CHAPTER 2

Building Hosts with Puppet

In Chapter 1 we installed and configured Puppet, created our first module, and applied that module and its configuration via the Puppet agent to a host. In this chapter, we're going to extend this process to build some more complete modules and hosts with Puppet for a hypothetical company, Example.com Pty Ltd. Each host's functionality we build will introduce new Puppet concepts and ideas.

Example.com Pty Ltd has four hosts we're going to manage with Puppet: a Web server, a database server, a mail server and our Puppet master server located in a flat network. You can see that network in Figure 2-1.

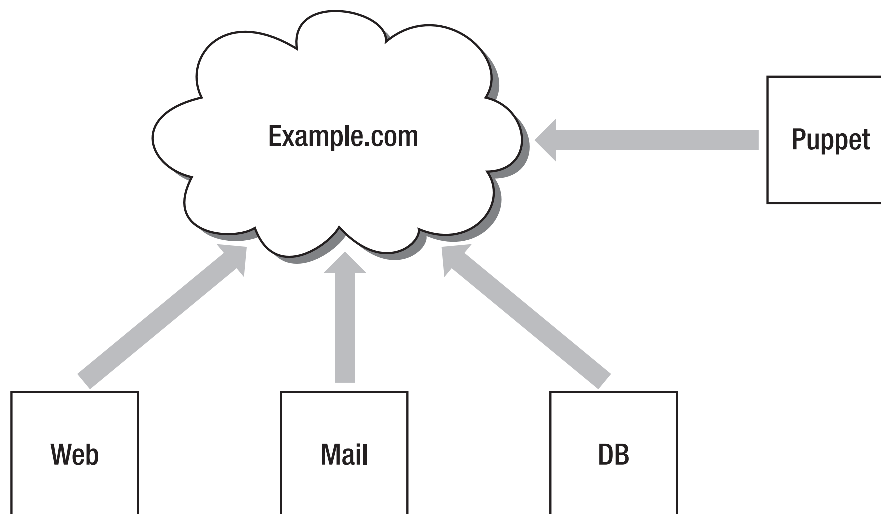

Figure 2-1. *The Example.com Pty Ltd Network*

Like many organizations, though, Example.com is not a very homogenous environment and each host uses a different operating system, as follows:

- `mail.example.com` – (Red Hat Enterprise Linux 5)
- `db.example.com` – (Solaris 10)
- `web.example.com` – (Ubuntu 10.04)
- `puppet.example.com` – (Ubuntu 10.04)

To solve this problem, we'll begin by working through how we use Puppet in a multiple operating system environment. Be sure you've installed the base operating system on these hosts as described in Chapter 1, because we'll perform some basic configuration on the hosts. We'll start with configuring `SSH` for each host, then we'll install and configure some role-specific applications for the hosts as follows:

- Postfix (`mail.example.com`)
- MySQL (`db.example.com`)
- Apache and a website (`web.example.com`)
- Manage the Puppet master with Puppet (`puppet.example.com`)

As we configure each host, we'll introduce some of the different features and functions available in Puppet. By the end of the chapter you'll have a firm grasp of the basics. In subsequent chapters, we'll build on this knowledge and introduce some of Puppet's more advanced features.

Getting Started

Before proceeding, we must have the proper setup, so we need to install the Puppet master and agent and then create node definitions for each of our hosts.

■ **Note** As we mentioned in Chapter 1, the Puppet software is called the "agent." Puppet calls the definition of the host itself a "node."

Installing Puppet

First, we need to install the Puppet master and agent. We're going to install the Puppet master on `puppet.example.com` and the Puppet agent on all our hosts, including `puppet.example.com`. We're installing the agent on the Puppet master because we're going to use Puppet to manage itself! We then need to connect, create and sign certificates for each host. To do this, you should follow the installation instructions for the relevant operating system from Chapter 1 on each of the four hosts. For example, for installation on the Red Hat Enterprise Linux host, use the instructions in the **Installing on Red Hat Enterprise Linux and Fedora** section. You can then move on to configuring the nodes (aka hosts).

■ **Tip** If you use a provisioning tool like Kickstart or Preseed, you can also include Puppet installation and signing as part of your build process. You can see an example of how to do that at `http://projects.puppetlabs.com/projects/1/wiki/Bootstrapping_With_Puppet`.

Configuring Nodes

After installing the Puppet master and associated agents, we need to create node definitions for each of our hosts in the node.pp file. We created this file in the /etc/puppet/manifests/ directory in Chapter 1. As you can see in Listing 2-1, we've created empty node definitions for each of the nodes in our network.

Listing 2-1. *Node defintions in nodes.pp*

```
node 'puppet.example.com' {
}

node 'web.example.com' {
}

node 'db.example.com' {
}

node 'mail.example.com' {
}
```

We haven't included any configuration on our node definitions – Puppet will just recognize the node as it connects and do nothing.

As you might imagine, if you've got a lot of nodes, the nodes.pp file could become quite large and complex. Puppet has some simple ways of dealing with this issue, described next.

Working With Similar Hosts

The first method works best for large number of similar hosts, such as Web servers, where the configuration of the host is largely identical. For example, if our environment had multiple hosts called web1, web2, web3, etc., we could specify:

```
node 'web1.example.com', 'web2.example.com', 'web3.example.com' { }
```

In version 0.25.0 and later, we can also specify these nodes in the form of a regular expression:

```
node /^web\d+\.example\.com$/ { }
```

This would match any host starting with web*x* where *x* is a digit or digits, such as web1 or web20.

Using External Sources

Puppet also has the ability to use external sources for your node data. These sources can include LDAP directories, databases or other external repositories. This allows you to leverage existing sources of information about your environment, such as asset management systems or identity stores. This functionality is called External Node Classification, or ENC, and we'll discuss it in more detail in Chapter 3.

Specifying a Default Node

You can also specify a special node called `default`. This is, as you'd imagine, a default node. If no other node definition exists, then the contents of this node are applied to the host.

```
node default {
    include defaultclass
}
```

Node Inheritance Structure

Lastly, Puppet has a simple node inheritance structure. You can use this to have one node inherit the contents of another node. Only one node can be inherited at a time. So, for example, we might want the node `web` host to inherit the contents of a node called `base`.

```
node base {
    include  sudo, mailx
}

node 'web.example.com' inherits base {
    …
}
```

Here we've defined the `base` node to include the modules `sudo` and `mailx` and then specified that the `web` node inherits the contents of this node. This means the `web` node would include `sudo` and `mailx` in addition to any classes included in its own node definition. Inheritance is cumulative and you can specify an inheritance structure like so:

```
node base {
    …
}

node webserver  inherits base {
    …
}

node 'web.example.com' inherits webserver {
    …
}
```

Here the `webserver` node inherits the contents of the `base` node, and then in turn the `web.example.com` node cumulatively inherits the contents of both nodes.

■ **Caution** When starting out with Puppet it is common to structure the assignment of classes to nodes using inheritance and a base node. This structure allows classes common to every node to be placed in the base node. This organization structure may pose a problem in the future as the number of nodes and the scale of puppet increases and base classes need to be added or removed from only a subset of all nodes. In order to avoid future refactoring, avoid using node inheritance in preference of a flat node classification tree. A good alternative to the base node and class inheritance is to employ conditional statements, which we'll introduce later in this chapter, to determine which classes a node should and should not receive instead of relying on node inheritance.

