module's name.

There's more to know, but this will get us started. Let's turn our NTP class into a real module:

```
# cd /etc/puppetlabs/puppet/modules
# mkdir -p ntp/manifests
# touch ntp/manifests/init.pp
```

Edit this init.pp file, and paste your ntp class definition into it. Be sure not to paste in the `include` statement; it's not necessary here.

```
# /etc/puppetlabs/puppet/modules/ntp/manifests/init.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

  package { 'ntp':
    ensure => installed,
  }
  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/examples/answers/${conf_file}"
  }
  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }
}
```

Declaring Classes From Modules

Now that we have a working module, you can edit your site.pp file: if there are any NTP-related resources left in it, be sure to delete them, then add one line:

```
include ntp
```

Turn off the NTP service, then do a foreground puppet agent run so you can see the action:

```
# service ntpd stop
# puppet agent --test

notice: /Stage[main]/Ntp/Service[ntp]/ensure: ensure changed 'stopped' to
'running'
```

It worked!

More About Declaring Classes

Once a class is stored in a module, there are actually several ways to declare or assign it. You should try each of these right now by manually turning off the ntpd service, declaring or assigning the class, and doing a Puppet run in the foreground.

Include

We already saw this: you can declare classes by putting `include ntp` in your main manifest.

The `include` function declares a class, if it hasn't already been declared somewhere else. If a class HAS already been declared, `include` will notice that and do nothing.

This lets you safely declare a class in several places. If some class depends on something in another class, it can declare that class without worrying whether it's also being declared in site.pp.

Resource-Like Class Declarations

These look like resource declarations, except with a resource type of "class:"

```
    class {'ntp':}
```

These behave differently, acting more like resources than like the `include` function. Remember we've seen that you can't declare the same resource more than once? The same holds true for resource-like class declarations. If Puppet tries to evaluate one and the class has already been declared, it will fail compilation with an error.

However, unlike `include`, resource-like declarations let you specify class parameters. We'll cover those in a later chapter, and go into more detail about why resource-like declarations are so strict.

The PE Console

You can also assign classes to specific nodes using PE's web console. You'll have to add the class to the console, then navigate to a node's page and assign the class to that node.

We'll go into more detail later about working with multiple nodes.

Module Structure, Part 2

We're not quite done with this module yet. Notice how the `source` attribute of the config file is pointing to an arbitrary local path? We can move those files inside the module, and make everything self-contained:

```
# mkdir /etc/puppetlabs/puppet/modules/ntp/files
# mv /root/examples/answers/ntp.conf.*
/etc/puppetlabs/puppet/modules/ntp/files/
```

Then, edit the init.pp manifest; we'll use the special `puppet:///` URL format to tell Puppet where the files are:

```
    # ...
      file { 'ntp.conf':
        path    => '/etc/ntp.conf',
        ensure  => file,
        require => Package['ntp'],
        source  => "puppet:///modules/ntp/${conf_file}",
      }
    }
```

Now, everything the module needs is in one place. Even better, a puppet master can actually serve those files to agent nodes over the network now — when we were using `/root/examples/etc...` paths, Puppet would only find the source files if they already existed on the target machine.

The Other Subdirectories

We've seen two of the subdirectories in a module, but there are several more available:

- `manifests/` — Contains all of the manifests in the module.
- `files/` — Contains static files, which managed nodes can download.
- `templates/` — Contains templates, which can be referenced from the module's manifests. More on templates later.
- `lib/` — Contains plugins, like custom facts and custom resource types.
- `tests/` or `examples/` — Contains example manifests showing how to declare the module's classes and defined types.

- `spec/` — Contains test files written with rspec-puppet.

> Our printable Module Cheat Sheet shows how to lay out a module and explains how in-manifest names map to the underlying files; it's a good quick reference when you're getting started. The Puppet reference manual also has a page of info about module layout.

This is a good time to explain more about how the `manifests` and `files` directories work:

Organizing and Referencing Manifests

Each manifest in a module should contain exactly one class or defined type. (More on defined types later.)

Each manifest's filename must map to the name of the class or defined type it contains. The init.pp file, which we used above, is special — it always contains a class (or defined type) with the same name as the module. Every other file must contain a class (or defined type) named as follows:

```
<MODULE NAME>::<FILENAME>
```

…or, if the file is inside a subdirectory of `manifests/`, it should be named:

```
<MODULE NAME>::<SUBDIRECTORY NAME>::<FILENAME>
```

So for example, if we had an apache module that contained a mod_passenger class:

- File on disk: `apache/manifests/mod_passenger.pp`
- Name of class in file: `apache::mod_passenger`

You can see more detail about this mapping at the namespaces and autoloading page of the Puppet reference manual.

Organizing and Referencing Files

Static files can be arranged in any directory structure inside the `files/` directory.

When referencing these files in Puppet manifests, as the `source` attributes of file resources, you should use `puppet:///` URLs. These have to be structured in a certain way:

Protocol	3 slashes	"modules"/	Name of module/	Name of file
`puppet:`	`///`	`modules/`	`ntp/`	`ntp.conf.el`

Note that the final segment of the URL starts inside the `files/` directory of the module. If there are any extra subdirectories, they work like you'd expect, so you could have something like `puppet:///modules/ntp/config_files/linux/ntp.conf.el`.

The Puppet Forge: How to Avoid Writing Modules

Now that you know how modules work, you can also use modules written by other users.

The Puppet Forge is a repository of free modules you can install and use. Most of these modules are open source, and you can easily contribute updates and changes to improve or enhance these modules. You can also contribute your own modules.

The Puppet Module Subcommand

Puppet ships with a module subcommand for installing and managing modules from the Puppet Forge. Detailed instructions for using it can be found in the Puppet reference manual's "installing modules" page. Some quick examples:

Install the puppetlabs-mysql module:

```
$ sudo puppet module install puppetlabs-mysql
```

List all installed modules:

```
$ sudo puppet module list
```

USER NAME PREFIXES

Modules from the Puppet Forge have a user name prefix in their names; this is done to avoid name clashes between, for example, all of the Apache modules out there.

The puppet module subcommand handles these user name prefixes automatically — it preserves them as metadata, but installs the module under its common name. That is, your Puppet manifests would refer to a `mysql` module instead of the `puppetlabs-mysql` module.

Exercises

Exercise: Apache Again

Building on your work from two chapters ago, create an Apache module and class, which ensures Apache is installed and running and manages its config file. Bonus work: Make Puppet manage the DocumentRoot folder, and put a custom 404 page and default index.html in place. You can also use conditional statements to set any files or package/service names that might vary per OS; if you don't want to research the names used by other OSes, you can just have the class fail if it's not used on CentOS.

Next

Next Lesson:

What's with that `templates/` folder in the module structure? And can we do anything more interesting with config files than just replacing them with static content? Find out in the Templates chapter.

Off-Road:

Since you know how to install free modules from the Puppet Forge, and how to declare the classes inside those modules, search around and try to find some modules that might be useful in your infrastructure. Then download Puppet Enterprise for free, follow the quick start guide to get a small environment installed, and try managing complex services on some of your test nodes.

Learning Puppet — Templates

Begin

Let's make a small adjustment to our NTP module from the last chapter: remove the `source` attribute from the file resource, and replace it with a `content` attribute using a new function. (Remember that `source` specifies file contents as a file, and `content` specifies file contents as a string.)

```
    # /etc/puppetlabs/puppet/modules/ntp/manifests/init.pp

    class ntp {
      #...
      #...
      #...
      file { 'ntp.conf':
        path    => '/etc/ntp.conf',
        ensure  => file,
        require => Package['ntp'],
        content => template("ntp/${conf_file}.erb"),
      }
    }
```

Then, copy the config files into the templates directory:

```
# cd /etc/puppetlabs/puppet/modules/ntp
# mkdir templates
# cp files/ntp.conf.el templates/ntp.conf.el.erb
# cp files/ntp.conf.debian templates/ntp.conf.debian.erb
```

The module should work the same way it's been working, but your config files are no longer static files; they're templates.

Stopping the Static Content Explosion

So consider our NTP module.

Right now, we're shipping around two different config files, which resemble the defaults for Red Hat-like and Debian-like OSes. What if we wanted to make a few small and reasonable changes? For example:

- Use different NTP servers for a small number of machines
- Adjust the settings on virtual machines, so NTP doesn't panic if the time jumps

We could end up maintaining eight or more different config files! Let's not do that. Instead, we can manage a bunch of small differences in one or two template files.

