3. Configure Puppet Master

The main configuration file for Puppet is etc/puppet/puppet.conf.
All Puppet related settings such as the definition of Puppet master, Puppet agent, Puppet apply and certificates are defined in this file.

Puppet configuration file mainly consists of the following config sections.

Main: This is known as the global section which is used by all the commands and services in Puppet. One defines the default values in the main section which can be overridden by any section present in puppet.conf file.

Master: This section is referred by Puppet master service and Puppet cert command.

Agent: This section is referred by Puppet agent service.

User: It is mostly used by Puppet apply command as well as many of the less common commands.

Following are the sample examples for config sections on both puppet master and agents.

Example agent config
[main]
certname = agent01.example.com
server = puppet
environment = production
runinterval = 1h

Example master config
[main]
certname = puppetmaster01.example.com
server = puppet
environment = production
runinterval = 1h
strict_variables = true

[master]
dns_alt_names = puppetmaster01,puppetmaster01.example.com,puppet,puppet.example.com
reports = puppetdb
storeconfigs_backend = puppetdb
storeconfigs = true
environment_timeout = unlimited

The basic changes to be made on puppet master configuration file

```
vi /etc/puppet/puppet.conf


[main]
certname = puppetmaster
```

The basic changes to be made on puppet agent configuration file

```
vi /etc/puppet/puppet.conf


[agent]

server = puppetmaster
```

The above commands update the puppet master and agent info in the configuration

4. SSL Sign Certificate Setup

When the Puppet agent software runs for the first time on any Puppet node, it generates a certificate and sends the certificate signing request to the Puppet master.

The below command from agent will request for the certificate from the master.

```
# puppet agent --verbose --no-daemonize -onetime
```

Before the Puppet server is able to communicate and control the agent nodes, it must sign that particular agent node's certificate. In the following sections, we will describe how to sign and check for the signing request.

List Current Certificate Requests.

On the Puppet master, run the following command to see all unsigned certificate requests.

```
$ sudo puppet cert list
```

As we have just set up a new agent node, we will see one request for approval. Following will be the output.

```


"agent01.example.com" (SHA259)

15:90:C2:FB:ED:69:A4:F7:B1:87:0B:BF:F7:ll:B5:1C:33:F5:76:67:F3:F6:45:AE:07:4B:F
```

```
6:E3:ss:04:11:8h
```

It does not contain any + (sign) in the beginning, which indicates that the certificate is still not signed.

Sign a Request

In order to sign the new certificate request which was generated when the Puppet agent run took place on the new node, the Puppet cert sign command would be used, with the host name of the certificate, which was generated by the newly configured node that needs to be signed.

As we have agent01.example.com's certificate, we will use the following Command.

```
$ sudo puppet cert sign agent01.example.com
```

Following will be the output.

Notice: Signed certificate request for agent01.example.com

Once the above is done, we have our infrastructure ready in which the Puppet master is now capable of managing newly added nodes.

5. Creating environments

In the IT industry, we can find different teams like development team, testing team, DB admin team etc. To manage infrastructure for each team separately we can create environments in puppet.

Enabling Directory Environments in Open Source Puppet

Directory environments are disabled by default. To enable them, you must:
Edit the config file
Create at least one directory environment

Edit puppet.conf
To enable directory environments, set environmentpath = $confdir/environments in the Puppet master's puppet.conf (in the [main] or [master] section).
Optionally, you can also:
Use the basemodulepath setting to specify global modules that should be available in all environments. Most people are fine with the default value.

See the section below about settings for more details.
Once you edit puppet.conf, directory environments will be enabled and config file environments will be disabled.

Create a Directory Environment

You must have a directory environment for every environment that any nodes are assigned to. At minimum, you should have a production environment. Nodes assigned to non existent environments cannot fetch their catalogs.
To create your first environment, create a directory named production in your environmentpath. (If a production directory doesn't exist, the Puppet master will try to create one when it starts up.) Once it is created, you can add modules, a main manifest, and an environment.conf file to it.

Restart the Puppet Master
Restart the web server that manages your Puppet master, to make sure the Puppet master picks up its changed configuration.
Global Settings for Configuring Environments

Puppet uses five settings in puppet.conf to configure the behaviour of directory environments:

- environmentpath is the list of directories where Puppet will look for environments.
- basemodulepath lists directories of global modules that all environments can access by default.
- default_manifest specifies the main manifest for any environment that doesn't set a manifest value in environment.conf.
- disable_per_environment_manifest lets you specify that all environments should use a shared main manifest. This requires default_manifest to be set to an absolute path.
- environment_timeout sets how often the Puppet will refresh information about environments. It can be overridden per-environment.