Variable Scoping

The concept of node inheritance is a good place to talk about an important and sometimes tricky concept in Puppet: variable scoping. Let's imagine we've decided to configure some variables in our nodes, for example:

```
node base {
  $location = "dc1"
  …
  $location = "dc2"
}
```

In most programming languages, the `$location` variable would start out with a value of `"dc1"` and then, when it was next assigned, it would change to a value of `"dc2"`. In Puppet, these same two statements cause an error:

```
err: Cannot reassign variable location at /etc/puppet/manifests/node.pp:4
```

Why is this? Puppet is declarative and hence dynamically scoped. Allowing variable reassignment would have to rely on order in the file to determine the value of the variable and order does not matter in a declarative language. The principal outcome of this is that you cannot redefine a variable inside the same scope it was defined in, like our node. Let's take another example, of a class this time instead of a node:

```
class ssh_sudo {
      $package = "openssh"
      package { $package: ensure => installed }

      $package = "sudo"
      package { $package: ensure => installed }
}
```

You can see that we've tried to define the `$package` variable twice. If we were to try to compile and apply this configuration, the Puppet agent would return the following error:

```
err: Cannot reassign variable package at /etc/puppet/modules/ssh/manifests/init.pp:5
```

■ **Note** The error helpfully also tells us the file, and line number in the file, where we've tried to redefine the variable.

So what's a scope? Each class, definition, or node introduces a new scope, and there is also a top scope for everything defined outside of those structures. Scopes are created hierarchically and the important thing you need to remember about scope hierarchy is that it is created when Puppet code is evaluated, rather than when it is defined, for example:

```
$package = "openssh"

class ssh {
  package { $package:
    ensure => installed,
  }
}

class ssh_server
  include ssh
  $package = "openssh-server"
}

include ssh_server
```

Here a top level scope, in which `$package` is defined, is present. Then there's a scope for the `ssh_server` class and a scope below that for the `ssh` class. When Puppet runs the `$package` variable will have a value of `"openssh-server"` because this is what the variable was when evaluation occurred.

Naturally, in these different scopes, you can reassign the value of a variable:

```
class apache {
      $apache = 1
}

class passenger {
      $apache = 2
}
```

The same variable can be used and defined in both the `apache` and `passenger` classes without generating an error because they represent different scopes.

Going back to node inheritance, you can probably see how this dynamic scoping is going to be potentially confusing, for example:

```
class apache {
  $apacheversion = "2.0.33"
  package { "apache2":
    ensure => $apacheversion,
  }
}
```

```
node 'web.example.com' {
  include  apache
}

node 'web2.example.com' inherits 'web.example.com' {
  $apacheversion = "2.0.42"
}
```

Here we've created a class called apache and a package resource for the apache2 package. We've also created a variable called $apacheversion and used that as the value of the ensure attribute of the package resource. This tells Puppet that we want to install version 2.0.33 of Apache. We've then included our apache class in a node, web.example.com.

But we've also decided to create another node, web2.example.com, which inherits the contents of the web.example.com node. In this case, however, we've decided to install a different Apache version and therefore we specified a new value for the $apacheversion variable. But instead of using this new value, Puppet will continue to install the 2.0.33 version of Apache because the $apacheversion variable is maintained in its original scope of the web.example.com node and the new variable value is ignored.

There is a work-around for this issue that you can see here:

```
class apache {
  $apacheversion = "2.0.33"
  package { "apache2":
    ensure => $apacheversion,
  }
}

class base {
  include  apache
}

node 'web.example.com' {
  $apacheversion = "2.0.42"
  include base
}
```

Instead of defining a base node we've defined a base class that includes the apache class. When we created our node, we specified the $apacheversion we want and then included the base class, ensuring we're in the right scope. We could put other like items in our base class and specify any required variables.

■ **Note** You can learn more about variable scoping, workarounds and related issues at http://projects.puppetlabs.com/projects/puppet/wiki/Frequently_Asked_Questions#Common+Misconceptions.

With Puppet installed and node definitions in place, we can now move on to creating our modules for the various hosts. But first, let's do a quick refresher on modules in general.

Making (More) Magic With Modules

In Chapter 1, we learned about modules: self-contained collections of resources, classes, files that can be served, and templates for configuration files. We're going to use several modules to define the various facets of each host's configuration. For example, we will have a module for managing Apache on our Web server and another for managing Postfix on our mail server.

Recall that modules are structured collections of Puppet manifests. By default Puppet will search the module path, which is by default `/etc/puppet/modules/` and `/var/lib/puppet/modules`, for modules and load them. These paths are controlled by the `modulepath` configuration option. This means we don't need to import any of these files into Puppet – it all happens automatically.

It's very important that modules are structured properly. For example, our `sudo` module contains the following:

```
sudo/
sudo/manifests
sudo/manifests/init.pp
sudo/files
sudo/templates
```

Inside our `init.pp` we create a class with the name of our module:

```
class sudo {
configuration…
}
```

Lastly, we also discovered we can apply a module, like the `sudo` module we created in Chapter 1, to a node by using the `include` function like so:

```
node 'puppet.example.com' {
  include sudo
}
```

The included function adds the resources contained in a class or module, for example adding all the resources contained in the `sudo` module here to the node `puppet.example.com`.

Let's now see how to manage the contents of our modules using version control tools as we recommended in Chapter 1.

■ **Note** You don't have to always create your own modules. The Puppet Forge at `http://forge.puppetlabs.com` contains a large collection of pre-existing modules that you can either use immediately or modify to suit your environment. This can make getting started with Puppet extremely simple and fast.

Version Controlling Your Modules

Because modules present self-contained collections of configuration, we also want to appropriately manage the contents of these modules, allowing us to perform change control. To manage your content, we recommend that you use a Version Control System or VCS.

Version control is the method most developers use to track changes in their application source code. Version control records the state of a series of files or objects and allows you to periodically capture that state in the form of a revision. This allows you to track the history of changes in files and objects and potentially revert to an earlier revision should you make a mistake. This makes management of our configuration much easier and saves us from issues like undoing inappropriate changes or accidently deleting configuration data.

In this case, we're going to show you an example of managing your Puppet manifests with a tool called Git, which is a distributed version control system (DVCS). Distributed version control allows the tracking of changes across multiple hosts, making it easier to allow multiple people to work on our modules. Git is used by a lot of large development projects, such as the Linux kernel, and was originally developed by Linux Torvalds for that purpose. It's a powerful tool but it'sd easy to learn the basic steps. You can obviously easily use whatever version control system suits your environment, for example many people use Subversion or CVS for the same purpose.

First, we need to install Git. On most platforms we install the `git` package. For example, on Red Hat and Ubuntu:

```
$ sudo yum install git
```

or,

```
$ sudo apt-get install git
```

Once Git is installed, let's identify ourselves to Git so it can track who we are and associate some details with actions we take.

```
$ git config --global user.name "Your Name"
$ git config --global user.email your@email.address.com
```

Now let's version control the path containing our modules, in our case `/etc/puppet/modules`. We change to that directory and then execute the `git` binary to initialize our new Git repository.

```
$ cd /etc/puppet/modules
$ git init
```

This creates a directory called `.git` in the `/etc/puppet/modules` directory that will hold all the details and tracking data for our Git repository.