Templates are documents that contain a mixture of static and dynamic content. By using a small amount of conditional logic and variable interpolation, they let you maintain one source document that can be rendered into any number of final documents.

For more details on the behavior of Puppet templates, see the guide for Using Puppet Templates; we'll cover the basics right here.

Template Files

Templates are saved as files with the .erb extension, and should be stored in the `templates/` directory of any module. There can be any number of subdirectories inside `templates/`.

Rendering Templates

To use a template, you have to render it to produce an output string. To do this, use Puppet's built-in `template` function. This function takes a path to one or more template files and returns an output string:

```
file {'/etc/foo.conf':
  ensure  => file,
  require => Package['foo'],
  content => template('foo/foo.conf.erb'),
}
```

Notice that we're using the output string as the value of the `content` attribute — it wouldn't work with the `source` attribute, which expects a URL rather than the actual content for a file.

Refererring to Template Files in Modules

The `template` function expects file paths to be in a specific format:

`<MODULE NAME>/<FILENAME INSIDE TEMPLATES DIRECTORY>`

That is, `template('foo/foo.conf.erb')` would point to the file `/etc/puppetlabs/puppet/modules/foo/templates/foo.conf.erb`.

> Note that the path to the template doesn't use the same semantics as the path in a `puppet:///` URL. Sorry about the inconsistency.

Inline Templates

Alternately, you can use the `inline_template` function, which takes a string containing a template and returns an output string.

This is less frequently useful, but if you have a very small template, you can sometimes embed it in the manifest instead of making a whole new file for it.

> #### Aside: Functions in General
>
> We've seen several functions already, including `include`, `template`, `fail`, and `str2bool`, so this is as good a time as any to explain what they are.
>
> Puppet has two kinds of functions:
>
> - Functions that return a value
> - Functions that do something else, without returning a value
>
> The `template` and `str2bool` functions both return values; you can use them anywhere that requires a value, as long as the return value is the right kind. The `include` and `fail` functions do something else, without returning a value — declare a class, and stop catalog compilation, respectively.
>
> All functions are run during catalog compilation. This means they run on the puppet master, and don't have access to any files or settings on the agent node.
>
> Functions can take any number of arguments, which are separated by commas and can be surrounded by optional parentheses:
>
> `function(argument, argument, argument)`
>
> Functions are plugins, so many custom plugins are available in modules.
>
> Complete documentation about functions are available at the functions page of the Puppet reference manual and the list of built-in functions.

Variables in Templates

Templates are powerful because they have access to all of the Puppet variables that are present when the template is rendered.

- Facts, global variables, and local variables from the current scope are available to a template as Ruby instance variables — instead of Puppet's `$` prefix, they have an `@` prefix. (e.g. `@fqdn, @memoryfree, @operatingsystem`, etc.)
- Variables from other scopes can be accessed with the `scope.lookupvar` method, which takes a long variable name without the `$` prefix. (For example, `scope.lookupvar('apache::user')`.)

The ERB Templating Language

Puppet doesn't have its own templating language; instead, it uses ERB, a common Ruby-based template language. (The Rails framework uses ERB, as do several other projects.)

ERB templates mostly look like normal configuration files, with the occasional `<% tag containing Ruby code %>`. The ERB syntax is documented here, but since tags can contain any Ruby code, it's possible for templates to get pretty complicated.

In general, we recommend keeping templates as simple as possible: we'll show you how to print variables, do conditional statements, and iterate over arrays, which should be enough for most tasks.

Non-Printing Tags

ERB tags are delimited by angle brackets with percent signs just inside. (There isn't any HTML-like concept of opening or closing tags.)

```
<% document = "" %>
```

Tags contain one or more lines of Ruby code, which can set variables, munge data, implement control flow, or… actually, pretty much anything, except for print text in the rendered output.□

Printing an Expression

For that, you need to use a printing tag, which looks like a normal tag with an equals sign right after the opening delimiter:

```
<%= sectionheader %>
  environment = <%= gitrevision[0,5] %>
```

The value you print can be a simple variable, or it can be an arbitrarily complicated Ruby

expression.

Comments

A tag with a hash mark right after the opening delimiter can hold comments, which aren't interpreted as code and aren't displayed in the rendered output.

```
<%# This comment will be ignored. %>
```

Suppressing Line Breaks and Leading Space

Regular tags don't print anything, but if you keep each tag of logic on its own line, the line breaks you use will show up as a swath of whitespace in the final file. Similarly, if you're indenting for readability, the whitespace in the indent can mess up the format of the rendered output.

If you don't like that, you can:

- Trim line breaks by putting a hyphen directly before the closing delimiter
- Trim leading space by putting a hyphen directly after the opening delimiter

```
<%- document += thisline -%>
```

An Example: NTP Again

Let's make the templates in your NTP module a little more clever.

First, make sure you change the file resource to use a template, like we saw at the top of this page. You should also make sure you've copied the config files to the `templates/` directory and given them the .erb extension.

Adjusting the Manifest

Next, we'll move the default NTP servers out of the config file and into the manifest:

```
# /etc/puppetlabs/puppet/modules/ntp/manifests/init.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name    = 'ntpd'
      $conf_file   = 'ntp.conf.el'
      $default_servers = [ "0.centos.pool.ntp.org",
                           "1.centos.pool.ntp.org",
                           "2.centos.pool.ntp.org", ]
    }
    debian, ubuntu: {
      $service_name    = 'ntp'
      $conf_file   = 'ntp.conf.debian'
```

```
        $default_servers = [ "0.debian.pool.ntp.org iburst",
                             "1.debian.pool.ntp.org iburst",
                             "2.debian.pool.ntp.org iburst",
                             "3.debian.pool.ntp.org iburst", ]
      }
    }

    $servers_real = $default_servers

    package { 'ntp':
      ensure => installed,
    }

    service { 'ntp':
      name      => $service_name,
      ensure    => running,
      enable    => true,
      subscribe => File['ntp.conf'],
    }

    file { 'ntp.conf':
      path    => '/etc/ntp.conf',
      ensure  => file,
      require => Package['ntp'],
      content => template("ntp/${conf_file}.erb"),
    }
  }
```

We're storing the servers in an array, so we can show how to iterate within a template. Right now, we're not providing the ability to change the list of servers, but we're paving the way to do so in the next chapter.

Editing the Templates

First, make each template use the `$servers_real` variable to create the list of `server` statements:

```
<%# /etc/puppetlabs/puppet/modules/ntp/templates/ntp %>

# Managed by Class['ntp']
<% @servers_real.each do |this_server| -%>
server <%= this_server %>
<% end -%>

# ...
```

What's this doing?

- Using a non-printing Ruby tag to start a loop. We reference the `$servers_real` Puppet variable by the name `@servers_real`, then call Ruby's `each` method on it. Everything between `do |server| -%>` and the `<% end -%>` tag will be repeated for each item in the `$servers_real` array, with the value of that array item being assigned to the temporary `this_server` variable.

- Within the loop, we print the literal word `server`, followed by the value of the current array item.

This snippet will produce something like the following:

```
# Managed by Class['ntp']
server 0.centos.pool.ntp.org
server 1.centos.pool.ntp.org
server 2.centos.pool.ntp.org
```

Next, let's use the `$is_virtual` fact to make NTP perform better if this is a virtual machine. At the top of the file, add this:

```
    <% if @is_virtual == "true" -%>
    # Keep ntpd from panicking in the event of a large clock skew
    # when a VM guest is suspended and resumed.
    tinker panic 0

    <% end -%>
```

Then, below the loop we made for the server statements, add this (being sure to replace the similar section of the Red Hat-like template):

```
    <% if @is_virtual == "false" -%>
    # Undisciplined Local Clock. This is a fake driver intended for backup
    # and when no outside source of synchronized time is available.
    server 127.127.1.0 # local clock
    fudge 127.127.1.0 stratum 10

    <% end -%>
```

By using facts to conditionally switch parts of the config file on and off, we can easily react to the type of machine we're managing.

Next

Next Lesson:

We've already seen that classes should sometimes behave differently for different kinds of systems, and have used facts to make conditional changes to both manifests and templates.