Creating an environment

We can create file structure for an environment, running the command

mkdir -p /etc/puppet/environments/production/{modules,manifests}

Sample environment.conf File

```
[master]
manifest= $confdir/environments/Production/manifests/site.pp
modulepath= $confdir/environments/Production/modules
```

Where "$confdir" represents "/etc/puppet".

6. Site.pp

/etc/puppet/environments/Production/manifests/site.pp

It is the file where we define the nodes and what changes to be made to get the node to a desired state in the production environment.

Syntax

```
# /etc/puppet/environments/production/manifests/site.pp
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

In the example above, only www1.example.com would receive the apache and squid classes, and only db1.example.com would receive the mysql class.

Node definitions look like class definitions. The general form of a node definition is:

- The node keyword
- The name(s) of the node(s), separated by commas (with an optional final trailing comma)
- An opening curly brace
- Any mixture of class declarations, variables, resource declarations, collectors, conditional statements, chaining relationships, and functions
- A closing curly brace

To create a group of multiple nodes for same node definition

```
node 'www1.example.com', 'www2.example.com', 'www3.example.com' {
  include common
  include apache, squid
}
```

Note: The function “include” can call puppet code from the manifests files. It is explained in detail in the next chapters

7. Modules

/etc/puppet/environments/Production/modules

Modules are how Puppet finds the classes and defined types it can use — it automatically loads any class or defined type stored in its modules.

It is a directory where we find all the manifest files, templates and files that are required to prepare the catalog for the nodes.

Creating Modules

By running the command below, we can create the file structure for the module.

puppet module generate <USERNAME>-<MODULE NAME> --environment < ENVIRONMENT NAME>

Module Layout

On disk, a module is simply **a directory tree with a specific, predictable structure:**

- <MODULE NAME>
  - manifests
  - files
  - templates
  - lib
  - facts.d
  - tests
  - spec

Example

This example module, named "my_module," shows the standard module layout in more detail:

15. my_module — This outermost directory's name matches the name of the module.
    1. manifests/ — Contains all of the manifests in the module.
        1. init.pp — Contains a class definition. **This class's name must match the module's name.**
        2. other_class.pp — Contains a class named **my_module::other_class.**
        3. my_defined_type.pp — Contains a defined type named **my_module::my_defined_type.**
        4. implementation/ — This directory's name affects the class names beneath it.
            1. foo.pp— Contains a class named **my_module::implementation::foo.**
            2. bar.pp — Contains a class named **my_module::implementation::bar.**
    2. files/ — Contains static files, which managed nodes can download.
        1. service.conf — This file's source => URL would be **puppet:///modules/my_module/service.conf.** Its contents can also be accessed with the file function, like content => file('my_module/service.conf').
    3. lib/ — Contains plugins, like custom facts and custom resource types. These will be used by both the Puppet master server and the Puppet agent service, and they'll be synced to all agent nodes whenever they request their configurations. See "Using Plugins" for more details.
    4. facts.d/ — Contains external facts, which are an alternative to Ruby-based custom facts. These will be synced to all agent nodes, so they can submit values for those facts to the Puppet master. (Requires Facter 2.0.1 or later.)
    5. templates/ — Contains templates, which the module's manifests can use. See "Templates" for more details.
        1. component.erb — A manifest can render this template with template('my_module/component.erb').
        2. component.epp — A manifest can render this template with epp('my_module/component.epp'). (The epp function is only available with the future parser enabled.)
    6. tests/ — Contains examples showing how to declare the module's classes and defined types.
        1. init.pp
        2. other_class.pp — Each class or defined type should have an example in the tests directory.
    7. spec/ — Contains spec tests for any plugins in the lib directory.

8. Manifests

Manifest is a collection of resources which are coupled inside the function or classes to configure any target system. They contain a set of Ruby code to configure a system.

Writing Manifest files
Puppet programs are called manifests. Manifests are composed of puppet code and their filenames use the .pp extension.

9. Classes

Classes are named blocks of Puppet code, which are stored in modules for later use and are not applied until they are invoked by name. They can be added to a node's catalog by declaring them in your manifests.

Defining a class makes it available by name, but doesn't automatically evaluate the code inside it. Before we can use a class, we must define it, which is done with the class keyword, a name, curly braces, and a block of code:

```
Class <CLASS NAME> {
  ... puppet code ...
}
```

This manifest does nothing.

Declaring a class evaluates the code in the class, and applies all of its resources.
This one actually does something.

We declare the classes in the main manifest file **/etc/puppet/manifests/site.pp**

```
node '<AGENT NAME>' {
  include <CLASS NAME 1>
  include <CLASS NAME 2>
.
.
}
```

Include Function:

It is used for declaration of one or more classes, which results in evaluating all the resources present inside those classes and finally add them to a catalog. The way it works is, include function accepts a class name, list of classes or a comma separated list of class names.

While defining classes, we should place puppet code in the manifest files which are made of
“ Resources ”

Resources are the fundamental unit for modelling system configurations. Each resource describes some aspect of a system, like a service that must be running or a package that must be installed. The block of Puppet code that describes a resource is called a resource declaration.

Simplified syntax

Resource declarations have a lot of features, but beginners can accomplish a lot with just a subset of these. For more advanced syntax (including expressions that declare multiple resources at once), see Resources (Advanced).
A resource declaration:

```
file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  mode   => '0600',
}
```

Every resource has a **resource type,** a **title,** and a set of **attributes:**
<TYPE> { '<TITLE>':
<ATTRIBUTE> => <VALUE>,
}

After
The form of a resource declaration is:

- The **resource type,** which is a word with no quotes.
- An opening curly brace ({).
- The **title,** which is a string.
- A colon (:).
- Optionally, any number of **attribute and value pairs,** each of which consists of:
  - An attribute name, which is a lowercase word with no quotes.
  - A => (called an arrow, “fat comma,” or “hash rocket”).
  - A value, which can have any data type.
  - A trailing comma.
- A closing curly brace (}).

10. Validating the syntax of manifest file(s)

$puppet parser validate [manifest] [manifest ...]

DESCRIPTION

This action validates Puppet DSL syntax without compiling a catalog or syncing any resources. If no manifest files are provided, it will validate the default site manifest.

When validating multiple issues per file are reported up to the settings of max_error, and max_warnings. The processing stops after having reported issues for the first encountered file with errors.

EXAMPLES

To validate the default site manifest at /etc/puppet/environments/production/modules/sample_module/manifests/init.pp:
$ puppet parser validate init.pp

To validate two arbitrary manifest files

```
$ puppet parser validate init.pp vhost.pp
```

11. Applying modules on Puppet agent

After validating the puppet code, we need the apply that module on the node. To achieve that we have to run the command

```
$puppet agent --test
```

The puppet agent is configured to run at a specific interval. The default is 30 minutes. You can change how often agent pulls the catalog by modifying the “runinterval” setting.

Example

```
root@puppetagent: puppet agent --test
Info: Retrieving plugin
Info: Caching catalog for puppetagent
Info: Applying configuration version '135555737643'
Finished catalog run in 0.10 seconds
```

12. Configuring the run interval

The Puppet agent service defaults to doing a configuration run every 30 minutes. You can configure this with the runinterval setting in puppet.conf:

```
# /etc/puppet/puppet.conf
[agent]
  runinterval = 2h
```

Disabling and re-enabling Puppet runs

Regardless of how you're running Puppet agent, you can prevent it from doing any Puppet runs by running

```
$sudo puppet agent --disable
```

You can re-enable it with

```
$sudo puppet agent --enable.
```

13. Deep Dive into Puppet Coding

Variables

Syntax

Assignment

```
$content = "some content\n"
```

Variable names are prefixed with a $ (dollar sign). Values are assigned to them with the = (equal sign) assignment operator.

Resolution

```
file {'/tmp/testing':
   ensure  => file,
   content => $content,
 }
```

In the above example, A file will be created with the information(of any data type) that is stored in the variable "$content"

Array

If we want to pass multiple parameters in puppet code, Puppet allows the use of arrays in multiple areas.

Syntax

Arrays are written as comma-separated lists of values surrounded by square brackets. An optional trailing comma is allowed between the final value and the closing square bracket.

['one', 'two', 'three']

The values in an array can be any data type.

Example

you can specify the packages in an array …