We can now add files to this repository using the `git` binary with the `add` option.

```
$ git add *
```

This adds everything currently in our path to Git. You can also use `git` and the `rm` option to remove items you don't want to be in the repository.

```
$ git rm filename
```

This doesn't mean, however, that our modules are already fully tracked by our Git repository. Like Subversion and other version control systems, we need to "commit" the objects we'd like to track. The commit process captures the state of the objects we'd like to track and manage, and it creates a revision to mark that state. You can also create a file called `.gitignore` in the directory. Every file or directory specified in this file will be ignored by Git and never added.

Before we commit though, we can see what Git is about by using the `git status` command:

```
$ git status
```

This tells us that when we commit that Git will add the contents to the repository and create a revision based on that state.

Now let's commit our revision to the repository.

```
$ git commit -a -m "This is our initial commit"
```

The `-m` option specifies a commit message that allows us to document the revision we're about to commit. It's useful to be verbose here and explain what you have changed and why, so it's easier to find out what's in each revision and make it easier to find an appropriate point to return to if required. If you need more space for your commit message you can omit the `-m` option and Git will open your default editor and allow you to type a more comprehensive message.

The changes are now committed to the repository and we can use the `git log` command to see our recent commit.

```
$ git log
```

We can see some information here about our commit. First, Git uses SHA1 hashes to track revisions; Subversion, for example, uses numeric numbers – 1, 2, 3, etc. Each commit has a unique hash assigned to it. We will also see some details about who created the commit and our commit message telling us what the commit is all about.

Every time you add a new module or file you will need to add it to Git using the `git add` command and then commit it to store it in the repository. I recommend you add and commit changes regularly to ensure you have sufficiently granular revisions to allow you to easily roll back to an earlier state.

■ **Tip** If you're interested in Git, we strongly recommend Scott Chacon's excellent book *Pro Git* – also published by Apress. The book is available in both dead tree form and online at `http://progit.org/book/`. Scott is also one of the lead developers of the Git hosting site, GitHub – `http://www.github.com`, where you can find a number of Puppet related modules.

Our simple `sudo` module is a good introduction to Puppet, but it only showcased a small number of Puppet's capabilities. It's now time to expand our Puppet knowledge and develop some new more advanced modules, starting with one to manage SSH on our hosts. We'll then create a module to manage Postfix on `mail.example.com`, one to manage MySQL on our Solaris host, `db.example.com`, another to manage Apache and web sites, and finally one to manage Puppet with Puppet itself.

We'll also introduce you to some best practices for structuring, writing and managing modules and configuration.

Creating a module to Manage SSH

We know that we first need to create an appropriate module structure. We're going to do this under the `/etc/puppet/modules` directory on our Puppet master.

```
$ cd /etc/puppet/modules
$ mkdir -p ssh/{manifests,templates,files}
$ touch ssh/manifests/init.pp
```

Next, we create some classes inside the `init.pp` file and some initial resources, as shown in Listing 2-2.

Listing 2-2. *The ssh module*

```
class ssh::install {
  package { "openssh":
    ensure => present,
  }
}

class ssh::config {
  file { "/etc/ssh/sshd_config":
    ensure = > present,
    owner => 'root',
    group => 'root',
    mode => 0600,
    source => "puppet:///modules/ssh/sshd_config",
    require => Class["ssh::install"],
    notify => Class["ssh::service"],
  }
}
class ssh::service {
  service { "sshd":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enable => true,
    require => Class["ssh::config"],
  }
}

class ssh {
  include ssh::install, ssh::config, ssh::service
}
```

We've created three classes: `ssh`, `ssh::install`, `ssh::config`, and `ssh::service`. As we mentioned earlier, modules can be made up multiple classes. We use the `::` namespace syntax as a way to create structure and organization in our modules. The `ssh` prefix tells Puppet that each class belongs in the `ssh` module, and the class name is suffixed.

■ **Note** We'd also want to create a `sshd_config` file in the `ssh/files/` directory so that our `File["/etc/ssh/sshd_config"]` resource can serve out that file. The easiest way to do this is to copy an existing functional `sshd_config` file and use that. Later we'll show you how to create template files that allow you to configure per-host configuration in your files. Without this file Puppet will report an error for this resource.

In Listing 2-2, we created a functional structure by dividing the components of the service we're managing into functional domains: things to be installed, things to be configured and things to be executed or run.

Lastly, we created a class called `ssh` (which we need to ensure the module is valid) and used the `include` function to add all the classes to the module.

Managing Our Classes

Lots of classes with lots of resources in our `init.pp` file means that the file is going to quickly get cluttered and hard to manage. Thankfully, Puppet has an elegant way to manage these classes rather than clutter the `init.pp` file. Each class, rather than being specified in the `init.pp` file, can be specified in an individual file in the manifests directory, for example in a `ssh/manifests/install.pp` file that would contain the `ssh::install` class:

```
class ssh::install {
  package { "openssh":
    ensure => present,
  }
}
```

When Puppet loads the `ssh` module, it will search the path for files suffixed with `.pp`, look inside them for namespaced classes and automatically import them. Let's quickly put our `ssh::config` and `ssh::service` classes into separate files:

```
$ touch ssh/manifests/{config.pp,service.pp}
```

This leaves our `init.pp` file containing just the `ssh` class:

```
class ssh
  include ssh::install, ssh::config, ssh::service
}
```

Our `ssh` module directory structure will now look like:

```
ssh
ssh/files/sshd_config
ssh/manifests/init.pp
ssh/manifests/install.pp
ssh/manifests/config.pp
ssh/manifests/service.pp
ssh/templates
```

Neat and simple.

■ **Tip** You can nest classes another layer, like `ssh::config::client`, and our auto-importing magic will still work by placing this class in the `ssh/manifests/config/client.pp` file.

The ssh::install Class

Now that we've created our structure, let's look at the classes and resources we've created. Let's start with the `ssh::install` class containing the `Package["openssh"]` resource, which installs the OpenSSH package.

It looks simple enough, but we've already hit a stumbling block – we want to manage SSH on all of Example.com's hosts, and across these platforms the OpenSSH package has different names:

- Red Hat: openssh-server
- Ubuntu: openssh-server
- Solaris: openssh

How are we going to ensure Puppet installs the correctly-named package for each platform? The answer lies with Facter, Puppet's system inventory tool. During each Puppet run, Facter queries data about the host and sends it to the Puppet master. This data includes the operating system of the host, which is made available in our Puppet manifests as a variable called `$operatingsystem`. We can now use this variable to select the appropriate package name for each platform. Let's rewrite our `Package["openssh"]` resource:

```
package { "ssh":
  name => $operatingsystem ?
    /(Red Hat|CentOS|Fedora|Ubuntu|Debian)/ => "openssh-server",
    Solaris => "openssh",
    },
  ensure => installed,
}
```

You can see we've changed the title of our resource to `ssh` and specified a new attribute called `name`. As we explained in Chapter 1, each resource is made up of a type, title and a series of attributes. Each resource's attributes includes its "name variable," or "namevar," and the value of this attribute is used to determine the name of the resource. For example, the Package and Service resources use the `name` attribute as their namevar while the File type uses the `path` attribute as its namevar. Most of the time we wouldn't specify the namevar, as it is synonymous with the title, for example in this resource:

```
file { "/etc/passwd":
  …
}
```

We don't need to specify the namevar because the value will be taken from the title, `"/etc/passwd"`. But often we're referring to resources in many places and we might want a simple alias, so we can give the resource a title and specify its namevar this way:

```
file { "passwd":
  path => "/etc/passwd",
  …
}
```

We can now refer to this resource as `File["passwd"]` as an aliased short-hand.