Sometimes, though, facts aren't enough — there are times when a human has to decide what makes a machine different, because that difference is a matter of policy. (For example, the difference between a test server and a production server.)

In these cases, we need to give ourselves a way to manually change the way a class works. We can do this by passing in data with class parameters.

Off-Road:

Are you managing any configuration on your real infrastructure yet? You've learned a lot by now, so why not download Puppet Enterprise for free, follow the quick start guide to get a small environment installed, and start automating?

Learning Puppet — Class Parameters

Begin

```
class echo_class ($to_echo = "default value") {
  notify {"What are we echoing? ${to_echo}.":}
}

class {'echo_class':
  to_echo => 'Custom value',
}
```

There's something different about that variable.

Investigating vs. Asking For Help

Most classes have to do slightly different things on different systems. You already know some ways to do that — all of the modules you've written so far have switched their behaviors by looking up system facts. Let's say that they "investigate:" they expect some information to be in a specific place (in the case of facts, a top-scope variable), and go looking for it when they need it.

But this isn't always the best way to do it, and it starts to break down once you need to switch a module's behavior on information that doesn't map cleanly to system facts. Is this a database server? A local NTP server? A test node? A production node? These aren't necessarily facts; usually, they're decisions made by a human.

In these cases, it's often best to just configure the class, and tell it what it needs to know when you declare it. To enable this, classes need some way to ask for information from the outside world.

Class Parameters

When defining a class, you can give it a list of parameters. Parameters go in an optional set of parentheses, between the name and the first curly brace. Each parameter is a variable name, and can have an optional default value; each parameter is separated from the next with a comma.

```
class mysql ($user = 'mysql', $port = 3306) {
  ...
}
```

This is a doorway for passing information into the class:

```
class {'mysql':
  user => mysqlserver,
}
```

- If you declare the class with a resource-like class declaration, the parameters are available as resource attributes.
- Inside the definition of the class, they appear as local variables.

Default Values

When defining the class, you can give any parameter a default value. This makes it optional when you declare the class; if you don't specify a value, it will use the default. Parameters without defaults become mandatory when declaring the class.

The Deal With Resource-Like Class Declarations

IN PUPPET ENTERPRISE 2.X

In Puppet 2.7, which is used in the Puppet Enterprise 2.x series, you must use resource-like class declarations if you want to specify class parameters; you cannot specify parameters with `include` or in the PE console. If every parameter has a default and you don't need to override any of them, you can declare the class with `include`; otherwise, you must use resource-like class declarations.

Resource-like declarations don't play nicely with `include`, and if you're using them, you need to organize your manifests so that they never attempt to declare a class more than once. This has traditionally been a pain, but class parameters are still superior to older ways of configuring classes, and the best practices developed over the course of the Puppet 2.7 series have made them much easier to deal with.

The best way to deal with class parameters in the Puppet Enterprise 2.x series is to create "role" and "profile" modules that combine your functional classes into more complete node descriptions. Once you find yourself managing multiple nodes with Puppet, you should read Craig Dunn's "Roles and Profiles" essay, which matches the best practices used by Puppet Labs's services engineers.

To make your roles and profiles more flexible and avoid repeating yourself, you can also install and configure Hiera on your puppet master and specify Hiera lookup functions as the values of class parameters.

WHY `INCLUDE` CAN'T DIRECTLY TAKE CLASS PARAMETERS

The problem is that classes are singletons, parameters configure the way they behave, and `include` can declare the same class more than once.

If you were to declare a class multiple times with different parameter values, which set of values should win? The question didn't seem to have a good answer. The older method of using magic

variables actually had this same problem — depending on parse order, there could be several different scope-chains that provided a given value, and the one you actually got would be effectively random. Icky.

The solution Puppet's designers settled on was that parameter values either had to be explicit and unconflicting (the restrictions on resource-like class declarations), or had to come from somewhere outside Puppet and be already resolved by the time Puppet's parsing begins (Puppet 3's automatic parameter lookup).

Older Ways to Configure Classes

Class parameters were added to Puppet in version 2.6.0, to address a need for a standard and visible way to configure clases.

Prior to that, people generally configured classes by choosing an arbitrary and unique external variable name and having the class retrieve that variable with dynamically-scoped variable lookup:

```
$some_variable
include some_class
# This class will reach outside its own scope, and hope
# it finds a value for $some_variable.
```

There were a few problems with this:

- Every class was competing for variable names in an effectively global name space. If you accidentally chose a non-unique name for your magic variables, something bad would happen.
- When writing modules to share with the world, you had to be very careful to document all of your magic variables; there wasn't a standard place a user could look to see what data a class needed.
- This inspired many many people to try and make intricate data hierarchies with node inheritance, which rarely worked and had a tendency to fail dramatically and confusingly.

Example: NTP (Again)

So let's get back to our NTP module. The first thing we talked about wanting to configure was the set of servers, so that's a good place to start. First, add a parameter:

```
    class ntp ($servers = undef) {
      ...
```

Next, we'll change how we set that `$servers_real` variable that the template uses:

```
      if $servers == undef {
        $servers_real = $default_servers
      }
      else {
        $servers_real = $servers
```

```
    }
```

If we specify an array of servers, use that; otherwise, use the defaults.

And… that's all it takes. If you declare the class with no attributes…

```
include ntp
```

…it'll work the same way it used to. If you declare it with a `servers` attribute containing an array of servers (with or without appended `iburst` and `dynamic` statements)…

```
class {'ntp':
  servers => [ "ntp1.example.com dynamic", "ntp2.example.com dynamic", ],
}
```

…it'll override the default servers in the `ntp.conf` file.

There's a bit of trickery to notice: setting a variable or parameter to `undef` might seem odd, and we're only doing it because we want to be able to get the default servers without asking for them. (Remember, parameters can't be optional without an explicit default value.)

Also, remember the business with the `$servers_real` variable? That was because the Puppet language won't let us re-assign the `$servers` variable within a given scope. If the default value we wanted was the same regardless of OS, we could just use it as the parameter default, but the extra logic to accomodate the per-OS defaults means we have to make a copy of the variable.

While we're in the NTP module, what else could we make into a parameter? Well, let's say you sometimes wanted to prevent the NTP daemon from being used as a server by other nodes. Or maybe you want to install and configure NTP, but not keep the daemon running. You could expose all of these as extra class parameters, and make changes in the manifest or the templates to use them.

All of these changes are based on decisions from the free puppetlabs/ntp module. [You can browse the source of this module] and see how these extra parameters play out in the manifest and templates.

Module Documentation

You have a fairly functional NTP module, at this point. About the only thing it's missing is some documentation:

```
# = Class: ntp
#
# This class installs/configures/manages NTP. It can optionally disable NTP
```

```
    # on virtual machines. Only supported on Debian-derived and Red Hat-derived
OSes.
    #
    # == Parameters:
    #
    # $servers:: An array of NTP servers, with or without +iburst+ and
    #            +dynamic+ statements appended. Defaults to the OS's defaults.
    # $enable::  Whether to start the NTP service on boot. Defaults to true.
Valid
    #            values: true and false.
    # $ensure::  Whether to run the NTP service. Defaults to running. Valid
values:
    #            running and stopped.
    #
    # == Requires:
    #
    # Nothing.
    #
    # == Sample Usage:
    #
    #   class {'ntp':
    #     servers => [ "ntp1.example.com dynamic",
    #                  "ntp2.example.com dynamic", ],
    #   }
    #   class {'ntp':
    #     enable => false,
    #     ensure => stopped,
    #   }
    #
    class ntp ($servers = undef, $enable = true, $ensure = running) {
      case $operatingsystem { ...
      ...
```

This doesn't have to be Tolstoy, but you should at least write down what the parameters are and what kind of data they take. Your future self will thank you. Also! If you write your documentation in RDoc format and put it in a comment block butted up directly against the start of the class definition, you can automatically generate a browsable Rdoc-style site with info for all your modules. You can test it now, actually:

```
# puppet doc --mode rdoc --outputdir ~/moduledocs --modulepath
/etc/puppetlabs/puppet/modules
```

(Then just upload that `~/moduledocs` folder to some webspace you control, or grab it onto your desktop with SFTP.)