```
$enhancers = [ 'screen', 'strace', 'sudo' ]
package { $enhancers:
                    ensure => 'installed',
                }
```

14. Conditionals

Conditions are situations when the user wishes to execute a set of statement or code when the defined condition or the required condition is satisfied. Puppet supports two types of conditions.

Puppet supports “if” and “unless” statements, case statements, and selectors.

“If” Statements

“If” statements take a Boolean condition and an arbitrary block of Puppet code, and will only execute the block if the condition is true. They can optionally include elsif and else clauses.

Syntax

```
if $osfamily == 'redhat' {
package {'httpd':
ensure => 'present',
}
elsif $osfamily  == 'debian'{
Package {'apache2':
ensure => 'present',
}
else {
notify { 'No Package Found':
}
```

The general form of an “if” statement is:

The if keyword
A **condition**
A pair of curly braces containing any Puppet code
Optionally: the elsif keyword, another condition, and a pair of curly braces containing Puppet code
Optionally: the else keyword and a pair of curly braces containing Puppet code

15. Case Statements

Like “if” statements, **case statements** choose one of several blocks of arbitrary Puppet code to execute. They take a control expression and a list of cases and code blocks, and will execute the first block whose case value matches the control expression.

Syntax

```
case $operatingsystem {
   'Solaris':          { include role::solaris } # apply the solaris class
   'RedHat', 'CentOS': { include role::redhat  } # apply the redhat class
   /^(Debian|Ubuntu)$/:{ include role::debian  } # apply the debian class
   default:            { include role::generic } # apply the generic class
}
```

The general form of a case statement is:

The case keyword
A **control expression** (see below)
An opening curly brace
Any number of possible matches, which consist of:
A **case** (see below) or comma-separated list of cases
A colon
A pair of curly braces containing any arbitrary Puppet code
A closing curly brace

Example

16. Selectors

Selector statements are similar to case statements, but return a value instead of executing a code block. Selectors are useful when the user wishes to specify a resource attribute and variables which are different from the default values based on the facts or other variables.

Syntax

Selectors resemble a cross between a case statement and the ternary operator found in other languages.

```
$rootgroup = $osfamily ? {
    'Solaris'          => 'wheel',
    /(Darwin|FreeBSD)/ => 'wheel',
    default            => 'root',
}

file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => $rootgroup,
}
```

In the example above, the value of $rootgroup is determined using the value of $osfamily.
The general form of a selector is:

A **control variable**
The ? (question mark) keyword
An opening curly brace
Any number of possible matches, each of which consists of:
A **case**
The => (fat comma) keyword
A **value**
A trailing comma
A closing curly brace

17. Relationships and ordering

By default, Puppet applies resources in the order they're declared in their manifest. However, if a group of resources must *always* be managed in a specific order, you should explicitly declare such relationships with relationship metaparameters.
Syntax: Relationship metaparameters

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
```

Puppet uses four metaparameters to establish relationships, and you can set each of them as an attribute in any resource. The value of any relationship metaparameter should be a resource reference pointing to one or more **target resources**.

before — Applies a resource **before** the target resource.
require— Applies a resource **after** the target resource.
notify — Applies a resource **before** the target resource.
The target resource refreshes if the notifying resource changes.
subscribe — Applies a resource **after** the target resource. The subscribing resource refreshes if the target resource changes.

If two resources need to happen in order, you can either put a before attribute in the prior one or a require attribute in the subsequent one; either approach creates the same relationship. The same is true of **notify** and **subscribe.**
The two examples below create the same ordering relationship:

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
file { '/etc/ssh/sshd_config':
  ensure  => file,
  mode    => '0600',
  source  => 'puppet:///modules/sshd/sshd_config',
```

```
  require => Package['openssh-server'],
}
```

The two examples below create the same notifying relationship:

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  notify => Service['sshd'],
}
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

Since an array of resource references can contain resources of differing types, these two examples also create the same ordering relationship:

```
service { 'sshd':
  ensure  => running,
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
package { 'openssh-server':
  ensure => present,
  before => Service['sshd'],
}

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  before => Service['sshd'],
}
```

---

18. ERB Templates

First, let’s adjust the file declaration from the previous section. We’ll remove the source attribute and replace it with a content attribute.

```
file { '/etc/puppet/puppet.conf':

ensure => ensure,

owner => 'root',

group => 'wheel',

mode => '0644',

content => template('puppet:///puppet/puppet.conf.erb'),

}
```

The template() function takes a single argument: the URI of the ERB template. The format of that URI is always puppet:///modulename/filename. The file should be placed in the templates directory of the module. ERB templates should end with the .erb extension to indicate that the file contains tags for the ERB template processor.

Let's create the ERB template file.