■ **Note** You should also read about the `alias` metaparameter, which provides a similar capability, at `http://docs.puppetlabs.com/references/latest/metaparameter.html#alias`.

In our current example, the name of the package we're managing varies on different hosts. Therefore, we want to specify a generic name for the resource and a platform-selected value for the actual package to be installed.

You can see that inside this new `name` attribute we've specified the value of the attribute as `$operatingsystem` followed by a conditional syntax that Puppet calls a "selector." To construct a selector, we specify the a variable containing the value we want to select on as the value of our attribute, here `$operatingsystem`, and follow this with a question mark (`?`). We then list on new lines a series of selections, for example if the value of `$operatingsystem` is Solaris, then the value of the `name` attribute will be set to `openssh,` and so on. Notice that we can specify multiple values in the form of simple regular expressions, like `/(Solaris|Ubuntu|Debian)/`.

■ **Note** Selector matching is case-insensitive. You can also see some other examples of regular expressions in selectors at `http://docs.puppetlabs.com/guides/language_tutorial.html#selectors`.

We can also specify a value called `default`.

```
default => "ssh",
```

This value is used if no other listed selection matches. If we don't specify a `default` value and no selection matches then the `name` attribute would be set to a nil value.

As can you imagine, this requirement to select the appropriate value for a particular platform happens a lot. This means we could end up scattering a lot of very similar conditional statements across our Puppet code. That's pretty messy; a best practice we recommend is to make this look a lot neater and more elegant by moving all your conditional checks to a separate class.

We usually call that class `module::params`, so in our current case it would be named `ssh::params`. Like before, we're going to store that class in a separate file. Let's create that file:

```
$ touch ssh/manifests/params.pp
```

We can see that class in Listing 2-3.

Listing 2-3. *The ssh::params class*

```
class ssh::params {
  case $operatingsystem {
    Solaris: {
      $ssh_package_name = 'openssh'
    }
    /(Ubuntu|Debian)/: {
      $ssh_package_name = 'openssh-server'
```

```
    }
    /(RedHat|CentOS|Fedora)/: {
      $ssh_package_name = 'openssh-server'
    }
  }
}
```

You can see that inside our `ssh::params` class we've created another type of conditional, the case statement. Much like a selector, the case statement iterates over the value of a variable, here `$operatingsystem`. Unlike a selector, case statements allow us to specify a block of things to do if the value of the variable matches one of the cases. In our case we're setting the value of a new variable we've created, called `$ssh_package_name`. You could do other things here, such as include a class or a resource, or perform some other function.

■ **Note** You can read more about case statements at `http://docs.puppetlabs.com/guides/language_tutorial.html#case_statement`. Also available is an if/else syntax that you can read about at `http://docs.puppetlabs.com/guides/language_tutorial.html#ifelse_statement`.

And finally, we need to include our new class in the `ssh` class:\

```
class ssh {
  include ssh::params, ssh::install, ssh::config, ssh::service
}
```

These includes tell Puppet that when you include the `ssh` module, you're getting all of these classes.

FUNCTIONS

The `include` directive we use to include our classes and modules is called a function. Functions are commands that run on the Puppet master to perform actions. Puppet has a number of other functions, including the `generate` function that calls external commands and returns the result, and the `notice` function that logs messages on the master and is useful for testing a configuration. For example:

```
notice("This is a notice message including the value of the $ssh_package variable")
```

Functions only run on the Puppet master and cannot be run on the client, and thus can only work with the resources available on the master.

You can see a full list of functions at `http://docs.puppetlabs.com/references/stable/function.html` and we'll introduce you to a variety of other functions in subsequent chapters. You can also find some documentation on how to write your own functions at `http://projects.puppetlabs.com/projects/puppet/wiki/Writing_Your_Own_Functions`, and we'll talk about developing functions in Chapter 10.

We're going to come back to the `ssh::params` class and add more variables as we discover other elements of our OpenSSH configuration that are unique to particular platforms, but for the moment how does including this new class change our `Package["ssh"]` resource?

```
package { $ssh::params::ssh_package_name:
  ensure => installed,
}
```

You can see our namespacing is useful for other things, here using variables from other classes. We can refer to a variable in another class by prefixing the variable name with the class it's contained in, here `ssh::params`. In this case, rather than our messy conditional, the package name to be installed will use the value of the `$ssh::params::ssh_package_name` parameter. Our resource is now much neater, simpler and easier to read.

■ **Tip** So how do we refer to namespaced resources? Just like other resources, `Package[$ssh::params::ssh_package_name]`.

The ssh::config Class

Now let's move onto our next class, `ssh::config`, which we can see in Listing 2-4.

Listing 2-4. *The ssh::config class*

```
class ssh::config {
  file { "/etc/ssh/sshd_config":
    ensure = > present,
    owner => 'root',
    group => 'root',
    mode => 0440,
    source => "puppet:///modules/ssh/sshd_config",
    require => Class["ssh::install"],
    notify => Class["ssh::service"],
  }
}
```

We know that the location of the sshd_config files will vary across different operating systems. Therefore, we're going to have to add another conditional for the name and location of that file. Let's go back to our `ssh::params` class from Example 2-3 and add a new variable:

```
class ssh::params {
  case $operatingsystem {
    Solaris {
      $ssh_package_name = 'openssh'
      $ssh_service_config = '/etc/ssh/sshd_config'
  }
…
}
```

We add the `$ssh_service_config` variable to each of the cases in our conditional and then update our file resource in the `ssh::config` class:

```
file { $ssh::params::ssh_service_config:
  ensure = > present,
  …
}
```

Again, we have no need for a messy conditional in the resource, we can simply reference the `$ssh::params::ssh_service_config` variable.

We can also see that the file resource contains two metaparameters, `require` and `notify`. These metaparameters both specify relationships between resources and classes. You'll notice here that both metaparameters reference classes rather than individual resources. They tell Puppet that it should create a relationship between this file resource and every resource in the referenced classes.

■ **Tip** It is a best practice to establish relationships with an entire class, rather than with a resource contained within another class, because this allows the internal structure of the class to change without refactoring the resource declarations related to the class.

For example, the `require` metaparameter tells Puppet that all the resources in the specified class must be processed prior to the current resource. In our example, the OpenSSH package must be installed before Puppet tries to manage the service's configuration file.

The `notify` metaparameter creates a notification relationship. If the current resource (the service's configuration file) is changed, then Puppet should notify all the resources contained in the `ssh::service` class. In our current case, a "notification" will cause the service resources in the `ssh::service` class restart, ensuring that if we change a configuration file that the service will be restarted and running with the correct, updated configuration.

■ **Tip** In Puppet 2.6.0, a shorthand method called "chaining" was introduced for specifying metaparameter relationships, such as `require` and `notify`. You can read about chaining at `http://docs.puppetlabs.com/guides/language_tutorial.html#chaining_resources`.