Next

Next Lesson:

Okay, we can pass parameters into classes now and change their behavior. Great! But classes are still always singletons; you can't declare more than one copy and get two different sets of behavior

simultaneously. And you'll eventually want to do that! What if you had a collection of resources that created a virtual host definition for a web server, or cloned a Git repository, or managed a user account complete with group, SSH key, home directory contents, sudoers entry, and .bashrc/.vimrc/etc. files? What if you wanted more than one Git repo, user account, or vhost on a single machine?

Well, you'd whip up a defined resource type.

Learning Puppet — Defined Types

Beyond Singletons

Say you wrote a chunk of Puppet code that takes parameters and configures an individual Apache virtual host. You put it in a class. It works fine, but since classes are singletons, Puppet won't ever let you declare more than one vhost.

What you want is something more like a resource type — you can't declare the same resource twice, but you can declare as many files or users as you want.

```
apache::vhost {'users.example.com':
  port    => 80,
  docroot => '/var/www/personal',
  options => 'Indexes MultiViews',
}
```

This turns out to be easy. To model repeatable chunks of configuration — like a Git repository or an Apache vhost — you should use defined resource types.

Defined types act like normal resource types and are declared in the same way, but they're composed of other resources.

Defining a Type

You define a type with the `define` keyword, and the definition looks almost exactly like a class with parameters. You need:

- The `define` keyword
- A name
- A list of parameters (in parentheses, after the name)
  - Defined types also get a special `$title` parameter without having to declare it, and its value is always set to the title of the resource instance. (The `$name` parameter acts the same way, and usually has the same value as `$title`.) Classes get these too, but they're less useful since a class will only ever have one name.

- A block of Puppet code

Like this:

```
define planfile ($user = $title, $content) {
  file {"/home/${user}/.plan":
    ensure  => file,
    content => $content,
    mode    => 0644,
    owner   => $user,
    require => User[$user],
  }
}

user {'nick':
  ensure     => present,
  managehome => true,
  uid        => 517,
}
planfile {'nick':
  content => "Working on new Learning Puppet chapters. Tomorrow: upgrading the LP virtual machine.",
}
```

This one's pretty simple. (In fact, it's basically just a macro.) It has two parameters, one of which is optional (it defaults to the title of the resource), and the collection of resources it declares is just a single file resource.

Special Little Flowers

So it's pretty simple, right? Exactly like defining a class? Almost: there's one extra requirement. Since the user might declare any number of instances of a defined type, you have to make sure that the implementation will never declare the same resource twice.

Consider a slightly different version of that first definition:

```
define planfile ($user = $title, $content) {
  file {'.plan':
    path    => "/home/${user}/.plan",
    ensure  => file,
    content => $content,
    mode    => 0644,
    owner   => $user,
    require => User[$user],
  }
}
```

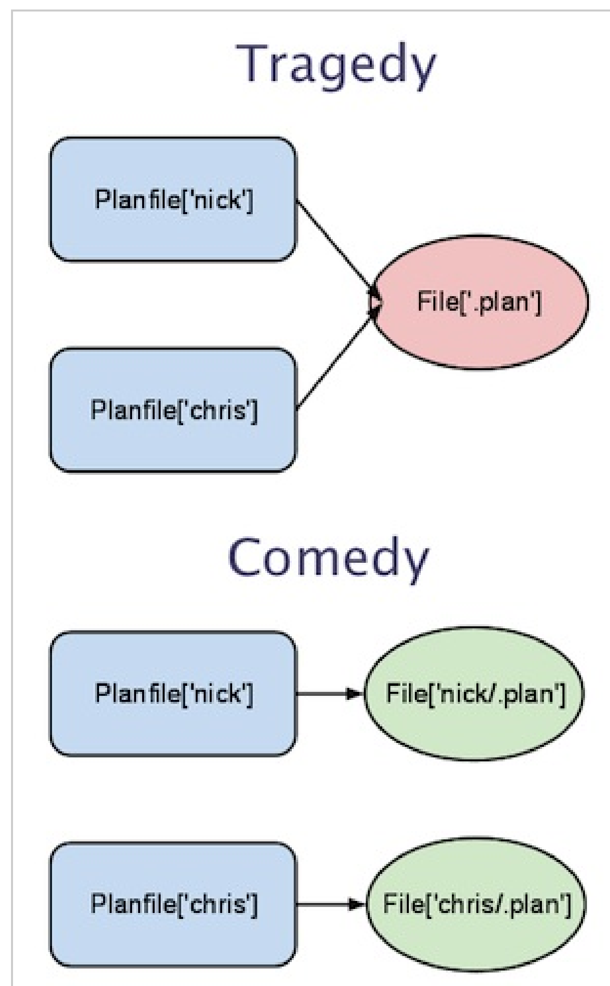
See how the title of the file resource isn't tied to any of the definition's parameters?

```
    planfile {'nick':
      content => "Working on new Learning Puppet chapters. Tomorrow: upgrading
the LP virtual machine.",
    }

    planfile {'chris':
      content => "Resurrecting a very dead laptop.",
    }
```

```
# puppet apply planfiles.pp
Duplicate definition: File[.plan] is already defined in file
/root/manifests/planfile.pp at line 9; cannot redefine at
/root/manifests/planfile.pp:9 on node puppet.localdomain
```

Yikes. You can see where we went wrong — every time we declare an instance of `planfile`, it's going to declare the resource `File['.plan']`, and Puppet will fail compilation if you try to declare the same resource twice.



To avoid this, you have to make sure that both the title and the name (or namevar) of every resource in the definition are somehow derived from a unique parameter (often the `$title`) of the defined type. (For example, we couldn't derive the file's title from the `$content` of the `planfile` resource, because more than one user might write the same .plan text.)

If there's a singleton resource that has to exist for any instance of the defined type to work, you should:

- Put that resource in a class.
- Inside the type definition, use `include` to declare that class.
- Also inside the type definition, use something like the following to establish an order dependency:

  ```
  # Make sure compilation will fail if 'myclass' doesn't get declared:
  Class['myclass'] -> Apache::Vhost["$title"]
  ```

  Establishing ordering relationships at the class level is generally better than directly requiring one of the resources inside it.

Defined Types in Modules

Defined types can be autoloaded just like classes, and thus used from anywhere in your manifests. Like with classes, each defined type should go in its own file in a module's `manifests/` directory, and the same rules for namespacing apply. (So the `apache::vhost` type would go somewhere like `/etc/puppetlabs/puppet/modules/apache/manifests/vhost.pp`, and if we were to keep the `planfile` type around, it would go in `/etc/puppetlabs/puppet/modules/planfile/manifests/init.pp`.)

Resource References and Namespaced Types

You might have already noticed this above, but: when you make a resource reference to an instance of a defined type, you have to capitalize every namespace segment in the type's name. That means an instance of the `foo::bar::baz` type would be referenced like `Foo::Bar::Baz['mybaz']`.