```
$ cd sample_module/templates

$ vi  templates/puppet.conf.erb
```

The contents of the file should look like this:

Generated by Puppet ERB template processor

```
[main]

log_level = <%= @loglevel %>

# This is used by "puppet agent"

[agent]

log_level = <%= @agent_loglevel %>

server = <%= @server -%>.example.net

# This is used for "puppet apply"

[user]

log_level = <%= @apply_loglevel %>
```

Each instance of <%=@variable %> is replaced with the value of the Puppet variable named after the @sign. The variables named with the @ sign must exist in the same scope (within the module class) as the template declaration.

There are many other things you can do within an ERB template. You can lookup variables from another class using the scope.lookupvar() function, or use scope[] as if it was a Hash.

You can call Puppet functions using scope.function_puppet_function(). For example, you could call the Hiera function to lookup Hiera values within templates (although this practice is strongly discouraged). This would be done by using scope.function_hiera() to call the same heira() function we used when introducing Hiera.

server = <%= scope.function_hiera(['puppet::server']) -%>

Best Practice: Avoid placing direct Hiera calls within the template, as it divides the source of data for the template between the manifest file and hiera, ensuring confusion. Instead, source the Hiera variables within the manifest so that all variables are within scope.

As ERB templates were intended for inline Ruby development, you can put any Ruby statement within <% ... %> tags without the equals sign. Here's an example that would limit duplicate assignment of loglevels which don't differ.

[user]
<% if @apply_loglevel != @loglevel -%>
log_level = <%= @apply_loglevel %>
<% end -%>

By wrapping this line of the template within the Ruby block, it will skip outputting the configuration line if the loglevel matches the main loglevel, thus simplifying the configuration file.

Go ahead and test this change right now with puppet apply. You will see the contents of the puppet configuration file get updated.

19. Iterating over Values

Here's an example where we use the Ruby each () function to iterate through an array of tags which should be used to limit which resources are applied to the node, as we discussed in Part I. This example uses the dash creatively to suppress linefeeds and output multiple Puppet servers on a single line:

```
[agent]

tags = <% @taglist.each do |tagname| -%>

<%= tagname + ',' -%>

<% end -%>
```

You'll note that we don't put an @ sign before the variable name. That is because we are not referencing a variable in the Puppet module class, but instead from the locall loop shown in this example.

---

20. Introducing Hiera

In Puppet, Hiera is a key-value lookup tool for configuration data. Hiera is integrated into Puppet to provide dynamic lookup of configuration data for the manifest files.
With Hiera, we can provide node-specific information to a Puppet module. Hiera uses a customizable hierarchy to lookup the data .\

For example, We can organize our data in this way:

1. Company-wide defaults
2. Operating system specific changes
3. Site-specific information

Creating Hiera Backends

Hiera has two built-in data file backends, YAML and JSON, and then the Puppet data provider. Each backend support four data types:

1. Arrays
2. Strings
3. true/false (Boolean)
4. Hashes

Let’s go through how to utilize these data types in each backend.

Hiera Data in YAML

The most common way to provide data to Hiera is using the YAML file format. The file has a .yaml file extension.

Files in YAML format always start with three dashes on the first line. The YAML format utilizes white spaces indentation to indicate the relationships between data. YAML should always be written using spaces, never tabs, for indentation.

Here are some examples of strings, Boolean, arrays, and hashes in YAML.

```
# strings
agent_running: 'running'
# boolean expression
agent_atboot: true
# arrays
puppet_components:
- facter
- puppet
# hash of values
puppet:
ensure: 'present'
version: '4.0.4'
#  variable loopup
hostname: %{facts::hostname}
```

We can organize all the data within a single hash That could look as simple as this:

```
puppet:
```

agent_running: 'running'
agent_atboot: true
components:
- 'facter'
- 'puppet'

We can see, YAML provides a clean, easy to read way to provide data without too much complicated syntax.

Configuring Hiera
Puppet looks for a Hiera configuration file at the location specified by the hiera_config configuration variable. By default this is $codedir/hiera.yaml, or /etc/puppet/code/hiera.yaml in Puppet.

Backends
The configuration key :backends should provide an array which lists the backend data providers that Hiera should use. There are three built-in backends, the two data types we discussed previously, YAML and JSON, plus Hiera can utilize data from Puppet.

If you wish to utilize both built-in file types, you could configure it as follows.
:backends:
- yaml
- json

We will only be utilizing YAML within this book.

Backend Conguration

For each backend data provider, you name in the backends array, you should create a top-level entry with the name of the provider. For each backend provide a hash of configuration data.

For the two built-in file-based backends the only configuration key necessary is :datadir, which identifies the directory in which the data files reside.

:yaml:
:datadir: /etc/puppetlabs/code/environments/%{::environment}/hieradata
:json:
:datadir: /etc/puppetlabs/code/environments/%{::environment}/hieradata

As the files read by each backend must be named differently, you can use the same data directory for both data sources as shown above.

You'll note that we're using the top-level environment variable (defined by puppet master or client) to allow different environment data in each environment. Let's go ahead and create the hieradata directory now in the environment directories we created in the last chapter.

mkdir /etc/puppetlabs/code/environments/test/hieradata
mkdir /etc/puppetlabs/code/environments/production/hieradata

For the Puppet backend place the name of a Puppet module which contains the data in a :datasource configuration key.
:puppet:
:datasource: hieradata

With this configuration variables from a module named heiradata would be accessed for Hiera lookups. As mentioned previously this is only useful when the module provides dynamic lookup of data.

Hierarchy

The final mandatory parameter is :hierarchy. The hierarchy defines the priority order for lookup of configuration data. For single values Hiera will proceed through the hierarchy until it finds a value and then stop. For arrays and hashes Hiera will merge data from each level of the hierarchy, selecting the winner of conflicts based on the :merge_behavior configuration setting.

There are two types of data sources: static and dynamic. Static data sources are files explicitly named in the hierarchy which contain data. Dynamic data sources are files which are named using interpolation of local configuration data, such as the host- name or operating system of the node.

In a larger enterprise, the data lookup hierarchy could be quite complex, however I recommend the following for a good starting point.

1. Put default values in a file named global.yaml.
2. Put all operating system specific information in a file named for the OS family as returned by Facter, e.g. RedHat.yaml, Debian.yaml, FreeBSD.yaml, etc.
3. Put information specific to a single node within a file named the full hostname of the node with a .yaml extension.

You would implement this hierarchy using the following configuration syntax. As you can see, we are interpolating data provided by Facter to choose which files will be read.

:hierarchy:
- defaults
- "%{::hostname}"
- "%{::osfamily}"
- global

Naturally you can extend this hierarchy to use information like the domain name of the node or any other facter-provided node value.

If you have multiple backends configured, then Hiera will evaluate the entire hierarchy for the first configured backend, then evaluate the entire hierarchy in order forthe 2nd configured backend, etc.

Complete Example
Following is a complete example of a Hiera configuration file. This example is what we will use for the remainder of this book. It enables YAML data input from etc/puppetlabs/code/hieradata, with a hierarchy that uses host-specific information in preference to operating system family information, finally defaulting to values global to every host.

:backends:

```
- yaml
:yaml:
:datadir: /etc/puppetlabs/code/hieradata
:hierarchy:
- defaults
- "%{facts::clientcert}"
- "%{facts::osfamily}"
- global
```

---

21. Resource Types

Resource Type: Cron

Description
Installs and manages cron jobs. Every cron resource created by Puppet requires a command and at least one periodic attribute (hour, minute, month, monthday, weekday, or special). While the name of the cron job is not part of the actual job, the name is stored in a comment beginning with # Puppet Name: . These comments are used to match crontab entries created by Puppet with cron resources.
If an existing crontab entry happens to match the scheduling and command of a cron resource that has never been synched, Puppet will defer to the existing crontab entry and will not create a new entry tagged with the # Puppet Name: comment.

Example:

```
cron { 'logrotate':
  command => '/usr/sbin/logrotate',
  user    => 'root',
  hour    => 2,
  minute  => 0,
}
```

Command
*(**Property:** This attribute represents concrete state on the target system.)*
The command to execute in the cron job. The environment provided to the command varies by local system rules, and it is best to always provide a fully qualified command. The user's profile is not sourced when the command is run, so if the user's environment is desired it should be sourced manually.
All cron parameters support absent as a value; this will remove any existing values for that field.

hour
*(**Property:** This attribute represents concrete state on the target system.)*