So why specify the whole `ssh::service` class rather than just the `Service["sshd"]` resource? This is another piece of simple best practice that allows us to simplify maintaining our classes and the relationships between them. Imagine that, instead of a single package, we had twenty packages. If we didn't require the class then we'd need to specify each individual package in our require statement, like this:

```
require => [ Package["package1"], Package["package2"], Package["package3"] ],
```

■ **Note** Adding [ ]s around a list creates a Puppet array. You can specify arrays as the values of variables and many attributes; for example, you can specify many items in a single resource: `package { [ "package1", "package2", "package3" ]: ensure => installed }`. In addition to arrays, Puppet also supports a hash syntax, which you can see at `http://docs.puppetlabs.com/guides/language_tutorial.html#hashes`.

We'd need to do that for every resource that required our packages, making our `require` statements cumbersome, potentially error prone, and most importantly requiring that every resource that requires packages be updated with any new package requirements.

By requiring the whole class, it doesn't matter how many packages we add to the `ssh::install` class – Puppet knows to install packages before managing configuration files, and we don't have to update a lot of resources every time we make a change.

■ **Tip** In our current example we could make use of arrays to extend the variables in the `ssh::params` class. For example, by changing `$ssh_package_name` to an array, we could specify multiple packages to be installed without needing to create another Package resource in the `ssh::install` class. Puppet is smart enough to know that if you specify a variable with a value of an array then it should expand the array, so changing the value of the `$ssh_package_name` variable to `[ openssh, package2, package3 ]` would result in the `ssh::install` class installing all three packages. This greatly simplifies the maintenance of our `ssh` module, as we only need to change values in one place to manage multiple configuration items.

The ssh::service Class

Let's look at our last class, `ssh::service`, and update it to reflect our new practice:

```
class ssh::service {
  service { $ssh::params::ssh_service_name:
    ensure => running,
    hasstatus => true,
    hasresstart => true,
    enable => true,
    require => Class["ssh::config"],
  }
}
```

We've added our new variable, `$ssh_service_name`, to the `ssh:params` class too:

```
class ssh::params {
  case $operatingsystem {
    Solaris {
      $ssh_package_name = 'openssh'
```

```
      $ssh_service_config = '/etc/ssh/sshd_config'
      $ssh_service_name = 'sshd'
  }
…
}
```

Let's also look at our `Service[$ssh::params::ssh_service_name]` resource (at the start of this section), as this is the first service we've seen managed. You'll notice two important attributes, `ensure` and `enable`, which specify the state and status of the resource respectively. The state of the resource specifies whether the service is running or stopped. The status of the resource specifies whether it is to be started at boot, for example as controlled by the `chkconfig` or `enable-rc.d` commands.

Puppet understands how to manage a variety of service frameworks, like SMF and init scripts, and can start, stop and restart services. It does this by attempting to identify the service framework your platform uses and executing the appropriate commands. For example, on Red Hat it might execute:

```
$ service sshd restart
```

If Puppet can't recognize your service framework, it will revert to simple parsing of the process table for processes with the same name as the service it's trying to manage. This obviously isn't ideal, so it helps to tell Puppet a bit more about your services to ensure it manages them appropriately. The `hasstatus` and `hasrestart` attributes we specified in the `ssh::service` class is one of the ways we tell Puppet useful things about our services. If we specify `hasstatus` as true, then Puppet knows that our service framework supports status commands of some kind. For example, on Red Hat it knows it can execute the following:

```
$ service sshd status
```

This enables it to determine accurately whether the service is started or stopped. The same principle applies to the `hasrestart` attribute, which specifies that the service has a restart command.

Now we can see Puppet managing a full service, if we include our new `ssh` module in our Puppet nodes, as shown in Listing 2-5.

Listing 2-5. *Adding the ssh Module*

```
class base {
  include sudo, ssh
}

node 'puppet.example.com' {
  include base
}

node 'web.example.com' {
  include base
}

node 'db.example.com' {
  include base
}
```

```
node 'mail.example.com' {
  include base
}
```

Here we've created a class called `base`, in which we're going to place the modules that will be base or generic to all our nodes. Thus far, these are our `sudo` and `ssh` modules. We then `include` this class in each node statement.

■ **Note** We talked earlier about node inheritance and some of its scoping issues. As we explained there, using a class instead of node inheritance helps avoids these issues. You can read about it at `http://projects.puppetlabs.com/projects/puppet/wiki/Frequently_Asked_Questions#Common+Misconceptions`.

With a basic SSH module in place, and we can now manage the SSH daemon and its configuration.

Creating a Module to Manage Postfix

Let's now create a module to manage Postfix on `mail.example.com`. We start with a similar structure to our SSH module. In this case, we know which platform we're going to install our mail server on so we don't need to include any conditional logic. However, if we had multiple mail servers on different platforms, it would be easy to adjust our module using the example we've just shown to cater for disparate operations systems.

```
postfix
postfix/files/master.cf
postifx/manifests/init.pp
postfix/manifests/install.pp
postfix/manifests/config.pp
postfix/manifests/service.pp
postfix/templates/main.cf.erb
```

The postfix::install class

We also have some similar resources present in our Postfix module that we saw in our SSH module, for example in the `postfix::install` class we install two packages, `postfix` and `mailx`:

```
class postfix::install {
  package { [ "postfix", "mailx" ]:
    ensure => present,
  }
}
```

Note that we've used an array to specify both packages in a single resource statement this is a useful shortcut that allows you specify multiple items in a single resource.

The postfix::config class

Next, we have the `postfix::config` class, which we will use to configure our Postfix server.

```
class postfix::config {
  File {
    owner => "postfix",
    group => "postfix",
    mode => 0644,
  }

  file { "/etc/postfix/master.cf":
    ensure = > present,
    source => "puppet:///modules/postfix/master.cf",
    require => Class["postfix::install"],
    notify => Class["postfix::service"],
  }

  file { "/etc/postfix/main.cf":
    ensure = > present,
    content => template("postfix/main.cf.erb"),
    require => Class["postfix::install"],
    notify => Class["postfix::service"],
  }
}
```

You may have noticed some new syntax: We specified the File resource type capitalized and without a title. This syntax is called a resource default, and it allows us to specify defaults for a particular resource type. In this case, all File resources within the `postfix::config` class will be owned by the user `postfix`, the group `postfix` and with a mode of `0644`. Resource defaults only apply to the current scope, but you can apply global defaults by specifying them in your `site.pp` file.

A common use for global defaults is to define a global "filebucket" for backing up the files Puppet changes. You can see the filebucket type and an example of how to use it globally at `http://docs.puppetlabs.com/references/stable/type.html#filebucket.`

■ **Tip** A common use for global defaults is to define a global "filebucket" for backing up the files Puppet changes. You can see the filebucket type and an example of how to use it globally at `http://docs.puppetlabs.com/references/stable/type.html#filebucket`.

METAPARAMETER DEFAULTS

Like resource defaults, you can also set defaults for metaparameters, such as `require`, using Puppet variable syntax. For example:

```
class postfix::config {
  $require = Class["postfix::install"]
  …
}
```

This would set a default for the `require` metaparameter in the `postfix::config` class and means we could remove all the `require => Class["postfix::install"]` statements from our resources in that class.