An Example: Apache Vhosts

Not that my .plan macro wasn't pretty great, but let's be serious for a minute. Remember your Apache module from a few chapters back? Let's extend it so we can easily declare vhosts. This example code is borrowed from the puppetlabs-apache module.

```
    # Definition: apache::vhost
    #
    # This class installs Apache Virtual Hosts
    #
    # Parameters:
    # - The $port to configure the host on
    # - The $docroot provides the DocumentationRoot variable
    # - The $template option specifies whether to use the default template or
override
    # - The $priority of the site
```

```
    # - The $serveraliases of the site
    # - The $options for the given vhost
    # - The $vhost_name for name based virtualhosting, defaulting to *
    #
    # Actions:
    # - Install Apache Virtual Hosts
    #
    # Requires:
    # - The apache class
    #
    # Sample Usage:
    #  apache::vhost { 'site.name.fqdn':
    #    priority => '20',
    #    port => '80',
    #    docroot => '/path/to/docroot',
    #  }
    #
    define apache::vhost(
        $port,
        $docroot,
        $template       = 'apache/vhost-default.conf.erb',
        $priority       = '25',
        $servername     = '',
        $serveraliases  = '',
        $options        = "Indexes FollowSymLinks MultiViews",
        $vhost_name     = '*'
      ) {

      include apache

      # Below is a pre-2.6.5 idiom for having a parameter default to the title,
      # but you could also just declare $servername = "$title" in the
parameters
      # list and change srvname to servername in the template.

      if $servername == '' {
        $srvname = $title
      } else {
        $srvname = $servername
      }
      case $operatingsystem {
        'centos', 'redhat', 'fedora': { $vdir   = '/etc/httpd/conf.d'
                                        $logdir = '/var/log/httpd'}
        'ubuntu', 'debian':           { $vdir   = '/etc/apache2/sites-enabled'
                                        $logdir = '/var/log/apache2'}
        default:                      { $vdir   = '/etc/apache2/sites-enabled'
                                        $logdir = '/var/log/apache2'}
      }
      file {
        "${vdir}/${priority}-${name}.conf":
          content => template($template),
          owner   => 'root',
          group   => 'root',
          mode    => '755',
          require => Package['httpd'],
          notify  => Service['httpd'],
      }
```

```
    }
```

```
    # /etc/puppetlabs/modules/apache/templates/vhost-default.conf.erb

    # ************************************
    # Default template in module puppetlabs-apache
    # Managed by Puppet
    # ************************************

    Listen <%= port %>
    NameVirtualHost <%= vhost_name %>:<%= port %>
    <VirtualHost <%= vhost_name %>:<%= port %>>
      ServerName <%= srvname %>
    <% if serveraliases.is_a? Array -%>
    <% serveraliases.each do |name| -%><%= "  ServerAlias #{name}\n" %><% end -
%>
    <% elsif serveraliases != '' -%>
    <%= "  ServerAlias #{serveraliases}" -%>
    <% end -%>
      DocumentRoot <%= docroot %>
      <Directory <%= docroot %>>
        Options <%= options %>
        AllowOverride None
        Order allow,deny
        allow from all
      </Directory>
      ErrorLog <%= logdir %>/<%= name %>_error.log
      LogLevel warn
      CustomLog <%= logdir %>/<%= name %>_access.log combined
      ServerSignature Off
    </VirtualHost>
```

And that’s more or less a wrap. You can apply a manifest like this:

```
    apache::vhost {'testhost':
      port => 8081,
      docroot => '/var/www-testhost',
      priority => 25,
      servername => 'puppet',
    }
```

…and (as long as the directory exists) you’ll immediately be able to reach the new vhost:

```
# curl http://puppet:8081
```

In a way, this is just slightly more sophisticated than the first example — it’s still only one `file` resource — but the use of a template makes it a LOT more powerful, and you can already see how much time it can save. And you can make it slicker as you build more types: once you’ve got a

custom type that handles firewall rules, for example, you can add something like this to the definition:

```
firewall {"0100-INPUT ACCEPT $port":
    jump  => 'ACCEPT',
    dport => "$port",
    proto => 'tcp'
}
```

Exercises

Take a minute to make a few more defined types, just to get used to modeling repeatable groups of resources.

- Try wrapping a `user` resource in a `human::user` type that manages that person's .bashrc file and manages one or more `ssh_authorized_key` resources for their account.
- If you're familiar with git, take a stab at writing a `git::repo` type that can clone from a repository on the network (and maybe even keep the working copy up-to-date on a specific branch!). This'll be harder — you'll probably have to make a `git` class to make sure git is available, and you'll have to use at least one `file` (`ensure => directory`) and at least one `exec` resource. Keep in mind that execs can be tricky, since you need to make sure they only run when necessary.

One Last Tip

Defined types take input, and input can get a little dirty — you might want to check your parameters to make sure they're the correct data type, and fail early if they're garbage instead of writing undefined stuff to the system.

If you're going to make a practice of validating your inputs (hint: DO), you can save yourself a lot of effort by using the validation functions in Puppet Labs' `stdlib` module. We ship a version of `stdlib` with PE 2.0, and you can also download it for free at either GitHub or the module forge. The functions are:

- `validate_array`
- `validate_bool`
- `validate_hash`
- `validate_re`
- `validate_string`

You can learn how to use these by running `puppet doc --reference function | less` on a system that has `stdlib` installed in its `modulepath`, or you can read the documentation directly in each of the functions' files — look in the `lib/puppet/parser/functions` directory of the module.

Next

Next Lesson:

There's more to say about modules — we still haven't covered data separation, patterns for making your modules more readable, or module composition yet — but there's more important business afoot. Continue reading to prepare your VMs (yes, plural) for agent/master Puppet.

Off-Road:

We've seen several Apache examples already, and it's pretty likely that you're running at least one web server in your own infrastructure. Why not use one of the off-the-shelf modules available, and see whether you can reproduce your own configuration in an automated way?

Download Puppet Enterprise for free, and follow the quick start guide to get a small environment installed on some test machines. Then, install one of the following modules:

- puppetlabs/apache
- simondean/iis (for IIS on Windows Server)
- Any of the many Nginx modules

Read the module's documentation to see how it works, then try managing the service and any relevant virtual hosts to match your manually configured infrastructure.

Learning Puppet — Preparing an Agent VM

We're now moving into exercises that involve both a puppet master and one or more agent nodes, and to get the most out of this section, you'll need to be running more than one node at a time. This interlude will help you get two nodes running and configured so that you're ready for the next few chapters.

Step 0: Get the Latest VM

Things have changed quite a bit since we posted the first Learning Puppet VM, so to keep these instructions simple, everything below will assume you're running a system with Puppet Enterprise 2.0 or later. You can check which version of PE you're running like this:

```
[root@learn ~]# puppet --version
2.7.9 (Puppet Enterprise 2.0.1)
```

If you're up to date, skip down to here. If you're running an older version, do one of the following:

Download the Latest VM

The latest version of the Learning Puppet VM is right over here. As ever, it's about 500 MB and free for everyone to download.

Before replacing your VM, make sure to save any manifests or modules from your previous copy. (After all, the whole point of Puppet is that you can use them to get right back to where you were.)

Or: Upgrade PE on Your Existing VM

If you can't download a whole new VM right now, you can:

- Download the latest version of Puppet Enterprise. Choose the EL 5 for i386 installer, which is about 50 MB.
- Copy the installer tarball to your VM and follow the upgrade instructions in the PE 2 User's Guide.

This is more advanced than just downloading the current VM, especially if you're upgrading from PE 1.0 or 1.1, but we've tried to document the process clearly. Follow the instructions for upgrading a combined master/console server.

Step 1: Duplicate Your Existing VM

There are any number of ways you could make an agent VM for the next steps, but the fastest and easiest is to simply copy your existing VM and reconfigure it. (The reconfiguring could be difficult, but there's a module for that.)

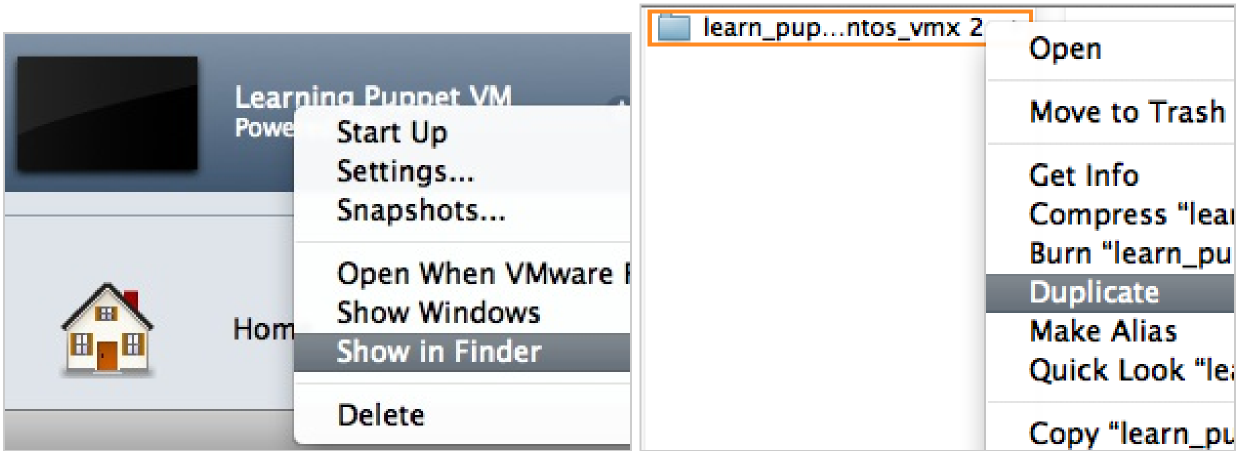
Below, we give instructions for copying the VM with VMware Fusion and with VirtualBox.

With VMware Fusion

(Note: although we don't provide a full walkthrough for VMware Workstation, the process should be similar.)