We've also introduced a new attribute in our `File["/etc/postfix/main.cf"]` resource – `content`. We've already seen the `source` attribute, which allows Puppet to serve out files, and we've used it in one of our File resources, `File["/etc/postfix/master.cf"]`. The `content` attribute allows us to specify the content of the file resources as a string. But it also allows us to specify a template for our file. The template is specified using a function called `template`.

As previously mentioned, functions are commands that run on the Puppet master and return values or results. In this case, the `template` function allows us to specify a Ruby ERB template (`http://ruby-doc.org/stdlib/libdoc/erb/rdoc/`), from which we can create the templated content for our configuration file. We specify the template like this:

```
content => template("postfix/main.cf.erb"),
```

We've specified the name of the function, "template," and inside brackets the name of the module that contains the template and the name of the template file. Puppet knows when we specify the name of the module to look inside the `postfix/templates` directory for the requisite file – here, `main.cf.erb`.

THE REQUIRE FUNCTION

In addition to the `include` function, Puppet also has a function called `require`. The `require` function works just like the `include` function except that it introduces some order to the inclusion of resources. With the `include` function, resources are not included in any sequence. The only exception is individual resources, which have relationships (using metaparameters, for example) that mandate some ordering. The `require` function tells Puppet that all resources being required must be processed first. For example, if we specified:

```
class ssh {
require ssh::params
include ssh::install, ssh::config, ssh::service
}
```

then the contents of `ssh::params` would be processed before any other includes or resources in the `ssh` class. This is useful as a simple way to specify some less granular ordering to your manifests than metaparameter relationships, but it's not recommended as a regular approach. The reason it is not

recommended is that Puppet does this by creating relationships between all the resources in the required class and the current class. This can lead to cyclical dependencies between resources. It's cleaner, more elegant and simpler to debug if you use metaparameters to specify the relationships between resources that need order.

In Listing 2-6 we can see what our template looks like.

Listing 2-6. *The Postfix main.cf template*

```
soft_bounce = no
command_directory = /usr/sbin
daemon_directory = /usr/libexec/postfix
mail_owner = postfix
myhostname = <%= hostname %>
mydomain = <%= domain %>
myorigin = $mydomain
mydestination = $myhostname, localhost.$mydomain, localhost, $mydomain
unknown_local_recipient_reject_code = 550
relay_domains = $mydestination
smtpd_reject_unlisted_recipient = yes
unverified_recipient_reject_code = 550
smtpd_banner = $myhostname ESMTP
setgid_group = postdrop
```

You can see a fairly typical Postfix `main.cf` configuration file with the addition of two ERB variables that use Facter facts to correctly populate the file. Each variable is enclosed in `<%=   %>` and will be replaced with the fact values when Puppet runs. You can specify any variable in a template like this.

This is a very simple template and ERB has much of the same capabilities as Ruby, so you can build templates that take advantage of iteration, conditionals and other features. You can learn more about how to use templates further at `http://docs.puppetlabs.com/guides/templating.html`.

■ **Tip** You can easily check the syntax of your ERB templates for correctness using the following command: `erb -x -T '-' mytemplate.erb | ruby -c`. Replace `mytemplate.erb` with the name of the template you want to check for syntax.

The postfix::service class

Next we have the `postfix::service` class, which manages our Postfix service:

```
class postfix::service {
  service { "postfix":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enable => true,
    require => Class["postfix::config"],
  }
}
```

And finally, we have the core `postfix` class where we include all the other classes from our Postfix module:

```
class postfix {
  include postfix::install, postfix::config, postfix::service
}
```

We can then apply our `postfix` module to the `mail.example.com` node:

```
node "mail.example.com" {
  include base
  include postfix
}
```

Now when the `mail.example.com` node connects, Puppet will apply the configuration in both the `base` and `postfix` modules.

CLASS INHERITANCE

As with nodes, Puppet classes also have a simple inherit-and-override model. A subclass can inherit the values of a parent class and potentially override one or more of the values contained in the parent. This allows you to specify a generic class and override specific values in subclasses that are designed to suit some nodes, for example:

```
class bind::server {
        service {
              "bind":
              hasstatus => true,
              hasrestart => true,
              enable => true,
        }
}

class bind::server::enabled inherits bind::server {
       Service["bind"] { ensure => running, enable => true }
}
class bind::server::disabled inherits bind::server {
       Service["bind"] { ensure => stopped, enable => false }
}
```

Here, class `bind::server` is the parent class and defines a service that controls the `bind` service. It uses the service resource type to `enable` the `bind` service at boot time and specify the service must be `stopped`. We then specify two new subclasses, called `bind::server::enabled` and `bind::server::disabled`, which inherit the `bind::server` class. They override the `ensure` and `enable` attributes, and specify that the `bind` service must be running for all nodes with the `bind::server::enabled` subclass included. If we wish to disable `bind` on some nodes, then we need to simply include `bind::server::disabled` rather than `bind::server::enabled`. The use of class inheritance allows us to declare the bind service resource in one location, the bind::server class, and

achieve the desired behavior of enabling or disabling the service without completely re-declaring the bind service resource. This organization structure also ensures we avoid duplicate resource declarations, remembering that a resource can only be declared once.

You can also add values to attributes in subclasses, like so:

```
class bind {
       service { "bind": require => Package["bind"] }
}

class bind::server inherits bind {
       Service["bind"] { require +> Package["bind-libs"] }
}
```

Here we have defined the `proxy` class containing the `bind` service, which in turn requires the `bind` package to be installed. We have then created a subclass called `bind::server` that inherits the `bind` service but adds an additional package, `bind-libs`, to the `require` metaparameter. To do this, we use the `+>` operator. After this addition, the `bind` service would now functionally look like this:

```
service { "bind":
       require => [ Package["bind"], Package["bind-libs"] ]
}
```

We can also unset particular values in subclasses using the `undef` attribute value.

```
class bind {
       service { "bind": require => Package["bind"] }
}

class bind::client inherits bind {
       Service["bind"] { require => undef }
}
```

Here, we again have the `bind` class with the `bind` service, which requires the `bind` package. In the subclass, though, we have removed the `require` attribute using the `undef` attribute value.

It is important to remember that class inheritance suffers from the same issues as node inheritance: variables are maintained in the scope they are defined in, and are not overridden. You can learn more at `http://projects.puppetlabs.com/projects/1/wiki/Frequently_Asked_Questions#Class+Inheritance+and+Variable+Scope`.

Managing MySQL with the mysql Module

Our next challenge is managing MySQL on our Solaris host, `db.example.com`. To do this we're going to create a third module called `mysql`. We create our module structure as follows:

```
mysql
mysql/files/my.cnf
mysql/manifests/init.pp
mysql/manifests/install.pp
mysql/manifests/config.pp
```

```
mysql/manifests/service.pp
mysql/templates/
```

The mysql::install class

Let's quickly walk through the classes to create, starting with `mysql::install`.

```
class mysql::install {
  package { [ "mysql5", "mysql5client", "mysql5rt", "mysql5test", "mysql5devel" ]:
    ensure => present,
    require => User["mysql"],
}

  user { "mysql":
    ensure => present,
    comment => "MySQL user",
    gid => "mysql",
    shell => "/bin/false",
    require => Group["mysql"],
}

  group { "mysql":
    ensure => present,
  }
}
```

You can see that we've used two new resource types in our `mysql::install` class, User and Group. We also created a `mysql` group and then a user and added that user, using the `gid` attribute, to the group we created. We then added the appropriate `require` metaparameters to ensure they get created in the right order.

The mysql::config class

Next, we add our `mysql::config` class:

```
class mysql::config {
  file { "/opt/csw/mysql5/my.cnf":
    ensure = > present,
    source => "puppet:///modules/mysql/my.cnf",
    owner => "mysql",
    group => "mysql",
    require => Class["mysql::install"],
    notify => Class["mysql::service"],
  }

  file { "/opt/csw/mysql5/var":
    group => "mysql",
    owner => "mysql",
    recurse => true,
    require => File["/opt/csw/mysql5/my.cnf"],
  }
}
```

You can see we've added a File resource to manage our /opt/csw/mysql5 directory. By specifying the directory as the title of the resource and setting the recurse attribute to true, we are asking Puppet to recurse through this directory and all directories underneath it and change the owner and group of all objects found inside them to mysql.

The mysql::service class

Then we add our mysql::service class:

```
class mysql::service {
  service { "cswmysql5":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enabled => true,
    require => Class["mysql::config"],
  }
}
```

Our last class is our mysql class, contained in the init.pp file where we load all the required classes for this module:

```
class mysql {
  include mysql::install, mysql::config, mysql::service
}
```

Lastly, we can apply our mysql module to the db.example.com node.

```
node "db.example.com" {
  include base
  include mysql
}
```

Now, when the db.example.com node connects, Puppet will apply the configuration in both the base and mysql modules.

AUDITING

In addition to the normal mode of changing configuration (and the --noop mode of modelling the proposed configuration), Puppet has a new audit mode that was introduced in version 2.6.0. A normal Puppet resource controls the state you'd like a configuration item to be in, like this for example:

```
file { '/etc/hosts':
   owner => 'root',
   group => 'root',
   mode => 0660,
}
```

This file resource specifies that the /etc/hosts file should be owned by the root user and group and have permissions set to 0660. Every time Puppet runs, it will check that this file's settings are correct and make changes if they are not. In audit mode, however, Puppet merely checks the state of the resource and reports differences back. It is configured using the audit metaparameter.

Using this new metaparameter we can specify our resource like this:

```
file { '/etc/hosts':
   audit => [ owner, group, mode ],
 }
```

Now, instead of changing each value (though you can also add and mix attributes to change it, if you wish), Puppet will generate auditing log messages, which are available in Puppet reports (see Chapter 9):

```
audit change: previously recorded value owner root has been changed to owner daemon
```

This allows you to track any changes that occur on resources under management on your hosts. You can specify this audit metaparameter for any resource and all their attributes, and track users, groups, files, services and the myriad of other resources Puppet can manage.

You can specify the special value of all to have Puppet audit every attribute of a resource rather than needing to list all possible attributes, like so:

```
file { '/etc/hosts':
   audit => all,
 }
```

You can also combine the audited resources with managed resources, allowing you to manage some configuration items and simply track others. It is important to remember though, unlike many file integrity systems, that your audit state is not protected by a checksum or the like and is stored on the client. Future releases plan to protect and centralise this state data.

Managing Apache and Websites

As you're starting to see a much more complete picture of our Puppet configuration, we come to managing Apache, Apache virtual hosts and their websites. We start with our module layout:

```
apache
apache/files/
apache/manifests/init.pp
apache/manifests/install.pp
apache/manifests/service.pp
apache/manifests/vhost.pp
apache/templates/vhost.conf.erb
```

The apache::install class

Firstly, we install Apache via the apache::install class:

```
class apache::install {
  package { [ "apache2" ]:
```

```
    ensure => present,
  }
}
```

This class currently just installs Apache on an Ubuntu host; we could easily add an `apache::params` class in the style of our SSH module to support multiple platforms.

The apache::service class

For this module we're going to skip a configuration class, because we can just use the default Apache configuration. Let's move right to an `apache::service` class to manage the Apache service itself.

```
class apache::service {
  service { "apache2":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enable => true,
    require => Class["apache::install"],
  }
}
```

This has allowed us to manage Apache, but how are we going to configure individual websites? To do this we're going to use a new syntax, the definition.

The Apache definition

Definitions are also collections of resources like classes, but unlike classes they can be specified and are evaluated multiple times on a host. They also accept parameters.

■ **Note** Remember that classes are singletons. They can be included multiple times on a node, but they will only be evaluated ONCE. A definition, because it takes parameters, can be declared multiple times and each new declaration will be evaluated.

We create a definition using the `define` syntax, as shown in Listing 2-7.

Listing 2-7. *The First Definition*

```
define apache::vhost( $port, $docroot, $ssl=true, $template='apache/vhost.conf.erb',
$priority, $serveraliases = '' ) {

  include apache

  file {"/etc/apache2/sites-enabled/${priority}-${name}":
    content => template($template),
    owner => 'root',
    group => 'root',
    mode => '777',
    require => Class["apache::install"],
```

```
    notify => Class["apache::service"],
  }
}
```

We gave a definition a title (`apache::vhost`) and then specified a list of potential variables. Variables can be specified as a list, and any default values specified, for example `$ssl=true`. Defaults will be overridden if the parameter is specified when the definition is used.

Inside the definition we can specify additional resources or classes, for example here we've included the `apache` class that ensures all required Apache configuration will be performed prior to our definition being evaluated. This is because it doesn't make sense to create an Apache VirtualHost if we don't have Apache installed and ready to serve content.

In addition to the `apache` class, we've added a basic file resource which manages Apache site files contained in the `/etc/apache2/sites-enabled` directory. The title of each file is constructed using the `priority` parameter, and the title of our definition is specified using the `$name` variable.

■ **Tip** The `$name` variable contains the name, also known as the title, of a declared defined resource. This is the value of the string before the colon when declaring the defined resource.

This file resource's `content` attribute is specified by a template, the specific template being the value of the `$template` parameter. Let's look at a fairly simple ERB template for an Apache VirtualHost in Listing 2-8.

Listing 2-8. *VirtualHost Template*

```
NameVirtualHost *:<%= port %>
<VirtualHost *:<%= port %>>
  ServerName <%= name %>
<%if serveraliases.is_a? Array -%>
<% serveraliases.each do |name| -%><%= "  ServerAlias #{name}\n" %><% end -%>
<% elsif serveraliases != '' -%>
<%= "  ServerAlias #{serveraliases}" -%>
<% end -%>
  DocumentRoot <%= docroot %>
  <Directory <%= docroot %>>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>
  ErrorLog /var/log/apache2/<%= name %>_error.log
  LogLevel warn
  CustomLog /var/log/apache2/<%= name %>_access.log combined
  ServerSignature On
</VirtualHost>
```

Each parameter specified in the definition is used, including the `$name` variable to name the virtual host we're creating.

You can also see some embedded Ruby in our ERB template:

```
<%if serveraliases.is_a? Array -%>
<% serveraliases.each do |name| -%><%= "  ServerAlias #{name}\n" %><% end -%>
<% elsif serveraliases != '' -%>
<%= "  ServerAlias #{serveraliases}" -%>
<% end -%>
```

Here we've added some logic to the `serveraliases` parameter. If that parameter is an array of values, then create each value as a new server alias; if it's a single value, then create only one alias.