1. If you still have the zipped VM archive you originally downloaded, you can extract it again for a fresh copy.

   Otherwise, shut down the VM by running `shutdown -h now` while logged in as root. Once the system is stopped, locate the folder or bundle that contains the VMX file — you can right-click its entry in the Virtual Machine Library window and choose "Show in Finder" — and duplicate that entire directory.



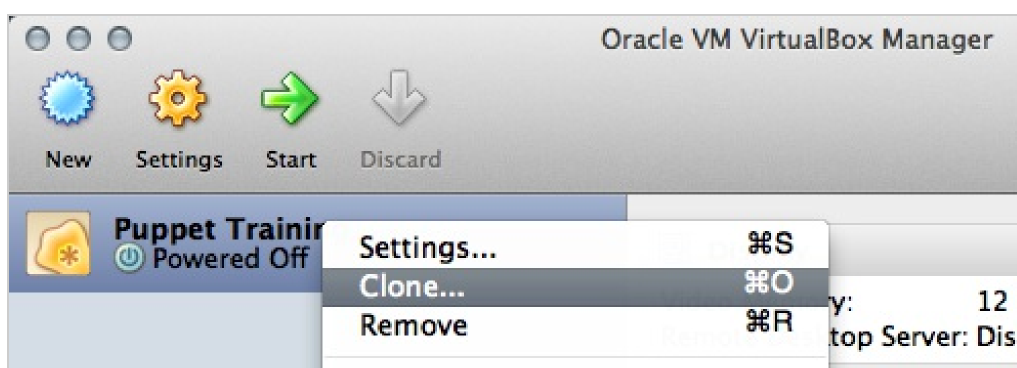
2. Your second copy of the files (whether from re-extracting or duplicating) contains a VMX file; drag it and drop it onto Fusion's Virtual Machine Library window. (If this window isn't displayed, the menu item to display it is in Fusion's "Window" menu.) This will import the virtual machine without automatically starting it up, which will give you a chance to change its RAM. (If you accidentally start it anyway, you can always change the RAM later or leave it as is.)
3. Once Fusion has the VM, you can right-click its entry in the Library window and choose "Settings" to change the amount of memory it will consume. (Use the "Processors & RAM" section of the settings window.) Although the original (puppet master) VM will need at least 512 MB of RAM, you can safely dial the agent VM down to 256 MB.

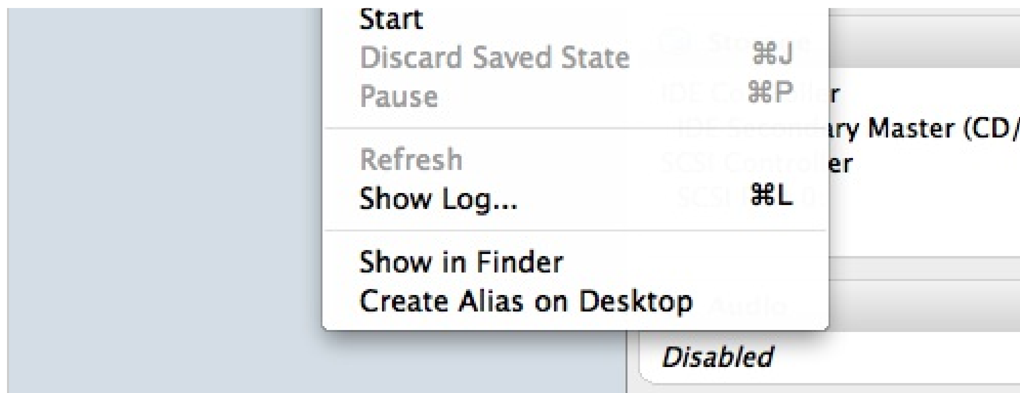
   You shouldn't need to change the networking settings from the default mode (NAT); with VMware, this will allow your VMs to access the internet, each other, and your host system. If you need other nodes on the network to be able to contact your VMs, you can change the networking mode to Bridged.
4. When you start the VM for the first time, Fusion will ask whether you moved it or copied it. You should answer that you copied it.

With VirtualBox

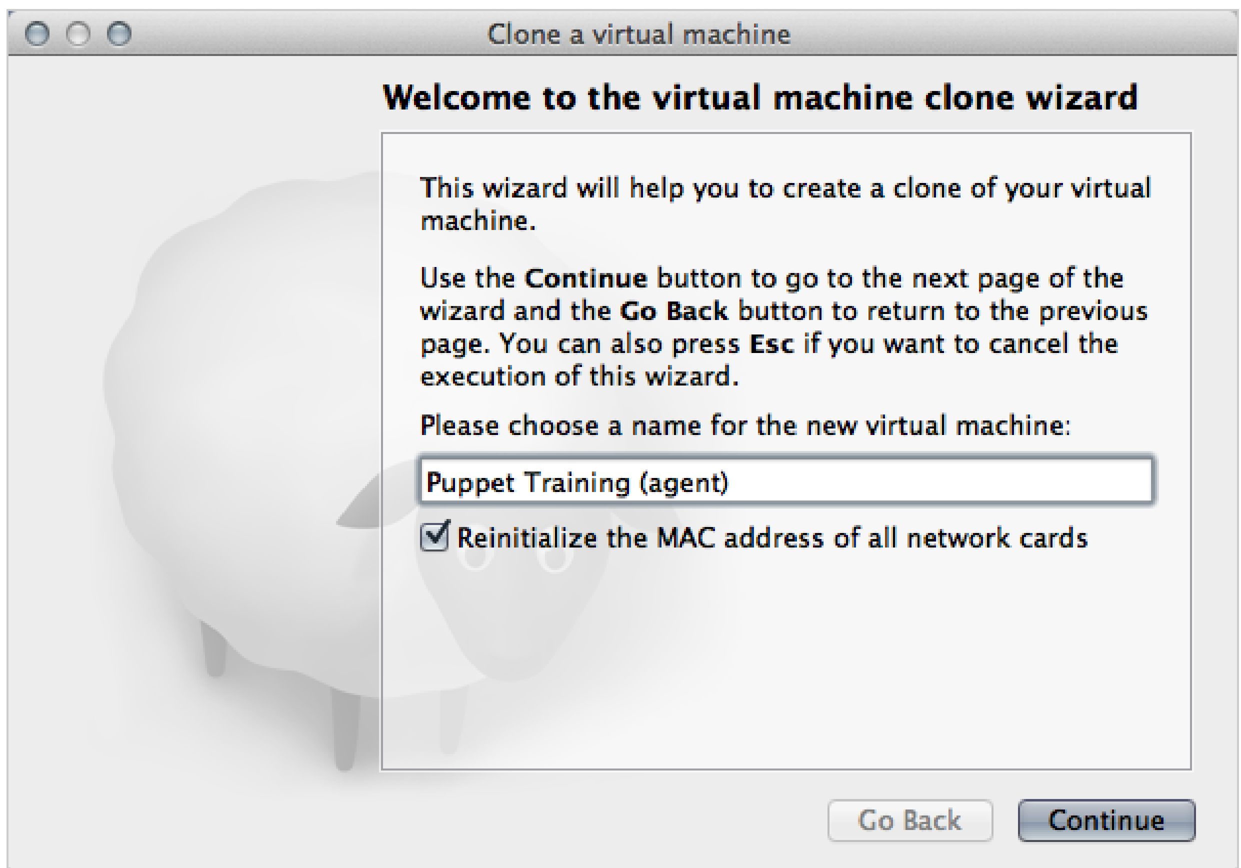
1. If you still have the folder with the original OVF file, you can re-import it into VirtualBox for a new VM.

   Otherwise, shut down the VM by running `shutdown -h now` while logged in as root. Once the system is stopped, right-click on the VM's entry in the VirtualBox Manager window, and select Clone. You will be presented with a series of dialog boxes.

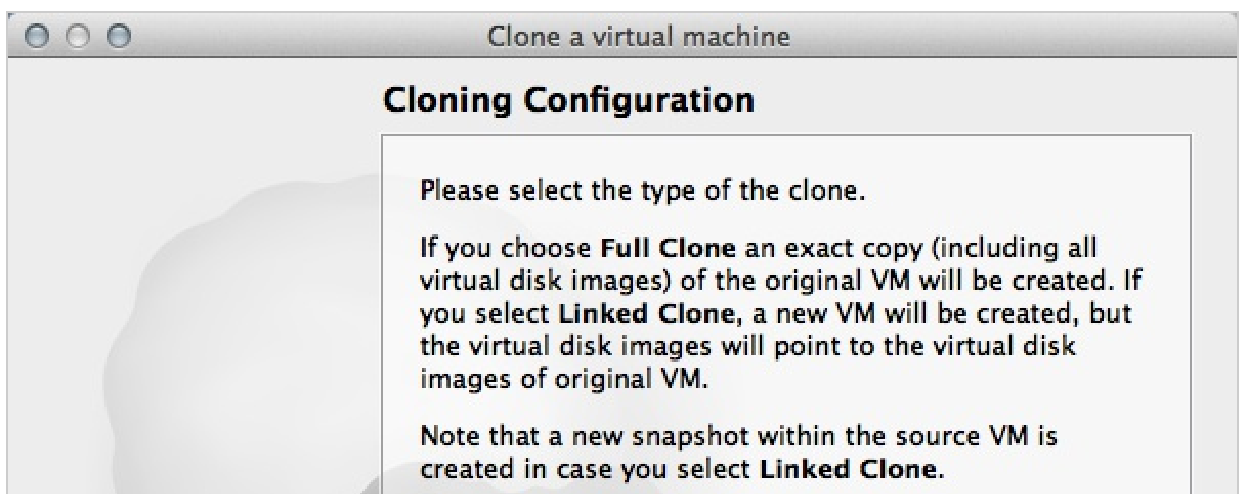


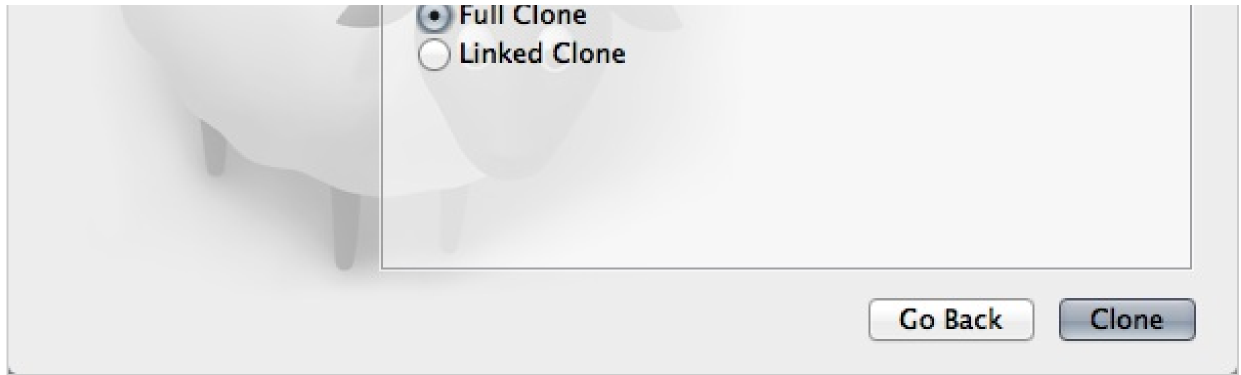


1. In the first one, choose a new name for the VM and make sure the “Reinitialize the MAC address of all network cards” box is checked.



2. In the second one, choose “Full Clone.”





2. Once VirtualBox has the new VM ready, change its network adapter to Bridged Adapter mode; otherwise, it will be unable to communicate with the puppet master. (You can also configure two network adapters instead of using bridged mode, but this is advanced and should not be done by most users.)
3. You can also click on the “System” settings to reduce the amount of memory the VM will consume. An agent node should only need 256 MB of RAM.

Step 2: Reconfigure the New Node With Puppet

Install this learningpuppet module on the agent VM, and apply the `learningpuppet::makeagent` class:

```
# wget http://docs.puppetlabs.com/learning/files/learningpuppet.tar.gz
# tar -xzf learningpuppet.tar.gz
# mv learningpuppet /etc/puppetlabs/puppet/modules/
# puppet apply -e "class {'learningpuppet::makeagent':}"
```

If you don’t give the class a `newname` attribute, it will default to `agent1`, which is probably what you want.

Step 3: Make Sure the VMs Can Communicate

For Puppet to work right, your two VMs must:

- Be able to reach each other by IP address
- Be able to reach the puppet master by name
- Have their firewalls turned off
- Have (reasonably) synchronized clocks

Ensure the VMs Can Reach Each Other by IP

WITH VMWARE FUSION

The VMs will be communicating via their `eth0` IP addresses. Find these addresses by running `facter ipaddress_eth0` on each system, then try to ping that IP from the other VM.

If the VMs can't communicate, examine each VM's settings and make sure:

- The networking mode is either NAT or Bridged.
- Both VMs have the same networking mode.

WITH VIRTUALBOX

If both VMs have a single network adapter in Bridged Adapter mode (recommended), they will be communicating via their `eth0` IP addresses. Find these addresses by running `facter ipaddress_eth0` on each system, then try to ping that IP from the other VM.

If you have configured the VMs to have two network adapters, examine their settings — the VMs will be communicating via whichever adapter is set to Host Only Adapter mode. Run `facter ipaddress_<ADAPTER>` to find these IP addresses.

Ensure the VMs Can Reach the Master by Name

Make sure both VMs' `/etc/hosts` files contain a line similar to the following:

```
172.16.158.151  learn.localdomain   learn puppet.localdomain puppet # This host
is required for Puppet's name resolution to work correctly.
```

The IP address should be the one you found for the puppet master in the previous step.

Once you've edited the files, test that both VMs can ping the master at both its full name and its aliases:

```
[root@agent1]# ping learn.localdomain
[root@agent1]# ping puppet
```

If this doesn't work, make sure that the `/etc/hosts` files don't have any conflicting lines — there should be only one line with those puppet master hostnames. If `/etc/hosts` looks good, you may also need to flush cached DNS information in each VM:

```
# nscd --invalidate=hosts
```

Ensure the Firewalls are Down

We shipped the VM with iptables turned off, but it's worth checking to make sure it's still down:

```
# service iptables status
Firewall is stopped.
```

(In a real environment, you'd add firewall rules for Puppet traffic instead of disabling the firewall.)

Ensure Both VMs Know the Time

Run `date -u` on both VMs, and compare the output. They should be within about a minute of each other.

Next

Your VMs are ready — now continue reading for a tour of the agent/master Puppet workflow.

Learning Puppet — Basic Agent/Master Puppet

This guide assumes that you've followed the previous walkthrough, and have a fresh agent VM that can reach your original master VM over the network. Both VMs should be running right now, and you'll need to be logged in to both of them as root.

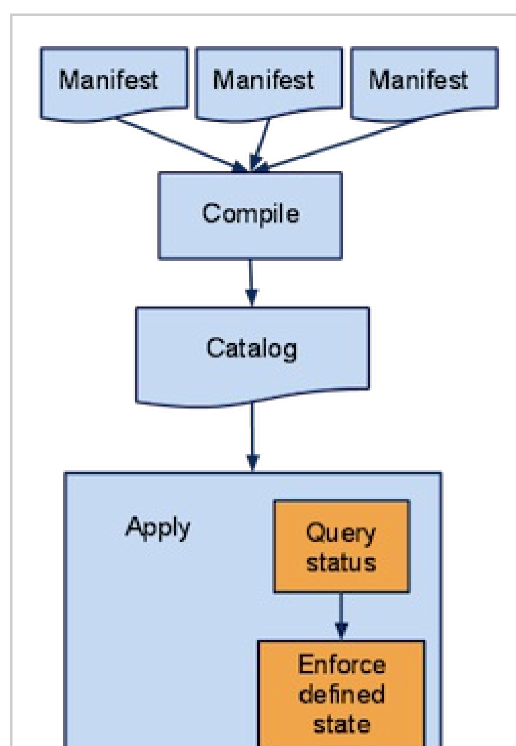
Introduction

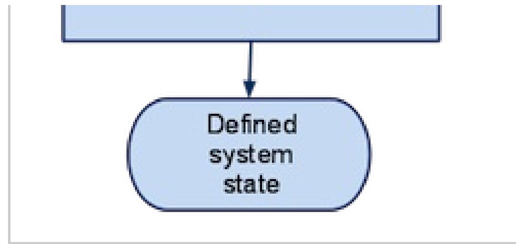
How Do Agents Get Configurations?