Let's now see how we would use this definition and combine our definition and template:

```
apache::vhost { 'www.example.com':
  port => 80,
  docroot => '/var/www/www.example.com',
  ssl => false,
  priority => 10,
  serveraliases => 'home.example.com',
}
```

Here we have used our definition much the same way we would specify a resource by declaring the apache::vhost definition and passing it a name, www.example.com (which is also the value of the `$name` variable). We've also specified values for the required parameters. Unless a default is already specified for a parameter, you need to specify a value for every parameter of a definition otherwise Puppet will return an error. We could also override parameters, for example by specifying a different template:

```
template => 'apache/another_vhost_template.erb',
```

So in our current example, the template would result in a VirtualHost definition that looks like Listing 2-9.

Listing 2-9. *The VirtualHost Configuration File*

```
NameVirtualHost *:80
<VirtualHost *:80>
  ServerName www.example.com
  ServerAlias home.example.com
  DocumentRoot /var/www/www.example.com
  <Directory /var/www/www.example.com>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>
  ErrorLog /var/log/apache2/www.example.com_error.log
  LogLevel warn
  CustomLog /var/log/apache2/www.example.com_access.log combined
  ServerSignature On
</VirtualHost>
```

The final class in our module is the apache class in the init.pp file, which includes our Apache classes:

```
class apache {
  include apache::install, apache::service
}
```

You can see we've included our three classes but not the definition, apache::vhost. This is because of some module magic called "autoloading." You learned how everything in modules is automatically imported into Puppet, so you don't need to use the import directive. Puppet scans your module and loads any .pp file in the manifests directory that is named after the class it contains, for example the install.pp file contains the apache::install class and so is autoloaded.

The same thing happens with definitions: The vhost.pp file contains the definition apache::vhost, and Puppet autoloads it. However, as we declare definitions, for example calling apache::vhost where we need it, we don't need to do an include apache::vhost because calling it implies inclusion.

Next, we include our classes into our www.example.com node and call the apache::vhost definition to create the www.example.com website.

```
node "www.example.com" {
  include base
  include apache

  apache::vhost { 'www.example.com':
    port => 80,
    docroot => '/var/www/www.example.com',
    ssl => false,
    priority => 10,
    serveraliases => 'home.example.com',
  }
}
```

We could now add additional web servers easily and create additional Apache VirtualHosts by calling the apache::vhost definition again, for example:

```
apache::vhost { 'another.example.com':
    port => 80,
    docroot => '/var/www/another.example.com',
    ssl => false,
    priority => 10,
}
```

Managing Puppet with the Puppet Module

In our very last module we're going to show you Puppet being self-referential, so you can manage Puppet with Puppet itself. To do this we create another module, one called puppet, with a structure as follows:

```
puppet
puppet/files/
puppet/manifests/init.pp
puppet/manifests/install.pp
```

```
puppet/manifests/config.pp
puppet/manifests/params.pp
puppet/manifests/service.pp
puppet/templates/puppet.conf.erb
```

Our first class will be the `puppet::install` class which installs the Puppet client package.

```
class puppet::install {
  package { "puppet" :
    ensure => present,
  }
}
```

All of the operating systems we're installing on call the Puppet package `puppet`, so we're not going to use a variable here.

We do, however, need a couple of variables for our Puppet module, so we add a `puppet::params` class.

```
class puppet::params {
  $puppetserver = "puppet.example.com"
}
```

For the moment, this class only contains a Puppet server variable that specifies the fully-qualified domain name (FQDN) of our Puppet master.

Now we create our `puppet::config` class:

```
class puppet::config {

include puppet::params

  file { "/etc/puppet/puppet.conf":
    ensure = > present,
    content => template("puppet/puppet.conf.erb"),
    owner => "puppet",
    group => "puppet",
    require => Class["puppet::install"],
    notify => Class["puppet::service"],
  }
}
```

This class contains a single file resource that loads the `puppet.conf.erb` template. It also includes the `puppet::params` class so as to make available the variables defined in that class. Let's take a look at the contents of our template too:

```
[main]
    user = puppet
    group = puppet
    report = true
    reports = log,store
```

```
[master]
    certname = <%= puppetserver %>

[agent]
    pluginsync = false
    report = true
    server = <%= puppetserver %>
```

This is a very simple template, which we can then expand upon, or you can easily modify to add additional options or customize for your own purposes. You'll notice we've included configuration for both our master and the client. We're going to manage one `puppet.conf` file rather than a separate one for master and client. This is mostly because it's easy and because it doesn't add much overhead to our template.

We can then add the `puppet::service` class to manage the Puppet client daemon.

```
class puppet::service {
  service { "puppet":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enable => true,
    require => Class["puppet::install"],
  }
}
```

We can then create an `init.pp` that includes the `puppet` class and the sub-classes we've just created:

```
class puppet {
  include puppet::install, puppet::config, puppet::service
}
```

Just stopping here would create a module that manages Puppet on all our clients. All we need to do, then, is to include this module on all of our client nodes, and Puppet will be able to manage itself. But we're also going to extend our module to manage the Puppet master as well. To do this, we're going to deviate slightly from our current design and put all the resources required to manage the Puppet master into a single class, called `puppet::master`:

```
class puppet::master {

include puppet
include puppet::params

package { "puppet-server":
  ensure => installed,
}

service { "puppetmasterd":
    ensure => running,
    hasstatus => true,
    hasrestart => true,
    enable => true,
```

```
    require => File["/etc/puppet/puppet.conf"],
  }
}
```

You can see that our class puppet::master includes the classes puppet and puppet::params. This will mean all the preceding Puppet configuration will be applied, in addition to the new package and service resources we've defined in this class.

We can now add this new module to our nodes, leaving them looking like this:

```
class base {
  include sudo, ssh, puppet
}

node 'puppet.example.com' {
  include base
  include puppet::master
}

node 'web.example.com' {
  include base
  include apache

  apache::vhost { 'www.example.com':
    port => 80,
    docroot => '/var/www/www.example.com',
    ssl => false,
    priority => 10,
    serveraliases => 'home.example.com',
  }
}

node 'db.example.com' {
  include base
  include mysql
}

node 'mail.example.com' {
  include base
  include postfix
}
```

We've added the puppet module to the base class we created earlier. This will mean it's added to all the nodes that include base. We've also added the puppet::master class, which adds the additional resources needed to configure the Puppet master, to the puppet.example.com node.

Summary

In this chapter, you've been introduced to quite a lot of Puppet's basic features and language, including:

- How to structure modules, including examples of modules to manage SSH, Postfix, MySQL and Apache.

- How to use language constructs like selectors, arrays and case statements
- A greater understanding of files and templates
- Definitions that allow you to manage configuration, such as Apache VirtualHosts
- Variable scoping

You've also seen how a basic Puppet configuration in a simple environment might be constructed, including some simple modules to manage your configuration. Also, Puppet Forge contains a large collection of pre-existing modules that you can either use immediately or modify to suit your environment.

In the next chapter, we'll look at how to scale Puppet beyond the basic Webrick server, using tools like Mongrel and Passenger and allowing you to manage larger numbers of hosts.

Resources

- Puppet Documentation: `http://docs.puppetlabs.com`
- Puppet Wiki: `http://projects.puppetlabs.com/projects/puppet/wiki`
- Puppet Forge: `http://forge.puppetlabs.com`