Puppet's agent/master mode is pull-based. Usually, agents are configured to periodically fetch a catalog and apply it, and the master controls what goes into that catalog. (For the next few exercises, though, you'll be triggering runs manually.)

What Do Agents Do, and What Do Masters Do?

Earlier, you saw this diagram of how Puppet compiles and applies a manifest:





Running Puppet in agent/master mode works much the same way — the main difference is that it moves the manifests and compilation to the puppet master server. Agents don't have to see any manifest files at all, and have no access to configuration information that isn't in their own catalog.

Agent
Request catalog (sends node name and facts)
Catalog
Apply
Query status
Enforce defined state
Defined system state
Master
Classify (Who is this, and what do they need?)
Class
Class
Class
Compile
Report

The Agent Subcommand

The puppet agent subcommand fetches configurations from a master server. It has two main modes:

1. Daemonize and fetch configurations every half-hour (default)
2. Run once and quit

We'll be using the second mode, since it gives a better view of what's going on. To keep the agent from daemonizing, you should use the `--test` option, which also prints detailed descriptions of what the agent is doing.

If you accidentally run the agent without `--test`, it will daemonize and run in the background. To check whether the agent is running in the background, run:

```
# /etc/init.d/pe-puppet status
```

To turn it off, run:

```
# /etc/init.d/pe-puppet stop
```

Saying Hi

Time to start! On your agent VM, start puppet agent for the first time:

```
[root@agent1 ~]# puppet agent --test
info: Creating a new SSL key for agent1.localdomain
warning: peer certificate won't be verified in this SSL session
info: Caching certificate for ca
warning: peer certificate won't be verified in this SSL session
warning: peer certificate won't be verified in this SSL session
info: Creating a new SSL certificate request for agent1.localdomain
info: Certificate Request fingerprint (md5):
FD:E7:41:C9:2C:B7:5C:27:11:0C:8F:9C:1D:F6:F9:46
warning: peer certificate won't be verified in this SSL session
warning: peer certificate won't be verified in this SSL session
warning: peer certificate won't be verified in this SSL session
Exiting; no certificate found and waitforcert is disabled
```

Hmm.

What Happened?

Puppet agent found the puppet master, but it got stopped at the certificate roadblock. It isn't authorized to fetch configurations, so the master is turning it away.

Troubleshooting

It's possible you didn't see the response printed above, and there are a number of possible culprits. Read back over the instructions for creating your agent VM and make sure you didn't miss anything; in particular, check that:

- The VMs can ping each other
- The agent can resolve the puppet master by host name
- The agent's `/etc/puppetlabs/puppet/puppet.conf` file has a `server` setting (in the `[agent]` block) of `puppet` or `learn.localdomain`
- The VMs' clocks are in sync

Signing the Certificate

So we'll authorize it! On your puppet master VM, check the list of outstanding certificate requests with `puppet cert list`. (More about this command later.)

```
[root@learn ~]# puppet cert list
  agent1.localdomain (FD:E7:41:C9:2C:B7:5C:27:11:0C:8F:9C:1D:F6:F9:46)
```

There's our agent node. And the request fingerprint matches, too. You know this node is okay, so go ahead and sign its certificate with `puppet cert sign`:

```
[root@learn ~]# puppet cert sign agent1.localdomain
notice: Signed certificate request for agent1.localdomain
notice: Removing file Puppet::SSL::CertificateRequest agent1.localdomain at
'/etc/puppetlabs/puppet/ssl/ca/requests/agent1.localdomain.pem'
```

Now that it's authorized, go back to the agent VM and run puppet agent again:

```
[root@agent1 ~]# puppet agent --test
warning: peer certificate won't be verified in this SSL session
info: Caching certificate for agent1.localdomain
info: Retrieving plugin
info: Caching certificate_revocation_list for ca
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Caching catalog for agent1.localdomain
info: Applying configuration version '1326210629'
notice: Finished catalog run in 0.11 seconds
```

It worked! That was a successful Puppet run, although it didn't do much yet.

What Happened?

Puppet uses SSL certificates to protect communications between agents and the master. Since agents can't do a full run without a certificate, our agent had to ask for one and then wait for the request to get approved.

We'll cover SSL in more detail later.

Serving a Real Configuration

So how can we make the agent do something interesting? Well, we already built some useful classes, and they're all available on the puppet master, so we'll use them. (If you haven't already copied the modules from your old VM into your puppet master's `/etc/puppetlabs/puppet/modules` directory, do so now.)

But how do we choose which classes go into an agent's catalog?

Site.pp

When we were using puppet apply, we would usually specify a manifest file, which declared all of the classes or resources we wanted to apply.

The puppet master works the same way, except that it always loads the same manifest file, which we usually refer to as site.pp. With Puppet Enterprise, it's located by default at `/etc/puppetlabs/puppet/manifests/site.pp`, but you can configure its location with the `manifest` setting.

You could declare classes and resources directly in site.pp, but that would make every node get the same resources in its catalog, which is of limited use. Instead, we'll hide the classes we want to declare in a node definition.

Node Definitions

Node definitions work almost exactly like class definitions:

```
# Append this at the bottom of /etc/puppetlabs/puppet/manifests/site.pp

node 'agent1.localdomain' {

  # Note the quotes around the name! Node names can have characters that
  # aren't legal for class names, so you can't always use bare, unquoted
  # strings like we do with classes.

  # Any resource or class declaration can go inside here. For now:

  include apache

  class {'ntp':
    enable => false,
    ensure => stopped,
  }
```

```
    }
```

But unlike classes, nodes are declared automatically, based on the name of the node whose catalog is being compiled. Only one node definition will get added to a given catalog, and any other node definitions are effectively hidden.

An agent node's name is almost always read from its `certname` setting, which is set at install time but can be changed later. The certname is usually (but not always) the node's fully qualified domain name.

More on node definitions later, as well as alternate ways to assign classes to a node.

Pulling the New Configuration

Now that you've saved site.pp with a node definition that matches the agent VM's name, go back to that VM and run puppet agent again:

```
[root@agent1 ~]# puppet agent --test
info: Retrieving plugin
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Loading facts in facter_dot_d
info: Caching catalog for agent1.localdomain
info: Applying configuration version '1326416535'
notice: /Stage[main]/Ntp/Package[ntp]/ensure: created
--- /etc/ntp.conf   2011-11-18 13:21:25.000000000 +0000
+++ /tmp/puppet-file20120113-5967-56l9xy-0  2012-01-13 01:02:23.000000000 +0000
@@ -14,6 +14,8 @@

 # Use public servers from the pool.ntp.org project.
 # Please consider joining the pool (http://www.pool.ntp.org/join.html).
+
+# Managed by puppet class { "ntp": servers => [ ... ] }
 server 0.centos.pool.ntp.org
 server 1.centos.pool.ntp.org
 server 2.centos.pool.ntp.org
info: /Stage[main]/Ntp/File[ntp.conf]: Filebucketed /etc/ntp.conf to main with
sum 5baec8bdbf90f877a05f88ba99e63685
notice: /Stage[main]/Ntp/File[ntp.conf]/content: content changed
'{md5}5baec8bdbf90f877a05f88ba99e63685' to
'{md5}35ea00fd40740faf3fd6d1708db6ad65'
notice: /Stage[main]/Apache/Package[apache]/ensure: created
notice: /Stage[main]/Apache/Service[apache]/ensure: ensure changed 'stopped' to
'running'
info: ntp.conf: Scheduling refresh of Service[ntp]
notice: /Stage[main]/Ntp/Service[ntp]: Triggered 'refresh' from 1 events
notice: Finished catalog run in 32.74 seconds
```

Success! We've pulled a configuration that actually does something.

If you change this node's definition in site.pp, it will fetch that new configuration on its next run

(which, in a normal environment, would happen less than 30 minutes after you make the change).

More Installments Coming Later

You now know how to:

- Run puppet agent interactively with `--test`
- Authorize a new agent node to pull configurations from the puppet master
- Use node definitions in site.pp to choose which classes go into a given node's catalog

But there are some important details we've glossed over. In a future installment, we'll talk more about certificates and node classification.

© 2010 Puppet Labs info@puppetlabs.com 411 NW Park Street / Portland, OR 97209 1-877-575-9775